ICOT Technical Memorandum: TM-0897

TM-0897

KL1プログラミング
入門編／初級編／中級編

瀧　和男

July, 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

KL1 プログラミング
入門編 / 初級編 / 中級編

新世代コンピュータ技術開発機構

1989 年 7 月 4 日

目次

はじめに

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 ICOT (新世代コンピュータ技術開発機構) にて設計され, 並列推論マシン PIM のソフトウェアとハードウェアのインタフェースとなる核言語として採用された, 汎用の並列プログラム言語である. KL1 は, 同じく ICOT で開発された並列論理型プログラム言語 GHC (Guarded Horn Clauses) 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言語である.

本テキストは全部で 3 部から構成されている. 第 1 部入門編では KL1 の基礎知識と基本的なプログラミング技法の習得を目的とし, 以下の事項を学ぶ.

- KL1 のプログラミングの基礎とその実行モデル
- 基本的なデータ構造
- 繰り返しの制御
- ユニフィケーションと同期制御
- プロセスの概念と使い方

KL1 の文法規則を詳しく解説する事はしないが, プログラム例を多く用いて説明していくため, 自然に KL1 の基本概念は理解できるはずである.

第 2 部初級編は本格的なプログラムを記述するのに必要な基礎知識の習得を目的とし, 以下の事項を学ぶ.

- プロセスを応用したプログラム例
- 節の選択方式の色々なバリエーション
- 組込み述語を用いた各種データ型の操作方法
- ゴールの優先度制御及び負荷分散制御の基本的な方法

第 3 部中級編は効率の良いプログラムを記述するのに必要なプログラミング技法の習得を目的とし, 以下の事項を学ぶ.

- 荘園機能とその使い方
- 計算の終了の判定
- 優先度制御の応用
- ストリーム通信あれこれ
- 要求駆動型プログラミング
- バッファリング
- 単一参照

なお, プログラム例は全て PDSS (Pimos Development Support System) で動かせるようなっている. また, 入出力などのわずかな変更で, マルチ PSI または擬似マルチ PSI でも動作させることができる. 理解を深めるために, 実際にプログラムを動かしてみることをお勧めする.

第Ⅰ部

入門編

第 1 章

基礎

本章では, KL1 の基本的な構文とその実行モデル, 及び簡単な制御構造について学び, KL1 のプログラミングとはどんなものかを理解する.

1.1 簡単なプログラムと実行モデル

ここではまず簡単な KL1 のプログラムを紹介して, 基本的な構文とその実行モデルを理解しよう. まず, 次の問題を KL1 でプログラムした例をみてみよう.

問題 1.1 図 1.1 のゴール木 (1) のように連結したゴール a, b, c を全て実行するプログラムを KL1 で作成せよ. なお, 実行する順序は矢印の方向に向かって行なうものとする. すなわち, この例ではゴールを a, b, c の順で実行する.

図 1.1: ゴール木 (1)

ここでゴールと言う用語が出てきたが, ここでは KL1 の処理の単位と考えておけばよい. このプログラムを作成するには, 次のような事項を理解する必要がある.

- KL1 の基本的な構文
- ゴールの実行の記述法
- ゴールの実行順序の制御法

では, 次のプログラムをみてみよう.

```
:- module tree1.
:- public a/0.
        a :- true | b.                  %(1)
        b :- true | c.                  %(2)
        c :- true | true.               %(3)
```

詳しい構文規則や用語の説明は後に回すものとして, まずこのプログラムが何を意味しているかを直感的にみてみよう.

最初の 2 行は前書きのようなものである. `:- module tree1.` は, このプログラムモジュールの名前を `tree1` と宣言した文である. また, モジュール内のゴールのうち, 他のモジュールからも呼びだし実行できるように宣言したものが, `:- public a/0.` である. 引数 (パラメタ) の数が 0 個で[1] a という名

[1] つまり, 引数はない. KL1 では C などとは異なり, 引数がない場合は引数リストのまわりの括弧も不要である.

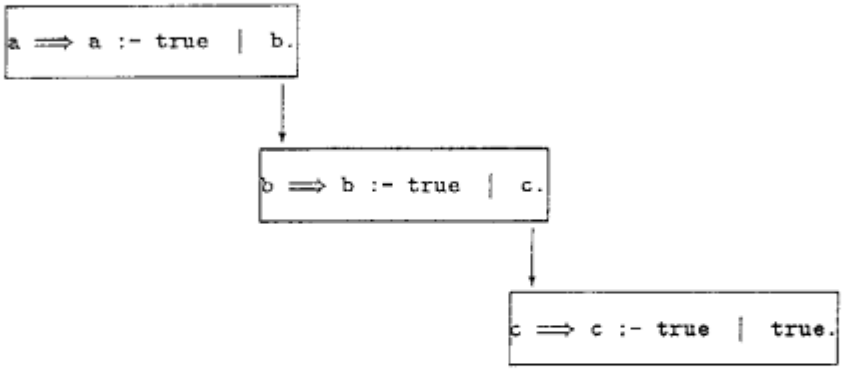


図 1.2: 実行過程 (1)

前のゴールを, 外部モジュールからも参照できるように宣言した文である. ここではこれ以上詳細な意味の説明は省く.

さて, このプログラムで, (1), (2), (3) はいずれも節と呼ばれる. 例えば, 節 (1) はゴール a がなすべき処理を定義したものである. ゴール a が呼ばれるとこの定義に従って実行が行なわれる. 記号 :- (コロンとマイナス, ネックと呼ぶこともある) と | (縦棒, コミットバーと呼ぶこともある) に囲まれた部分は, その節を選んでも良いか否かをチェックするための条件式である. この場合は true という無条件に成立するものが書かれている. この条件式が満たされると, この節を選ぶこととし, 次にコミットバーの後ろに書かれているゴール b を実行する.

節 (2) も全く同じである. 節 (3) も同様であるが, この場合はコミットバーの後にも true と無条件で成功するゴールが書かれているので, ゴール c の実行は直ちに終了するものと考えればよい.

では, このプログラムを実際に動かしてゴール a, b, c がこの順番で実行されるか確かめてみよう. KL1 のプログラムの実行過程をみるには, KL1 のデバッグツールのひとつであるトレーサを用いると良い. 従来型の言語では, ステッパに相当するツールである. トレーサの詳細な使用法は, 参考文献 [2] を参照のこと.

では, トレーサを使ってゴール a を実行してみよう.

```
?-tree1:a.                    %(1)
call    a                     %(2)
call    b                     %(3)
call    c                     %(4)
yes                           %(5)
?-
```

(1) の部分はユーザが入力した部分で, モジュール tree1 のゴール a を実行せよ, という意味である. また, (2), (3), (4) はシステムが表示した実行のトレースである. ゴールが実行されるたびに, その入り口で call というメッセージと共に, そのゴールの名前を表示する.

(5) の yes というメッセージは, ユーザが呼び出したゴールの実行が正しく終了した, という事を表わす. 表示から, ゴール a, b, c がこの順番で実行されていく過程がわかるであろう. この実行過程を図示すると, 図 1.2 のようになる.

図の中で四角で囲まれた部分が, 先のトレーサの出力で call の部分に相当する. また,

```
a ⟹ a:-true | b.
```

は, ゴール a の実行中に新たなゴール b の実行が行なわれる事を表わしている. 言い換えると, ゴール a の実行がゴール b の実行に置き換えられたことを表わしている.

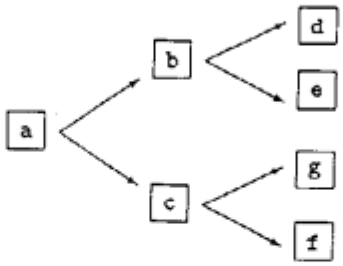


図 1.3: ゴール木 (2)

このように, KL1 のプログラムは幾つかのゴールの処理の定義から構成され, その実行は, 定義に従ってゴールを次々に実行していくものとみることができる. また見方を変えて KL1 のプログラムを論理式と見なすと, その実行は公理に従って論理式を次々と同値な論理式に書き換えていくような, 論理式の証明過程と見なすこともできる.

さて, 先のプログラム例では実行が明らかに逐次に進んでいった. では, 一体どうして KL1 が並列プログラミング言語なのであろうか. ここで, 図 1.3 に示されるようなゴール木をみてみよう. 先程と同じように, このゴール全てを実行するプログラムを書いてみよう. 実行する順番は矢印の方向に向かうものとする. 但し, ゴール a からはゴール b と c に枝分かれしており, このような場合はどちらを先に実行してもよいものとする.

```
:- module tree2.
:- public a/0.
        a:- true | b,c.          %(1)
        b:- true | d,e.          %(2)
        c:- true | g,f.          %(3)
        d:- true | true.         %(4)
        e:- true | true.         %(5)
        f:- true | true.         %(6)
```

このプログラムは, 先のプログラムと構造はほとんど同じである. ただ, コミットバーの後にゴールが複数ある部分が異なる. 節 (1) はゴール a の処理を定義したもので, ゴール b と c の両方を実行することを意味する. 問題の条件は b と c はどちらを先に実行してもよい, というものであったので, 順番は逆に書いてもかまわないであろう. では, またトレーサを使ってこのプログラムの動きをみてみよう.

```
?- tree2:a.
call a
call b
call d
call e
call c
call g
call f
?-
```

ゴール a, b, d, e, c, g, f がこの順番で実行された様子がわかる. 先程と同様に, 実行過程を表わした図 1.4 もみてみよう.

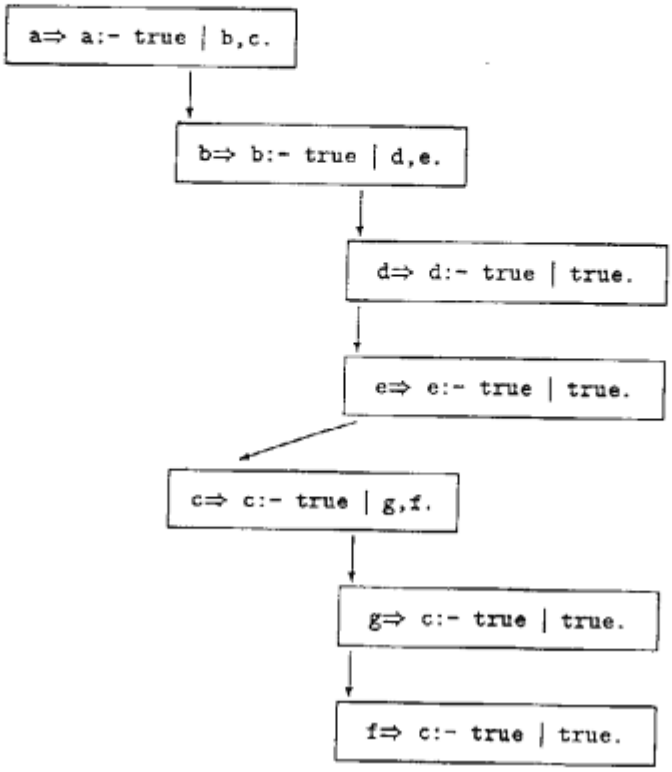


図 1.4: 実行過程 (2)

この実行過程を見る限り, 逐次的にプログラムが実行されていることに変わりはない. ではここで, もう一度プログラムをよくみてみよう.

節 (1) の中で, ゴール b と c の順番を逆に書くと, 実行する順番は変わるようにみえる. しかし実は b, c のように並べて書いたゴールは, どちらが先に実行されるか KL1 の言語仕様では決まっていないのである. また, プロセッサが複数あれば, これらは並列に実行される可能性もあるのである. ただし, ゴール b を実行してから d を実行する, という順番は変わらない. ゴール b を実行してみないと, ゴール d を実行しなければならないということはわからないのだから, これは当然だろう.

例えば, プロセッサが 2 台あるようなシステムで同じプログラムを実行した時に得られるようなトレース結果をみてみよう. また, 実行過程を表わした図 1.5も併せてみてみよう.

```
プロセッサ A   プロセッサ B
?-tree2:a.
call a
call b         call c
call d         call g
call e         call f
?-
```

図をみながら, 実行過程を追ってみよう.

1. まず, ゴール a がプロセッサ A で実行される.
2. ゴール a は, b と c に置き換えられる. これらはどちらを先に実行しても構わないし, またどのプロセッサで実行しても良い. この例では, b はプロセッサ A で実行され, c はプロセッサ B で実行されている.

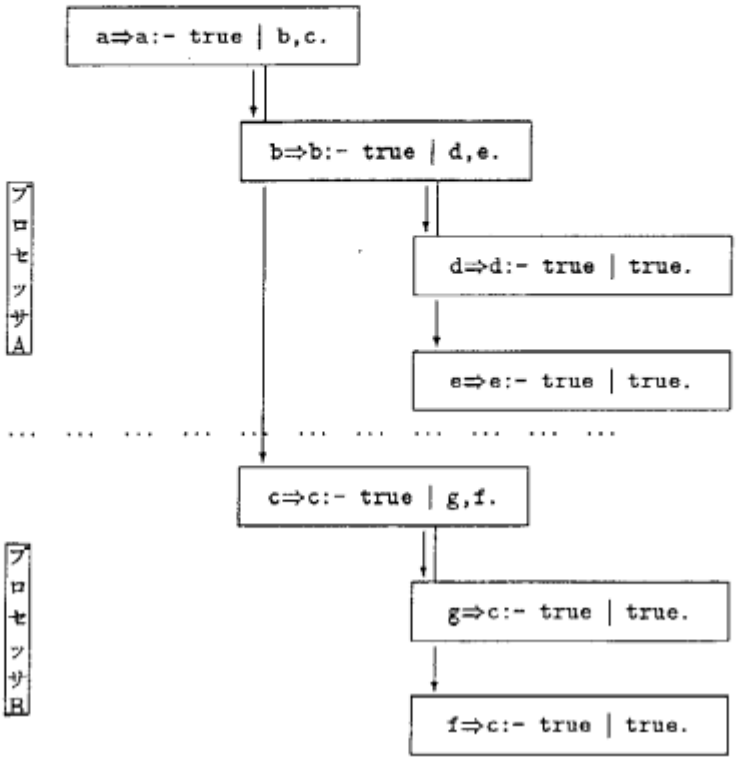


図 1.5: 実行過程 (3)

3. 次に，ゴール b は d と e に置き換えられる．これも先と同様に，実行する順番はどちらでも良く，どのプロセッサで実行しても良い．この例では，たまたま同じプロセッサ上で d，e の順で実行されている．

4. ゴール b が実行されるのと並列に，c は プロセッサ B 上で同様に実行されている．

5. 全てのゴールの実行が終わった時，ゴール a の処理は終わったことになる．

ここで，ゴールの実行順序についてまとめておこう．

> **KL1 では，コミットバー以降に並べられたゴールを実行する順番は，プログラム中で記述された順番とは無関係である．並列に実行可能であれば，これらは並列に実行されることもある．**

したがって，例ではたまたまゴールの実行がこのような順番で行なわれたが，もう一度実行すると，違った順番で行なわれる事もあるわけである．

実際には，並列実行は以下のどれかによって実現されることになろう．

- ユーザがプログラム中でそのゴールの実行プロセッサを指定する．
- コンパイラなどがコンパイル時に自動的にプロセッサ割り付けを行なう．
- オペレーティングシステムなどが実行時に自動的にプロセッサ割り付けを行なう．

現在のマルチ PSI システムでは，ユーザが積極的に指定する方法のみが実現されている．プロセッサへゴールを割り付ける方法に関しては，初級編にて詳しく説明する．

さて，このプログラムが先のトレース結果に示されるように実行されたとしよう．この場合，プロセッサが 1 台の時と 2 台の時では，実行速度は 4/7 に短縮されたと考えられる．但し，一般的な KL1 のプログラムでは，これほど単純に台数の効果が出るわけではない．例えば，ゴール c は b が終了した後でなければ実行できない，というように，同期をとる必要がある場合が多いためである．並列プログラミング言語においてはこの同期のメカニズムが大変重要であるが，これについては第 5 章にて詳しく説明する．

1.2 基本的な構文と条件分岐

この節では，もう少しプログラムらしいプログラム例を示し，KL1 の基本的な構文と，制御構造の例として条件分岐を勉強する．

まず，次のような問題を考えてみよう．

問題 1.2 引数に与えられたデータが 1 であれば結果として 0 を返し，0 が与えられれば 1 を返すようなプログラムを作れ．

このプログラムを作るためには，次のような事項を理解する必要がある．

- 引数の与え方．
- 条件の判定と分岐の方法．
- 結果の返し方．

次のプログラムがこの問題の解であるが，プログラムを説明する前に，これを用いて KL1 の構文を簡単に説明する．

```
:- module not.
:- public not/2.
        not(In, Out) :- In=:=0 | Out=1.    %(1)
        not(In, Out) :- In=:=1 | Out=0.    %(2)
```

まず節 (1) と節 (2) をみてみよう．

$$\text{predicate}\begin{cases}\texttt{not(In, Out) :- In=:=0 | Out=1.} \\ \texttt{not(In, Out) :- In=:=1 | Out=0.}\end{cases}$$

次に，節 (1) をみてみよう．

$$\underbrace{\underbrace{\underbrace{\texttt{not(In, Out)}}_{head}\ \texttt{:-}\ \underbrace{\texttt{In=:=0}}_{goal}}_{guard}\ \underbrace{\texttt{|}}_{commit}\ \underbrace{\texttt{Out=1.}}_{goal}}_{clause}\qquad \underbrace{\underbrace{\texttt{Out=1.}}_{goal}}_{body}$$

また *head* の部分に着目すると，

$$\underbrace{\texttt{not}}_{predicate\ name}\ \underbrace{\texttt{(}\ \overbrace{\texttt{In}}^{variable}\ \texttt{,}\ \overbrace{\texttt{Out}}^{variable}\ \texttt{)}}_{arguments}$$

となる．以下に各用語の説明をする．ただし，これはあくまでも以下の説明のための用語説明であり，厳密な定義ではない．

Predicate (述語) KL1 プログラムを構成する単位で, 手続型言語では関数, 或いはサブルーチンに相当するものである. KL1 でプログラムを書くと言う事は, 述語を定義する事に等しい.

述語はその名前 (述語名) と, 0 個以上の引数を持っている. また, その定義は 1 個以上の節を並べる事により行なう. 述語名が同じであってもその引数の数 (アリティ) が違えば, それは別の述語である点に注意する事.

Clause (節, 或いはガード付きホーン節) 述語定義を構成する要素である. ガード部とボディ部から構成される.

Commit operator (コミット演算子) 節の中のガード部とボディー部を区切る演算子.

Head (前頭部, 或いはヘッド) :- より左側の原子論理式 *(atomic formula)*. 即ち, 述語名とその引数のペアからなる `f(X,Y,…)` の形式.

Predicate name (述語記号, 或いは述語名) 述語の名前.

Variable (変数) 節の中で, 引数を受け渡すためのもの. 同じ節の中では, 同じ名前の変数は同一のものである. ただし, 別の節では同じ名前でも別のものである. 変数に関しては, 次章のデータ型の所で詳しく述べる.

Guard (ガード) 節の中のコミット演算子より左側の部分. 上記の not の定義では:- の右側にはゴールが 1 つしか記述されてないが, カンマ , で区切って複数のゴールを記述する事ができる. 一般的に, ガード部にはその節を選ぶかどうかを表す条件式を記述する. 詳しくは後で述べる.

Body (ボディ) 節の中のコミット演算子より右側の部分. ガードと同じくカンマ , で区切って複数のゴールを記述する事ができる.

Goal (ゴール) :- の右側の原子論理式をゴールと呼ぶ. ゴールには, ガード部に書かれるガードゴールと, ボディー部に書かれるボディーゴールがある.

ガードゴールは節の選択条件を記述したもので, システムに組込みの, しかも特定の述語しか書けない.

ボディーゴールはゴールの実行を表すもので, ユーザが定義した述語, およびシステムに組込みの述語を書く事ができる.

Arguments (引数) 述語の引数 (パラメタ). 引数は 0 個以上である.

Arity (アリティ, 或いは引数の数) 引数の数. 述語名が同じでも, アリティが違えば別の述語である.

ではプログラムの内容を見てみよう. まず, 節 (1) は述語名 not を定義した節の一つである. 引数としては In と Out を持つ. 頭文字が大文字の名前は変数であり, この場合は引数として In と Out いう変数が与えられている. また, 同一節の中で用いられている同じ名前の変数は, 同一の変数を表わす.

ガードの部分には, 述語を定義した複数の節のうち, どの節を選択するかを示す条件式が書かれている.

節 (1) では述語 not の引数 In が 0 であるかどうかをチェックする.
節 (2) では引数 In が 1 であるかどうかをチェックする.

ここで, ガード部のゴール =:= は左辺と右辺の値が等しいかどうかをチェックするシステムに組込みのもので, 組込み述語と呼ぶ. また, ガード部に記述できる特定の組込み述語なので, ガード組込み述語とも呼ぶ.

また, 節 (1) と (2) とはどちらも述語 not を定義したものであるが, このように 2 つ以上の節を並べることによって, ガード部に書かれた条件による条件分岐を記述する. 即ち, 引数 In が 0 の場合には節 (1) が選択され, 1 の場合には節 (2) が選択される. 従って, 節を並べる順番はどちらでも構わない. 但し,

本講座の上級編で述べるが，節を並べる順番が実行速度に影響を及ぼすこともある，ということを覚えておこう．

では，引数 In に値 1 を与えてゴール not を実行した時のプログラムの動きをみてみよう．

1. システムは述語 not の定義してある節を探す．この場合は節 (1) と節 (2) があるので，このうちのガード条件を満たす節を一つ選ぶ．どちらの節のガードを先にチェックするかはシステムに依存する．どちらが先にチェックされてもよいようにガード条件の記述には注意すること．

2. この場合は節 (2) がガード条件を満たすので，節 (2) を選ぶ．

3. 次に，処理はボディー部のゴールの実行に移る．

4. ボディー部のゴール Out=1 は，変数 Out の値を 1 に決める組込み述語である．
   従って，結果を返すための変数 Out の値が 1 に決まって，このプログラムの実行は終わる．

では，このプログラムを実際に動かしてみよう．今度はトレーサは使わずに実行する．

```
?- not:not(1,X).                    %(1)
yes                                 %(2)
?- not:not(1,X)|X.                  %(3)
X=0                                 %(4)
?- not:not(0,X)|X.                  %(5)
X=1                                 %(6)
?- not:not(2,X)|X.                  %(7)
fail                                %(8)
?-
```

(1) のところでは，引数 In に 1 を与えてゴール not を呼び出している．ゴールが呼び出されると，システムはそのゴールを実行する．実行が終了すると，変数 Out の値は 0 に決まっている筈である．しかし PDSS システムでは，このままではその値を表示してはくれない．(2) のように実行が正常に終了した事を表わすメッセージ yes が表示される．

実行後の変数の値を表示させるには，(3) のように | の後に値を表示したい変数名を書く．すると，(4) のように実行後の変数 Out の値を表示してくれる．(5),(6) は同様である．

(7) は，ガード部の条件がいずれも満たされないような呼び出しである．この場合は，(8) のようなメッセージ fail が表示される．これはゴールの実行が失敗したという意味であるが，これは即ちエラーである．エラーを起こすようなバグ付きのプログラムでは困るので，例えば次のようにプログラムを書き換えておこう．

```
:- module not.
:- public not/2.
        not(In,Out):- In=:=0 | Out=1.          %(1)
        not(In,Out):- In=:=1 | Out=0.          %(2)
        not(In,Out):- In=\=0,In=\=1 | Out=-1.  %(3)
```

このプログラムでは，節 (1) と (2) のいずれの条件も満たさない時の条件として，節 (3) が新たに付け加えられた．即ち，In が 0 でも 1 でもないときには，Out を -1 に決めるようにしたわけである．ガード部のゴール =\= は，等しくないかどうかを表わす組込み述語である．

今度は，同じプログラムを少し違った書き方をした例を示そう．次のプログラムは先のプログラムと全く等価なものである．

```
:- module not.
:- public not/2.
```

```
not(0, Out):- true | Out=1.              %(1)
not(1, Out):- true | Out=0.              %(2)
not(In,Out):- In=\=0,In=\=1 | Out=-1.    %(3)
```

まず節(1)のガード部をみてみよう．この部分には，先にも述べたように節を選択するための条件が記述される．例えば，節(1)ではヘッドの引数の位置には数0が書かれている．これは即ち，一番目の引数が0で呼ばれた場合には，節(1)が選択されることを意味する．従って，これはIn=:=0をガード部に書いた時と全く同じである．

このように，ヘッド部に条件を書くかガードゴール部に条件を書くかは，プログラムの見易さだけの問題であり，時と場合に応じて使い分ければよい．

さて，これまでに示した例ではガード部の条件は排他的に記述したが，どちらの節を選んでもよいような曖昧なプログラムを記述する場合には，このように排他的な条件を書かなくてもよい．例えば，次のような例をみてみよう．

```
:- module redundant.
:- public redundant/2.
redundant(In,Out):- true | Out=1.      %(1)
redundant(In,Out):- true | Out=2.      %(2)
```

引数Inがいかなる値であっても，節(1)と(2)のガード条件を満たす．従って，答を返す変数Outには1が返ってきたり，2が返って来たり，偶然に左右されて値が決まる．従って，ガード部の条件の記述の仕方には注意する必要がある．

では，これまでに学んだことを利用して，次の問題を解いてみよう．

問題: 2つの整数を引数として与え，どちらか大きい方を結果として返すようなプログラムを作れ．

ここまで勉強してきた人であれば，次のようなプログラムはすぐに作れるであろう．

```
:- module max.
:- public max.
max(X,Y,Max):- X >= Y | Max=X.         %(1)
max(X,Y,Max):- X =< Y | Max=Y.         %(2)
```

プログラムの意味はみれば明らかであろう．但し，XとYが等しい時には節(1)が選ばれるか節(2)が選ばれるかわからない．しかし答は正しいのでこれは正しいプログラムである．ガード部の条件を排他的に書きたければ次のように書けばよい．いずれのプログラムも正しく，どちらでも良い．

```
:- module max.
:- public max.
max(X,Y,Max):- X >= Y | Max=X.         %(1)
max(X,Y,Max):- X <  Y | Max=Y.         %(2)
```

実行例

```
?-max:max(1,2,V)|V.                    %(3)
  V=2.                                 %(4)
?-max:max(200,-200,V)|V.               %(5)
  V=200.                               %(6)
?-max:max(25,25,V)|V.                  %(7)
  V=25.                                %(8)
```

第 2 章

データ型

本章では，KL1 で扱える基本的なデータ型について説明する．但し，入門編で扱うプログラムで用いるものについてのみ詳しく説明し，その他については本講座の初級編にて説明する．

2.1 変数の取り扱い

変数については，これまでに何度も出てきたので，大体のイメージはつかめているであろう．ただし，KL1 の変数は他の言語の変数とは大変異なるものであり，その扱いを充分理解しておく必要がある．

一般の言語では，変数に値を入れることを代入すると呼ぶ．しかし，前章までの説明では，例えば変数の値を 1 に決める，という表現を用いてきた．これは，KL1 の変数がシングルアサインメント変数だからである．即ち，一般の言語では変数に一旦値を代入した後で，更に別の値を代入することができるのに対して，KL1 では一旦値が決まったらその値は変更することができないからである．

では，次に KL1 の変数の取り扱いについて説明する．変数は，プログラム中では次のように記述する．

- 頭文字が大文字で始まる文字の並び A,B,C,Var,Var1
- 下線から始まる文字の並び ..... _Var,_hensuu,_123
- 下線 ..... _

また，プログラム中で組込み述語を用いて動的に変数を作ることもできるが，入門編では説明しない．

同じ名前の変数は，同一節の中で用いられているものについては同じ変数を表わす．ただし，異なる節で用いられているものは，別の変数である．このような変数は論理変数と呼ばれる．

KL1 の変数には，次のような 2 つの状態がある．

1. 未定義の状態
   変数の値がまだ決まってない状態である．プログラム中に記述した変数は，そのゴールが呼ばれた時には最初は全てこの状態である．この状態の変数のことを，未定義変数と呼ぶ．

2. 定義の状態
   変数が整数などの値に決まった状態である．この状態の変数のことを定義変数，或いはバインドされた変数と呼ぶ．

これら 2 つの状態は，図 2.1 に示されるように状態遷移する．

ここで，変数が未定義から定義の状態への遷移 (1) は，例えば未定義変数 Out の値を 1 と決めた時の遷移である．このような操作をユニフィケーションと呼ぶ．また，操作する事をユニファイすると呼ぶ．すなわち，ユニフィケーションの結果 Out の値が 1 に決まったわけである．

変数は，1 のような整数とユニファイできるだけではない．次節で説明するデータ型全てのものとユニファイできる．なお，ユニフィケーションの詳細な説明は後に行なう．

未定義から未定義への遷移 (2) は，若干奇異に見えるかも知れない．これは，未定義変数同士のユニフィケーションを行なった時の遷移である．例えば，In と Out が未定義変数だったとしよう．組込み述語 = を用いて，次のようにして未定義変数同士のユニフィケーションをしたとしよう．

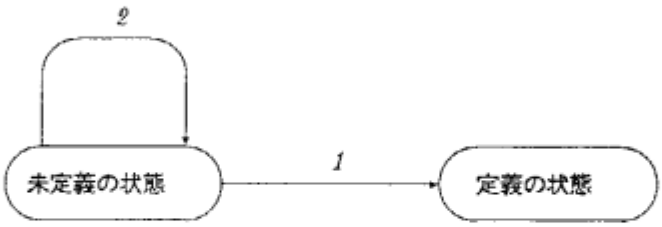


図 2.1: 変数の状態遷移

```
In = Out
```

これ以降は，両方の変数は同一の変数と同じ取り扱いになる．例えば，ここでInと1をユニファイすると，Outの値も1に決まる．

ユニフィケーションは，このような組込み述語を使ったものばかりではない．例えば，述語notを定義したプログラムとその実行例をみてみよう．

```
:- module not.
:- public not/2.
        not(In, Out):- In=:=0 | Out=1.          %(1)
        not(In, Out):- In=:=1 | Out=0.          %(2)

                実行例

        ?- not:not(0,Result)|Result.            %(3)
           Result=1                             %(4)
        ?-
```

節中の変数In,Outは，ゴールが呼ばれた時には未定義変数である．また，ゴールを呼び出した時の変数Resultも未定義変数である．ゴールを呼び出すと，次のように引数同士のユニフィケーションを行なう．

$$0 \Longleftrightarrow \mathtt{In}$$
$$\mathtt{Result} \Longleftrightarrow \mathtt{Out}$$

ここで，矢印の左側のデータが呼び出した側のもので，右側が呼び出された側のものである．このように，ゴール呼び出しの時にも引数同士のユニフィケーションが行なわれる．これを，ヘッドユニフィケーションと呼ぶ．

0とInのユニフィケーションは，変数を未定義状態から定義状態に変えるものである．ResultとOutとのユニフィケーションは，未定義変数同士のユニフィケーションである．

未定義変数同士のユニフィケーションを行なった後は，それらは同一変数として扱われると先に述べた．従って，ResultとOutをユニファイした後は，これらは同一の変数として扱われる．即ち，Outに1をユニファイすると，Resultの値は1に決まる．その結果，(4)のように表示されたわけである．

2.2 KL1で扱えるデータ型

KL1で扱えるデータ型には以下のものがある．

```
整数        ···  123,16'ADB,8'37
アトム      ···  abc,'ABC'
リスト      ···  [1,2,3],[1 | X]
ベクタ      ···  {1,2,3},{a,X,b},{}
ストリング  ···  "abc",""
```

- 整数
  整数データである．2 進数から 36 進数で記述することができる．

- アトム
  プログラム中では，アルファベットの小文字から始まる文字の並びで表わす．シングルクォートで括れば，大文字で始まるものも，数字もアトムとする事ができる．アトムは同一性だけが意味を持つデータであり，プログラム中に記述したもの以外にもアトムデータは用いられるが，その方法についてはここでは説明しない．つまり，アトムデータは必ずしも印字可能ではないとここでは理解しておこう．ここで，同一性だけが意味を持つデータとは，同じ印字名を持つアトムは同一のものである，という意味である．例えば，述語 not をアトムを用いて書き直して見よう．

```
:- module not.
:- public not/2.
        not(In, Out):- In=:=true  | Out=false.        %(1)
        not(In, Out):- In=:=false | Out=true.         %(2)

                        実行例

        ?- not:not(true,Result)|Result.               %(3)
           Result=false                               %(4)
        ?-
```

  このように，プログラム中の節 (1) に出現するアトム true と，ゴールの呼びだし (3) に出現するアトム true とは同一のものである．

- リスト
  その名前から類推できるように，データの並びである．リストのデータ構造とその基本的な操作に関しては，後ほど詳しく説明する．

- ベクタ
  配列型のデータである．PDSS システムでは，ベクタの各要素は 16 ビットである．なお，入門編ではベクタ操作は勉強しないが，その記法については後ほど詳しく説明する．

- ストリング
  文字列データである．各要素のサイズには，1 ビットのものから 32 ビットのものまで用意されている．プログラム中でダブルコートで括った文字列は，PDSS システムでは 8 ビットのストリングとなる．なお，入門編ではストリング操作は勉強しない．

2.3 リスト

本節では，KL1 では大変重要なデータ型であるリストのデータ構造とその基本的な操作方法を勉強する．

2.3.1 リストのデータ構造

リストとは，その名前から類推できるようにデータの並びである．個々のデータ，即ちリストの要素はリストセルと言う図 2.2に示すような構造をしている．

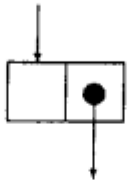


図 2.2: リストセルの構造

通常リストセルは1ワード, 或いは2ワードを使って実現されている. ここで, 左側の箱にはリストの要素データが格納される. この領域は, *Lisp* の流れで Car (カー) 部と呼ばれる. 例えば, 整数の1などがここに格納される. データ部に入り切らないデータ, 例えばベクタやストリングデータ等は, それらへのポインタが格納される. また KL1 の特徴であるが, 変数へのポインタも格納できる. 即ち, リストの要素として変数を格納できるわけである.

右側の箱には, 次の要素へのポインタが格納される. 即ち, 次のリストセルへのポインタである. この領域は Cdr (クダー) 部と呼ばれる. この図ではポインタデータが入っていることを, 黒丸にて表わしている. 従って, Car 部にポインタが格納された場合にも, 黒丸で表わされるものとする.

また, 最終要素を表わすリストセルには, ポインタ部に通常 □(空リスト或いはニルと読む) が入っている. なお, 空リストはアトム型のデータである.

ここで整数の 100 を要素とする1要素のリストデータの構造を, 図 2.3に示す. 要素の数が1なので1つのリストセルから構成され, Car 部には整数の 100, Cdr 部には [] が入っている. また, このリストは次のように表記する.

```
[100]
```

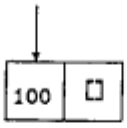


図 2.3: 1要素のリスト構造 [100]

次に, 先の1要素のリストの先頭に要素 200 を付け加えたリストの構造をみてみよう (図 2.4).

```
[200,100]
```

図をみてわかるように, この2要素のリスト構造は, 先の図 2.3のリストの前に, Car 部が 200 のリストセルを連結したものである. 言い換えると, Cdr 部が 200 のリストセルの Cdr 部に, Car 部が 100 の終端セルへのポインタをセットしたものである.

要素の数が多いリストも, これを繰り返した構造をしている. 次のような4要素のリスト構造を, 図 2.5に示す.

```
[1,2,3,4]
```

このように, リストデータは先頭要素の付け加えが簡単に行なえる, という特徴を備えており, KL1 では大変良く使われるデータである. 更に, 色々なリスト構造をみてみよう.

先に説明したように, リストセルの Car 部には全てのデータ型のデータを格納することができる. 例えば, 次のようなリストデータの構造を, 図 2.6 に示す.

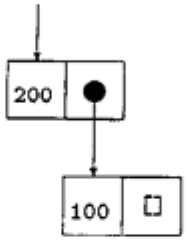


図 2.4: 2 要素のリスト構造 [200,100]

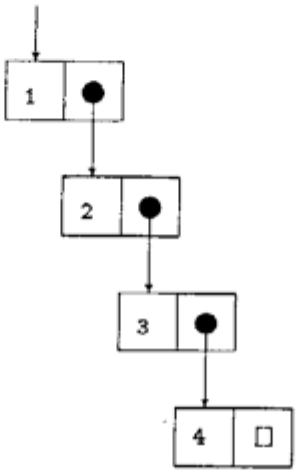


図 2.5: 4 要素のリスト構造 [1,2,3,4]

```
[1,[2,3],4,5]
```

このリストは 4 要素のリストで, 2 番目の要素が更に 2 要素のリスト [2, 3] になっている.

リストの表記には次のようなものもある.

```
[H|T]
```

ここで, H と T は変数である. このリストは, 先頭の要素が H で, 残りが T であるようなリストである. このように表記したリストを, D リスト *(Difference List)*, 或いは差分リストと呼ぶ. 厳密には, D リストはリストの使い方の一手法であるが, 詳しくは初級編にて述べる. 図 2.7 にその構造を示す.

D リスト [H|T] の変数 H を 100 とユニファイし, また, T を [] とユニファイすると, 図 2.3と同じリスト [100] になる. また, 次のように表記する事もできる.

```
[1,2,3|T]
```

このリストは, 最初の 3 要素がそれぞれ 1,2,3 で, 残りは変数であるようなリストである. この構造を図 2.8に示す.

このリストの変数 T をリスト [4] とユニファイすれば, 図 2.5と同じ構造のリスト [1,2,3,4] になる.

D リスト [H|T] は, 主にリストを分割する目的で用いられる. 次のような例を考えてみよう.

```
[1,2,3]⟸⟹[H|T]
```

このようなユニフィケーションを行なうと, リスト [1,2,3] を分割してその先頭要素 1 を H とユニファイし, 残りのリスト [2,3] を T とユニファイする. その様子を図 2.9に示す.

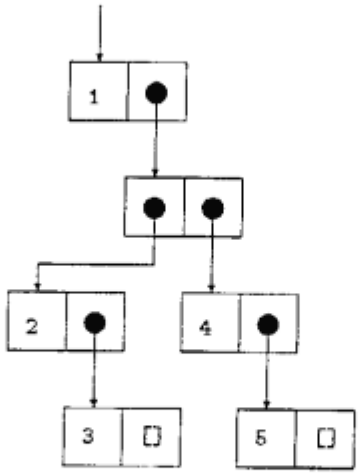


図 2.6: 複雑なリスト構造 [1,[2,3],4,5]

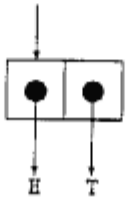


図 2.7: D リストの構造 (1) [H|T]

この例は，リストを分割するのに D リストを用いた例であるが，この性質を逆に利用してリストの頭に新たに要素を付けるのにも用いられる．次の例をみてみよう．

```
T⟸[1,2,3]
H⟸0
List⟸[H|T]
```

これは，3 回のユニフィケーション操作によって，リスト [1,2,3] の先頭に整数 0 を付け加える操作を表わしている．その結果，変数 List は [0,1,2,3] というリストになる．これらのユニフィケーション操作は，ボディーゴール中で組込み述語 = を用いて行なわれる．なお，リストの先頭に要素を付け加えるユニフィケーション操作を，今後は簡単に次のように記述することにする．

```
T⟸[1,2,3]
List⟸[0|T]
```

また，D リストを用いると 2 つのリストを結合することができる．次の例をみてみよう．

```
List1⟸[1,2|T]
List2⟸[3,4,5]
T⟸List2
```

この 3 回のユニフィケーションで，変数 List1 はリスト [1,2,3,4,5] になる．

これまでに学んだ D リストの操作を組み合わせると，リストの中間に要素を挿入することもできる．次の例をみてみよう．

```
List1⟸[1,2|T]
List2⟸[4,5]
T⟸[3|List2]
```

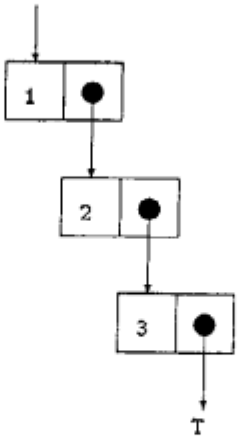


図 2.8: D リストの構造 (2) [1,2,3|T]

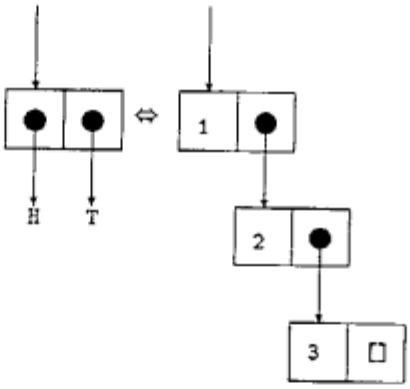


図 2.9: リストと D リストのユニフィケーション

この 3 回のユニフィケーションで, 変数 List1 はリスト [1,2,3,4,5] になる.

2.3.2 リスト操作

リストのデータ構造と, 簡単なリスト操作については前節で学んだ. ここでは, これを応用したプログラムを用いて, 更にリスト操作の基本を勉強する. まず, 次の問題を解いてみよう.

問題 2.1 Lisp の基本関数である car,cdr,cons を KL1 で書いてみよう.

まず *Lisp* の基本関数の説明を行なう.

- car　リストの先頭要素を得る関数である. ここでは, 引数としてリストを与え, その先頭要素を別の引数に返すようなプログラムを作成してみよう. なお, 空リストが与えられた時にはアトム □ を返すものとする.

- cdr　リストの先頭要素を取り除いた部分を得る関数である. ここでは, 引数としてリストを与え, その先頭要素を取り除いたリストを別の引数に返すようなプログラムを作成してみよう. なお, 空リストが与えられた時にはアトム error を返すものとする.

- cons　リストの先頭要素にデータを付け加える関数である．ここでは，引数としてデータとリストを与え，リストの先頭にデータを付け加えた新たなリストを別の引数に返すようなプログラムを作成してみよう．

次にプログラム例を示す．

```
:- module lisp.
:- public car/2,cdr/2,cons/3.

        car([H|T],V):- true | V=H.                          %(1)
        car([],   V):- true | V=[].                         %(2)

        cdr([H|T],V):- true | V=T.                          %(3)
        cdr([],   V):- true | V=error.                      %(4)

        cons(H,List,V):- true | V=[H|List].                 %(5)

                実行例

        ?- lisp:car([1,2,3,4,5],Result)|Result.             %(6)
           Result = 1                                       %(7)
        ?- lisp:cdr([1,2,3],Result)|Result.                 %(8)
           Result=[2,3]                                     %(9)
        ?- lisp:cons(we,[love,japan],Result)|Result.        %(10)
           Result=[we,love,japan]                           %(11)
```

- まず，car の定義をみてみよう．

  ヘッド引数に [H|T] と書かれている．これは，ゴールが呼ばれた時のヘッドユニフィケーションで，リストを分割する事を意味している．

  例えば，引数がリスト [1，2，3，4，5] でこのゴールが呼ばれた場合には，まず引数同士のユニフィケーションが行なわれ，変数 H は整数の 1，T はリスト [2，3，4，5] に決まる．

  ボディー部では，単に結果を返す変数と結果とのユニフィケーションを行なっているだけである．即ち，変数 V を H とユニファイする．この例では H は 1 に決まっているので，V も 1 に決まる．

- 次に，cdr の定義をみてみよう．

  car を定義したプログラムと性とんど同じである．ボディー部で変数 V を T とユニファイする点が異なる．

- 最後に，cons の定義をみてみよう．

  ヘッド部では，受け取った引数をボディー部に渡しているだけである．ボディー部では，リストの先頭に要素を付けた新たなリストと，結果を返す変数とのユニフィケーションを行なっている．実行例 (10) のように，引数としてアトム we とリスト [love，japan] を与えてゴールを呼び出すと，リストの先頭に要素を付け加えたリスト [we，love，japan] が Result の値に決まる．

2.4　ベクタ

ベクタとは，任意の長さを持った一次元の配列のことで，長さが 0 のものも含む．リストと同じように，各種のデータを格納しておくのに用いられる．各要素には KL1 の全てのデータ型が許される．即ち整数やアトムのほか，変数を格納することもできる．データを格納するという意味では，リストと良く似た使い方がされるが，次のような違いがあるので，目的によってうまく使い分ける必要がある．

1. アクセスの方法

   リスト シーケンシャルなアクセスである．即ち要素を取り出す時は，先頭要素から 1 つずつ取り出す．また，要素を加える時も先頭に要素を 1 つずつ加える．D リストを用いれば，中間に要素を挿入することが簡単にできる．

   ベクタ ランダムアクセスである．即ち，ポジションを指定してその内容の読み書きができる．中間に要素を挿入するのは大変である．

2. 大きさ

   リスト リスト操作を行なうことによって，動的に大きさが変わる．従って，あらかじめ格納すべき要素の数が決まってない時には，通常はリストを用いる．

   ベクタ 生成された時のベクタの大きさは変わらない．従って，通常は要素の数がわかっている時にベクタを用いる．要素の数がわかってない時には，あらかじめ大きなベクタを作っておく必要がある．

ベクタは，プログラム中では { と } で要素を括って，次のように表記する．

```
{1,2,3}          {1,2,3,4}
{we,love,japan} {we,love,{japan,yokohama}}
{A,B,C}          {1,A,2}
```

また，第 1 番目の要素がアトムの場合は次のような表記もでき，各要素を中括弧で括ったものと同じである．

```
we(love,japan)  we(love(japan,yokohama))
```

また，次のようにベクタ同士のユニフィケーションを行なえる．

$$\{1,2,3\} \Longleftrightarrow \{A,B,C\}$$

このようにベクタ同士のユニフィケーションを行なうと，ベクタのサイズが同じであれば各要素のユニフィケーションが行なわれる．従って，第 1 番目の要素は，整数の 1 と変数 A なので，A は整数 1 に決まる．同様に，B，C はそれぞれ 2, 3 に決まる．

なお，組込み述語を使ってプログラムの実行中に動的にベクタを生成することもできる．また，ベクタから要素を取り出したり，セットしたりするベクタ操作を行なう組込み述語もあるが，これらについては初級編で説明する．

第 3 章

繰り返し制御

本章では，制御構造の一つである繰り返し制御を勉強しよう．KL1 で繰り返し制御を行なうには，再帰呼び出しにより行なう．まずカウンタを用いた繰り返しを説明し，次にリストを用いた繰り返しを説明する．

3.1 カウンタを用いた繰り返し

再帰呼び出しとは述語が自分自身を呼びだすことであり，繰り返し制御の基本である．例えば，ある処理を x 回繰り返す述語を定義してみよう．

```
:- module recursive.
:- public loop/1.
                    loop(C):- C=:=0 | true.              %(1)
                    loop(C):- C=\=0 | NC:=C-1,loop(NC). %(2)
```

ゴール loop が呼ばれるとまず引数 C の値をチェックし，0 でなければ節 2 が選ばれて，カウンタ C の値を次々と減じて行きながら自分自身を再帰的に呼んでいる．そして，カウンタの値が 0 になった時点で節 (1) が呼ばれてこのループは終了する．

まず，ゴール loop を 2 回ループさせた時の動きを追ってみよう．

```
?-recursive:loop(2).
```

1. まず，ゴール loop の引数が 2 であるから，節 (2) のガード条件を満たす．従って節 (2) が選ばれ，ボディーゴールが次々と実行される．
2. ゴール NC := C - 1 が実行される．この時，C は 2 であるから，NC は 2 になる．
3. ゴール loop が引数 NC にて呼ばれる．即ち，ゴール loop(1) が実行される．
4. 引数が 1 でなので，先程と同じように節 (2) が選ばれ，ボディーゴールが次々と実行される．
5. ゴール NC := C - 1 が実行される．注意するのは，ここで現れて来た変数 NC は，先ほどのものとは名前が同じでも別のものである点である．即ち，述語の定義中に現れる変数は，ゴールが呼び出される度に新たに生成されるわけである．C は 1 であるから，NC は 0 になる．
6. ゴール loop が引数 NC にて呼ばれる．即ち，ゴール loop(0) が実行される．
7. 引数が 0 だから，今度は節 (1) が選ばれる．この時，ボディーゴールは直ちに成功する true なので，これでゴール loop の実行は終了である．

今度は，トレーサを使ってゴール loop を 5 回ループさせた時の動きをみてみよう．前章でトレーサを使った時は，引数のないゴールを実行させたので，ゴールが実行される入り口でその述語名のみが表示された．今度は引数が有るので，その時の引数の値も表示される．

```
?-recursive:loop(5).
  call loop(5)
  call loop(4)
  call loop(3)
  call loop(2)
  call loop(1)
  call loop(0)
?-
```

これで再帰呼び出しを用いたループの記述法がわかった．では次に，これを使ってもう少し実用的なプログラムを書いてみよう．

問題 3.1 階乗を計算する関数を定義して見よう．

階乗は，次のように再帰的に定義される．

$$FACT_0 := 1$$
$$FACT_n := FACT_{n-1}$$

これを KL1 でプログラムすると次のようになる．

```
:- module factorial.
:- public fact/2.
      fact(0,V):- true | V=1.                              %(1)
      fact(N,V):- N=\=0| N1:=N-1,fact(N1,V1),V:=N*V1. %(2)
```

以下にプログラムの説明をする．

1. 節 (1) は，0 の階乗を定義したものである．即ち，ゴール fact の引数が 0 の時には，V と 1 をユニファイする．

2. 節 (2) は，1 以上の数の階乗を定義したものである．即ち，ゴール fact の引数が 0 でないときには，ボディー部のゴールが実行される．

3. ゴール N1:=N-1 は，再帰的に呼び出す次のゴール fact の引数の値を計算するものである．

4. ゴール fact(N1,V1) は，ゴールを再帰的に呼び出している部分である．N-1 の階乗を計算して，その値を V1 にユニファイする．

5. ゴール V:=N*V1 は，N-1 の階乗 V1 と N の掛け算を行なって，N の階乗を計算している．

では，このプログラムをトレーサを使って実行した例をみてみよう．

```
?-factorial:fact(3,Result)|Result.      %(1)
  call fact(3,R1)                       %(2)
  call fact(2,R2)                       %(3)
  call fact(1,R3)                       %(4)
  call fact(0,R4)                       %(5)
  Result=6                              %(6)
?-
```

トレーサの出力だけでは，実行の様子が良くわからないので，少し補足情報を付加した例をみてみよう．

```
?-factorial:fact(3,Result)|Result.        %(1)
  call fact(3,R1)                         %(2)
     call fact(2,R2)                      %(3)
        call fact(1,R3)                   %(4)
           call fact(0,R4)                %(5)
                R4=1                      %(6)
             R3:=R4*1 即ち R3:=1          %(7)
          R2:=R3*2 即ち R2:=2             %(8)
        R1:=R2*3 即ち R1:=6               %(9)
  Result=6                                %(10)
?-
```

これに良く似た情報を表示するトレーサもある (マルチ PSI). また, 同じプログラムを次のように書くこともできる. 違いについては各自で勉強すること.

```
:- module factorial.
:- public fact/2.
        fact(N,V):- true     | fact(N,1,V).              %(1)

        fact(0,R,V):- true   | V=R.                      %(2)
        fact(N,R,V):- N=\=0  | NewR:=R*N,NewN:=N-1,      %(3)
                               fact(NewN,NewR,V).
```

また, 参考までに *Lisp* で定義した例を示す.

```
. (DEFUN FACT (n)
          (IF (ZEROP n)
               1
              (*  n (FACT (1- n)))))
```

PL/1 で定義した例も示す. これは, 再帰呼び出しを使わないプログラムである.

```
FACT :  PROC (X,Y) ;
        DCL  (X,Y) INTEGER ;
        IF X=0
           THEN Y=1 ;
           ELSE DO ;
                   Y=1 ;
                   DO I=1 TO X ;
                      Y=Y*I ;
                   END ;
                END ;
     END FACT ;
```

3.2 リストを用いた繰り返し

ここではリストを用いた繰り返し制御を勉強する. 即ち, リストの要素個数の回数ループするような制御のことである. では次の問題を解いてみよう.

問題 3.2 要素が全て整数のリストを与え, その中の最大値を求めるプログラムを作れ.

この問題を解くには, 次のような事項を学ばねばならない.

- リストの要素を次々と取り出す方法. しかし, これはリスト操作の節で既に学んだ.
- リストの要素が無くなるまでループさせる方法. リストの要素が無くなるとは, リストが [] になる事に等しい. 従って, ガード部の条件チェックでリストが [] であるかどうかを判定すれば良い.

では, 次のプログラムを見てみよう.

```
:- module maximum.
:- public max/2.
        max([H|T],Max):- true | max(T,H,Max).                    %(1)

        max([],   CMax,Max):- true       | Max=CMax.             %(2)
        max([H|T],CMax,Max):- H <  CMax  | max(T,CMax,Max).       %(3)
        max([H|T],CMax,Max):- H >= CMax  | max(T,H    ,Max).      %(4)
```

実行例

```
?- maximum:max([2,5,1,10,3,46,3],Result)|Result.
   Result = 46
```

1. 節 (1) は, 2引数の述語 max の定義である. ここでは, まず与えられたリストの先頭要素を取り出し, それを一時的な最大値としてボディーゴールを呼び出している. 3引数の max は, 1引数目にリストが与えられる. この場合は, 一時的な最大値を除いた T を引数とする. また, 2引数目には一時的な最大値が与えられる. この場合は, H を引数とする.
2. 節 (2),(3),(4) は, 3引数の述語 max の定義である.
3. 節 (2) は, リストが空リストの時には一時的な最大値を Max にユニファイする. すなわち, ループの終了条件である.
4. 節 (3),(4) は, ヘッドユニフィケーションにてリストの先頭要素 H を取り出す. ガード部では, それと一時的な最大値 CMax の大小比較を行なっている. CMax の方が H より小さいとき, 節 (3) が選択される. それ以外の時には節 (4) が選択される.
5. 節 (3),(4) のいずれかの節が選択されると, ボディーゴールの実行を行なう.
6. 節 (3) では, 一時的な最大値はそのままで max を再帰的に呼び出している.
7. 節 (4) では, 一時的な最大値を H として max を再帰的に呼び出している.
8. このように, リストの要素を先頭から次々に取り出していき, 最終的にリストが空になった時点で節 (2) が呼ばれて, ゴールの実行は終了する.

ここで, トレーサを使ってこのプログラムの動きを確認してみよう.

```
?- maximum:max([2,5,1,10,3,46,3],Result)|Result.  %(1)
   call max([2,5,1,10,3,46,3],Result)              %(2)
   call max([5,1,10,3,46,3],2,Result)              %(3)
   call max([1,10,3,46,3],5,Result)                %(4)
   call max([10,3,46,3],5,Result)                  %(5)
   call max([3,46,3],10,Result)                    %(6)
   call max([46,3],10,Result)                      %(7)
   call max([3],46,Result)                         %(8)
   call max([],46,Result)                          %(9)
   Result=46                                       %(10)
?-
```

第 4 章

ユニフィケーションと同期制御

本章では，ユニフィケーションと同期制御を勉強する．ユニフィケーションは論理型言語の基本要素であり，同期制御は並列言語の基本要素である．この双方を兼ね備えている点が KL1 の特徴である．KL1 では，同期制御をユニフィケーションの中断メカニズムを利用して実現している．本章ではこれら基本要素の勉強をし，同期を応用した並列プログラムの例を用いて並列言語としての KL1 の理解を深める．

4.1 ユニフィケーションとは

ユニフィケーションとは，従来の言語では変数に値を代入する操作に対応する事は既に学んだ通りである．つまり，未定義変数の値を具体的な値に決める事がユニフィケーションである．しかし，ユニフィケーションは更に強力な機能を備えている．

変数を含む 2 つのデータ同士を同一のパターンにする

ここでデータ同士を同一のパターンにするとは，次のような操作を行なう事である．

- 未定義変数の値を具体的な値に決める．
- 未定義変数と未定義変数の値を同一にする．
- アトムや整数の同一性のチェックをする．
- リスト，ベクタ，ストリング同士各要素のユニフィケーションをする．

論理学の世界でいうユニフィケーションは更に広い意味を持つが，ここでは説明しない．従って，以後ユニフィケーションをすると言った場合，以上の 4 つの操作を行なう事を指すものとする．

なお，KL1 ではクローズ中のガード部でユニフィケーションを行なうかボディ部で行なうかによってその呼び名が異なる．

$$\underbrace{\mathrm{Head} :- \mathrm{GuardGoal_1}, \cdots, \mathrm{GuardGoal_n} \mid}_{\textit{Guard Unification}} \underbrace{\mathrm{Goal_1}, \mathrm{Goal_2}, \cdots, \mathrm{Goal_m}}_{\textit{Body Unification}}$$

- ガード部で行なうユニフィケーションをガードユニフィケーション或いはパッシブユニフィケーションと呼ぶ．
- ボディ部で行なうユニフィケーションをボディユニフィケーション或いはアクティブユニフィケーションと呼ぶ．

いずれも，ユニフィケーションを行なう点では同じであるが，ガードユニフィケーションは一時的に中断する場合のある点が異なる．中断に関する詳しい説明はガードユニフィケーションの節で行なう．

4.2 ボディユニフィケーション

ボディユニフィケーションはボディ部に書かれた組込み述語 $Data_1 = Data_2$ によって $Data_1$ と $Data_2$ のユニフィケーションが行なわれる. *Prolog* を知っている人は, そのユニフィケーションと全く同じものであると理解しておけば良い.

表4.1に, $Data_1 = Data_2$ のようなボディユニフィケーションを行なった時の, 各データ型毎のユニフィケーション一覧表を示す. 横軸は $Data_1$ のデータ型を示し, 縦軸は $Data_2$ のデータ型を示す.

表 4.1: ボディユニフィケーション一覧表

□	未定義変数	整数	アトム	ベクタ	リスト	ストリング
未定義変数	○	○	○	○	○	○
整数	○	⊖	×	×	×	×
アトム	○	×	⊖	×	×	×
ベクタ	○	×	×	⊗	×	×
リスト	○	×	×	×	⊗	×
ストリング	○	×	×	×	×	⊗

表中, ○ はユニフィケーションが必ず成功することを表わす. 即ち, 一方が未定義変数で他方が具体的な値の時, 未定義変数の値は具体的な値に決まる. また, 双方が未定義変数の時には値が同一になる. × は必ずユニフィケーションが失敗することを表わしている. ⊖ は同一性をチェックすることを表わす. ⊗ は各要素毎のユニフィケーションを行なうことを表わす. なお, 要素毎のユニフィケーションは並列に行なわれる.

では, 幾つかボディーユニフィケーションの典型的な例をみてみよう.

```
?- not:not(1,Result).          %(1)

not(0,Out):- true | Out=1.     %(2)
not(1,Out):- true | Out=0.     %(3)
```

(1) でゴール not を呼んだ時, 未定義変数 Result に着目してみよう. このゴールを呼び出すと, ヘッドユニフィケーションによりガード条件のチェックが行なわれ, クローズ (2) が選択される. また, 未定義変数 Result と Out のユニフィケーションが行なわれる. ヘッドユニフィケーションはガードユニフィケーションであり, 詳しい説明は次節で行なうが, 引数の受け渡しのために未定義変数同士のユニフィケーションが行なわれると考えておけば良い. 即ち, ユニフィケーションの結果それぞれの未定義変数は同一の値に決まる.

次に, ボディ部で整数 1 と未定義変数 Out とのユニフィケーションが行なわれる. その結果, Out の値は 1 という整数に具体化される. また, 呼びだし側の未定義変数 Result は Out と同一の値を持っているので, Out が具体化されると Result も具体化される.

別の例をみてみよう.

```
?- foo:through(1,Output).      %(1)

through(In,Out):- true | In=Out. %(2)
```

クローズ (2) は, 引数 In にどんなデータがきても引数 Out にそれを伝えるような述語の定義である. (1) のようにゴール through を呼び出すと, ガードでは何も条件をチェックすることがないので, 直ちに節 (2) が選ばれてボディーユニフィケーションが行なわれる. この場合は, 未定義変数 Out と整数 1 のユニフィケーションが行なわれ, Out は整数 1 に具体化される.

もう一つ例をみてみよう.

```
?- foo:through(Input,Output).     %(1)

through(In,Out):- true | In=Out. %(2)
```

先の例と違うのは, (1) でゴールを呼び出す際に引数がいずれも未定義変数のまま呼んでいる点である. この場合, 節 (2) のガード条件は直ちに満たされるので, ボディーユニフィケーションで未定義変数同士のユニフィケーションが行なわれる. 即ち, 未定義変数 In と Out の値を同一にする. その結果, 呼びだし側の未定義変数 Input の値と Output の値も同一になる.

このように, ボディーユニフィケーションでは呼びだし側の未定義変数の値を能動的に具体化したり, 未定義変数同士の値を能動的に同一にするため, アクティブユニフィケーションと呼ばれる.

4.3 ガードユニフィケーションと中断機構

ガードユニフィケーションは, ヘッドユニフィケーション及びガードゴール中のユニフィケーションのことである. 既に勉強したように, ガード部では呼び出したゴールの引数がクローズのガード条件を満たすか否かのチェックを行なう. ガード条件のチェックは, 複数あるクローズ毎に論理的には同時に行なわれる.

従って, ガードユニフィケーションによって呼び出し側の未定義変数の値を具体化したり, 未定義変数同士の値を同じにしたりすると, 正しくガード条件のチェックはできない. そこで, このようなガードユニフィケーションは中断 (サスペンド) する. また, 呼び出し側ゴールの引数が未定義であるためにガード条件をチェックする事ができない場合にも, ガードユニフィケーションは中断する.

次に, ガードユニフィケーションが中断する条件をまとめる.

ガードユニフィケーションが中断する条件
(1) 呼び出し側の未定義変数の値を具体化しようとしたとき
(2) 呼び出し側の未定義変数同士の値を同一にしようとしたとき
(3) 呼び出し側の未定義変数の値が決まらないと ガード条件のチェックができない時

ガードユニフィケーションが中断すると, クローズの選択処理が中断してゴールの実行が中断する. 呼び出し側の未定義変数の値が決まると, ゴールの実行が再開してクローズの選択処理も再開し, ガードユニフィケーションが再開する. 更にガードユニフィケーションが中断すると, 中断しなくなるまで以上の処理を繰り返す.

このように, ガードユニフィケーションは呼び出し側の未定義変数に対して受動的なユニフィケーションを行なうため, パッシブユニフィケーションと呼ばれる.

表 4.2: ガードユニフィケーション一覧表

	呼び出し側						
呼び出された側	—	未定義変数	整数	アトム	ベクタ	リスト	ストリング
	未定義変数	○/*Suspend*	○	○	○	○	○
	整数	*Suspend*	⊖	×	×	×	×
	アトム	*Suspend*	×	⊖	×	×	×
	ベクタ	*Suspend*	×	×	⊗	×	×
	リスト	*Suspend*	×	×	×	⊗	×
	ストリング	*Suspend*	×	×	×	×	⊗

ここで, ガードユニフィケーションの一覧表を表 4.2に示す. 横軸はゴールを呼び出した側の引数のデータ型を表わし, 縦軸は呼ばれた側の引数のデータ型を示す. 表中, ○ は直ちにユニフィケーションが

行なわれて必ず成功することを表わす. × は直ちにユニフィケーションが行なわれて必ず失敗することを表わす. *Suspend* は, 呼び出した側の未定義変数の値が決まるまで中断することを表わす. ⊖ は直ちにユニフィケーションが行なわれ, 同一性をチェックすることを表わす. ⊗ は直ちにユニフィケーションが行なわれ, 要素毎のガードユニフィケーションを行なうことを示す. なお, 要素毎のユニフィケーションは逐次に行なわれる.

では, ガードユニフィケーションの理解を深めるため, 幾つかのタイプ毎に分類してそれぞれの典型的な例を用いて説明しよう. ガードユニフィケーションは, 呼び出し側の引数が未定義変数であるか否かによって分類できる. また, 呼び出された側の引数が未定義であるか否かによっても分類できる. 更に, ガード条件のチェックを ヘッドユニフィケーションで行なうかガードゴールで行なうか, 或いはチェックを行なわないか否かによっても分類できる. 表 4.3にガードユニフィケーションの分類表を示す. なお, 表 4.2と 4.3からそれぞれの例が予想できる人は, 以下は読み飛ばしてもかまわない.

表 4.3: ガードユニフィケーションのタイプ

<table>
<tr><td colspan="3"></td><td colspan="2">呼び出し側の引数</td></tr>
<tr><td colspan="3"></td><td>未定義</td><td>定義</td></tr>
<tr><td rowspan="4">呼ばれた側の引数</td><td rowspan="3">未定義</td><td>ガード
チェック
なし</td><td colspan="2">タイプ (1)</td></tr>
<tr><td>ガード
ゴール</td><td>タイプ (2)</td><td>タイプ (3)</td></tr>
<tr><td>ヘッド
ユニフィ
ケーション</td><td colspan="2">タイプ (4)</td></tr>
<tr><td>定義</td><td>ヘッド
ユニフィ
ケーション</td><td>タイプ (5)</td><td>タイプ (6)</td></tr>
</table>

(1) のタイプのガードユニフィケーション

このタイプのガードユニフィケーションは, 引数を受け渡すために用いられる. では, ボディユニフィケーションの説明で用いた例をもう一度みてみよう.

```
?- foo:through(Input,Output).        %(1)

through(In,Out):- true | In = Out.  %(2)
```

ここで, ゴールを (1) のように呼び出すとガード部ではチェックする条件が全くないので, 引数の受け渡しのみを行なって直ちにボディユニフィケーションが実行される. その結果呼び出した側の未定義変数 Input と Output は同一の値になる.

(2) のタイプのガードユニフィケーション

このタイプの呼び出しでは, 呼び出し側の引数が未定義変数であるが故にガード条件のチェックを行なう事ができない. 従って, 値が決まってチェックが可能になるまでガードユニフィケーションは中断し, ゴールの実行も中断する.

```
?- not:not(Input,Result).          %(1)

not(In,Out):- In =:= 0 | Out=1. %(2)
not(In,Out):- In =:= 1 | Out=0. %(3)
```

(1) でゴール not が呼ばれると，まずクローズ (2) か (3) のどちらかを選ばねばならない．しかし，この時点ではガード部のチェックは不可能である．従って，未定義変数 Input の値が決まるまでガードユニフィケーションは中断する．しかし，このまま放っておくといつまでも値は決まらず，デッドロックの状態になる．次のような例をみてみよう．

```
?- not:not(Input,Result),Input=0. %(1)

not(In,Out):- In =:= 0 | Out=1.    %(2)
not(In,Out):- In =:= 1 | Out=0.    %(3)
```

ここで，ゴール not がゴール = より先に実行されたとしよう．すると，ガードユニフィケーションが中断してゴール not の実行は中断する．その後に Input の値が整数 0 に決まってゴールの実行が再開し，ガードユニフィケーションが再開してこのゴールはデッドロックすることなく実行される．

では別の例をみてみよう．

```
?- foo:through(abc,abc).           %(1)

through(In,Out):- In = Out | true.  %(2)
```

クローズ (2) のガードゴールでは，変数 In と Out の同一性をチェックしている．この場合，どちらも同一のアトムなのでガードユニフィケーションは直ちに実行された後ボディ部が実行される．しかし，次の例をみてみよう．

```
?- foo:through(abc,Output).        %(1)

through(In,Out):- In = Out | true.  %(2)
```

この場合は，変数 Out の値がまだ決まってないので同一性のチェックができないのでガードユニフィケーションは中断する．言い換えると，呼び出し側の未定義変数 Output の値を具体化しようとしているので中断する．次の例も同様に中断する．

```
?- foo:through(Input,Output).      %(1)

through(In,Out):- In = Out | true.  %(2)
```

この場合呼び出し側の未定義変数 Input と Output の値をガードユニフィケーションで同一にしようとしているので，中断する．

(3) のタイプのガードユニフィケーション

このタイプの呼び出しは (2) のタイプとは違って直ちにガード条件のチェックが可能である．ガード条件を満たすクローズが選択されてボディ部が実行される．

```
?- not:not(1,Result).              %(1)

not(In,Out):- In =:= 0 | Out=1. %(2)
not(In,Out):- In =:= 1 | Out=0. %(3)
```

(4) のタイプのガードユニフィケーション

このタイプの呼び出しは引数同士の同一性をチェックするために用いる.

```
?- foo:through(Input,Output).       %(1)

through(Same,Same):- true | true.  %(2)
```

これは *Prolog* で引数同士のユニフィケーションを行なう目的で良く使うプログラミングテクニックの一つであるが, KL1 では中断が起きるため注意して用いるべきものである. このプログラムの意味は, 一番目の引数と二番目の引数が同一の値であるかどうかをチェックせよ, である. すなわち, (2) のタイプの例で用いたプログラムと全く同じである.

```
?- foo:through(Input,Output).       %(1)

through(In,Out):- In = Out | true.  %(2)
```

従って, 変数 In と Out の双方の値が決まるまでガードユニフィケーションは中断する. もう一つ例をみてみよう.

```
?- foo:through(Bow,Bow).            %(1)

through(Same,Same):- true | true.  %(2)
```

(1) でゴールを呼び出した時, 2 つの引数は同じ未定義変数 Bow であるが, 値が具体化されるまでガードユニフィケーションは中断する. このように, KL1 では未定義変数同士の同一性のチェックは行なわないので, *Prolog* のプログラミングに慣れている人は特に注意する事.

(5) のタイプのガードユニフィケーション

このタイプの呼び出しでは, ヘッドユニフィケーションでそれぞれの引数同士のユニフィケーションを行なおうとする. しかし, 呼び出し側の未定義変数の値を決めようとする操作なので中断する.

```
?- not:not(Input,Result),Input=0 %(1)

not(0,Out):- true | Out=1.        %(2)
not(1,Out):- true | Out=0.        %(3)
```

(1) でゴール not を呼び出すと, まずヘッドユニフィケーションが行なわれ, 引数 Input と (2) の 0, 或いは (3) の 1 とのユニフィケーションを行なおうとする. しかし, ゴールが呼び出された時には引数 Input が未定義変数であっても, その直後には値が 0 に決まるかも知れないし, また 1 に決まるかも知れない. 従って, 呼び出し側の未定義変数 Input の値が決まるまでガードユニフィケーションは中断する.

(6) のタイプのガードユニフィケーション

このタイプの呼び出しは (3) のタイプと同じで, ガード条件のチェックをヘッドユニフィケーションで行なうかガードゴールで行なうかの違いだけである.

```
?- not:not(1,Result).             %(1)

not(0,Out):- true | Out=1.        %(2)
not(1,Out):- true | Out=0.        %(3)
```

4.4 同期制御の基礎知識

並列プログラミング言語においては，同期制御の機能は不可欠である．KL1 では同期制御をユニフィケーションの中断機構を利用して実現していることは先にも述べた．ここでは，まず同期とは何であるかを説明し，KL1 での同期制御の実現方法を説明する．

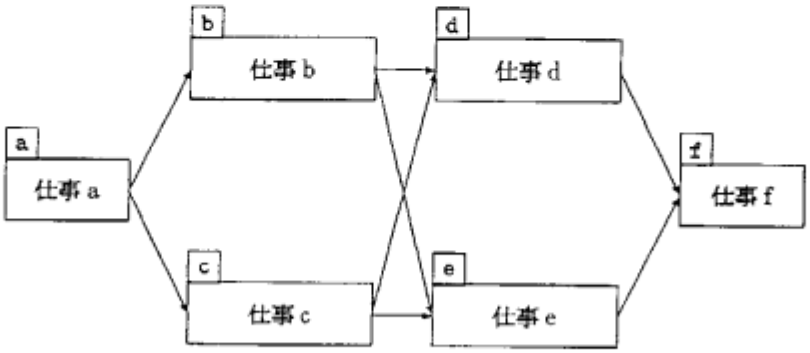


図 4.1: 仕事の工程

まず同期とは何かを簡単に説明する．例えばここに何か大きな仕事があり，それは図 4.1 に示されるような工程で作業するものだとしよう．図の中で，四角で囲まれた部分が個々の仕事であり，それらは矢印の方向に向かって逐次進んでいくものとする．即ち，仕事 a が終わってから b を行なうことができる．

また，縦に 2 つ並んだ仕事はそれぞれ互いに独立した仕事であり，並行して行なうことができるものとする．例えば，仕事 a が終わると b と c は並行して行なうことができる．

また，2 つの矢印でさされている仕事は，両方が終わらないとその仕事は開始できないことを表わしている．即ち，仕事 b と c の両方が終われば d 及び e を開始できることを表わしている．

このように，ある仕事が終わると次の仕事ができるように制御することを同期制御と呼ぶ．また言い換えると，ある仕事が終わるまで次の仕事は中断して待たせるような制御を同期制御と呼ぶ．

では，同期制御を KL1 ではどのように記述するか，例で示した仕事をシミュレートするような次のプログラムで見て見よう．

```
:- module sim.
:- public simulate/1.
        simulate(Go,End):- true |          %(1)
                a(Go,A),                   %(2)
                b(A,B),                    %(3)
                c(A,C),                    %(4)
                d(B,C,D),                  %(5)
                e(B,C,E),                  %(6)
                f(D,E, End).               %(7)
        a(ok,End):- true | End = ok.       %(8)
        b(ok,End):- true | End = ok.       %(9)
        c(ok,End):- true | End = ok.       %(10)
        d(ok,ok,End):- true | End = ok.    %(11)
        e(ok,ok,End):- true | End = ok.    %(12)
        f(ok,ok,End):- true | End = ok.    %(13)
```

実行例

```
?- sim:simulate(ok,End)|End.      %(14)
   End=ok
```

クローズ(1)は，このシミュレータのトップレベルの呼び出し述語 simulate を定義したクローズである．ボディーゴールでは，仕事の各工程を呼び出している．ゴールが実行される順序は，プログラムに記述した順ではなく，中断せずに実行可能なものから順に行なわれる．

ここで，(14)のようにゴールを呼び出した場合をみてみよう．中断せずに直ちに実行できるゴールは(2)だけである．従ってクローズ(8)が選択されて，ボディユニフィケーションで未定義変数 A の値がアトム ok に決まる．

ここで，ゴール(3)と(4)とその定義(9)，(10)をみてみよう．いずれも未定義変数 A の値がアトム ok に決まるのを待って中断している．従って，ゴール(2)の実行が終わって A の値が ok に決まると，ゴール(3)，(4)は共に実行を再開する．ただし，どちらが先に実行されるかは，わからない．

次に，(5)のゴール d とその定義(11)をみてみよう．未定義変数 B と C の両方の値が決まるまでこのゴールの実行は中断する．

ここでは，これ以上このプログラムの動きは追わないことにするが，これが KL1 で同期制御を行なった最も基本的なプログラム例である．

4.5 同期制御を応用したプログラム例

ここでは，同期制御を応用したさまざまなプログラム例を示す．ただし，本格的に同期制御を応用した KL1 の典型的なプログラムは，次章で説明するプロセスである．ここではプロセスの説明は行なわないが，プロセスを理解するための基本的なプログラミング手法を用いたプログラムの説明を行なう．

リストと未定義変数を用いた同期制御の基礎

KL1 で同期制御を行なう際，ヘッドユニフィケーション時の未定義変数とリストの中断メカニズムを利用する場合が多い．まず入門編で勉強した *Lisp* の基本関数プログラムを思い出してみよう．

```
:- module lisp.
:- public car/2,cdr/2,cons/3.
        car([H|T],V):- true | V=H.                          %(1)
        car([],   V):- true | V=[].                         %(2)

        cons(H,List,V):- true | V=[H|List].                 %(5)

        ?- lisp:car(List,Result)|Result.                    %(6)
              デッドロック発生．
        ?- lisp:cons(Var,[1,2,3],Result)|Result.            %(7)
           Result=[Var,1,2,3]                               %(8)
        ?- lisp:cons(Var1,Var2,Result)|Result.              %(9)
           Result=[Var1|Var2].                              %(10)
        ?- lisp:car(List,Result),List=[1,2,3]|Result.       %(11)
           Result=1                                         %(12)
```

入門編でこのプログラムを勉強した際には，まだ中断メカニズムは勉強していなかった．しかし，ゴールを呼ぶ際に引数を未定義変数のまま呼び出すと，これは中断するであろうということが今ではわかるであろう．

例えば，(6)のようなゴール呼び出しをすると，ヘッドユニフィケーションが中断してそのゴールの実行は中断し，この場合はデッドロックする．

(7) のようにゴールを呼び出すと，ガード条件のチェックがないのでゴールは直ちに実行される．その結果，未定義変数 Result の値は (8) のような第 1 要素が変数であるようなリストに決まる．

(9) のようにゴールを呼び出すと，(10) のような先頭要素が未定義変数 Var1 で残りが Var2 であるようなリストに決まる．

では，(11) のようなゴール呼び出しをした場合はどのようになるであろうか．ここで，ゴール car が先に実行された時のことを考えてみよう．まず，引数 List が未定義変数なので，ヘッドユニフィケーションは中断する．次に未定義変数 List の値がリストに決まるとヘッドユニフィケーションは再開する．

リストと未定義変数を用いた同期制御の応用

リストと未定義変数を用いた同期制御を応用すると，次のようなプログラムを書く事ができる．

0 から 100 までの自然数を次々に生成し，
偶数と奇数に振り分けるプログラム．

```
:- module furui.
:- public go/2.
      go(Even,Odd):- true |                              %(1)
              generate(0,Numbers),                       %(2)
              furui(Numbers,Even,Odd).                   %(3)
      generate(N,Numbers):- N > 100 | Numbers=[].        %(4)
      generate(N,Numbers):- N =<100 |                    %(5)
              Numbers = [N|NewNumbers],                  %(6)
              N1:= N+1,generate(N1,NewNumbers).          %(7)
      furui([H|T],Even,Odd):- (H mod 2) =:= 0  |         %(8)
              Even = [H|NewEven],                        %(9)
              furui(T,NewEven,Odd).                      %(10)
      furui([H|T],Even,Odd):- (H mod 2) =:= 1  |         %(11)
              Odd  = [H|NewOdd],                         %(12)
              furui(T,Evem,NewOdd).                      %(13)
     furui([],Even,Odd):- true | Even=[],Odd=[].         %(14)
```

クローズ (1) は述語 go を定義したものであり，ガード部では何もチェックしていない．このゴールを呼び出すと，引数の Even と Odd にはそれぞれ 0 から 100 までの偶数のリストと奇数のリストが帰ってくる．

ゴール go を呼び出すと，ボディー部ではゴール generate と furui を呼び出す．なお，ゴール generate は再帰呼び出しを使って 0 から 100 までの自然数の生成を行なっている．ガード部では N が 100 以下かどうかの判定を行なっている．このゴールは，生成した自然数を引数 Numbers にリスト形式で返し続ける．

また，(8) から (13) までは述語 furui を定義したクローズである．まず，(8) のクローズをみてみよう．このクローズは，ヘッドユニフィケーションで 1 番目の引数にの値がリストに決まるまで中断する．なお中断が解かれるのは，(2) で呼び出したゴール generate が引数 numbers の値をリストに決めた時である．即ち，自然数が一つ生成される度にゴール furui は 1 サイクル実行される．

このスタイルのプログラムは，KL1 で同期制御を行なう典型的な例である．
また，generate や furui のように再帰呼び出しを使って実行するゴールをプロセスと呼ぶ．また，プロセスの間で共有の変数 Request を使ってデータをやり取り (通信) する事をストリームにより通信を行なうと呼び，変数 Resuest をストリームと呼ぶ．なお，プロセスとストリームを使ったプログラミング手法については，次章にて詳しく説明する．

第 5 章

プロセス

KL1 のプログラムはプロセスと呼ばれる基本要素を組み合わせて構成されるのが普通である．各プロセスはメッセージを用いて通信しながら計算を進めていき，半ば独立に並列に動き得る．メッセージは次から次へと送られるのでメッセージ列となる．このメッセージ列をストリームと呼び，KL1 ではリストで表現される．リストの car 部分が最初のメッセージ，cdr 部分が残りのメッセージ列を表わす．KL1 ではこのストリームで各プロセスを連結し全体として一つの計算を進めていくという考えでプログラミングすることが多い．このようなプログラミング技法をストリームプログラミングと呼ぶこともある．

プロセスは一般的には，m 個の入力ストリームと n 個の出力ストリームと内部状態を持つものとして表現することができる (図 5.1)．

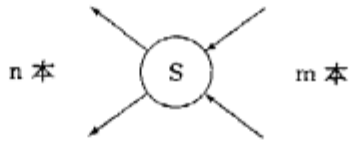


図 5.1: プロセス

入力ストリームからメッセージを受け取り内部状態を変化させ，出力ストリームからメッセージを送出するというのがプロセスの標準的な 1 サイクルである．KL1 においてプロセスは再帰呼びだしと局所的 (他のプロセスから参照されない) 変数によって実現される．再帰呼出しがプロセスの 1 サイクルに，局所変数が内部状態に対応している．

本章では，典型的なプロセスの例題を幾つか取り上げる．

5.1 not プロセス

not プロセスは図 5.2に示すような 1 入力 1 出力のプロセスで表わされる．

図 5.2: not プロセス

入力ストリーム In からメッセージ 0 を受け取ったらメッセージ 1 を，メッセージ 1 を受け取ったら 0 を出力ストリームに出すプロセスである (図 5.3)．

```
:- module not.
:- public not/2.
```

図 5.3: not プロセスの機能

```
not([], Out) :- true | Out = [].                          %(1)
not([0|In], Out) :- true | Out = [1|Out1], not(In, Out1).  %(2)
not([1|In], Out) :- true | Out = [0|Out1], not(In, Out1).  %(3)
```

not/2 の第 1 引数は入力ストリームで第 2 引数は出力ストリームである．このように入力ストリーム，出力ストリームを持ち入力メッセージを加工してメッセージを出力ストリームへ送り出すプロセスを一般にフィルタプロセスという．

節 (1) は入力ストリームが閉じられた時の処理が記述されている．これはプロセスの終了時の処理に相当し，本プログラムの場合の処理では出力ストリームを閉じている．入力ストリームからメッセージ 0 が届いた場合は，出力ストリームにメッセージ 1 を送り，次の処理のために再帰呼出しをする (節 (2))．また入力ストリームからメッセージ 1 が届いた場合は，出力ストリームにメッセージ 0 を送り，そして次の処理のために再帰呼出しをする (節 (3))．このメッセージを受け取ってから再帰呼出しをするまでの過程が，プロセスの 1 サイクルである．

入力ストリームが閉じられておらずまた何のメッセージも届いていない時の処理は書かれていないないが，KL1 の機構によって入力ストリームが閉じられるかメッセージが届くまで，このプロセスは自動的に中断 (suspend) させられる．この機構を中断機構という．この機構については次節において，詳しく述べる．

5.2 中断機構

並列計算においてはプロセス間の同期のとりかたが重要になる．KL1 においてプロセス間の同期は中断機構を利用して行なわれる．この中断機構はガード部の実行において『その呼びだしゴール側の変数に値を代入することはしない』という制限を設けることにより実現される．この制限の下でガード部の実行が成功する時その節を成功節，失敗する時失敗節という．成功節が複数ある場合その中からただ 1 つの節が選択される．全て節が失敗節となった時その呼びだしゴールの失敗となる[1]．

節の種類

成功節 ガードの実行に成功する節

失敗節 ガードの実行に失敗する節

中断している節 上記以外の節

ガード部の実行において呼びだしゴール側の変数の値を決めなければ成功しないとき，ガード部の実行は中断させられる．成功節がなく中断している節がある場合，そのゴール自身必要な変数の値が決められるまで実行を中断させられる．このように変数の値に『未定義』という概念が入っているので『変数の値が決まるまで待つ』という形で，同期を実現できる．

本節ではこの中断機構を具体例を幾つか例示しながら説明する．ヘッド部に書いてあるものが変数，アトミックなデータ，構造体かにより場合を分けて例題を用いて中断機構の解説を行なっていく．

[1] 通常これはプログラムのバグであることが多い．

5.2.1 ヘッド部に現れる変数

ヘッド部の変数は値を受け取ることに使われる.

例 5.1 (成功節) X にはアトム a が代入され, Y と B は同一視[2]される.

```
呼びだしゴール  p(a, B)
節定義  p(X, Y) :- true | ....
```

例 5.2 (失敗節) X にはアトム a が代入されるが, integer(X) の実行に失敗する.

```
呼びだしゴール  p(a)
節定義  p(X) :- integer(X) | ....
```

例 5.3 (中断する節) A と B の値が決まるまで中断する. A と B の値が決まったところでその値の比較が行なわれ, 同じだったら成功節となり違う場合は失敗節となる.

```
呼びだしゴール  p(A, B)
節定義  p(X, X) :- true | ....
```

5.2.2 ヘッド部に現れるアトミック

例 5.4 (成功節) ゴールの第 1 引数 a とヘッド部の第 1 引数 a の比較が行なわれ同一なので成功する. また Y にはベクタ {b} が代入される.

```
呼びだしゴール  p(a, {b})
節定義  p(a, Y) :- true | ....
```

例 5.5 (失敗節) ゴールの第 1 引数 {a} とヘッド部の第 1 引数 a の比較が行なわれ同一ではないので失敗する.

```
呼びだしゴール  p({a}, {b})
節定義  p(a, Y) :- true | ....
```

例 5.6 (中断する節) A の値が決まるまで中断する. A の値が決まったところでその値と a の比較が行なわれ, 同じだったら成功節となり違う場合は失敗節となる.

```
呼びだしゴール  p(A, B)
節定義  p(a, X) :- true | ....
```

5.2.3 ヘッド部に現れる構造体

例 5.7 (成功節) ヘッド部の意味は, 「第 1 引数が a 第 2 引数が 1 要素ベクタならばこの節を選択せよ」というものである. 呼びだしゴールはこれに合致するので成功し, Y には b が代入される.

```
呼びだしゴール  p(a, {b})
節定義  p(a, {Y}) :- true | ....
```

例 5.8 (失敗節) ゴールの第 2 引数 {b} とヘッド部の第 2 引数 {c} の比較が行なわれ同一ではないので失敗する.

```
呼びだしゴール  p(a, {b})
節定義  p(a, {c}) :- true | ....
```

例 5.9 (中断する節) B の値が決まるまで中断する. B の値が決まりそれが 1 要素ベクタの場合成功節となり Y の値をその要素とする. それ以外の場合は失敗節となる.

```
呼びだしゴール  p(a, B)
節定義  p(a, {Y}) :- true | ....
```

[2] Y と B には同じものしか代入できなくなる.

5.3 stack プロセス

stack プロセスは 1 入力のプロセスであり内部状態としてスタックを保持している (図 5.4). stack プロセスに対しメッセージ push(Data) によりデータを格納でき, メッセージ pop(Data) によりデータを取り出すことができる.

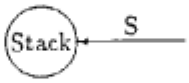


図 5.4: stack プロセス

```
:- module stack
:- public stack/1.
stack(S) :- true | stack(S, []).                                    %(1)

stack([], _) :- true | true.                                        %(2)
stack([push(Data)|S], Stack) :- true | stack(S, [Data|Stack]).      %(3)
stack([pop(Data)|S], [Top|Stack]) :- true |
        Data = Top, stack(S, Stack).                                %(4)
```

stack プロセスは 2 つの引数を持つ. 第 1 引数は入力ストリームであり, 第 2 引数はスタックの中身を保持するためのもので要素のリストで表わされている. これが stack プロセスの内部状態に相当する. stack プロセスは呼出し stack(S) によって起動され, まずスタックを空に初期化する (節 (1)).

入力ストリームが閉じられた場合, ただ単に終了する (節 (2)). メッセージが push(Data) の時は, Data をスタックの上に積む (節 (3)). メッセージが pop(Data) の時はスタックの一番上の要素 Top を Data とユニファイする (節 (4)).

ここでメッセージ push(Data), pop(Data) が, 未定義変数 Data を含んでいて, それが結果の返信用に使われていることに注目して欲しい. push(), pop() という骨格は stack プロセスが受け取る部分であるが, その引数は stack プロセスがメッセージの送り手に値を返す部分なのである. これら push(Data), pop(Data) のような変数を含んだメッセージのことを, 未完成メッセージ (incomplete message) という.

トレース 5.1 (stack) スタックプロセスにメッセージ列 push(1), push(2), pop(X), pop(Y) を送った時の様子を追いかけてみる.

```
?- stack:stack([push(1), push(2), pop(X), pop(Y)]) | X, Y.
call stack([push(1), push(2), pop(X), pop(Y)])
    call stack([push(1), push(2), pop(X), pop(Y)], [])
        call stack([push(1), push(2), pop(X), pop(Y)], [1])
            call stack([pop(X), pop(Y)], [2, 1])
                call X = 2
                call stack([pop(Y)], [1])
                    call Y = 1
                    call stack([], [])
X = 2
Y = 1

yes
```

5.4 queue プロセス

次にデータの取りだし順が FIFO の queue プロセスを定義してみる．このプロセスもやはり 1 入力のプロセスであり内部状態としてキューを保持している (図 5.5).

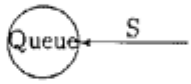


図 5.5: queue プロセス

```
:- module queue.
:- public queue/1.

queue(S) :- true | queue(S, T, T).                          %(1)

queue([], _, _) :- true | true.                             %(2)
queue([enqueue(Data)|S], Head, Tail) :- true |
        Tail = [Data|NewTail], queue(S, Head, NewTail).     %(3)
queue([dequeue(Data)|S], Head, Tail) :- true |
        Head = [Data|NewHead], queue(S, NewHead, Tail).     %(4)
```

このプログラムはキューを差分リストを利用して表現している．差分リストはリストの頭の部分へのポインタと尾の部分へのポインタの対でリストを表現する方法である．対は，これら 2 つのポインタに挟まれた部分のリストを表わしている．

いま，図 5.6のようなリストを考える．このリストは，全体が H であり，その初めの n 個の要素が $X_1, X_2, \ldots, X_n$ であり，その後にリスト T がつながっている．すなわち，H = [$X_1, X_2, \ldots, X_n$|T] である．このリストにおいて，H と T の間に挟まれた部分 $X_1, X_2, \ldots, X_n$ を一つのリスト考え，それを H と T とから作られる差分リストという．つまり差分リストはポインタの対 (H, T) により表現することができる．

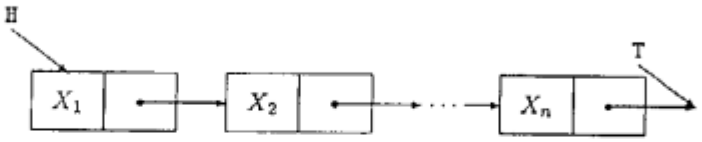


図 5.6: 差分リスト (H, T)

図 5.6 の差分リスト (H, T) の先頭に要素 X_0 を加えた差分リスト (NH, T) は NH = [X_0|H] により得られる (図 5.7).

次に T が未定義の場合を考える (図 5.8)． この場合差分リスト (H, T) の後尾に要素 X_{n+1} を加えた差分リスト (H, NT) を作ることができる． それは，T=[X_0 | NT] により得られる (図 5.9)．このように差分リストでは後尾への要素の追加が，普通のリストに比べて容易にできる利点がある．

さてここで queue プログラムの説明をする． まず最初にキューを空に初期化している (節 (1) 図 5.10)． 入力ストリームよりメッセージ enqueue(Data) を受け取ると (節 (2))，Data を差分リストの後尾に加え (Tail = [Data|NewTail])，NewTail をキューの新しい後尾とする (queue(S, Head, NewTail))． 入力ストリームよりメッセージ dequeue(Data) を受け取ると (節 (3))，Data

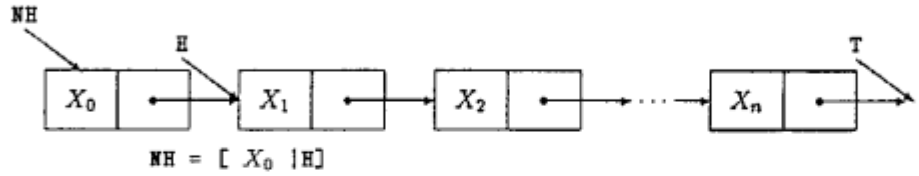


図 5.7: 差分リスト (NH, T)

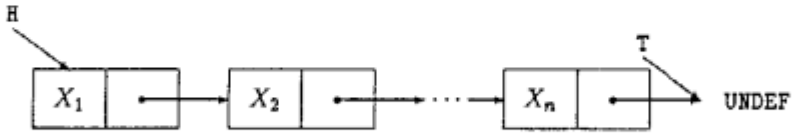


図 5.8: T : 未定義

を差分リストの先頭より取りだし (Head = [Data|NewHead]), NewHead をキューの新しい頭とする (queue(S, NewHead, Tail)).

トレース 5.2 (queue) queue プロセスにメッセージ列 enqueue(1), enqueue(2), dequeue(X), dequeue(Y) を送った時の様子をキューに着目して追いかけてみる. まずキューを空に初期化する (図 5.10). メッセージ enqueue(1) によりキューの後尾に 1 が入る (図 5.11). 次にメッセージ enqueue(2) によりキューに 2 が入る (図 5.12). さて dequeue(X) により X に 1 が代入される (図 5.13). dequeue(Y) も同様に Y に 2 が代入される (図 5.14).

```
?- queue:queue([enqueue(1), enqueue(2), dequeue(X), dequeue(Y)]) | X, Y.
call queue([enqueue(1), enqueue(2), dequeue(X), dequeue(Y)])
    call queue([enqueue(1), enqueue(2), dequeue(X), dequeue(Y)], T, T)
        call T = [1|NT0]
        call queue([enqueue(2), dequeue(X), dequeue(Y)], [1|NT0], NT0)
            call NT0 = [2|NT1]
            call queue([dequeue(X), dequeue(Y)], [1, 2|NT1], NT1)
                call [1, 2|NT1] = [X|H0]
                call queue([dequeue(Y)], [2|NT1], NT1)
                    call [2|NT1] = [Y|H1]
                    call queue([], NT1, NT1)
X = 1
Y = 2
```

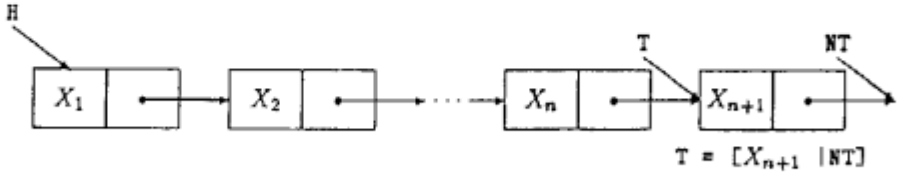


図 5.9: 差分リスト (H, NT)

```
yes
```

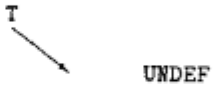


図 5.10: キューの初期化

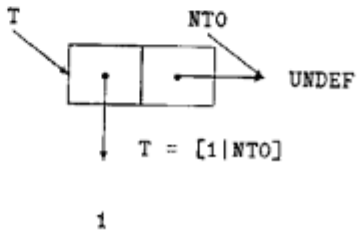


図 5.11: キューの状態 (1)

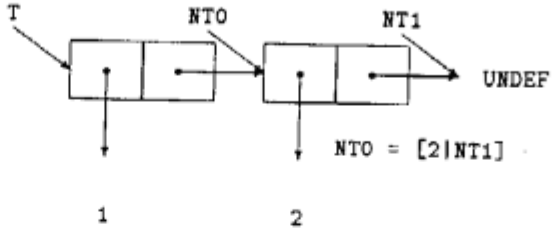


図 5.12: キューの状態 (2)

5.5 merge プロセス

merge プロセスは 2 入力 1 出力のプロセス (図 5.15) であり，2 つの入力ストリームから来たメッセージを来た順に出力ストリームに送出するプロセスであり，次のように定義される．

```
:- module merge.
:- public merge/3.

merge([], Ys, Zs) :- true | Ys = Zs.  %(1)
merge(Xs, [], Zs) :- true | Xs = Zs.                                  %(2)
merge([X|Xs], Ys, Zs) :- true | Zs = [X|Zs1], merge(Xs, Ys, Zs1).     %(3)
merge(Xs, [Y|Ys], Zs) :- true | Zs = [Y|Zs1], merge(Xs, Ys, Zs1).     %(4)
```

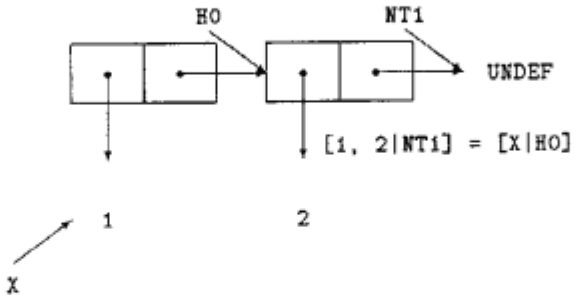


図 5.13: キューの状態 (3)

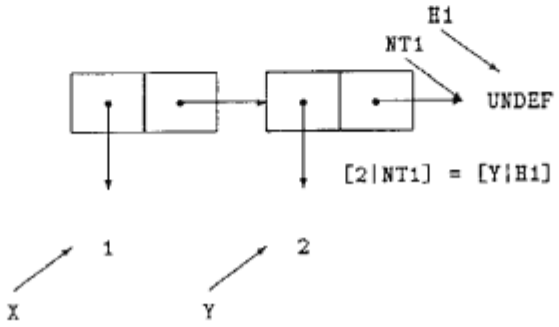


図 5.14: キューの状態 (4)

節 (1) と (2) は入力ストリームの一方が閉じた場合の処理が記述してあり，閉じられていない入力ストリームと出力ストリームを直接つないでいる (図 5.16)．節 (3) と (4) は入力ストリームにメッセージが届いた場合の処理であり，メッセージの届いた順に出力ストリーム (Zs) にメッセージを出力している．

merge プロセスが今までのプロセスと違うのはガード部が排他的でないということである．not，stack，queue プロセスは複数のガード同時に成功することはなかった．それに対して merge プロセスでは，例えば，両入力ストリームにメッセージが届いている場合，節 (3) と (4) のガード部が成功するので，どちらかが選択されるかは非決定的である (図 5.17)．このような非決定性を *don't care* 非決定性と呼び，Prolog のバックトラックを伴うような *don't know* 非決定性と区別している．KL1 では *don't care* な非決定性のみを扱える．

プロセスは幾つかの入出力ストリームを持ち，それを使って他のプロセスと通信をしながら計算を進めていく．幾つかからのメッセージをまとめて処理をしたいときに merge プロセスは利用される．この場合 merge プロセスの木状ネットワークが形成される (図 5.18)．しかし，この merge プログラムを利用してプロセス間通信をしていると，merge プロセスが非常に多くなり本来の処理効率を落としてしまう．そこで KL1 では多入力ストリームをもつ組込みのシステム述語を用意している．この使い方は初級編で解説する．組込み merge を利用すると図のような merge ネットワークは 1 つの merge プロセスで実現される．

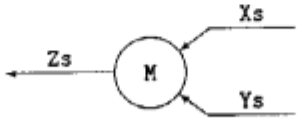


図 5.15: Merge

図 5.16: ストリームの閉鎖

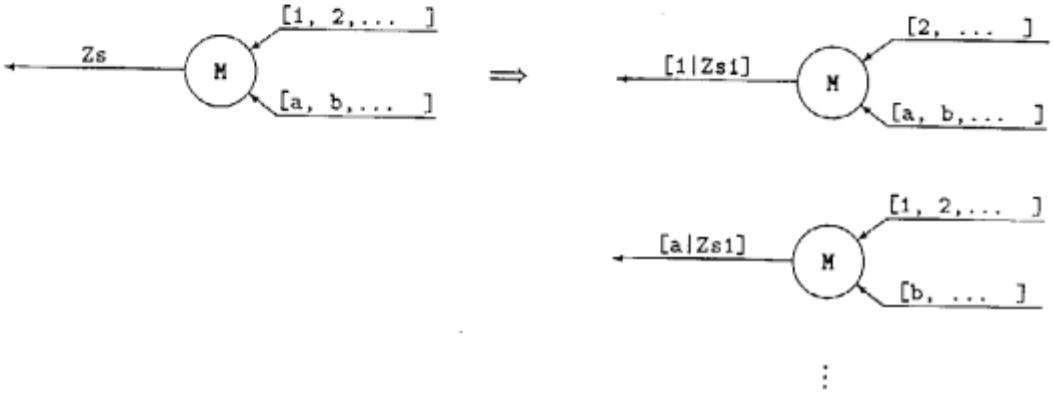


図 5.17: 非決定性

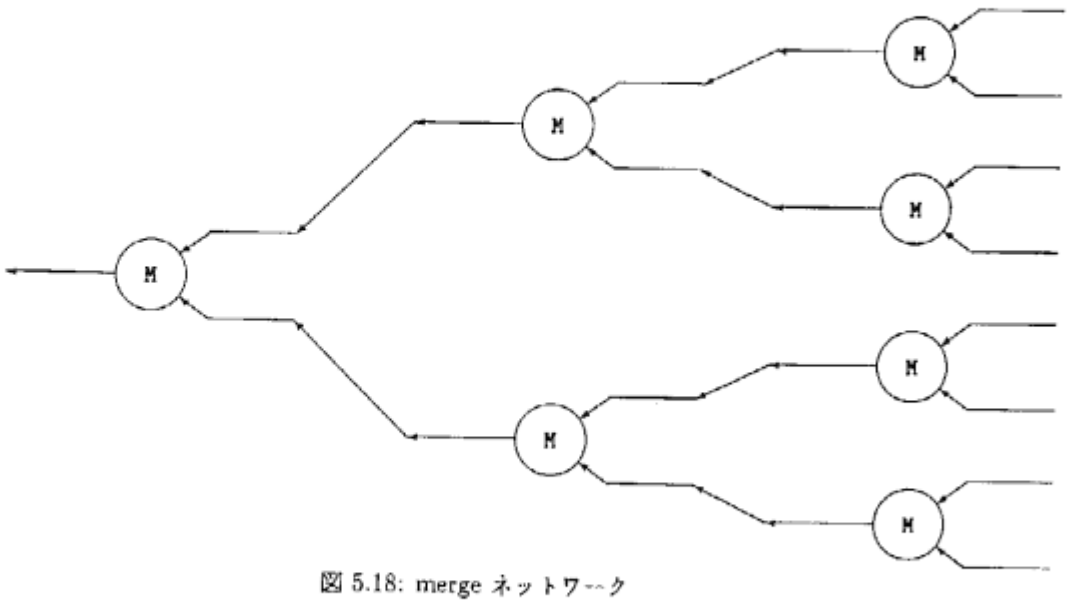


図 5.18: merge ネットワーク

第 6 章

入門編練習問題

この章では、入門編で勉強した事の復習をするための練習問題の説明を行なう。解答は次章に載せるが，なるべく各自でプログラムを作成して PDSS 等で動作を確認する事.

6.1 DO ループ

DO ループを書いてみよう

次に DO ループの例を示す。

(1)

```
DO I=1,N
   CALL FUNCTION(I)
END
```

(2)

```
DO I=1,N,S
   CALL FUNCTION(I)
END
```

この 2 つの DO ループと同じ機能のものを、KL1 で書いてみよう。色々な書き方があると思うが、思い付く限りのものを作成する事。

6.2 総和を求める

全ての要素が全て整数のリストを与え、その総和を求めるプログラムを作成せよ。

リスト [1,2,3,4,5] が与えられた時、全ての要素の総和を求めて、15 を得るようなプログラムを作成せよ。

実行例

```
?- ex:sigma([1,2,3,4,5],Sigma)|Sigma.          %(1)
   Sigma=15                                    %(2)
?- ex:sigma([10,-10,20,-20],Sigma)|Sigma.      %(3)
   Sigma=0                                     %(4)
?-
```

6.3 リスト要素の並べ変え

与えられたリストの要素を逆順に並べ変えるプログラムを作成せよ。

リスト [1,2,3,4,5] が与えられた時、この要素を逆順に並べ変えて、[5,4,3,2,1] を生成するようなプログラムを作成せよ。

実行例

```
?- ex:reverse([1,2,3,4,5],List)|List.          %(1)
   List=[5,4,3,2,1]                            %(2)
?- ex:reverse([a,[b,c],d,e],List)|List.        %(3)
   List=[e,d,[b,c],a]                          %(4)
?-
```

6.4 2つのリストの結合 *(append)*

つのリストを与え、それらを結合して 1 つにするプログラムを作成せよ。

リスト [1,2] と [3,4] を与えたとしよう。
この 2 つのリストを結合して、[1,2,3,4] を生成するようなプログラムを作成せよ。なお、*Prolog* の教科書に載っているプログラムは、そのままでは KL1 では動かない。その意味を良く考えてみよ。

実行例

```
?- ex:append([1,2],[3,4],List)|List.
   List=[1,2,3,4]
?-
```

6.5 フィボナッチ数列を書いてみよう

リスト $[a_1,a_2,\cdots,a_m]$ は 次の漸化式を満たすものとする。 このような数列をフィボナッチ数列と呼ぶ。フィボナッチ数列を生成するプログラムを作成せよ。

$$a_1 = 1$$
$$a_2 = 1$$
$$a_n = a_{n-2}+a_{n-1}$$

なお、無限数列を生成すると止まらないプログラムになるので、引数で生成する個数を与えるようにせよ。

実行例

```
?- fib:fib(10,X)|X.
   X=[1,1,2,3,5,8,13,21,34,55]
```

6.6 簡単な入出力機能の使い方

入門編では入出力機能の勉強は行なわなかった。しかし、実際に練習問題を解いているとやはり入出力を行ないたくなる。詳細な説明は中級編以降で行なうが、PDSS 上でウィンドウを使ったプログラムを載せるので、これを参考にしてこれまでに勉強したプログラムに入出力機能を付けてみよう。このプログラムは、先に作成したフィボナッチ数列を端末に出力するものである。

```
:- module io_sample.
:- public test/0.
       test :- true |  window:create(IOS,"sample"),          %(1)
                       fib(100,S),                           %(2)
                       output(S,IOS).                        %(3)
```

```
output([H|T],IOS):- true | IOS=[putt(H),nl|NIOS],      %(4)
                    output(T,NIOS).                     %(5)
output([],IOS):-    true | IOS=[getc(_)].               %(6)
```

プログラムの説明はここでは行なわないので、理解したい人はマニュアル [2] を参照の事。

6.7 整数生成プログラム

2 つの整数 $M,N(M \leq N)$ が与えられたとき，M より始まり N までの整数列をストリームとして次々に生成するプログラムを書け．

6.8 ストリームの圧縮

与えられた入力ストリームから同一要素を全て除去したストリームを作るプログラムを書け（図 6.1)．

図 6.1: プロセス

6.9 行列の掛け算

行列 M,N が与えられた時，その積 MN を求めるプログラムを書け．

第 7 章

入門編練習問題の解答

7.1 DO ループ

DO ループを書いてみよう

次に DO ループの例と KL1 のプログラムを示す。

(1)

```
DO I=1,N
   CALL FUNCTION(I)
END
```

```
:- module do_loop.
:- public loop_1/2.
loop_1(I,N):- I =< N |          %(1)
        I1:= I+1,               %(2)
        function(I),            %(3)
        loop_1(I1,N).           %(4)
loop_1(I,N):- I > N | true .    %(5)
```

呼びだし方

```
?- do_loop:loop_1(1,100).
```

(2)

```
DO I=1,N,S
   CALL FUNCTION(I)
END
```

```
:- module do_loop.
:- public loop_2/3.
loop_2(I,N,S):- I =< N |        %(1)
        I1:= I+S,               %(2)
        function(I),            %(3)
        loop_1(I1,N).           %(4)
loop_2(I,N,_):- I > N | true .  %(5)
```

呼びだし方

```
?- do_loop:loop_1(1,100,2).
```

7.6 整数生成プログラム

2 つの整数 $M, N(M \leq N)$ が与えられたとき，M より始まり N までの整数列をストリームとして次々に生成するプログラムを書け．

```
:- module integer.
:- public generate/3.

generate(N,N, IntS) :- true | IntS = [].
generate(M,N, IntS) :- M < N |
        IntS = [M|IntS1],
        M1 := M + 1,
        generate(M1,N, IntS1).
```

実行例

```
| ?- integer:generate(2,8,X)|X.
X = [2,3,4,5,6,7,8]

yes.
| ?-
```

7.7 ストリームの圧縮

与えられた入力ストリームから同一要素を全て除去したストリームを作るプログラムを書け．

```
:- module stream.
:- public compact/2.

compact([], Ys) :- true | Ys = [].
compact([X|Xs], Ys) :- true |
        Ys = [X|Ys1],
        filter(X, Xs, Zs),
        compact(Zs, Ys1).

filter(_, [], Ys) :- true | Ys = [].
filter(K, [K|Xs], Ys) :- true | filter(K, Xs, Ys).
filter(K, [X|Xs], Ys) :- K \= X |
        Ys = [X|Ys1], filter(K, Xs, Ys1).
```

実行例

```
| ?- stream:compact([1,2,3,4,5,4,3,2,1],Y)|Y.
Y = [1,2,3,4,5]

yes.
| ?-
```

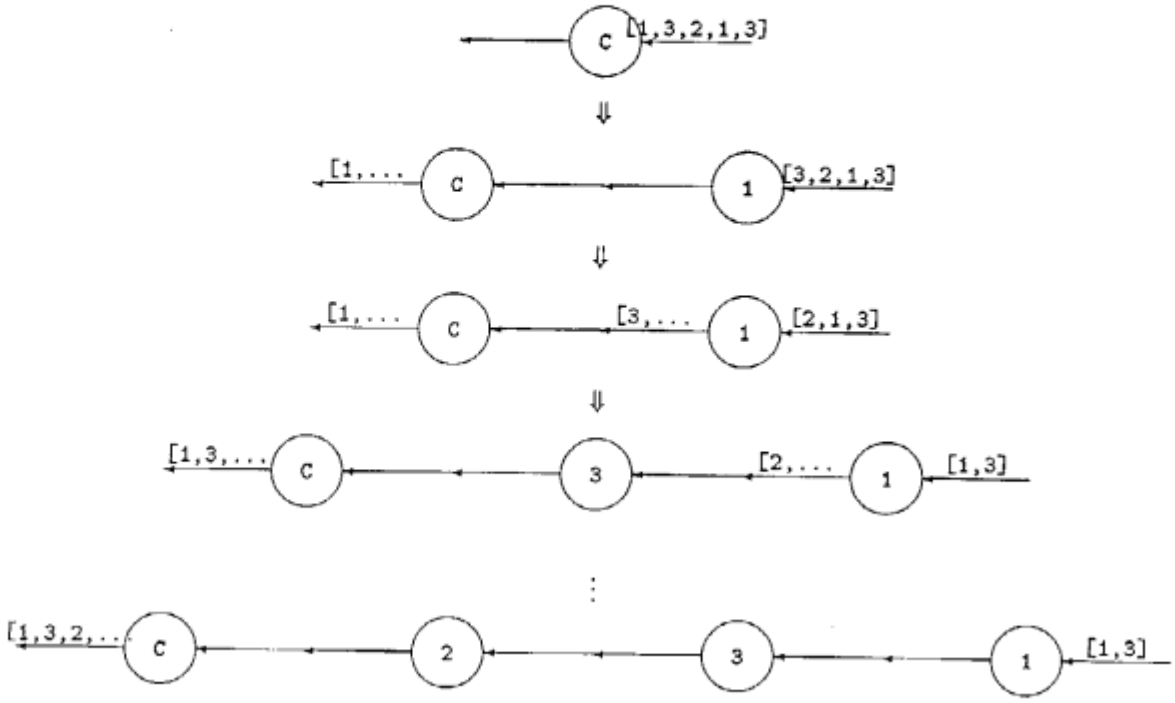


図 7.1: プロセス生成図

7.8 行列の掛け算

行列 M,N が与えられた時，その積 MN を求めるプログラムを書け．
行列の行ベクタをリストで表現し，そのリストのリストで行列を表現することにする．

```
:- module matrix.
:- public multi/3.

multi([],_, AB) :- true | AB = [].
multi([ARow|A],B, AB) :- true |
        multiRow(ARow, B, ABRow),
        AB = [ABRow|AB1],
        multi(A, B, AB1).

multiRow(_, [[]|_], ABRow) :- true | ABRow = [].
multiRow(ARow, [BRow|B], ABRow) :- BRow \= [] |
        map([BRow|B], BCars, BCdrs),
        innerProduct(ARow, BCars, IP),
        ABRow = [IP|ABRow1],
        multiRow(ARow, BCdrs, ABRow1).

innerProduct(V1,V2, IP) :- true |
        innerProduct(V1,V2, 0,IP).

innerProduct([],[], IP,NIP) :- true | NIP := IP.
innerProduct([E1|V1],[E2|V2], IP,NIP) :- true |
```

```
        IP1 := E1 * E2 + IP,
        innerProduct(V1,V2, IP1,NIP).

map([], XCars,XCdrs) :- true | XCars = [], XCdrs = [].
map([X|Xs], XCars,XCdrs) :- true |
        carCdr(X, XCar,XCdr),
        XCars = [XCar|XCars1], XCdrs = [XCdr|XCdrs1],
        map(Xs, XCars1,XCdrs1).

carCdr([H|T], Car,Cdr) :- true | Car = H, Cdr = T.
```

実行例

```
| ?- matrix:multi([[1,2],[3,4],[5,6]],[[6,5,4],[3,2,1]],X)|X.
X = [[12,9,6],[30,23,16],[48,37,26]]

yes.
| ?-
```

第 II 部

初級編

第 8 章

プロセスを利用したプログラム例

KL1 プログラミングでは単純なプロセスを連結したプロセスネットワークを意識してプログラミングすることが多い．本章では簡単なプロセスを幾つか連結しプロセスネットワークを形成するプログラム例として，クイックソートプログラムと最適経路問題プログラムを取り上げる．

8.1 クイックソートプログラム

クイックソートプログラムを例にとって，差分リストの有効性と並列実行について考えてみる．

```
:- module sort.
:- public quicksort/2.

quicksort(Xs, Ys) :- true | qsort(Xs, Ys, []).          %(1)

qsort([], Ys0, Ys1) :- true | Ys0 = Ys1.                %(2)
qsort([X|Xs], Ys0, Ys3) :- true |
        partition(Xs, X, S, L), qsort(S, Ys0, Ys1),
        Ys1 = [X|Ys2], qsort(L, Ys2, Ys3).              %(3)

partition([], P, S, L) :- true | S = [], L = [].        %(4)
partition([X|Xs], P, S, L) :- X < P |
        S = [X|S0], partition(Xs, P, S0, L).            %(5)
partition([X|Xs], P, S, L) :- X >= P |
        L = [X|L0], partition(Xs, P, S, L0).            %(6)
```

quicksort(Xs, Ys) は数のリスト Xs を整列し結果を Ys に返すプログラムである（図 8.1）．

図 8.1: quicksort プロセス

qsort(Xs, Ys0, Ys1) は数のリスト Xs を整列し，結果を差分リスト (Ys0, Ys1) で返すものである．最初に qsort を呼ぶ時には差分リストの後尾は空であるので，後尾に空リスト [] を代入して呼ばなければならない（節 (1)）．

partition(Xs, P, S, L) は数のリスト Xs を数 P より小さい数のリスト S と P 以上の数のリスト L に分割するプログラムである（図 8.2, 8.3 節 (4), (5), (6)）．

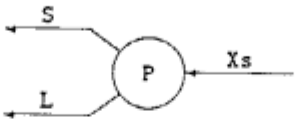


図 8.2: partition プロセス

図 8.3: partition プロセスの機能

節 (3) が，このプログラムで一番重要な部分である．まず受け取ったリスト [X|Xs] を X より小さい数のリスト S と X 以上の数のリストに分け (partition(Xs, X, S, L))，S と L を再帰的に整列する (qsort(S, Ys0, Ys1), qsort(L, Ys2, Ys3))．プロセス qsort(S, Ys0, Ys1) と qsort(L, Ys2, Ys3) は独立な仕事なので[1]，並列に実行して結果 (Ys0, Ys1) と (Ys2, Ys3) を同時に生成することが可能になる．もう 1 つのゴール Ys1 = [X|Ys2] は要素が X のみの差分リスト (Ys1, Ys2) を作る．このように並列に作られた 3 つの差分リスト (Ys0, Ys1), (Ys1, Ys2), (Ys2, Ys3) は，Ys1 と Ys2 がつなぎ役になり，長い差分リスト (Ys0, Ys3) を構成する (図 8.4)．これが節 (3) の返す値である．

節 (3) は差分リストを作る 3 つのゴールの実行順序を定めていないだけではなく，これらとゴール partition との実行順序も定めていない．これは入力リストの分割と分割リストの整列を並列に実行することも可能である．partition(Xs, X, S, L) は S と L を一挙にではなく，Xs の頭の方が決まるに従い，S と L を一挙にではなく，徐々に決めていく．一方，qsort の 2 つの再帰呼出しも，S と L の値を一挙に調べるのではなく，頭の方から 1 要素ずつ分解して調べる．つまり，分割作業が終わらなくても，整列作業を始めることができる．

partition と qsort 間の並列性が qsort 間の並列性と異なる点は，qsort の実行が partition の実行の進み具合によって制約を受ける点である．qsort は partition の生成するリストを入力としているので，partition がまだ作っていない部分にたどりついたら，中断機構により qsort の実行は中断する．このように KL1 では，リストを生成するプロセスと消費するプロセスの間での同期[2]は，特にプログラマが意識しないでも取られる．

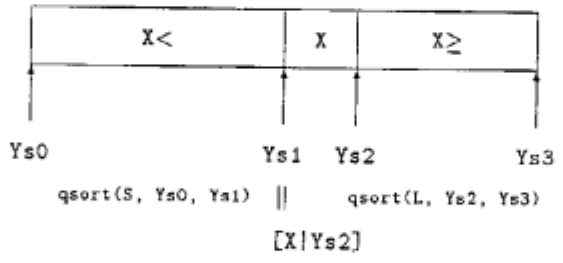


図 8.4: qsort(Xs, Ys0, Ys3) の結果の構成

[1] 一方が他方の結果を利用することはない

[2] リストの消費はリストの生産に先行してはならないという意味

8.2 最適経路問題プログラム

プロセスネットワークを意識できる問題として，ネットワークの最適経路問題プログラムを取り上げる．

問題 8.1 (最適経路問題) ネットワーク各辺にそれぞれ非負の値 (コスト) が割当てられている時，その上の 2 点 (開始点と終了点) 間を結ぶ経路のうちコスト最小の経路を求めよ．

例えば，図 8.5 のようなネットワークの場合 点a から点p への最適経路は [a, e, i, j, f, g, k, o, p] で，コストは 23 である．

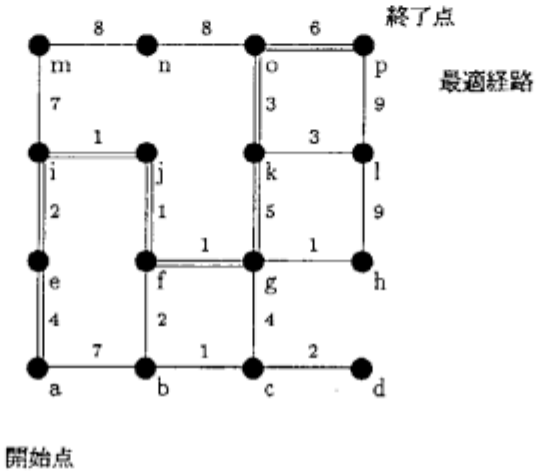


図 8.5: ネットワーク

この問題の解法は色々考えられるだろうが，ここではプロセスのネットワークを利用した 1 解法を示すことにする．

何をプロセスにするか? ネットワークのノード． 辺に対応するように双方向にストリームでプロセスをつなぐ．

何をメッセージとして流すのか? 各辺のコストに相当する情報．

8.2.1 アルゴリズム

与えられたネットワークの各ノード毎にプロセスを生成し，ノード間のメッセージ通信によって最適解を求めていく．

メッセージには，開始点からそのメッセージを受信したノードまでのコストと経路が含まれている．最初に開始点へコスト 0 のメッセージ cp(0, _) が送られることによりメッセージ通信が開始される．

また，各ノードは，常にその時点までに受信したメッセージのうちの最小コストとその経路を保持している． 最初に保持しているコストは ∞ である (図 2 参照)．

ノード n がメッセージ cp(Cost, Path) を受信した時，値 Cost がその時点で保持しているコストの値 C より小さい場合は，それまで保持していたものを捨てて新しい値 Cost, Path を保持することにし，さらにすべての隣接ノード m へメッセージ cp(Cost + C', [n|Path]) を送る． ただし，C' はノード n からノード m へのコストである (図 8.7 参照)． 一方，Cost が C と等しいかまたは Cost がより大きい場合，このメッセージは捨てられ，ノードは何もしない (自分の保持している値 C の方がより最適解に近いことを意味する)．

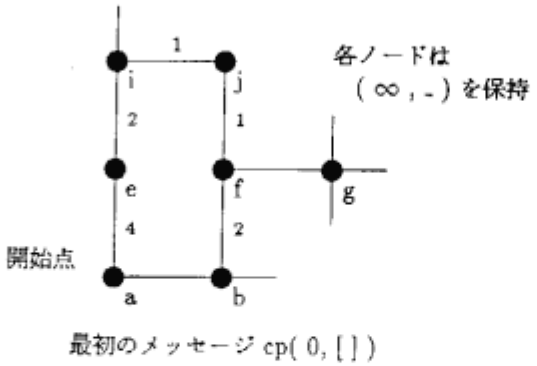


図 8.6: 最初のメッセージ

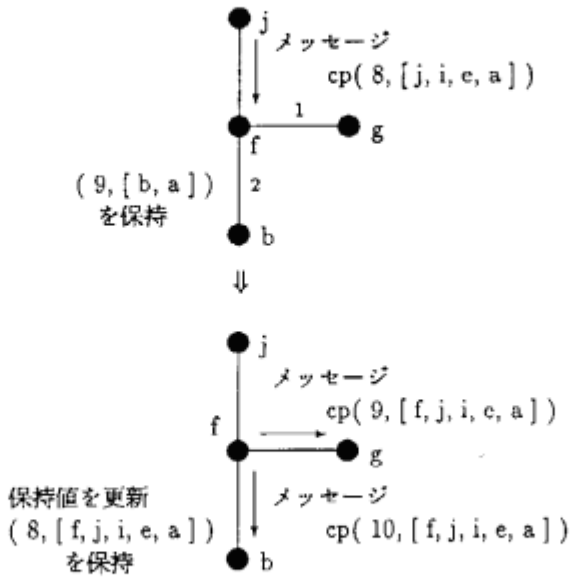
このようにしてすべてのメッセージがネットワーク上から消滅した時, 各ノードは, 開始点からそれぞれのノードまでの最適経路とそのコストを保持している.

終了判定の方法として, ネットワーク上のすべてのメッセージが消滅したことを知るために, 実際に送るメッセージにショートサーキットのスイッチをつける. これは 2 つの変数から成っている. あるノードがメッセージを受信した時, さらに他のノードへメッセージを送る場合は, 受信したメッセージに含まれるスイッチを分解してそれぞれ新しいメッセージにつけて送る. すなわち, 受け取ったあるメッセージ cp(C, P, S_0, S_n)(図 8.8) に対して (S_0, S_n はショートサーキットのスイッチを示す変数), 新たに n 個のメッセージを送信する場合, n 個のメッセージは n-1 個の新出の変数 S_1, S_2, $\cdots$, S_{n-1} を用いて cp(C_i, P', S_{i-1}, S_i) で表される (図 8.9). また, 受信したメッセージを消滅させる場合は, スイッチを閉じる. スイッチを閉じるには, スイッチを表す 2 つの変数 S_{i-1}, S_i をユニファイする (図 8.10). このようにして, 実行開始時に投げられる最初のメッセージのスイッチの一方をアトムなどに具体化しておくと, もう一方の変数が具体化した時, 実行が終了したことを知ることができる (図 8.11).

8.2.2 プログラム

8.2.1 アルゴリズムで述べた方法をプログラムしてみたのが次のプログラムである.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       node process                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% node(In, Outs, Cout, C,P, End, RO-R, Cs, N)
%
%       In      : 入力ストリーム
%       Outs    : 出力ストリームのリスト
%       C       : いままでで分かっている開始点からの最小コスト
%       P       : コスト C の道
%       End     : ショートサーキットフラグ
%       RO-R    : 結果を返す差分リスト
%       Cs      : Outs に対応する辺のコストのリスト
%       N       : ノードの識別子
%
node([cp(Cost, Path, TO-T)|In], Outs, C,P, End, RO-R, Cs, N) :-
    Cost >= C |
    TO = T,
    node(In, Outs, C,P, End, RO-R, Cs, N).                          %(1)
```



ノードIについて，コスト 9 の経路 [b, a] (開始点 a から自分までの経路 [a, b, f] を示している) を
それまでの最適解として保持している時，メッセージ cp(8, [j, i, e, a]) を受信した場合の動作:
新しく到着した情報の方がより最適解に近いので，コスト 8, 経路 [j, i, e, a] を保持することにし，
ノード g へメッセージ cp(8+1, [f, j, i, e, a]), ノード b へメッセージ cp(8+2, [f, j, i, e, a]) を送る.

図 8.7: 途中のメッセージ

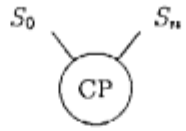


図 8.8: ショートサーキット (1)

```
node([cp(Cost, Path, T0-T)|In], Outs, C,P, End, R0-R, Cs, N) :-
    Cost < C |
    send_cp(Outs, Cs, Cost, [N|Path], T0-T, NewOuts),
    node(In, NewOuts, Cost, Path, End, R0-R, Cs, N).                %(2)
node(_, Outs, C,P, end, R0-R, _,N) :- true |
    R0 = [N-C-P|R],
    closeOuts(Outs).                                                %(3)

send_cp([], _,_, _, T0-T, NewOuts) :- true | T0 = T, NewOuts = [].  %(4)
send_cp([Out|Outs], [C|Cs], Cost, NewPath, T0-T, NewOuts) :- true |
    NewCost := Cost+C,
    Out = [cp(NewCost, NewPath, T0-T1)|Out1],
    NewOuts = [Out1|NewOuts1],
    send_cp(Outs, Cs, Cost, NewPath, T1-T, NewOuts1).               %(5)

closeOuts([]) :- true | true.                                       %(6)
```

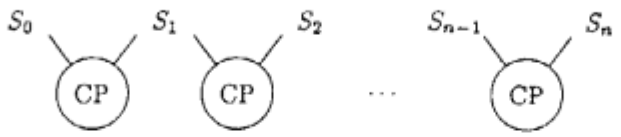


図 8.9: ショートサーキット (2)

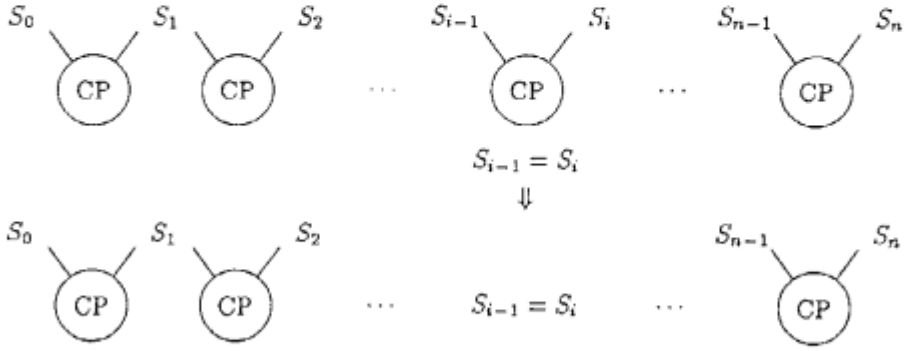


図 8.10: ショートサーキット (3)

```
closeOuts([Out|Outs]) :- true | Out = [], closeOuts(Outs).      %(7)
```

メッセージに含まれるコスト (Cost) が, 自分の持っているコスト (C) 以上の場合, ショートサーキットのスイッチをつけて (T0-T), メッセージを捨てる (節 (1)). 一方, メッセージに含まれるコストが自分の持っているコストより小さい場合 (節 (2)), 古いコストと道順を捨て, 新しいコストと道順を覚える (node(In, NewOuts, Cost, Path, End, R0-R, Cs, N)).

そして, その道順 (Path) に自分の識別子を付加 ([N|Path]) し, 次のノードに各辺のコストを加えたメッセージを送る (節 (5)). この時ショートサーキット用の変数 (T1) を新たに生成する.

このプログラムを用いると, 開始点から他の全ての点への最適経路を求めることができる. 図 8.12 の場合次のような呼出しで, 解を求めることができる.

```
bestPath(Answer) :- true |
        node([cp(0,[],end-End)|AIn],AOuts,99999,_,End,Answer-Answer1,ACs,a),
                merge(AIn1,AIn2,AIn),
                AOuts = [BIn1,DIn1],ACs = [7,4],
        node(BIn,BOuts,99999,_,End,Answer1-Answer2,BCs,b),
```

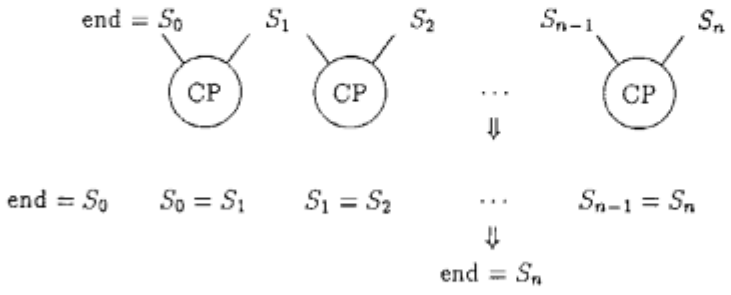


図 8.11: ショートサーキット (4)

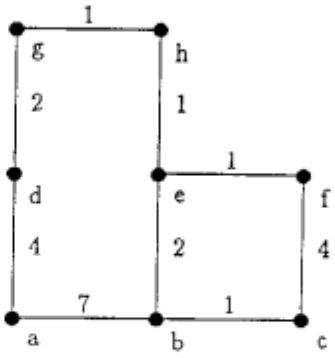


図 8.12: 例題

```
        merge(BIn1,BIn2,BIn0),merge(BIn0,BIn3,BIn),
        BOuts = [AIn1,CIn1,EIn1],BCs = [7,1,2],
node(CIn,COuts,99999,_,End,Answer2-Answer3,CCs,c),
        merge(CIn1,CIn2,CIn),
        COuts = [BIn2,FIn1],CCs = [1,4],
node(DIn,DOuts,99999,_,End,Answer3-Answer4,DCs,d),
        merge(DIn1,DIn2,DIn),
        DOuts = [AIn2,GIn1],DCs = [4,2],
node(EIn,EOuts,99999,_,End,Answer4-Answer5,ECs,e),
        merge(EIn1,EIn2,EIn0),merge(EIn0,EIn3,EIn),
        EOuts = [BIn3,FIn2,HIn1],ECs = [2,1,1],
node(FIn,FOuts,99999,_,End,Answer5-Answer6,FCs,f),
        merge(FIn1,FIn2,FIn),
        FOuts = [CIn2,EIn2],FCs = [4,1],
node(GIn,GOuts,99999,_,End,Answer6-Answer7,GCs,g),
        merge(GIn1,GIn2,GIn),
        GOuts = [DIn2,HIn2],GCs = [2,1],
node(HIn,HOuts,99999,_,End,Answer7-[],HCs,h),
        merge(HIn1,HIn2,HIn),
        HOuts = [EIn3,GIn2],HCs = [1,1].
```

bestPath の定義は，ネットワークの複雑さが増せば増すほど複雑になる．そこで，ネットワークの形状をデータとして与えると，node プロセスのネットワークを形成するプログラムを定義すると便利であるが，それはここでの目的ではないので触れない．興味ある読者は試みて欲しい．

第 9 章

節の選択

KL1 では，ガード部の成功した節のうち任意の 1 つが選択される．これにより条件分岐を記述できる．ガード部に書ける述語は組込みのガード用述語に限られる．本章ではこれら組込み述語の解説を行う．そして，ガード部を書き易くまた読みやすくするための otherwise 機能と，節選択に優先順位をつけるための alternatively 機能を説明する．

9.1 ガードの組込み述語

KL1 の組込み述語は，ボディ部用とガード用に分けられる．ボディ部用組込み述語の主な機能はデータの生成及びデータの操作であり，ガード用組込み述語はもっぱら節選択のためのテストに用いられる．ここではガード用組込み述語の概略とその使い方を述べる．ボディ部用組込み述語については次章で説明する．述語の詳しい仕様は言語マニュアルを参照されたい．

ガード部とボディ部の実行メカニズムの大きな違いはボディ部は並列実行されるのに対して，ガード部は左から右への逐次実行であるという点である．したがってボディ部と違って，ガード部の述語の順番を変えると意味が変わることもあるので注意が必要である[1]．

引数のモード　以降の説明の便宜のために述語の引数のモードについて解説する．引数のモードには入力引数と出力引数の 2 種類ある．以下の説明において，述語の引数に ^ のついている引数は出力引数，ついていない引数は入力引数とする[2]．

入力引数　この引数をあるデータに具体化することが実行開始の必要条件であるもの．

出力引数　実行によりこの引数があるデータにユニファイされる．

入力引数が未定義の場合その組込み述語の実行は中断し，定められたデータに具体化された時に実行を開始し出力引数は定められたデータにユニファイされる．

9.1.1 型判定述語

型別の処理を行いたい時に使用するガード述語である．

`atom(Atom)`	Atom の型はアトムである．
`integer(Integer)`	Integer の型は整数である．

`list(List)` List の型はリストである．ここで注意しなければならないのは，□ はアトムなので `list(□)` は失敗し，`atom(□)` は成功することである．

例 9.1 (成功) □ はアトムである．

[1] 通常書くようなプログラムではこのようなことはあまり起こらないので，最初はあまり気に止めなくてもよい．

[2] 記号 ^ は注釈であり言語シンタックスとは関係ない．

呼びだしゴール　p(□)
節定義　p(X) :- atom(X) | ...

例 9.2 (失敗) □ はリストではない

呼びだしゴール　p(□)
節定義　p(X) :- list(X) | ...

string(String, ^Size, ^ElementSize) String の型はストリングであり, そのサイズは Size 要素サイズは ElementSize である.

例 9.3 (成功) "ghc" が要素サイズ 8 ビットのストリングの時, S に 3 が具体化され ES に 8 が具体化される.

呼びだしゴール　p("ghc")
節定義　p(X) :- string(X, S, ES) | ...

例 9.4 (失敗) "ghc" のサイズは 3 なので失敗する.

呼びだしゴール　p("ghc")
節定義　p(X) :- string(X, 2, 8) | ...

vector(Vector, ^Size) Vector の型はベクタであり, そのサイズは Size である.

例 9.5 (成功) S に 3 が具体化される.

呼びだしゴール　p({K, L, ONE})
節定義　p(X) :- vector(X, S) | ...

例題 9.1 (型判定) 入出力ストリームをそれぞれ一本ずつ持ち, 入力ストリームのメッセージの型を判定し, それを出力ストリームに出力するプログラム.

```
:- module type.
:- 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).
```

実行例

```
| ?- type:check([1, a,"ds", {A, B}], X) |X.
```

```
X = [integer, atom, string, vector]

yes.
| ?-
```

上記の型判定述語は，入力引数が未定義の時には中断し，入力引数が具体化されるとその型により成功か失敗をするガード用述語であった．この他に値によらず成功する述語 wait(X) がある．この述語は X が未定義なら中断し，それ以外では成功する述語である．

例 9.6 (中断) X が具体化されるまで (X = a が実行されるまで) 中断する．

```
呼びだしゴール  p(X), ..., X = a, ..
節定義  p(X) :- wait(X) | ...
```

9.1.2 算術比較

数の大小比較により条件分岐をしたいときには，算術比較述語を用いる．これには以下のようなものがある．

X < Y	X < Y
X > Y	Y < X と同じ．
X =< Y	$X \leq Y$
X >= Y	Y =< X と同じ．
X =:= Y	X, Y は同じ値の数である．
X =\= Y	X, Y は違う値の数である．

例題 9.2 (算術比較) 与えられた数のリストから最大の数を求めるプログラム

```
:- module integer.
:- public max/2.

max([Int|IntegerList], MaxInteger) :- true |
        max(IntegerList, Int, MaxInteger).

max([], Cand, Max) :- true | Max = Cand.
max([I|Is], Cand, Max) :- I =< Cand | max(Is, Cand, Max).
max([I|Is], Cand, Max) :- I >= Cand | max(Is, I, Max).
```

実行例

```
| ?- integer:max([2, 6,9, 2,5], X) | X.
X = 9

yes.
| ?-
```

9.2 Otherwise

otherwise は否定を書くための略記であり，節のガード部を排他的にするのに用いられる．例えば入門編で扱った not プロセスを考えてみる．

```
:- module not.
:- public not/2.
```

```
not([], Out) :- true | Out = [].                          %(1)
not([0|In], Out) :- true | Out = [1|Out1], not(In, Out1).  %(2)
not([1|In], Out) :- true | Out = [0|Out1], not(In, Out1).  %(3)
```

このスタックプロセスに 0, 1 以外の数を送るとどうなるであろうか. その場合 (1), (2), (3) の節ガードが全て失敗してしまい, not プロセス自体が失敗してしまう. そこで 0, 1 以外の数が来た場合はそれを無視することにする. それには次のような節を加えれば良い.

```
not([M|In], Out) :- M =\= 0, M =\= 1 | not(In, Out).      %(4)
```

節 (4) 処理は, 節 (1), (2), (3) のガードが全て失敗してしまった時の処理である. このような場合, いちいち M =\= 0, M =\= 1 のようにそれらガード条件の否定を書くのは, 面倒なものである. 節が増えるとなおさらである. このような場合 KL1 では次のように otherwise. を用いて書くことができる[3].

```
not([], Out) :- true | Out = [].                          %(1)
not([0|In], Out) :- true | Out = [1|Out1], not(In, Out1).  %(2)
not([1|In], Out) :- true | Out = [0|Out1], not(In, Out1).  %(3)
otherwise.
not([M|In], Out) :- true | not(In, Out).                  %(4')
```

otherwise. は節と節との間に書くことができ, 1 つの述語定義に複数個書くことも可能である. 意味は「otherwise. 以前に書かれた節のガード部が全て失敗してから, otherwise. 以降の節のガードが試される」ということである. otherwise. の前の節と後の節は排他的節となる.

9.3 Alternatively

alternatively は節間に優先順位をつけるものである. 次のような簡単な数式計算プログラムを考える.

```
:- module arithmetic.
:- public compute/2.

compute(X, Z) :- integer(X) | Z := X.                     %(1)
compute(X+Y, Z) :- true |
        compute(X, X1), compute(Y, Y1),
        Z := X1 + Y1.                                     %(2)
compute(X-Y, Z) :- true |
        compute(X, X1), compute(Y, Y1),
        Z := X1 - Y1.                                     %(3)
compute(X*Y, Z) :- true |
        compute(X, X1), compute(Y, Y1),
        Z := X1 * Y1.                                     %(4)
compute(X/Y, Z) :- true |
        compute(X, X1), compute(Y, Y1),
        Z := X1 / Y1.                                     %(5)
```

compute/2 は第 1 引数の数式を計算し第 2 引数に返すプログラムである[4]. 乗算 (節 (4)) の場合 X, Y の計算結果 X1, Y1 のどちらかが 0 であることが分かれば, 一方の計算結果を待たずに乗算結果は 0 と分かる. そのように節 (4) を書き換えると次のようになる.

---

[3] 正確にいうとこのプログラムの意味は前のプログラムの意味と違う. 前のプログラムでは '数' 以外のメッセージが来た場合失敗するが, このプログラムではそのような場合でも節 (4') が機能する.

[4] compute の引数 X+Y, X-Y などは X+Y, X-Y というパターンを表現している. KL1 ではこれらはベクタ {+, X, Y}, {-, X, Y} と同じである.

```
compute(X*Y, Z) :- true |
        compute(X, X1), compute(Y, Y1),
        multiwait(X1, Y1, Z).

multiwait(0, Y, Z) :- true | Z := 0.
multiwait(X, 0, Z) :- true | Z := 0.
multiwait(X, Y, Z) :- X =\= 0, Y =\= 0 | Z := X * Y.
```

multiwait/3 では乗算の引数の計算結果の一方が 0 と分かれば, 他方の計算結果を待たずに直ちに乗算結果 0 を返す. この時他方の計算は最後まで実行され, その計算結果は捨てられる. つまりこのような場合, compute(X, X1), compute(Y, Y1) のどちらか一方は無駄な計算を最後まで続けることになる. そこで乗算計算の時に, 一方の計算結果が 0 と分かった時点でもう一方の計算を終了させるためにプログラムを次のように書き換える.

```
:- module arithmetic.
:- public compute/2.

compute(X, Z) :- true | compute(X, Z, Stop).                          %(1)

compute(X, Z, stop) :- true | true.                                   %(2)
compute(X, Z, Stop) :- integer(X) | Z := X.                           %(3)
compute(X+Y, Z, Stop) :- true | subcompute(add, X, Y, Z, Stop).       %(4)
compute(X-Y, Z, Stop) :- true | subcompute(subtract, X, Y, Z, Stop).  %(5)
compute(X*Y, Z, Stop) :- true | subcompute(multiply, X, Y, Z, Stop).  %(6)
compute(X/Y, Z, Stop) :- true | subcompute(divide, X, Y, Z, Stop).    %(7)

subcompute(Op, X, Y, Z, Stop) :- true |
        compute(X, X1, NewStop), compute(Y, Y1, NewStop),
        waitSubcompute(Op, X1, Y1, Z, Stop, NewStop).                 %(8)

waitSubcompute(Op, X, Y, Z, stop, NStop) :- true | NStop = stop.      %(9)
waitSubcompute(add, X, Y, Z, Stop, NStop) :- integer(X), integer(Y) |
        Z := X + Y.                                                   %(10)
waitSubcompute(subtract, X, Y, Z, Stop, NStop) :- integer(X), integer(Y) |
        Z := X - Y.                                                   %(11)
waitSubcompute(divide, X, Y, Z, Stop, NStop) :- integer(X), integer(Y) |
        Z := X / Y.                                                   %(12)
waitSubcompute(multiply, 0, Y, Z, Stop, NStop) :- true |
        NStop = stop, Z := 0.                                         %(13)
waitSubcompute(multiply, X, 0, Z, Stop, NStop) :- true |
        NStop = stop, Z := 0.                                         %(14)
waitSubcompute(multiply, X, Y, Z, Stop, NStop) :- X =\= 0, Y =\= 0 |
        Z := X * Y.                                                   %(15)
```

乗算引数の一方の計算結果が 0 であることを, もう一方の引数を計算している compute に知らせるために変数 Stop が用いられる. 一方の計算結果が 0 であることが分かった場合, そのことがもう一方に知らされる (節 (13)(14)). そしてそのことが通知されると計算を終了 (節 (2)) したり, さらに副プログラムに通知 (節 (9)) したりする.

このプログラムで, 節 (2) と (3)~(7) のガード部は排他的ではない. つまり (2) と (3)~(7) のいずれかのガードの両方が成功する場合がある. このような場合どちらの節を選択するかは, 非決定的であっ

た. したがって, compute/3 に計算を終了しても良いことが知らされていても, 節 (2) が選ばれずに他の節が選ばれ続ける場合もある. waitSubcompute/6 でも同様なことがいえる. このように複数の節が選択の候補になる時に, ある節を優先的に選択して欲しい場合, alternatively を用いる.

```
:- module arithmetic.
:- public compute/2.

compute(X, Z) :- true | compute(X, Z, Stop).                          %(1)

compute(X, Z, stop) :- true | true.                                   %(2)
alternatively.
compute(X, Z, Stop) :- integer(X) | Z := X.                           %(3)
compute(X+Y, Z, Stop) :- true | subcompute(add, X, Y, Z, Stop).       %(4)
compute(X-Y, Z, Stop) :- true | subcompute(subtract, X, Y, Z, Stop).  %(5)
compute(X*Y, Z, Stop) :- true | subcompute(multiply, X, Y, Z, Stop).  %(6)
compute(X/Y, Z, Stop) :- true | subcompute(divide, X, Y, Z, Stop).    %(7)

subcompute(Op, X, Y, Z, Stop) :- true |
        compute(X, X1, NewStop), compute(Y, Y1, NewStop),
        waitSubcompute(Op, X1, Y1, Z, Stop, NewStop).                 %(8)

waitSubcompute(Op, X, Y, Z, stop, NStop) :- true | NStop = stop.      %(9)
alternatively.
waitSubcompute(add, X, Y, Z, Stop, NStop) :- integer(X), integer(Y) |
        Z := X + Y.                                                   %(10)
waitSubcompute(subtract, X, Y, Z, Stop, NStop) :- integer(X), integer(Y) |
        Z := X - Y.                                                   %(11)
waitSubcompute(divide, X, Y, Z, Stop, NStop) :- integer(X), integer(Y) |
        Z := X / Y.                                                   %(12)
waitSubcompute(multiply, 0, Y, Z, Stop, NStop) :- true |
        NStop = stop, Z := 0.                                         %(13)
waitSubcompute(multiply, X, 0, Z, Stop, NStop) :- true |
        NStop = stop, Z := 0.                                         %(14)
waitSubcompute(multiply, X, Y, Z, Stop, NStop) :- X =\= 0, Y =\= 0 |
        Z := X * Y.                                                   %(15)
```

alternatively. は節と節との間に書くことができ, 1 つの述語定義に複数個書くことも可能である. 意味は 「複数の節のガードが成功することが分かっていて, それらの中に alternatively. 以前に書かれている節と以降に書かれている節があるある場合, alternatively. 以前に書かれている節を選択する. 」ということである. otherwise. と違って alternatively により, プログラムの論理的意味は変わらない. alternatively は「この節はこの節より優先的に選択して欲しい」という処理系へのコメントと考えることもできる[5].

[5] 待ち合わせするデータが他 PE にある場合自 PE で同に合う節で済ませてしまうこともある. 例えば, Stop = stop が PE_i で実行され PE_j において compute/3 が実行されようとする場合を考えてみる. Stop が stop に具体化されたことが PE_j 内ではまだ分からず, 第 1 引数が X+Y であることが PE_j 内で分かっている場合, 節 (2) が選択されずに節 (4) を選択するかもしれない.

第 10 章

組込みデータ型の操作

本章では KL1 のデータ操作用組込み述語の機能について述べる．これらの述語は主にボディ部用述語として用意されている．整数型に関しては加減乗除などの算術演算，ベクタとストリングに関してはデータの生成，要素の参照や書き換えといった機能をこれらの述語は持つ．また，入力ストリームの数が動的に変えられる組込みの merge の機能についても説明をする．

以下組込み述語の説明において，次のような記法を適宜用いる．

```
vector(X, ^Size) :: G
      ↑              ↑
   呼出し形式      記述可能な場所
```

記述可能な場所が G であればガード部のみで記述できる述語を意味し，B であればボディ部のみで記述可能な述語である．GB はガード部でもボディ部でも記述可能であることを意味する．

10.1 整数演算

KL1 では現在，数として整数型[1]だけを提供している．整数演算[2]には加減乗除，ビット論理演算，シフトがある．これらはガード部ボディ部どちらにも書ける．

10.1.1 加減乗除

`Z := X + Y`	Z は X と Y を加えたものである．
`Z := X - Y`	Z は X から Y を引いたものである．
`Z := X * Y`	Z は X と Y を掛けたものである．
`Z := X / Y`	Z は X を Y で割った商である．
`Z := X mod Y`	Z は X を Y で割った余りである．

10.1.2 ビット論理演算

`Z := X /\ Y`	Z は X と Y 各ビットの論理積である．
`Z := X \/ Y`	Z は X と Y 各ビットの論理和である．
`Z := X xor Y`	Z は X と Y 各ビットの排他的論理和である．
`Z := \(X)`	Z は X の 各ビットを反転したものである．

10.1.3 シフト演算

シフト幅 (下では Y) は,0 以上 31 以下の整数でなければならない．

`Z := X << Y`	Z は X を Y ビット左にシフトしたものである．
`Z := X >> Y`	Z は X を Y ビット右にシフトしたものである．

[1] 現在は 32 ビットで内部表現されている．

[2] PDSS ではオーバフローは検出されない．Multi-PSI V2 ではオーバフローはガード部では失敗，ボディ部では例外となる．

以上のものはマクロであり, 例えば次のように展開される.

$$Z := X + Y \Rightarrow add(X, Y, Z)$$

10.2 ストリーム操作

入門編で扱った 2 入力マージャ (図 10.1) を利用して 3 入力マージャを構成すると図 10.2のようになる.

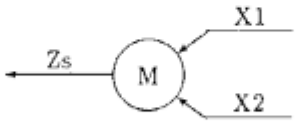


図 10.1: 2 入力マージャ

一般に, 2 入力マージャで n 入力マージャを構成しようとすると 2 入力マージャが n-1 個必要になる. KL1 のプログラムではストリームによるプロセス間通信が基本的な技法の 1 つとなっており, n 入力マージャが頻繁に必要となる. このような場合に 2 入力マージャしかないと 2 入力マージャが多く生成されてしまい, 本来の処理の効率にも影響を与える.

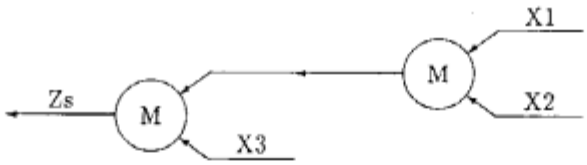


図 10.2: 3 入力マージャ

KL1 ではこのために組込みの多入力マージャを提供している (図 10.3). 組込みのマージャは最初 1 入力のマージャとして生成される (図 10.4).

```
merge(In, Out) :: B
```

組込みのマージャを生成する.

そして, この入力にストリームではなくベクタをユニファイすることによって, そのベクタ要素を入力ストリームとする多入力マージャを作ることができる (図 10.5).

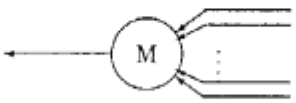
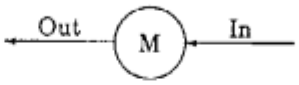


図 10.3: 多入力マージャ

このようにリストの代わりにベクタをユニファイすることにより, マージャの入力を動的に増やすことができる (図 10.6). ストリームを閉じるには [] をユニファイすればよい (図 10.7). また, 入力ストリームが全て閉じられると merge は出力ストリームを閉じ終了する (図 10.8).



merge(In,Out)

図 10.4: 組込みマージャ (1)

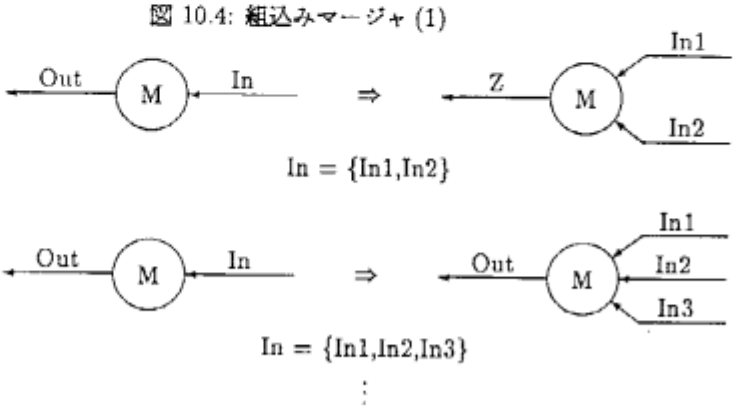


図 10.5: 組込みマージャ (2)

10.3 ストリング (文字列) 操作

ストリングは長さ 0 以上の固定ビット長[3]で表現されるデータの列である. 各要素へのアクセスは一定時間内で実行できる. テキスト上 "(ダブルクォート) で囲った文字列は,PDSS では ASCII コードを右詰めにした 8 ビットのストリングに, Multi-PSI V2 では JIS コードの 16 ビットのストリングとなる. インデックスはゼロオリジンである.

10.3.1 生成

```
new_string(^String, Size, ElememementSize) :: B
```

要素サイズ ElementSize ビットの長さ Size のストリング String を生成する. 各要素は全て整数 0 で初期化される (図 10.9).

10.3.2 要素の参照及び更新

要素の参照は string_element 更新は set_string_element で行われる.

[3]現在のところ,PDSS では 1 ビット以上 32 ビット以下, Multi-PSI V2 では 1,8,16,32 ビットのみが許される.

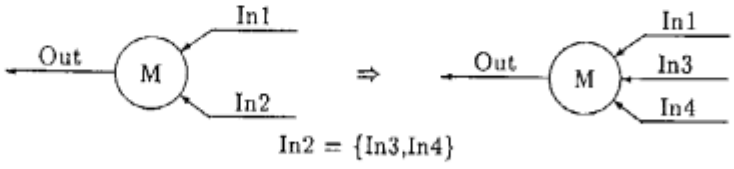


図 10.6: 組込みマージャ (3)

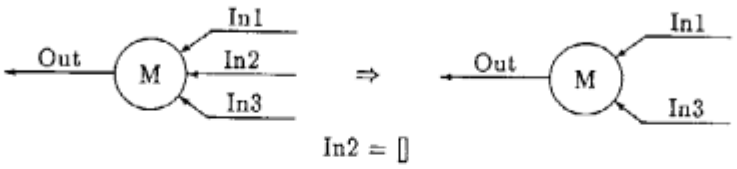
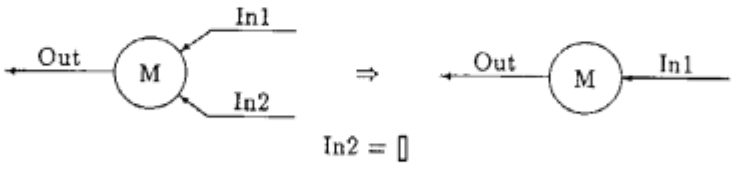


図 10.7: 組込みマージャ (4)



⇓ In1 = []

Out = []

図 10.8: マージャの終了

`string_element(String, Position, ^Element) :: G`

ストリング String の第 Position 要素を整数として Element とユニファイする (図 10.10).

`string_element(String, Position, ^Element, ^NewString) :: B`

ストリング String の第 Position 要素を整数として Element とユニファイすると共に String を NewString とユニファイする.(図 10.11).

`set_string_element(String, Position, NewElement, ^NewString) :: B`

ストリング String の第 Position 番目の要素を NewElement に書き換えたストリングを生成し NewString とユニファイする (図 10.12).NewElement の String の要素サイズを越える部分は無視される.

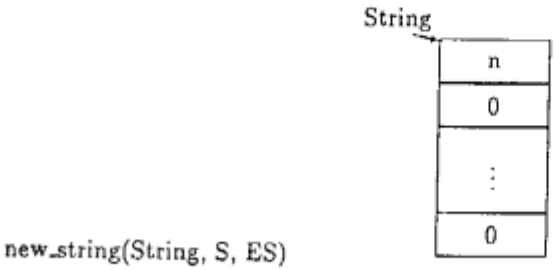


図 10.9: ストリングの生成

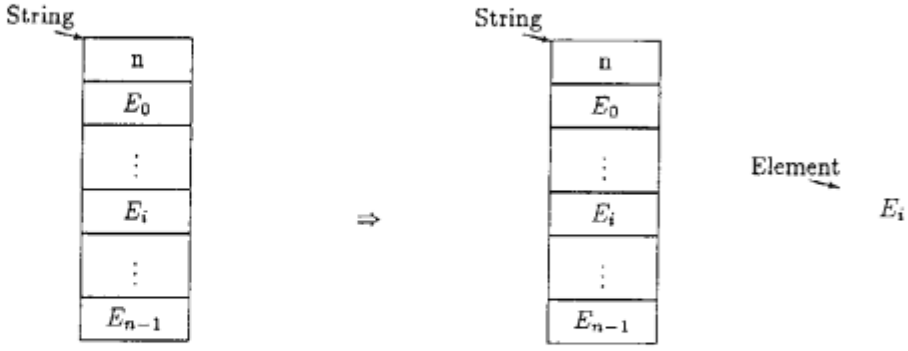


図 10.10: ストリングの要素の参照 (1)

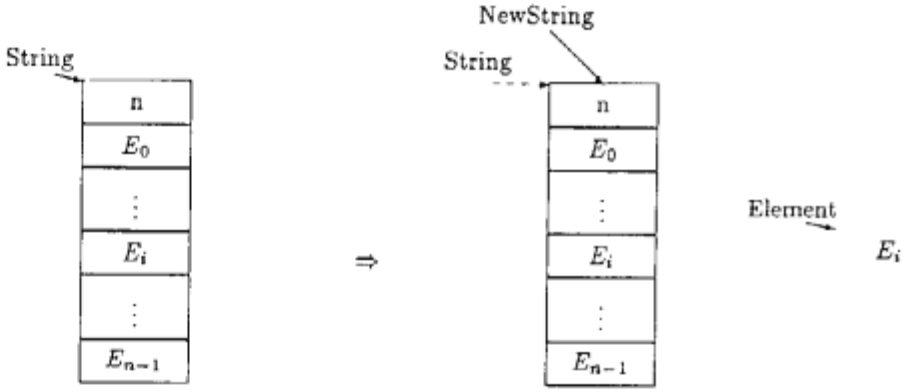


図 10.11: ストリングの要素の参照 (2)

set_string_element(String, Position, NewElement,^NewString) では基本的には String と Position 番目の要素だけ違うストリングを新たに生成し NewString とユニファイする. したがってこの述語の実行には, String のサイズ分の手間が必要ということになる. しかし, もし元の String がもはや参照されないということが分かっていれば, String を直接更新して NewString とユニファイしてよい (図 10.13). String が set_string_element/4 以外で参照されていないことを KL1 処理系が分かっている場合は, このように String を直接更新する. このときはこの述語の実行は, 一定の手間でできる. string_element/4 が 4 引数であるのは「参照が 1 つである」という性質を保つためのものである.

例題 10.1 (ストリングの連結) 2 つのストリングを連結するプログラム

```
:- module string.
:- public append/3.

append(Str1,Str2, Str) :- string(Str1, S1, SE), string(Str2, S2, SE) |  %(1)
```

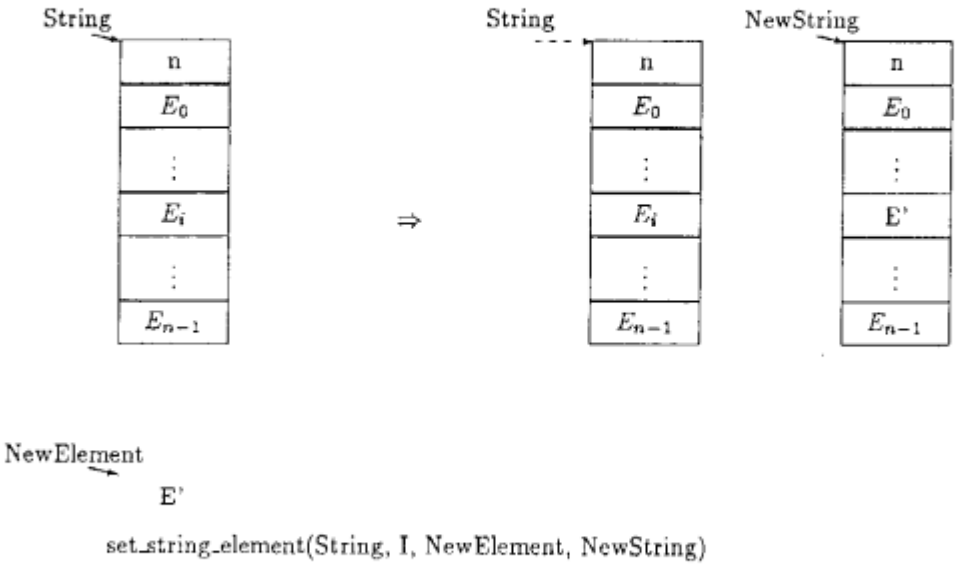


図 10.12: ストリングの要素の更新 (1)

```
        S12 := S1 + S2,                                          %(2)
        new_string(NStr, S12, SE),                               %(3)
        appendArgs(0,S1, 0, Str1, NStr,NStr1),                   %(4)
        appendArgs(0,S2, S1, Str2, NStr1,Str).                   %(5)

appendArgs(N,N, _, _, Str,NStr) :- true | NStr = Str.           %(6)
appendArgs(M,N, SM, OStr, Str,NStr) :- M < N,                    %(7)
        string_element(OStr, M, E) |                             %(8)
        set_string_element(Str, SM, E, Str1),                    %(9)
        M1 := M + 1, SM1 := SM + 1,                              %(10)
        appendArgs(M1,N, SM1, OStr, Str1,NStr).                  %(11)
```

append/3 は連結すべき 2 つのストリングを受け取ると, それらの長さを調べ (1), その和の長さのストリングを生成する (2)(3). appendArgs(From,To, SFrom, OString, String,NewString) はストリング OString の第 From 要素から第 To 要素までを, ストリング String 第 SFrom 要素から順に詰めていき, その結果を NewString に返すプログラムである. OString から要素を取りだし (8), String にその要素を入れている (9).appendArgs/6 により結果となるストリングの前の部分に Str1 を後ろの部分に Str2 を詰め込んでいる (4)(5).

10.4 ベクタ操作

ベクタは任意の KL1 データの列である. ベクタの各要素はなんであってもよく, 未定義変数であってもよい. このところがストリングと大きく違うところである. 各要素へのアクセスは一定時間内で実行できる.

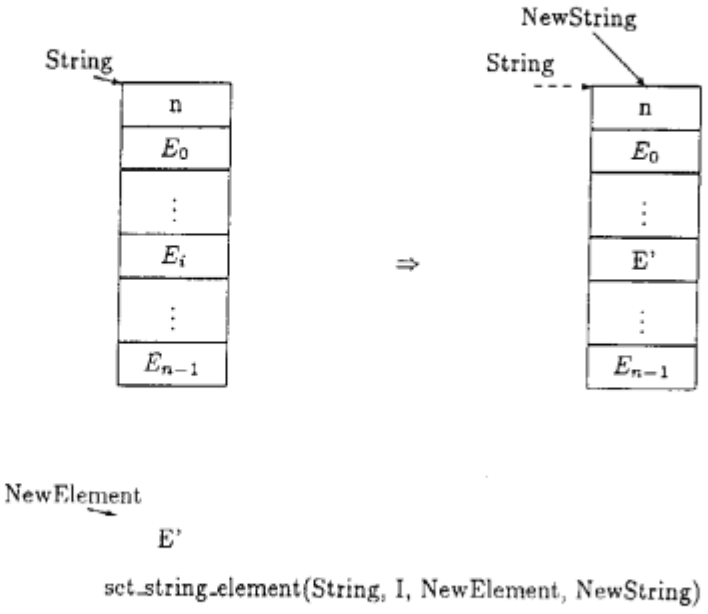


図 10.13: ストリングの要素の更新 (2)

10.4.1 生成

`new_vector(^Vector, Size) :: B`

長さ Size のベクタ Vector を生成する. 各要素は全て整数 0 で初期化される (図 10.14).

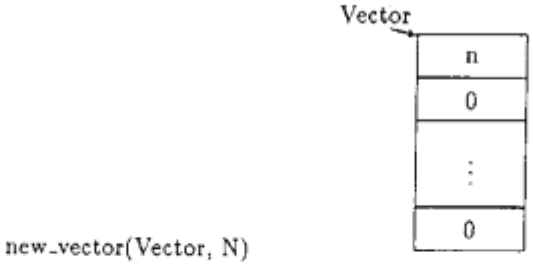


図 10.14: ベクタの生成

10.4.2 要素の参照及び更新

要素の参照・更新は set_vector_element/5 を用い同時に行うことができる.

`set_vector_element(Vector,Position,^OldElement,NewElement, ^NewVector) :: B`

ベクタ Vector の第 Position 番目の要素を OldElement とユニファイすると共に Vector の Position 番目を NewElement と置き換えた新しいベクタを生成し NewVector とユニファイする (図 10.15).

ストリングの場合と同様に,set_vector_element(Vector, Position, OldElement, NewElement, NewVector) では基本的には図 10.15 のように,Vector と Position 番目の要素だけ違うベクタを新

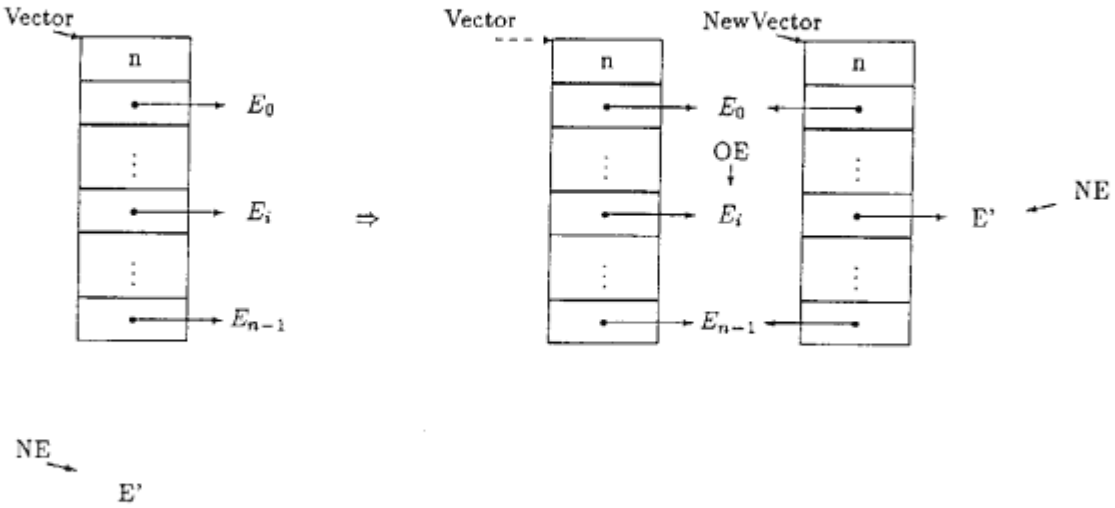


図 10.15: ベクタの要素の更新 (1)

たに生成し, NewString とユニファイする. しかし Vector の参照が 1 つであれることが, 処理系で分かっている場合は Vector を直接更新して, NewVector とユニファイする.(図 10.16). 「参照が 1 つである」という性質を保つために, ベクタの要素の参照と更新はペアで同時に行うのが KL1 では普通である.

ベクタの要素を参照するためには以下のような述語が用意されているが, これら述語は参照される要素に関して, 「参照が 1 つである」という性質を保存しないことに注意して欲しい.

```
vector_element(Vector,Position,-Element) :: G
```

ベクタ Vector の第 Position 番目の要素を Element とユニファイする (図 10.17).

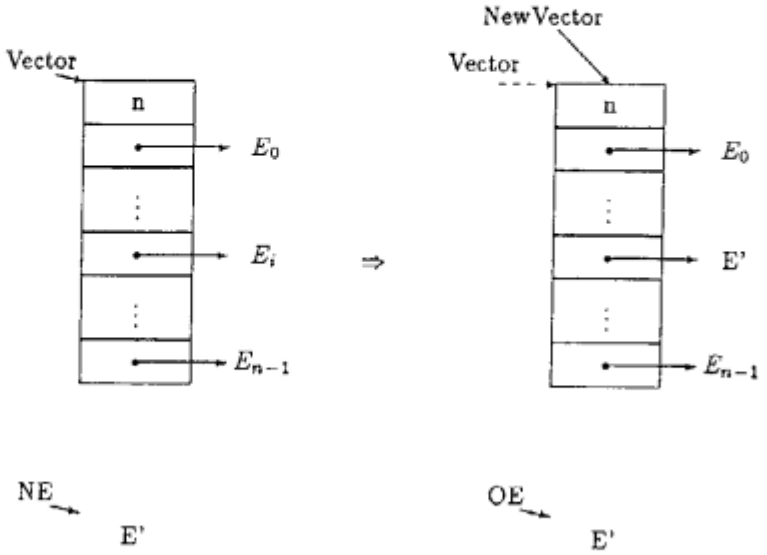


図 10.16: ベクタの要素の更新 (2)

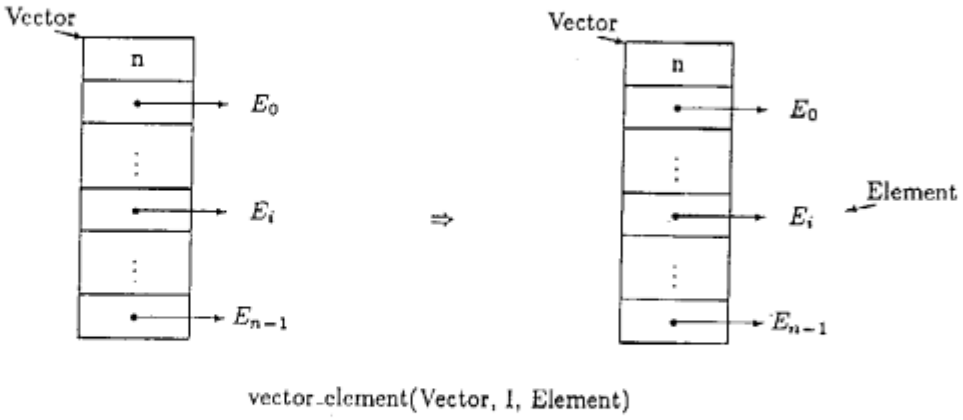


図 10.17: ベクタの要素の参照 (1)

`vector_element(Vector,Position,^Element,^NewVector) :: B`

ベクタ Vector の第 Position 番目の要素を Element とユニファイすると共に Vector を NewVector にユニファイする (図 10.18).

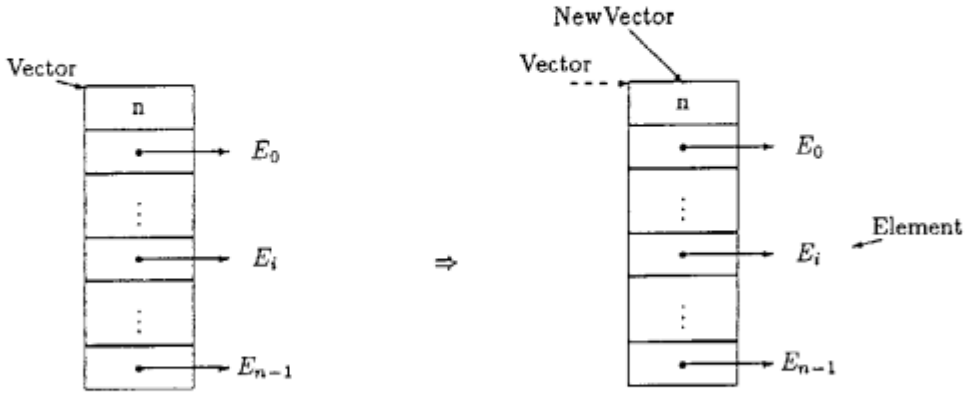


図 10.18: ベクタの要素の参照 (2)

例題 10.2 (転置行列) $m \times n$ の行列が与えられた時その転置行列を求めるプログラム

```
:- module matrix.
:- public transpose/2.

%
% transpose(+Matrix, -TransposedMatrix)
%
%       行列 Matrix の転置行列 TransposedMatrix を求めるプログラム
%
transpose(M,TM) :- vector(M, NumberOfRow),                          %(1)
        vector_element(M,0,M0), vector(M0, NumberOfCol) |           %(2)
        newMatrix(NumberOfCol,NumberOfRow, TM0),                    %(3)
        transpose(M, NumberOfCol,NumberOfRow, TM0,TM).              %(4)

%
% newMatrix(+NumberOfRow,+NumberOfColumn, -Matrix)
%
%       (NumberOfRow,NumberOfColumn) のゼロ行列 Matrix を作る.
%
newMatrix(RowN,ColN, M) :- true |                                   %(5)
        new_vector(M0, RowN),                                       %(6)
        newMatrixArgs(0,RowN, ColN, M0,M).                          %(7)
```

```
newMatrixArgs(To,To, _, M,NM) :- true | NM = M.                    %(8)
newMatrixArgs(From,To, Size, M,NM) :- From < To |                   %(9)
        new_vector(Row, Size),                                      %(10)
        set_vector_element(M, From, _, Row, M1),                    %(11)
        From1 := From + 1,                                          %(12)
        newMatrixArgs(From1,To, Size, M1,NM).                       %(13)

%
% transpose(+Matrix, +NumberofOfRow,+NumberOfColumn,
%               +TransposedMatrix,-TransposedMatrix)
%
%       (NumberOfColumn,NumberOfRow) の行列 Matrix からその
%       (NumberOfRow,NumberOfColumn) の転置行列 TransposedMatrix
%       を作る.
%
transpose(M, RowN,ColN, TM,NTM) :- true |                           %(14)
        transposeArgs(0,RowN, ColN, M, TM,NTM).                     %(15)

transposeArgs(To,To, _, _, TM,NTM) :- true | NTM = TM.              %(16)
transposeArgs(From,To, ColN, M, TM,NTM) :- From < To |              %(17)
        set_vector_element(TM, From, R,NR, TM1),                    %(18)
        transposeArgsRow(0,ColN, From, R,NR, M,M1),                 %(19)
        From1 := From + 1,                                          %(20)
        transposeArgs(From1,To, ColN, M1, TM1,NTM).                 %(21)

%
% transposeArgsRow(+From,+To, +ColumnPosition, +Row,-Row
%               +Matrix,-Matrix)
%
%       転置行列の第 ColumnPosition 行ベクトル Row を作る.
%       (From,ColumnPosition) -> (ColumnPosition,From)
%       Matrix : 元の行列
%
transposeArgsRow(To,To, _, Row,NRow, M,NM) :- true |                %(22)
        NRow = Row, NM = M.                                         %(23)
transposeArgsRow(From,To, ColN, Row,NRow, M,NM) :- From < To |      %(24)
        set_vector_element(M, From, MRow,NMRow, M1),                %(25)
        set_vector_element(MRow, ColN, E,0, NMRow),                 %(26)
        set_vector_element(Row, From, _,E, Row1),                   %(27)
        From1 := From + 1,                                          %(28)
        transposeArgsRow(From1,To, ColN, Row1,NRow, M1,NM).         %(29)
```

行列は行ベクタのベクタで表現されるものとする. 例えば行列

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

は {{1,2,3},{4,5,6}} と表現するものとする. プログラムをトップダウンにみてみる. まず与えられた行列の行数 NumberOfRow と列数 NumberOfCol を調べ (1)(2), NumberOfCol × NumberOfRow の行列を作り (3) その各要素に求められる値を入れていく (4).

newMatrix(NumberOfRow,NumberOfColumn, -Matrix) は NumberOfRow × NumberOfColumn のゼロ行列を生成する. サイズ NumberOfRow のベクタを作り (6), その各要素にサイズ NumberOfColumn のベクタを代入していく (7).

transpose(Matrix, NumberOfRow,NumberOfColumn, TransposedMatrix,NewTransposedMatrix) は受け取ったゼロ行列 TransposedMatrix の各要素を元の行列 Matrix を参照しながら, 求められる値で置き換えていき転置行列 NewTransposedMatrix を求める. 置き換えは行ベクタ毎に行われる (18)(19).transposeArgsRow/7 は行ベクタ R を受け取り, その各要素を適当な値で置き換えて NR にする. ここで, ベクタの要素は未定義変数であっても良かったことを思い出して欲しい. つまり,NR の値が決まらなくても TM1(19) はそこの部分が未定義のままベクタとユニファイされる (図 10.19)[4]. 各 transposeArgsRow は TM のことなる要素を参照しそれの書き換えを行っているので, 並列に実行できる. つまり, ここで転置行列の行数分の並列度が期待できるわけである.

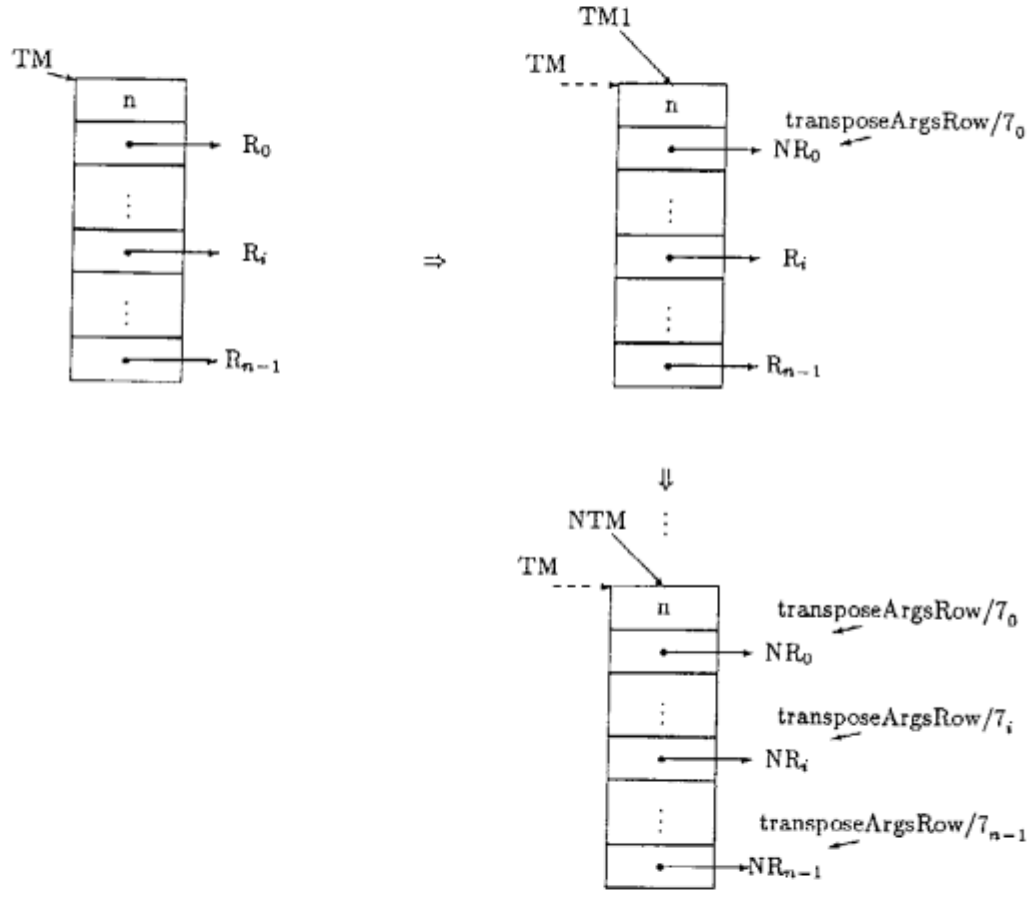


図 10.19: 転置行列

transposeArgsRow/7 において実際の転置が行われている. 元の行列 M の (From,Coln) 要素を (25)(26) 転置行列の (Coln,From) においている (27). 並列度に関してはここでも transposeArgs/6 の議論が成り立ち, 各 transposeArgsRow/7 について転置行列の列数分の並列度が期待できる. したがって全体としては, 行列の要素数分の並列度が期待できるわけである.

KL1 ではこのように本来独立にできる処理は, 自然に並列に実行され得る. しかしこれは, 理想的な計算処理系を仮定しているためであって, 実際の並列計算機では物理的制約により効率良く並列に実行されるとは限らない. このことに関しては, まだ研究の段階であって「こうすれば並列計算機上で効率良く並列に実行される」という一般的な指針もあまり確立されていない. 現在のところ各応用プログラムごとにその制御を考える必要がある. 次章ではこの制御の指定の仕方について説明をする.

[4] このプログラムの場合,TM への参照は 1 つである.

第 11 章

実行の制御

これまでに我々はKL1で実行順序を制御するための方法として,次のようなものを学んだ.

- 再帰呼び出しを用いた繰り返し制御
- 中断メカニズムを利用した同期制御

本章では,KL1で更に細かなゴールの実行制御を行なう方法を勉強する.まず,効率良くプログラムを実行するためのゴールの優先度付けの方法と,実際のプログラムへの応用例を勉強する.次に,ゴールの実行を別プロセッサで行なうための負荷分散指定の方法と応用例を勉強し,負荷分散の基本的な考え方を勉強する.

11.1 優先度制御の目的

逐次型計算機では,処理の順序はプログラムに記述した順序で決まる. しかし,KL1ではボディーゴールの実行はプログラム中に記述した順番とは全く関係なく行なわれる. 言い換えると,これはボディーゴールは並列に実行しても良いと言うことに相当し,並列性を余り意識せずにプログラムしても,実行可能な部分は並列に実行できるという特徴をKL1は備えている.

ただし,システムが勝手にゴールの実行順序を決めてしまっては,プログラムは効率的には動かない場合が多い. それを解決するための機能がゴールの優先度制御である. なお,実行順序を論理的に明確にしなければならない場合は既に勉強した同期制御を行なえば良い. 優先度制御は,実行順序が論理的にはどのようになっても構わない場合に,プログラムを効率的に実行するために行なう.

> ゴールの優先度制御は,
> プログラムを効率的に動かすために行なう.

ここで,優先度を付ける目的をOS (オペレーティングシステム) を例に考えてみよう. OSは種々のプロセスから構成されており,資源管理やユーザプロセスの管理等を行なっている. もしユーザゴールとOSゴールが同じ優先度で動いていると,ユーザゴールの制御をOSが速やかに行なう事はできない. 例えば,ユーザのジョブの中止処理を,ユーザが端末からキー入力する事によって行なう場合を考えてみよう. ここでもし端末からの入力処理を行なうOSのゴールの優先度が,中止したいジョブの各ゴールと同じ優先度だったとしたら,キー入力をしてから実際にジョブが中止されるまでにそのジョブは相当走り続けてしまうであろう. 応答を速くするには,端末からの入出力処理を行なうゴールの優先度をユーザのジョブのゴールより高くしてやれば良い.

ゴールに優先度を付けるための一般的な戦略を以下にまとめる.

- 緊急を要する処理を行なうゴール,或いは直ちに動かしたいゴールは優先度を高くする.
- プロセッサが暇になった時にのみ動けば良いようなゴールは優先度を低くする.

11.2 優先度の指定方法

ゴールの実行優先度は，システムが用意している実行優先度の上限と下限の範囲内に割り付けられる．システムが用意している優先度を物理優先度と呼び，例えば，マルチ PSI では 0 から $2^{12}(=4096)$ まで $2^{12}+1$ 段階の優先度を持っている．なお，物理優先度はシステムに依存するものであるが，プログラムを記述する際にはあまり意識する必要はない．

プログラム中ではゴールの実行優先度を論理優先度で指定する．指定の方法には割合指定と相対指定の 2 種類がある．システムは，ユーザが指定した論理優先度を物理優先度に変換するが，物理優先度は例えば $2^{12}+1$ 段階と限定されるため，ユーザが優先度を変えたつもりでも同じ物理優先度になる場合がある点を注意しておく事．

ゴールの実行優先度を指定しない場合には親ゴールと同じ優先度で実行される．また，組込み述語には優先度を指定する事はできず，親ゴールと同じ優先度で実行される．

優先度割合指定の方法

プログラムが動く時には，その時にゴールが取り得る論理優先度の上限と下限がある．その上限と下限に対する割合で優先度を指定する方法であり，次のような形式で書く．初級編では，この上限と下限についての詳しい説明は行なわないので，上限と下限がある事をここでは理解しておく事．

Goal @ priority(*, 割合)

このときゴール Goal の論理優先度は，

$$\text{下限値} + (\text{上限値} - \text{下限値}) \times \frac{\text{割合}}{4096} \qquad (0 \leq \text{割合} \leq 4096)$$

で決まる．

優先度相対指定の方法

ゴールを呼び出す際，そのゴールは親ゴールの論理優先度を持っており，これを自己論理優先度と呼ぶ．この自己論理優先度に対する相対値を指定する方法であり，次のように書く．

Goal @ priority($, 割合)

このときゴール Goal の論理優先度は，呼び出すゴールの自己論理優先度を Cp とすると，

$$Cp + (\text{上限値} - Cp) \times \frac{|\text{割合}|}{4096} \qquad (0 \leq \text{割合} \leq 4096)$$

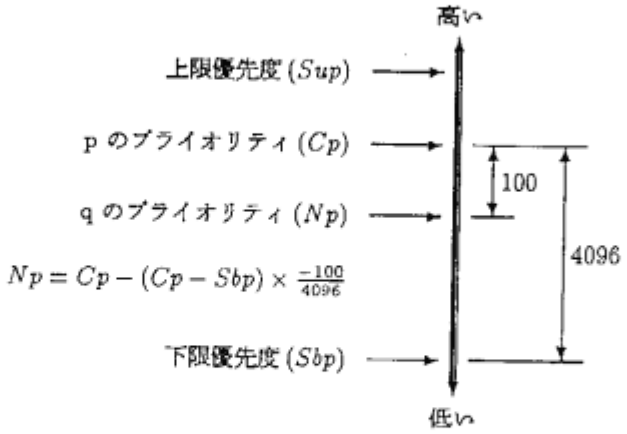
または，

$$Cp - (Cp - \text{下限値}) \times \frac{|\text{割合}|}{4096} \qquad (-4096 \leq \text{割合} < 0)$$

で決まる．

例を示そう．次のプログラムで Goal の論理優先度が Cp とすると，ゴール q の論理優先度 Np は下のように計算される．

```
Goal:       ?- p.
Clause:     p :- true | q @ priority($, -100).
```



なお, どのような場合にどちらの指定方法を選択するかを一般化するのは難しいが, 次のような観点で選択すれば良い[1]. また, 必ずしもこの条件だけで決められるものではなく, 実際にプログラムを書く際に色々試してみるのが良かろう.

割合指定 プロセスの構成がほぼ静的に決まっており, 個々のプロセス毎に付ける実行優先度がわかっている時に用いる.

相対指定 プロセスの構成が動的に変わり, 生成するプロセスと親プロセスの優先度の関係のみがわかっている時に用いる.

11.3 優先度制御を応用したプログラム例

本章では, 優先度制御を使った効率的なプログラム例を示す.

簡単な探索問題

簡単な探索問題を KL1 でプログラムしてみよう. 探索問題は, 探索木の各ノードを探索して解の含まれているリーフ (葉) を見つける問題である. 探索する順序には関係なく, 最悪全てのリーフを探索すれば答えは見つかる. しかし, 探索問題の多くのものは経験的に木の左の方とか右の方とか, どのあたりに解が含まれている確率が高いかわかっているものが多い. ここでは, そのようなヒューリスティックを適用できる探索問題を考えてみる.

なお, 同期制御を行なえば木の左から探索するようにプログラムを書く事は可能であるが, それでは逐次実行しかできない. ここでは, 並列実行ができかつヒューリスティックを適用するように, 優先度制御を使ってみよう.

図 11.1に示されるような 2 分探索木があった時, here を含むリーフを 1 つ見つけるプログラムを書け.

図 11.1において四角はノードを表わし, 楕円はリーフを表わす. また, ノードには a から l の名前が付けてあり, リーフは整数或いはアトムを値として持つ. この探索木をここでは次の様なリストデータで表わす事にする.

```
[[[1,[here,[2,here]]],[here,here]],[[[3,[4,5]],6],[[here,7],8]]]
```

ではまず, ヒューリスティクを用いずにこの探索問題を解くプログラムをみてみよう. ここでは, here が見つかったらそのアトムを要素とするリストを返し, 見つからなかったら空リストを返すようにプログラムしてある. また, 探索を行なうプロセスは一斉に生成する. 解を一つ見つけたプロセスは, 探索中の他

[1] 経験的には, どのような場合も割合指定を用いる事が多い

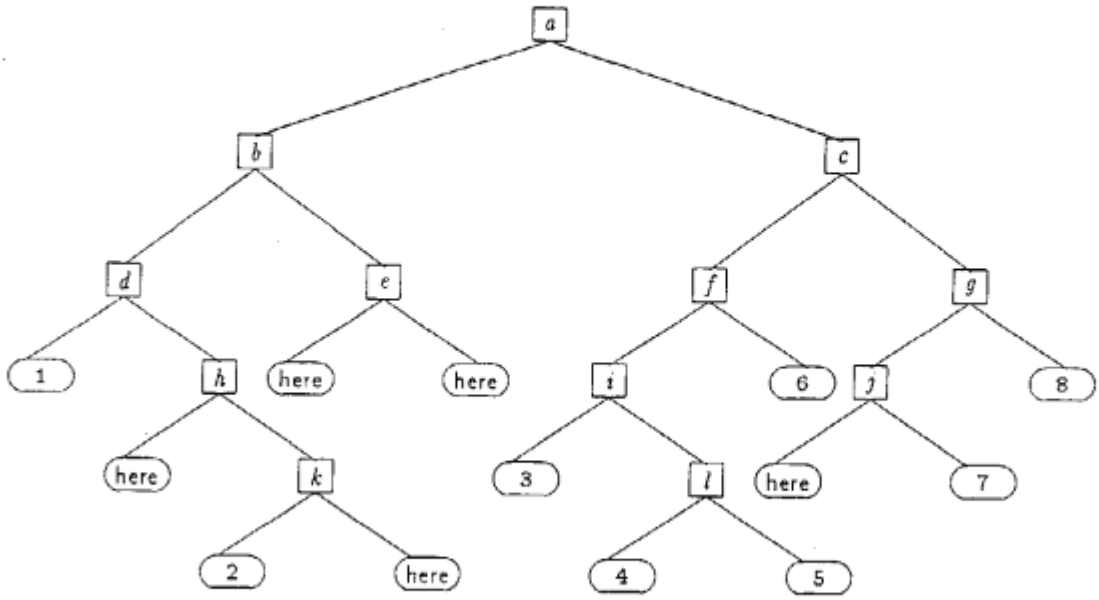


図 11.1: 探索木

のプロセスに対してこれ以上探索しなくてもよい旨を通知する．なお，一斉に通知するために全ての探索プロセスは同一の制御用変数を持ち廻る．各探索プロセスは，この変数の値が決まるのを alternatively の機能を使ってチェックする．

```
:- module simple_search.
:- public search/2,fork/3.
        search(Tree,Output):- true |                    %(1)
                merge(Results,Output),                  %(2)
                control(Output,Cont),                   %(3)
                fork(Tree,Results,Cont).                %(4)
        fork(_,R,stop):-   true | R=[].                 %(5)
    alternatively.                                      %(6)
        fork(H,R,Cont):- integer(H) |R=[].              %(7)
        fork(here,R,Cont):- true |                      %(8)
                R=[here].                               %(9)
        fork([TL,TR],R,Cont):-                          %(10)
                R={R0,R1},                              %(11)
                fork(TL,R0,Cont),                       %(12)
                fork(TR,R1,Cont).                       %(13)
        control([here|_],Cont):- true | Cont=stop.      %(14)
        control([],Cont):- true | true .                %(15)
                実行例
        ?- simple_search:search(Tree,Out),              %(16)
        Tree=[[[1,[here,[2,here]]],[here,here]],
             [[[3,[4,5]],6],[[here,7],8]]]|Out.         %(17)
          Out=[here]                                    %(18)
```

では，プログラムの説明を行なう．

search このプログラムのトップレベルの述語である．マージャプロセス merge を生成し，出力ストリームを Output，入力ストリームを Results とする．また，解が 1 つ見つかった時に探索を終了するための制御を行なうプロセス control と探索を行なうプロセス fork を生成する．

fork クローズ (5) では，変数 Cont の値がアトム stop に決まったらそれ以降の探索を中止する．Alternatively を用いており，このクローズは必ず最初に選択される．Cont の値が決まっていなかったら，(7) 以降のクローズの選択を行なう．

fork クローズ (7) では，H が整数ならばそれはリーフなのでそれ以上探索は行なわない．また，それは解ではないので解を集めるストリームには [] を流す．

fork クローズ (8) では，H が here なら，それはリーフなのでそれ以上探索は行なわない．また，それは解なのでストリームには here を流す．

fork クローズ (10) では，探索木が 2 要素のリストで各ノードを表わしている場合に，その左の探索木 TL と 右の探索木 TR のそれぞれを探索するプロセスを (12) と (13) で生成している．なお，(11) の部分ではここで生成した 2 つのプロセスから返ってくる解のストリーム R0 と R1 をマージインしている．

control クローズ (14) では，解を集めるストリーム Output に解が 1 つでも見つかった時に，他の探索を行なうプロセスを中止するために変数 Cont の値を stop に決めている．

ここで，fork プロセスを生成する部分 (12) と (13) をみてみよう．(12) はノードの左を探索するプロセスを生成しており，(13) ではノードの右を探索するプロセスを生成している．このプログラムを実行すると，もしかしたら右側の探索が優先的に実行され，例えばノード c の下の探索ばかりが実行されるかも知れないが，この下には解が 1 つしかないので効率は良くない．

では，左方向深さ優先のヒューリスティクスを，ゴールの優先度制御を用いてこのプログラムに適用してみよう．優先度を付ける戦略としては，探索木の左側を探索するゴール (12) の優先度を右側のもの (13) より高くするようにする．即ち，ゴール (12) を呼び出すところで優先度を次々に低くしてやれば良い．しかし，せっかく優先度を低くしても別のゴールと優先度がぶつかってしまってはその効果は小さい．そこで，次のようにしてゴール (13) のプライオリティを決める事にする．

1. 探索を行なう各プロセス fork は，自分が使っても良い優先度の最高 P_{high} と最低 P_{low} を保持する．

2. 自分自身は，その時の最高の優先度 P_{high} で実行する．

3. 枝分かれする際，探索木の左を探索する fork は自分と同じ優先度，即ち最高の優先度 P_{high} で実行する．同時に，そのプロセスが使ってもよい最高 PC_{high} と最低 PC_{low} の優先度を知らせる．

$$PC_{high} = P_{high}$$
$$PC_{low} = \frac{P_{high} - P_{low}}{2} + 1$$

4. また，探索木の右を探索する fork は自分が使っても良い最低の優先度より低くする．即ち，$P_{low} - 1$ で実行する．同時に，そのプロセスが使ってもよい最高 PC_{high} と最低 PC_{low} の優先度を教えてやる．

$$PC_{high} = \frac{P_{high} - P_{low}}{2}$$
$$PC_{low} = P_{low}$$

このアルゴリズムに従って優先度を付けると，探索木の各ノード及びリーフにおけるプロセス fork の優先度は図 11.2のようになる．図中，各ノード及びリーフの右肩或いは左肩の数字は優先度を表わす．また，続けてプログラムをみてみよう．

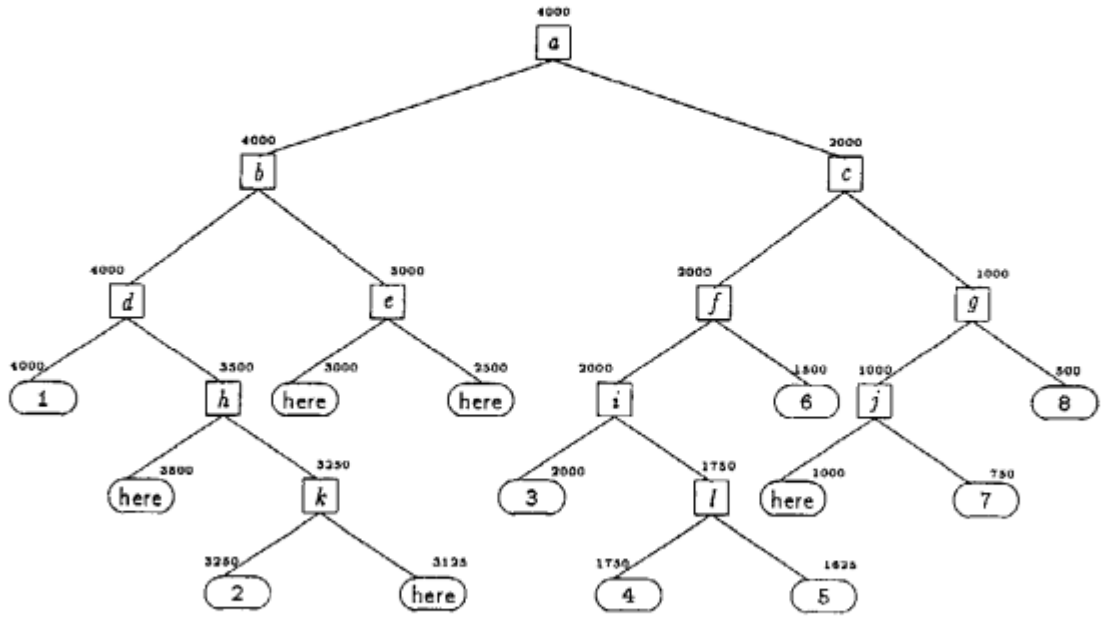


図 11.2: 探索木の各ノードプロセスの優先度

```
:- module search_pri.
:- public search/4,fork/5.
        search(Tree,Output,Phigh,Plow):- true |          %(1)
                merge(Results,Output),                   %(2)
                control(Output,Cont),                    %(3)
                fork(Tree,Results,Phigh,Plow,Cont).      %(4)
        fork(_,R,_,_,stop):- true | R=[].                %(5)
        alternatively.                                   %(6)
        fork(H,R,_,_,_):- integer(H) | R=[].             %(7)
        fork(here,R,_,_,Cont):- true |                   %(8)
                R=[here].                                %(9)
        fork([TL,TR],R,Phigh,Plow,Cont):-true |          %(10)
                R={R0,R1},                               %(11)
                PChighR:= (Phigh-Plow)/2,                %(12)
                PClowL := PChighR+1,                     %(13)
                fork(TL,R0,Phigh,PClowL,Cont),           %(14)
                fork(TR,R1,PChighR,Plow,Cont)
                        @priority(*,PChighR).            %(15)
        control([here|_],Cont):- true | Cont=stop.       %(16)
        control([],Cont):- true | true .                 %(17)

                実行例
?- search_pri:search(Tree,Out,4000,0),                   %(18)
   Tree=[[[1,[here,[2,here]]],[here,here]],
         [[[3,[4,5]],6],[[here,7],8]]]|Out.              %(19)
```

```
Out=[here]                                          %(20)
```

先のプログラムから変更した部分は，次の通りである．

search 引数を2つ増やして，探索を行なうプロセスが使っても良い最高と最低の優先度をパラメタで渡せるようにした．

fork 引数を2つ増やして，自分の子プロセスが使える最高と最低の優先度をパラメタで渡せるようにした．

(12) と (13) の部分で，子プロセスが使っても良い最高と最低の優先度を計算している．

(15) の部分で右を探索するプロセスを生成する際，優先度を下げている．

なお，この例では探索がある程度以上深くなると優先度はこれ以上分割できなくなるが，実際にはその前に物理優先度がぶつかる場合が多い．しかし，優先度制御を行なう事によってプログラムは効率的に動く事は分かったであろう．

11.4 負荷分散制御の目的と考え方

本節では，負荷分散制御の目的と考え方を説明する．一般的に，プログラム中には逐次にしか実行できない部分と並列に実行できる部分がある．にも関わらず，並列に実行できる部分も逐次計算機では逐次に実行するしか方法がない．しかし，並列計算機では逐次に実行できる部分は同じプロセッサで実行し，並列に実行できる部分は別プロセッサで実行でき，その点で問題を素直にプログラム化する事ができる．

KL1では，ボディーゴールの実行はプログラム中に記述した順番とは全く関係なく実行される．また，実行可能な部分は並列に実行される．ただし，逐次マシン上のKL1処理系は並列に実行が可能であるとは言ってもプロセッサは1台しかないので，ゴールは同じプロセッサ上で優先度に基づいて擬似並列的に実行される．従って，本当に並列に実行するためにはマルチPSIやPIM等の並列推論マシンで動かさねばならない．

KL1では，ゴールを別プロセッサで実行させる事を負荷を分散させると呼ぶ．負荷を分散させるには，KL1ではプログラム中でそのための指定を行なう (図11.3) [2]．指定の方法についての詳細は次節で説明する．

負荷分散を行なうにあたっては，プロセスとストリームの構造，及び各ゴールの処理の重さ等についてあらかじめ良く考えておく必要がある．というのも，ゴールの実行を他のプロセッサに依頼するにはそれなりのコストが必要であり，並列な部分を全て別プロセッサに投げれば良いと言うものではない．

また，幾ら並列な部分で処理の大きなゴールであっても，最終的に莫大なデータを返すようなものであっては，通信時間のオーバヘッドが顕著に現れて，結局全体的には並列効果が現れなくなる場合もある．

以下に，負荷分散制御を行なう際に充分気をつけねばならない事項についてまとめてある．これを参考にして負荷分散制御を行なうと，並列効果の良いプログラムを書く指標になる．

(1) 処理の小さなゴールは負荷分散させない． 幾ら並列性があるとは言っても，処理の小さなゴールまでも負荷分散させるのは各種のオーバヘッドによって負荷分散の効果は相殺される．ゴールを投げると，投げるのに要するコストと終了した事を知らせるコストが余分にかかる．これらのコストの和よりも処理の小さなゴールは，負荷分散させない方が良い．このような処理の小さなゴールを負荷分散するには，幾つかの関連のあるゴールをまとめて一つのゴールとし，それを負荷分散する．

(2) なるべくデータのあるプロセッサ上で実行させる． プロセスとデータのあるプロセッサが異なると，その間の通信のオーバヘッドが増大する．プロセスとデータは同じプロセッサ上にあるのが一番望ましい．各プロセスの処理が充分大きくて別プロセッサにする方が効率が良い場合には，通信の多いものを同じプロセッサ上で処理するようにする事．

[2] 指定方法はシステム毎に違い，これはマルチPSIの例である

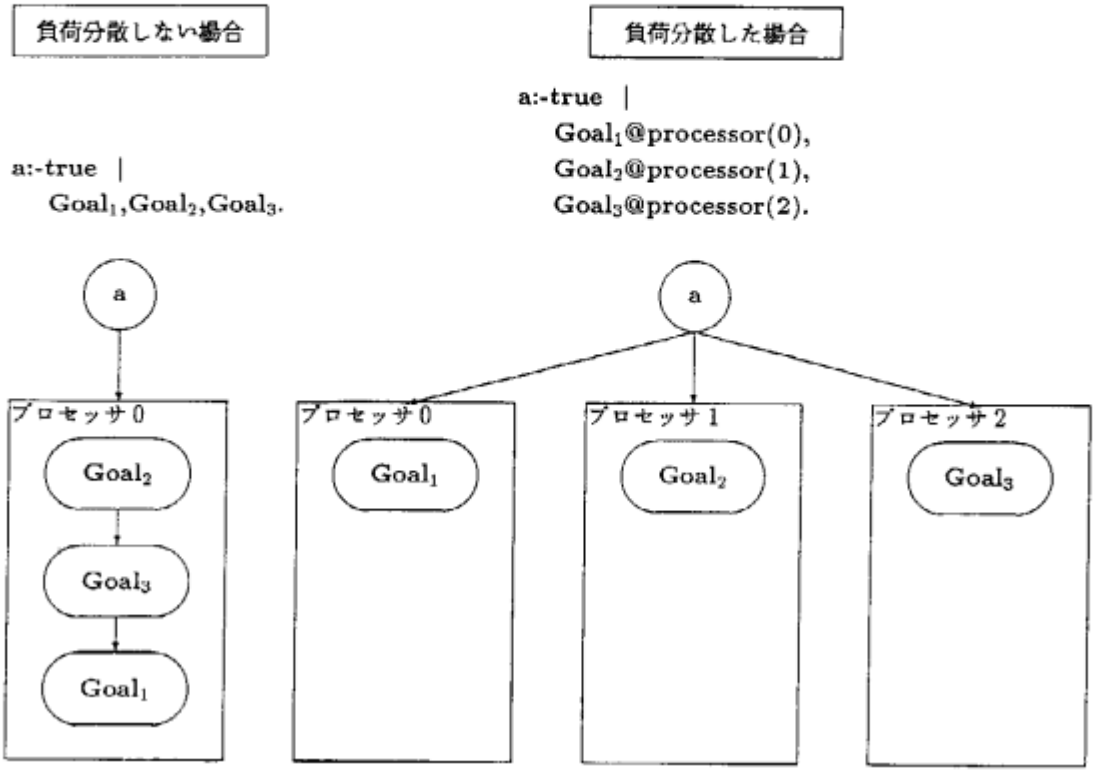


図 11.3: KL1 における負荷分散

(3) 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のが望ましい. 共通のデータベースを複数のプロセスからアクセスする場合には, データベースのあるプロセッサに通信が集中する. アクセスの多いデータベースの分割が可能な場合には, できる限り分散管理を行なって通信が集中しないようにする事.

(4) 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. 投げるゴールの実行に必要なデータは, なるべくコンパクトにまとめてられていないと通信のオーバヘッドが増大する. また, その結果もコンパクトにまとめれている必要がある.

なお, 今後の研究次第で更に留意する点が次々と出てくるであろう. ここでは, 安易に負荷分散を行なうと並列効果を期待できないという事を良く理解しておき, その基本的な考え方を示した.

11.5 負荷分散の指定方法

負荷分散の指定は, ボディの各ゴール毎にどのプロセッサで実行するかを次のようにして指定する.

goal@process(プロセッサ番号)

なお, プロセッサ番号は 0 以上の整数であり, 実行するシステムによって台数は異なる. また, システムが異なると指定方法も変わるので注意すること. この指定方法はマルチ PSI 及び *PDSS* システムのものである. そのシステムのプロセッサ台数を得るには, 次のような組込み述語を用いる.

current_processor(PE,X,Y)

なお, ここで PE はこの組込み述語が実行されたプロセッサの番号で, X は X 軸方向のプロセッサ台数, Y は Y 軸方向のプロセッサ台数である. 従って, 全体のプロセッサ台数を得るには X×Y を計算

すればよい. プロセッサ指定と同様に, この組込み述語はシステムにより異なる. この例はマルチ PSI 及び *PDSS* システムのものであり, *PDSS* の場合には PE と X と Y はいずれも 1 である. なお, 参考のために図 11.4と 11.5にそれぞれ8台プロセッサと 64 台プロセッサのマルチ PSI のシステム構成を示す.

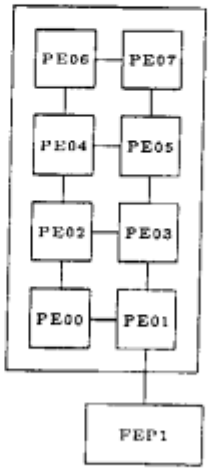


図 11.4: マルチ PSI のシステム構成 (8 プロセッサ)

では, ここで簡単な例を用いて負荷分散の具体的な指定方法を説明しよう. 次のプログラムは, ゴール a をプロセッサの 0 と 1 と 2 に投げたもので, プロセッサの番号は整数で指定する.

```
:- module foo.
:- public foo/0.
        foo:- true |                    %(1)
                a@processor(0),         %(2)
                a@processor(1),         %(3)
                a@processor(2).         %(4)
```

また, 変数を用いてプロセッサ番号を指定することもできる.

```
:- module foo.
:- public foo/3.
        foo(P0,P1,P2):- true |          %(1)
                a@processor(P0),        %(2)
                a@processor(P1),        %(3)
                a@processor(P2).        %(4)
                  実行例
        ?- foo:foo(0,1,2).              %(5)
```

なお, ゴールを別プロセッサに投げた後には, そのゴールは元と同じ優先度で実行される. 従って, 例えばプロセッサ 0 で優先度 1000 で動いていたゴールはプロセッサ 1 でも優先度 1000 で実行される. その時プロセッサ 0 では優先度 1000 以上のゴールがなかった場合には, このゴールはプロセッサ 1 では直ちに実行される. しかし, 投げた先のプロセッサ 1 では優先度 2000 のゴールが忙しく実行されているかも知れない. その場合, 投げられたゴールはすぐには実行されない. 優先度制御を行なっている場合には, このようなことが起こり得るということを覚えておこう.

なお, プロセッサの割り付け方法には色々あるが, 幾つかの代表的なものを例にして, 負荷分散制御の方法を実際のプログラムでみてみよう.

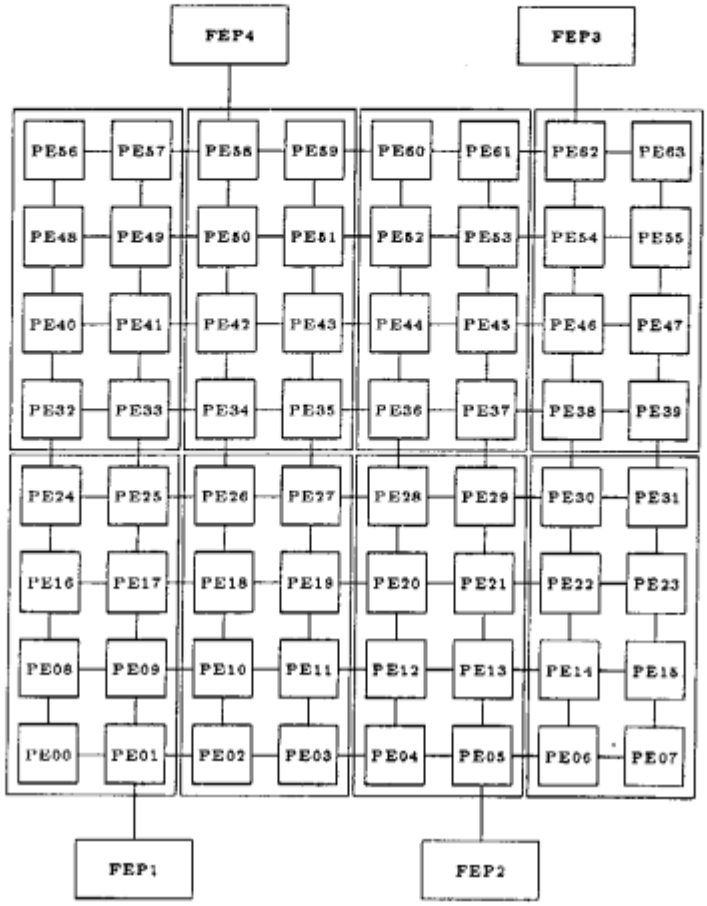


図 11.5: マルチ PSI のシステム構成 (64 プロセッサ)

サイクリック割り付け

2×2 の構成のシステム, 即ち 4 台のプロセッサにゴール spawn のプロセスを 8 個サイクリックに割り付けたプログラム例を次に示す. 生成するプロセスの個数は引数で指定し, システムのプロセッサ台数も組込み述語で調べているため, 汎用的に使える例である.

```
:- module foo.
:- public distribute/1.
        distribute(N):- true |                  %(1)
                current_processor(_,X,Y),       %(2)
                PEs := X+Y,                     %(3)
                fork(N,PEs,0).                  %(4)
        fork(0,PEs,PE):- true | true.           %(5)
        otherwise.
        fork(N,PEs,PE):- PeNo:=PE mod PEs|      %(6)
                spawn@processor(PeNo),          %(7)
                NextPE:=PE+1,                   %(8)
                N1:=N-1,                        %(9)
                fork(N1,PEs,NextPE).            %(10)
```

実行例

```
?- foo:distribute(8).                    %(11)
```

(11) のようにゴールを実行すると，spawn プロセスはプロセッサの 0, 1, 2, 3, 0, 1, 2, 3 に割り付けられる．

ランダム割り付け

プロセッサをサイクリックに割り付けると，規則性のあるプログラムでは特定のプロセッサに負荷が集中してしまう場合がある．それを回避する一つの割り付け手法がランダム割り付けである．割り付けるプロセッサの番号を乱数によって決める手法である．次に示したプログラムは spawn プロセスを 100 個擬似乱数を使ってプロセッサに割り付けた例である．

```
:- module foo.
:- public distribute/1.
       distribute(N):- true |                          %(1)
                current_processor(_,X,Y),              %(2)
                PEs := X*Y,                            %(3)
                random(Rs,23,PEs),                     %(4)
                fork(N,Rs).                            %(5)
      fork(0,Rs):- true | Rs=□.                        %(6)
   otherwise.
      fork(N,Rs):- true |                              %(7)
                Rs=[get(PeNo)|NewRs],                  %(8)
                spawn@processor(PeNo),                 %(9)
                N1:=N-1,                               %(10)
                fork(N1,NewRs).                        %(11)
      random([get(Rnd)|RS],Seed,R):- true |            %(12)
                Rnd:= Seed mod R,                      %(13)
                NewSeed:=(125*Seed+1) mod 4096,        %(14)
                random(RS,NewSeed,R).                  %(15)
      random(□,_,_):- true | true.                     %(16)

                実行例
      ?- foo:distribute(100).                          %(17)
```

(17) のようにゴールを呼び出すと，このシステムが 10×10 のシステムであれば，(8) の部分で乱数生成プロセスから 0 から 99 までの一様乱数を得，(9) の部分でプロセッサの割り付けを行なう．この例の場合では，プロセッサは 23, 76, 49, 10, 99, 32, *cdots* と割り付けられる．

暇なプロセッサへの動的割り付け

この方式は，暇なプロセッサが誰であるかを OS に尋ねて，ゴールを投げようとした時に一番暇だと思われるプロセッサに割り付ける方式である．ただし，残念ながら現在我々が使えるシステムの OS にはこのような機能はないので，本格的に動的割り付けを行なうのは今後の研究成果に期待しよう．

しかし，OS の機能を使わなくても似た機能を実現することはできる．まず，単純なカウンタゴール[3]を用意する．また，あるプロセッサには全プロセッサのカウンタ値を管理するマネージャゴールを用意し，これに最高優先度を与える．次に，全てのプロセッサにカウンタゴールを投げ，それにシステムの最低優先度を与える．即ち，各プロセッサは他に何も実行するゴールがなくなった時にこのカウンタゴールを実

[3]再帰呼び出しを使ってループした回数をカウントするゴール

行する. カウンタゴールは, 一定時間おきに自分のカウンタ値をマネージャに通知する. もし, プロセッサが忙しければ小さなカウント値がマネージャには通知される. 或いは, 忙し過ぎて通知が遅れるかも知れない. また, プロセッサが暇であれば大きなカウント値が通知される. マネージャゴールには最高優先度が与えられているので, 各プロセッサからカウント値が通知されれば直ちにそれを集める事ができる. このマネージャに対して問い合わせすれば, カウント値を元にして, どのプロセッサが暇であるかが大体わかる.

このようにして暇そうなプロセッサを見つけることは可能であるが, 一般的に並列マシンで本当に暇なプロセッサを見つけるのは難しい. 例えば, OS に尋ねた瞬間には本当に暇であっても, そのプロセッサにゴールを投げてそれが到着した時には実は大変忙しいかも知れない. このような動的な負荷分散の方式は並列計算機一般の現在の研究課題であり, 興味のある人は是非トライしてみる価値があるだろう.

11.6 負荷分散制御を応用したプログラム

本節では, 負荷分散制御を応用したプログラム例を 2 種類示す. ここで示すプログラム及び負荷分散アルゴリズムは一例である. 各自でプログラムを書き換えたり, また負荷分散アルゴリズムを変えて色々評価をしてみると大変よいと思う.

簡単な探索問題

ゴールの優先度を勉強した際に用いた簡単な探索問題プログラムに, 今度は更に負荷分散制御を応用してみよう. プログラムは先に用いたものを拡張し, 負荷分散アルゴリズムはサイクリック割り付けを用いる.

また, 負荷分散は探索木が別れる部分で行なう. ただし, あまり細かく分散すると負荷分散のためのオーバヘッドが増大するので, 探索がある程度のレベル (この例の場合は 2) に達するまでは同じプロセッサで実行し, レベルに達したところでサイクリックにプロセッサ割り付けを行なって負荷分散する. この探索木は 2 分木なので, 探索レベル 3 のところ兄弟の数は 2^3 個になり, 8 台のプロセッサに負荷分散する事になる.

なお, 負荷分散した先ではそれ以上の負荷分散は行なわず, 優先度制御のみを行なう. この例では, 先に用いた search_pri モジュールのゴールをただ呼び出しているだけである. また, プロセッサの番号を得るためのサーバプロセスを設け, 負荷分散を行なう前にはこのプロセスに対してプロセッサ番号を問い合わせるものとする.

では, まず図 11.6 に各ノードのプロセスがどのプロセッサで実行されるかを示す. また, 各プロセッサ内での優先度も示す. 図中, 各ノード及びリーフの右肩或いは左肩の数字は優先度を表わす. また, 右或いは左の数字は実行されるプロセッサを表わす. 続けてプログラムをみてみよう.

```
:- module search_load_balancing.
:- public search/4.
        search(Tree,Output,Phigh,Plow):- true |          %(1)
                current_processor(PE,X,Y),               %(2)
                PEs := X*Y,                              %(3)
                merge(R,Output),                         %(4)
                merge(Next,PeServer),                    %(5)
                fork(Tree,R,Phigh,Plow,0,Next,Cont),     %(6)
                next_pe(PeServer,PE,PEs).                %(7)
        fork(_,R,_,_,_,Next,stop):- true |R=[],Next=[]. %(8)
        alternatively.                                   %(9)
        fork(H,R,_,_,_,Next,Cont):- integer(H) |         %(10)
                R=[],Next=[].                            %(11)
        fork(here,R,_,_,_,Next,Cont):- true |            %(12)
                R=[here],Next=[],Cont=stop.              %(13)
```

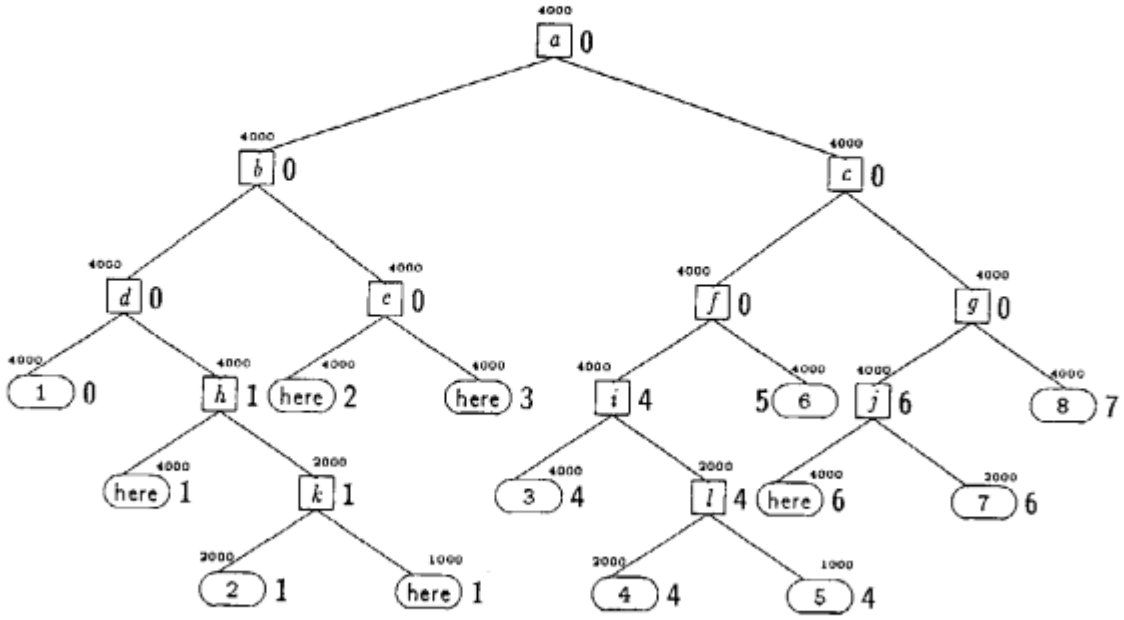


図 11.6: 探索木の各ノードプロセスの実行されるプロセッサと優先度

```
fork([TL,TR],R,Phigh,Plow,Level,Next,Cont):-        %(14)
        Level =< 1|                                 %(15)
        L1:=Level+1,                                %(16)
        R={R0,R1},                                  %(17)
        Next = {N0,N1},                             %(18)
        fork(TL,R0,Phigh,Plow,L1,N0,Cont),          %(19)
        fork(TR,R1,Phigh,Plow,L1,N1,Cont).          %(20)
fork([TL,TR],R,Phigh,Plow,Level,Next,Cont):-        %(21)
        Level =:= 2|                                %(22)
        Next = [get(PE1),get(PE2)],                 %(23)
        R={R0,R1},                                  %(24)
        search_pri:fork(TL,R0,Phigh,Plow,Cont)
                @processor(PE1),                    %(25)
        search_pri:fork(TR,R1,Phigh,Plow,Cont)
                @processor(PE2).                    %(26)
next_pe([],_,_):- true | true .                     %(27)
next_pe([get(PeNo)|Next],PE,PEs):-true |            %(28)
        PE1 := PE+1,                                %(29)
        PeNo := PE mod PEs,                         %(30)
        next_pe(Next,PE1,PEs).                      %(31)

        実行例
?- search_load_balancing:search(Tree,Out,4000,0),   %(34)
   Tree=[[[1,[here,[2,here]]],[here,here]],
         [[[3,[4,5]],6],[[here,7],8]]]|Out          %(35)
```

```
Out=[here]                                          %(36)
```

ではプログラムをみてみよう.

search このプログラムを呼び出すトップレベルの述語である.引数は先の例と全く同じである. ボディゴールでは, まず組込み述語で自分のプロセッサ番号と全体のプロセッサ台数を計算している (2), (3). また, (4) では答を返すストリームのマージャを作っている. (5) ではプロセッサ番号をサイクリックに得るためのストリームのマージャを作っている. (6) では探索を行なうプロセスを生成している. (7) ではプロセッサ番号を得るサーバプロセスを生成している.

fork クローズ (14) 探索のレベルが 1 以下の時には同じプロセッサで実行する. なお, ここで (20) の優先度を変えていないが, これは負荷分散作業を最優先で実行するためである. 即ち, 優先度を下げると探索作業と負荷分散作業が並行して行なわれ, プロセッサに投げる処理が遅れるからである.

> 負荷分散が完了するまでは,負荷分散作業を行なうゴールのプライオリティは他のゴールのプライオリティよりも低くならないようにするように注意しなければならない.

fork クローズ (21) 探索のレベルが 2 に達した時, 左右の探索を別プロセッサで実行させるようゴールを投げている (25), (26). いずれのゴールも, 実行する優先度の範囲はトップレベルのゴール呼び出しで与えられた引数をそのまま渡している.
ここで search_pri: の後にゴールを記述しているが, これは別モジュールのゴールを呼び出すための表記である. 優先度制御は, このモジュール の中で行なわれる. なお, (23) の部分で投げるべきプロセッサ番号を得ている.

next_pe プロセッサ番号をサイクリックに得るためのサーバである.

負荷分散制御をしない 8 クイーン問題

では次に, プログラミングの例題として良く用いられる 8 クイーン問題 を KL1 で書いてみよう. まずは問題の説明をかねて, 負荷分散は考えずにプログラムを書いてみよう.

8 クイーンは, 図 11.7のような 8×8 のチェスの盤面にクイーンを互いに取られないようにおく全ての答を求める全解探索問題である. なおクイーンは盤面上を縦横斜めに動く事ができる.

では, まずこの問題の解き方を考えてみよう. 盤面を表わすデータにはリストを用い, 例えば 11.7は次のようなリストで表わされるものとする.

```
[6,3,5,7,1,4,2,8]
```

このリストは, 第 1 要素が 1 行目のクイーンの位置を表わし, 第 2 要素が 2 行目の位置を表わす. クイーン問題は, 次のようなアルゴリズムで解く.

1. まず 8 行目の 8 列目にクイーンを置く事を考え, 探索を開始する.
2. 8 行目の 7 列目, 6 列目, ···, 1 列目にクイーンを置いた場合の探索も同時に開始される.
3. 次に 7 行目にクイーンを置く事を考える. それと同時に 7 列目, 6 列目, ···, 1 列目にクイーンを置いた場合の探索を行なう. まず, 取り敢えずクイーンを置いてみてそれが他のクイーンに取られないかどうか調べる.
4. 取られる事がわかれば, それは解ではないのでその先の探索は行なわない.
5. 取られない事がわかれば, 次の 6 行目にクイーンを置く事を考える.
6. 後は 2 から 5 までの処理を 1 行目にクイーンを置くまで繰り返す.
7. 1 行目の処理を終えたら, この探索は終わりである.

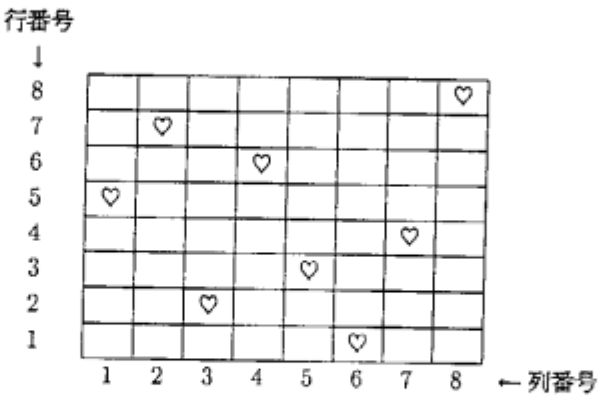


図 11.7: 8クイーン問題

8. 以上の探索処理により全ての解が求まる.

では, プログラムをみてみよう.

```
:- module queens.
:- public queens/2.
        queens(N,R) :- N>0 |                          %(1)
                merge(R0,R),                          %(2)
                queens(N,N,[],R0).                    %(3)


        queens(N,I,B,R) :- I>0 |                      %(4)
                next_queen(N,I,N,B,R).                %(5)
        queens(_,0,B,R) :- true | R=[B].              %(6)

        next_queen(N,I,J,B,R) :- J>0 |                %(7)
                R={R0,R1},                            %(8)
                try(N,I,J,B,R0),                      %(9)
                next_queen(N,I,~(J-1),B,R1).          %(10)
        next_queen(_,_,0,_,R) :- true | R=[].         %(11)

        try(N,I,J,B,R) :- true |                      %(12)
                check(B,J,1,Res),                     %(13)
                if_succeeded(N,I,J,B,R,Res).          %(14)

        if_succeeded(N,I,J,B,R,yes) :- true |         %(15)
                queens(N,~(I-1),[J|B],R).             %(16)
        if_succeeded(_,_,_,_,R,no) :- true |          %(17)
                R=[].                                 %(18)

        check([K|_],K,_,Res) :- true | Res=no.        %(18)
        check([K|_],J,D,Res) :- J=:=K+D | Res=no.     %(19)
        check([K|_],J,D,Res) :- J=:=K-D | Res=no.     %(20)
        otherwise.
```

```
check([_|B],J,D,Res) :- true |          %(21)
        check(B,J,~(D+1),Res).           %(22)
check([],_,_,Res) :- true | Res=yes.     %(23)
```

プログラムの説明を行なう.

queens(N,R) このプログラムのトップレベルの述語で, N はクイーンの数, 即ち盤面の大きさを表わす. R は解を集めるストリームで, 盤面を表わしたリストが次々と流れて来る.

ボディ部では, マージャプロセスと4引数の queen プロセスを生成する. 呼びだしゴール queens(N, N, [], R) の各引数は次のような意味である.

第 1 引数: N クイーンの数.

第 2 引数: N 最初に探索を行なう行の番号, 即ち 8 行目.

第 3 引数: [] 盤面を表わすデータで, 最初は盤面上に何も置いてない事を表わすため, 初期値 [] を与えている.

第 4 引数: R 解を集めるストリーム.

queens(N, I, B, R) N はクイーンの数である. また I はこれからクイーンを置こうとする行番号である. B は盤面を表わすデータである. I の行にクイーンを置く探索を行なうトップレベルの述語である.

ボディ部では, next_queens(N, I, N, B, R) を呼び出してその行の各列にクイーンを置く探索を行なう (5). ここで, このゴールの各引数は次のような意味である.

第 1 引数: N クイーンの数.

第 2 引数: I これから探索する行の番号.

第 3 引数: N これから探索する列の番号. 初期値は N である.

第 4 引数: B 盤面を表わすデータ.

第 5 引数: R 解を集めるストリーム.

I が 0 になった時, 即ち全ての行の探索を終えたら, この時の盤面データ B を解を集めるストリーム R に流し, このプロセスは終了する (6).

next_queens(N, I, J, B, R) N はクイーンの数である. また I と J それぞれこれからクイーンを置こうとする行番号と列番号である. I と J で示される場所にクイーンを置く述語である.

ボディ部では, その行の全ての列に置けるかどうかの探索をするため, J を 1 ずつ減算しながら再帰呼び出しをしている (10). なお, ここで (10) で ~(J-1) なる表記は KL1 のマクロであり, J-1 を計算してその値を引数に渡す, という意味である. 全ての列を探索した結果は R1 ストリームに流れて来る.

また, try を呼び出している部分ではその置き方が解であるかどうかの判定をしている. もし解であれば, その次の行以降の探索をこのゴールの中で行ない, 結果は R0 ストリームに流れて来る (9).

列の探索結果 R1 と行の探索結果 R0 はマージする (8).

列 1 の探索を終えると, ストリームを閉じてこのプロセスの実行は終了する.

try(N, I, J, B, R) 行 I と列 J にクイーンを置くのが解であるかどうか調べる. 実際に判定するのは check であり, その結果は Res にアトム yes か no にて知らされる (13). この判定結果をゴール if_succeeded の引数に渡す (14).

if_succeeded(N, I, J, B, R, YesNo) check で解かどうかの判定した結果の YesNo によって条件分岐する.

yes の場合には, 次の行の探索を行なうために I を 1 減算して queens を呼び出している (16). また, ここで盤面データに新たに列番号 J を付け加える.

no の場合は, これは解ではないと言う意味であるからストリームを閉じてプロセスの実行を終了する (17).

check(B, J, D, YesNo) 列 J にクイーンを置けるかどうかをチェックする. B はそれまでにクイーンを置いた盤面を表わすデータである. D は斜め方向のチェックをするために用いるもので, (14) で初期値は 1 に設定されている.

クローズ (18) では同じ列であるかどうかを調べる. クローズ (19), (20) では隣り合うものがあるかどうか, 及び斜めに位置するものがあるかどうかを調べる. クローズ (21) では斜めかどうかを調べるために D をパラメタとして再帰呼びだししている.

クローズ (22) では, その位置にクイーンが置けるので, YesNo に yes をバインドしている. それ以外の置けなかった場合には no をバインドしている.

負荷分散制御を行なった 8 クイーン問題

では次に, 先に紹介した 8 クイーンプログラムに負荷分散制御機能を付けてみよう. まず, このプログラムのどの部分が並列に実行可能かを考えてみる. プログラム中の並列性を見つけるには, プロセスを多く生成している部分にまず着目する.

まず next_queen の定義 (7) から (11) までをみてみよう. この述語は再帰呼び出しを行なって, 一つの列に対して 8 つの next_queen プロセスを生成している. それぞれのプロセスは互いに独立であり, この部分は並列実行できる.

次に, if_succeeded の定義 (15) から (18) までをみてみよう.
この部分は, 1 つの行に対して全ての列を探索している部分で, queens のプロセスを生成している. それぞれのプロセスは互いに独立である. しかし, このプロセスを別プロセッサで実行させると, プロセスの処理は大変小さなものになってしまう.

これ以外のプロセスは全て処理が小さいもので, (10) の next_queen を別プロセッサに投げるように負荷分散してみよう. この負荷分散を簡単に説明すると, 次のようにまとめられる.

- 8 行目の各列にクイーンを置いた盤面を 8 つ作る.
- 8 つの盤面毎に別プロセッサで探索を行なう.
- 7 行目の各列にクイーンを置いた盤面を 8 つ作る.
- 8 つの盤面毎に別プロセッサで探索を行なう.
- 以下はこの繰り返し.

しかし, この負荷分散を次々に行なうと, 探索のレベルが深くなるにつれて個々のプロセッサが担当する仕事が小さくなり過ぎる. 従って, ある程度の探索レベルに達したら負荷分散を行なわないようにしなければならない. プロセッサの割り付け方法は, サイクリックに行なう.

例えば, 探索レベル 1 まで負荷分散した時のプロセッサ割り付けを図 11.8 に示す. 図をみればわかるように, 8 行目にクイーンを置く時の 8 通りの置き方についての探索ゴールをそれぞれ別プロセッサに投げ, その後は同じプロセッサで実行していることがわかる.

次に, 探索レベル 2 まで負荷分散した時のプロセッサ割り付けを図 11.9 に示す. 探索レベル 2 に達した時に隣のプロセッサに投げられるように, 探索レベル 1 でプロセッサにゴールを投げる時には, その下の探索木が何台のプロセッサを必要とするか予測している. この場合は各探索レベルで 8 台ずつプロセッサを必要とするので, 8^{Depth} を計算すれば予測できる.

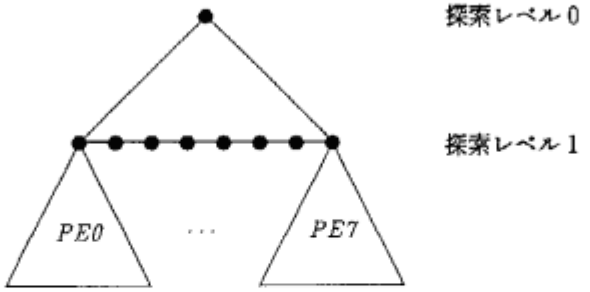


図 11.8: 探索レベル 1 まで負荷分散する時のプロセッサ割り付け

次に示すプログラムでは, 使用するプロセッサの台数をパラメタで与える. また, 負荷分散を行なう探索レベルもパラメタで指定する. このプログラムについては詳しい説明は行なわないが, プログラム中のコメントを参考にして, 負荷分散を行なう部分とそれを打ち切るメカニズムに着目して各自読んでみる事.

```
:- module queens.
:- public queens/4.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% queens(N,R,PE,D)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% トップレベルの呼びだし述語.
% ガード部では引数のエラーチェックをしている.
% ボディ部の check_pes では入力されたプロセッサ台数を
%       実際のシステムの台数と比較してチェックしている.
% N:    クイーンの数
% R:    解を集めるストリーム
% PE:   負荷分散するプロセッサの台数
% D:    負荷分散を打ち切る探索の深さ
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
queens(N,R,PE,D) :- N>0,PE>0,D>=0 |                    %(1)
        check_pes(PE,PE1),                              %(2)
        merge(R0,R),                                    %(3)
        queens(N,N,[],R0,D,0,PE1).                      %(4)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% queens(N,I,B,R,D,Pi,P)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 各行にクイーンを置くための探索を行なう述語
% D が 0 より大きい時にはプロセッサを投げるためのゴールを呼ぶ.
% D が 0 以上の時には, 同じプロセッサで実行するためのゴールを呼ぶ.
% N:    クイーンの数
% I:    行番号
% B:    盤面データ
% R:    解を集めるストリーム
% D:    負荷分散を打ち切る探索の深さ
```

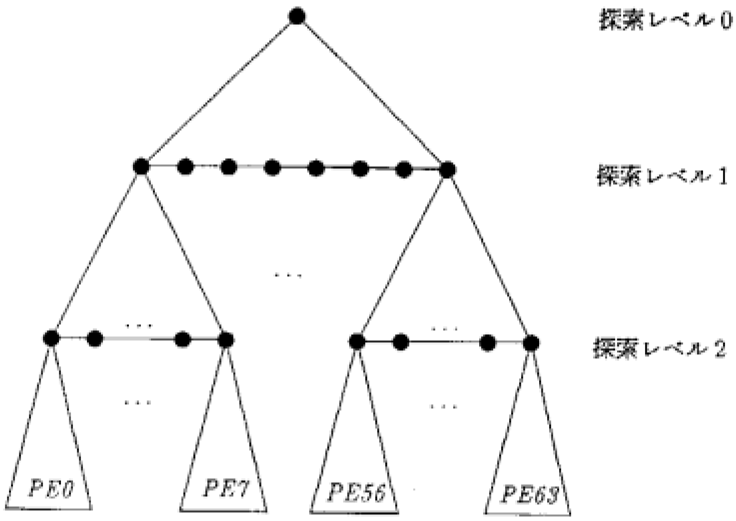


図 11.9: 探索レベル 2 まで負荷分散する時のプロセッサ割り付け

```
% Pi:    ゴールを投げるプロセッサ番号（初期値は 0)
% P:     負荷分散を行なうプロセッサ台数
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

queens(N,I,B,R,D,Pi,P) :- I>0,D>0 |                  %(5)
        next_queen_ext(N,I,N,B,R,D,Pi,P).             %(6)
queens(N,I,B,R,D,_,_) :- I>0,D=<0 |                   %(7)
        next_queen(N,I,N,B,R).                        %(8)
queens(_,0,B,R,_,_,_) :- true | R=[B].                %(9)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% next_queen_ext(N,I,J,B,R,D,Pi,P)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 各列にクイーンを置くための探索を行なう述語
% 負荷分散を行なっている間に呼び出され，(15) でゴールを投げる．
% 投げるプロセッサ番号は next_pe にて得る．
% J:    列番号
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

next_queen_ext(N,I,J,B,R,D,Pi,P) :- J>0 |            %(10)
        next_pe(D,N,1,Pi,P,Pi1),                      %(11)
        R={R1,R2},                                    %(12)
        try(N,I,J,B,R1,D,Pi,P),                       %(13)
        next_queen_ext(N,I,~(J-1),B,R2,D,Pi1,P) %(14)
                              @processor(Pi1).%(15)
next_queen_ext(_,_,0,_,R,_,_,_) :- true | R=[]. %(16)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% next_queen(N,I,J,B,R)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 各列にクイーンを置くための探索を行なう述語
% 負荷分散を打ち切った後に呼び出される.
% J:    列番号
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

next_queen(N,I,J,B,R) :- J>0 |                          %(17)
        R={RO,R1},                                      %(18)
        try(N,I,J,B,RO,0,_,_),                          %(19)
        next_queen(N,I,~(J-1),B,R1).                    %(20)
next_queen(_,_,0,_,R) :- true | R=[].                   %(21)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% try(N,I,J,B,R,D,Pi,P)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% その場所にクイーンを置く事ができるか調べ，置けたら次の行を調べる
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

try(N,I,J,B,R,D,Pi,P) :- true |                         %(22)
        check(B,J,1,Res),                               %(23)
        if_succeeded(N,I,J,B,R,D,Pi,P,Res).             %(24)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% if_succeeded(N,I,J,B,R,D,Pi,P,YesNo)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% その場所にクイーンを置けるか否かによって条件分岐する述語
% 置けたら次の行を調べる.
% YesNo:yes ならば置ける，no ならば置けない事を表わす
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if_succeeded(N,I,J,B,R,D,Pi,P,yes) :- true |            %(25)
        queens(N,~(I-1),[J|B],R,~(D-1),Pi,P).           %(26)
if_succeeded(_,_,_,_,R,_,_,_,no) :- true | R=[].%(27)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% check(B,J,D,YesNo)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% その場所にクイーンを置けるか否かをチェックする述語
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

check([K|_],K,_,Res) :- true | Res=no.                 %(28)
check([K|_],J,D,Res) :- J=:=K+D | Res=no.               %(29)
check([K|_],J,D,Res) :- J=:=K-D | Res=no.               %(30)
otherwise.                                              %(31)
check([_|B],J,D,Res) :- true |                          %(32)
```

```
        check(B,J,~(D+1),Res).                    %(33)
check([],_,_,Res) :- true | Res=yes.              %(34)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% check_pes(N,PE)/check_pes(N,PE,NO)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 入力されたプロセッサの台数がシステムのプロセッサの台数以下で
% あるかどうかをチェックする．もし大きかったら，システムのプロ
% セッサ台数を使用するプロセッサ台数とする．
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

check_pes(N,PE) :- true |                         %(35)
        current_processor(_,X,Y),                 %(36)
        check_pes(N,PE,~(X*Y)).                   %(37)
check_pes(N,PE,NO) :- N=<NO | PE:=N.              %(38)
check_pes(N,PE,NO) :- N>NO | PE:=NO.              %(39)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% next_pe(D,N,M,Pi,P,Pi1)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 次に負荷分散を行なうプロセッサの番号を得る述語
% ８のＤ乗を計算して自分より下の探索で使用するプロセッサの台数を
% 計算する．
% D:    探索の深さ
% N:    クイーンの数
% M:    ある探索の深さにおける枝の数
% Pi:   前にゴールを投げたプロセッサの番号
% P:    使用するプロセッサの台数
% Pi1:  次にゴールを投げるプロセッサの番号
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

next_pe(D,N,M,Pi,P,Pi1) :- D>1 |                  %(40)
        next_pe(~(D-1),N,~(M*N),Pi,P,Pi1).        %(41)
next_pe(1,_,M,Pi,P,Pi1) :- true |                 %(42)
        Pi1:=(Pi+M) mod P.                        %(43)
```

第 12 章

初級編練習問題

この章では、初級編で勉強した事の復習をするための練習問題の説明を行なう。解答は次章に載せるが，なるべく各自でプログラムを作成して PDSS 等で動作を確認する事.

12.1 ストリング操作

PDSS の組込み述語である substring/5, set_substring/4 を書け.

12.2 ベクタ操作

行列をベクタで表現した場合の，行列の積を求めるプログラムを書け.

12.3 n 次元配列

ベクタを用い n 次元配列を作り，その要素を参照・更新するプログラムを書け.

12.4 探索問題をベクタを使って書いてみよう

第 5 章で用いた探索問題では，探索木をリストで表わしていた．これをベクタを用いて書いてみよう．また，別の負荷分散を考えてみよ.

12.5 8 クイーンを高速化してみよう

第 5 章で用いた 8 クイーンは最も単純な解き方であまり最適化を考えていない．例えば，8 行目の 8 列目にクイーンを置けば，7 行目の 8 列目にはクイーンを置けないことは明らかである．また，7 列目に置けないのも明らかである．このように，簡単な枝刈りを行なうだけで効率は上がる筈である．できれば段階別に最適化を行ない，それぞれの実行時間の計測と評価をせよ.

12.6 8 クイーンの負荷分散方式を変えてみよう

第 5 章で用いた 8 クイーンは，ある探索レベル 0 でゴールを別プロセッサに投げていた．これを，ある探索レベルまでは同じプロセッサで実行し，その探索レベルに達したら負荷分散を行ない，その後は同じプロセッサで実行するようにせよ．また，できたら他にも負荷分散方式を考えてみよ.

第 13 章

初級編練習問題の解答

13.1 ストリング操作

PDSS の組込み述語である substring/5,set_substring/4 を書け.

substring(String, Position, Length, ^ SubString, ^ NewString)

String が未定義なら中断. ストリング以外ならば例外. Position が未定義な らば中断. 非負整数以外または String の要素以上ならば例外. Length が未定義ならば中断. 正整数以外または Position + Length が要素を越えていれば例外. それ以外の場合には String の Position 番目から長さ Length 分をコピーし新たなストリングを生成し, SubString が未定義なら中断. String と同じタイプのストリング以外なら例外. また, Position+(SubString の長さ) が String の要素数を越えていれば例外. それ以外の場合には String の Position 番目からを SubString で示されたストリングで置き換えたストリングを生成し NewString とユニファイする.

set_substring(String, Position, SubString, ^ NewString)

String が未定義なら中断. ストリング以外ならば例外. Position が未定義ならば中断. 非負整数以外または String の要素以上ならば例外. SubString が未定義ならば中断. String と同じタイプのストリング以外なら例外. また, Position+(SubString の長さ) が String の要素数を越えていれば例外. それ以外の場合には String の Position 番目から長さ Length 分をコピーし新たなストリングを生成し, SubString が未定義なら中断. また, Position+(SubString の長さ) が String の要素数を越えていれば例外. それ以外の場合には String の Position 番目からを SubString で示されたストリングで置き換えたストリングを生成し NewString とユニファイする.

プログラム例

```
:- module my_string.
:- public my_substring/5, my_set_substring/4.

    my_substring(OldString, Position, Length, SubString, ValString):-
        string(OldString, OldLength, ElementSize),
        integer(Position),
        integer(Length),
        0 =< Position,
        Position < OldLength,
        0 =< Length,
```

```
        Position + Length =< OldLength |
        new_string(NewString, Length, ElementSize),
        copy_substring(OldString, Position, NewString, 0,
                                Length, ValString, SubString).


    my_set_substring(OldString, Position, SubString, NewString):-
        string(OldString, OldLength, ElementSize),
        string(SubString, SubLength, ElementSize),
        integer(Position) ,
        0 =< Position,
        Position < OldLength,
        Position + SubLength =< OldLength |
        copy_substring(SubString, 0, OldString, Position,
                                SubLength, _, NewString).

%ストリングを後ろからコピーする
    copy_substring( OldString1, Position1, OldString2, Position2,
                                CopyLength, ValString1, ValString2):-
        CopyLength > 0|
        NewLength2 := CopyLength - 1,
        string_element(OldString1, ~(Position1 + NewLength2),
                                            Element, NewString1),
        set_string_element(OldString2, ~(Position2 + NewLength2),
                                            Element, NewString2),
        copy_substring( NewString1, Position1,NewString2, Position2,
                                NewLength2, ValString1, ValString2).


    copy_substring( OldString1, Position1, OldString2, Position2,
                                CopyLength, ValString1, ValString2):-
        CopyLength =:= 0|
        ValString1 = OldString1,
        ValString2 = OldString2.
```

実行例

```
| ?- my_string:my_substring("abcdef",2,3,SubString, NewString)|SubString,NewString.
SubString = "cde"
NewString = "abcdef"

yes.
| ?- my_string:my_set_substring("abcdefg",2,"hij",New)|New.
New = "abhijfg"

yes.
```

13.2 ベクタ操作

行列をベクタ表現した場合の、行列の積を求めるプログラムを書け.

```
mul(OldA, OldB, ^ NewA, ^ NewB, ^ Val)
```

OldA	行列 A を表わすベクタ.
OldB	行列 B を表わすベクタ.
NewA	OldA とユニファイしたもの.
NewB	OldB とユニファイしたもの.
Val	A*B 積を表わすベクタ.

プログラム例

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ b_{11} & b_{12} & b_{13} \end{pmatrix} \Longrightarrow \{\{a_{11}, a_{12}, a_{13}\}, \{b_{11}, b_{12}, b_{13}\}\}$$

のようにベクタを用いて行列を表わすことにする.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} \end{pmatrix}$$

の場合は次のような表現になる。

$$\{\{a_{11}, a_{12}, a_{13}\}, \{a_{21}, a_{22}, a_{23}\}\} \times \{\{b_{11}, b_{12}\}, \{b_{21}, b_{22}\}, \{b_{31}, b_{32}\}\}$$

$$= \{\{a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31}, a_{11}b_{12}+a_{12}b_{22}+a_{13}b_{32}\}, \{a_{21}b_{11}+a_{22}b_{21}+a_{23}b_{31}, a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32}\}\}$$

- メモリ効率を優先した解答例

```
:- module matrix.
:- public mul/5.

    mul(OldA, OldB, NewA, NewB, NewVal):-
        vector(OldA, NRowA),
        vector(OldB, NRowB),
        vector_element(OldA, 0, RowA),
        vector_element(OldB, 0, RowB),
        vector(RowA, NColA),
        vector(RowB, NColB),
NColA = NRowB |
        new_vector(OldVal, NRowA),
        newmatrix(NRowA, NColB, OldVal, Val), % 新しい行列
        multi(NRowA, NColB, NRowB,  OldA, OldB, Val, NewA, NewB, NewVal).

% 新しい行列 (ベクタ) を作る。
    newmatrix(NRowA,NColB, OldVal, NewVal):-
        NRowA > 0|
        NNewRowA := NRowA - 1,
        new_vector(Element, NColB),
        set_vector_element(OldVal, NNewRowA, _, Element, Val),
```

```
        newmatrix(NNewRowA, NColB, Val, NewVal).

    newmatrix(NRowA,NColB, OldVal, NewVal):-
        NRowA =:= 0|
        OldVal = NewVal.


% 列の計算ループ
    multi(0, NColB, NRowB, OldA, OldB, OldVal, NewA, NewB, NewVal):-
        true|
        OldA = NewA,
        OldB = NewB,
        OldVal = NewVal.


    multi(NRowA,NColB, NRowB, OldA, OldB, OldVal, NewA, NewB, NewVal):-
        NRowA > 0|
        NNewRowA := NRowA - 1,
        NNewColB := NColB - 1,
        NNewRowB := NRowB - 1,
        set_vector_element(OldA, NNewRowA, OldColA, NewColA, A),
        multiply_col(NNewColB, NNewRowB, OldColA, NewColA, OldB, B, OldV,NewV),
        set_vector_element(OldVal, NNewRowA, OldV, NewV, Val),
        multi(NNewRowA, NColB, NRowB, A, B, Val, NewA, NewB, NewVal).


% 行の計算ループ
    multiply_col(NColB, NRowB, OldColA, NewColA, OldB, NewB, OldV, NewV):-
        NColB >= 0|
        multiply_col(~(NColB-1), NRowB, ColA, NewColA,B, NewB, V, NewV),
        multiply_element(NRowB, NColB, OldColA, ColA, OldB, B, 0, NVal),
        set_vector_element(OldV, NColB, _, NVal, V).

    multiply_col(NColB, NRowB, OldColA, NewColA, OldB, NewB, OldV, NewV):-
        NColB < 0|
        OldColA = NewColA,
        OldB = NewB,
        OldV = NewV.

%計算して和を求めるループ
    multiply_element(NRowB,NColB,OldColA,NewColA,OldB,NewB, NOldVal, NNewVal):-
        NRowB >= 0|
        multiply_element(~(NRowB-1), NColB,ColA, NewColA, B, NewB,
                                    ~(NOldVal+ElementA*ElementB), NNewVal),
        vector_element(OldColA, NRowB, ElementA, ColA),
        set_vector_element(OldB, NRowB, ColB, NewColB, B),
        vector_element(ColB, NColB,  ElementB, NewColB).

    multiply_element(NRowB,NColB,OldColA,NewColA, OldB, NewB,NOldVal,NNewVal):-
```

```
        NRowB < 0|
        OldColA = NewColA,
        OldB = NewB,
        NOldVal = NNewVal.
```

- 並列性を優先した解答例

```
:- module matrix.
:- public mul/3.

    mul(OldA, OldB, NewVal):-
        vector(OldA, NRowA),
        vector(OldB, NRowB),
        vector_element(OldA, 0, RowA),
        vector_element(OldB, 0, RowB),
        vector(RowA, NRowB),
        vector(RowB, NColB)|
        new_vector(OldVal, NRowA),
        newmatrix(NRowA, NColB, OldVal, Val),
        multi(NRowA, NColB, NRowB,  OldA, OldB, Val, NewVal).

    newmatrix(NRowA,NColB, OldVal, NewVal):-
        NRowA > 0|
        NNewRowA := NRowA - 1,
        new_vector(Element, NColB),
        set_vector_element(OldVal, NNewRowA, _, Element, Val),
        newmatrix(NNewRowA, NColB, Val, NewVal).

    newmatrix(NRowA,NColB, OldVal, NewVal):-
        NRowA =:= 0|
        OldVal = NewVal.


    multi(0, NColB, NRowB, OldA, OldB, OldVal, NewVal):-
        true|
        OldVal = NewVal.


    multi(NRowA,NColB, NRowB, OldA, OldB, OldVal, NewVal):-
        NRowA > 0|
        NNewRowA := NRowA - 1,
        NNewColB := NColB - 1,
        NNewRowB := NRowB - 1,
        vector_element(OldA, NNewRowA, OldColA, A),
        multiply_col(NNewColB, NNewRowB, OldColA, NewColA, OldB,OldV, NewV),
        set_vector_element(OldVal, NNewRowA, OldV, NewV, Val),
        multi(NNewRowA, NColB, NRowB, OldA, OldB, Val, NewVal).
```

```
multiply_col(NColB, NRowB, OldColA, NewColA, OldB, OldV, NewV):-
    NColB >= 0|
    multiply_col(~(NColB-1), NRowB, ColA, NewColA,OldB, V, NewV),
    multiply_element(NRowB, NColB, OldColA, ColA, OldB, 0, NVal),
    set_vector_element(OldV, NColB, _, NVal, V).

multiply_col(NColB, NRowB, OldColA, NewColA, OldB, OldV, NewV):-
    NColB < 0|
    OldColA = NewColA,
    OldV = NewV.

multiply_element(NRowB, NColB, OldColA, NewColA, OldB, NOldVal, NNewVal):-
    NRowB >= 0|
    multiply_element(~(NRowB-1), NColB,ColA, NewColA, OldB,
                                ~(NOldVal+ElementA*ElementB), NNewVal),
    vector_element(OldColA, NRowB, ElementA, ColA),
    set_vector_element(OldB, NRowB, ColB, NewColB, _),
    vector_element(ColB, NColB,  ElementB, NewColB).

multiply_element(NRowB, NColB, OldColA, NewColA, OldB, NOldVal, NNewVal):-
    NRowB < 0|
    OldColA = NewColA,
    NOldVal = NNewVal.
```

実行例

```
| ?- matrix:mul({{1,2,3},{4,5,6}},{{1,2},{3,4},{5,6}},A,B,V)|all.
A = {{1,2,3},{4,5,6}}
B = {{1,2},{3,4},{5,6}}
V = {{22,28},{49,64}}

yes.
| ?-
```

13.3 n 次元配列

ベクタを用い n 次元配列を作り，その要素を参照・更新するプログラムを書け．

```
dim:create(Stream, Def, Status)
```

Def で ([100,20,30] のように) リストを用いて定義した n 次元配列を作成し，その配列に繋がるコマンドストリームを Stream とユニファイする。Status には以下のものがユニファイされる．

normal	成功
abnormal	配列宣言が異常

Stream へ送るメッセージの種類は以下のとおり．

```
set_array_element(Position, ^ OldElement, NewElement)
```

n 次元配列の Position 位置の内容を OldElement とユニファイし、NewElement と置き換える.

```
array_element(Position, Element)
```

n 次元配列の Position 位置の内容を Element とユニファイする.

プログラム例ここでは, n 次元の配列を 1 次元のベクタで表わす。配列をアクセスする時に配列位置を計算するために宣言した配列の大きさを持ち回る。

```
:- module dim.
:- public create/3.

    create(Stream, Def, Status):-
        list(Def)|
        Status = normal,
        arraysize(Def, 1, N, NewDef),
        new_vector(Array, N),
        eval(Stream, Array, NewDef).

otherwise.

    create(Stream, Def, Status):-
        true|
        Status = abnormal.

    arraysize([Car|Cdr], Old, New, NewDef):-
        true|
        arraysize(Cdr, Old, N, Def),
        New := N * Car,
        NewDef = [N|Def].

    arraysize([], Old, New, NewDef):-
        true|
        Old = New,
        NewDef = [].

    eval( [], _, _):-
        true|
        true.

    eval([set_array_element(Position, OldElement, NewElement)|T], Array, Def):-
        true|
        set_array_element(Array, Position, Def, OldElement, NewElement, NewArray),
        eval(T, NewArray, Def).

    eval([array_element(Position, Element)|T], Array, Def):-
        true|
        array_element(Array, Position, Def, Element, NewArray),
```

```
        eval(T, NewArray, Def).

    set_array_element(OldArray, Position,Def, OldElement, NewElement, NewArray):-
        list(Position)|
        position(Position, Def, P),
        set_vector_element(OldArray, P, OldElement, NewElement, NewArray).

    array_element(OldArray, Position, Def, OldElement, NewArray):-
        list(Position)|
        position(Position, Def, P),
        vector_element(OldArray, P, OldElement, NewArray).

    position([Car],_,P):-
        true| P = Car.

otherwise.

    position([Car|Cdr], [DefCar|DefCdr], NewP):-
        true|
        OldP := Car*DefCar,
        position(Cdr, DefCdr, P),
        NewP := OldP + P.
```

実行例

```
| ?- dim:create(Stream,[5,5,5], Status),
    Stream = [set_array_element([1,1,1],O1,a),
              set_array_element([2,2,2],O2,b),
              set_array_element([3,3,3],O3,c),
              set_array_element([1,1,1],A1,d),
              set_array_element([2,2,2],A2,e),
              array_element([1,1,1],B1)]|all.

Stream = [set_array_element([1,1,1],0,a),
          set_array_element([2,2,2],0,b),
          set_array_element([3,3,3],0,c),
          set_array_element([1,1,1],a,d),
          set_array_element([2,2,2],b,e),
          array_element([1,1,1],d)]
Status = normal
O1 = 0
O2 = 0
O3 = 0
A1 = a
A2 = b
B1 = d
```

```
yes.
| ?-
```

第 III 部

中級編

第 14 章

荘園機能とその使い方

荘園機能はオペレーティングシステム等システムプログラムを記述するために導入した機能であり，ひとかたまりの計算の実行，即ちあるゴールとそのサブゴール群の実行を，その詳細を知る事なく制御するための機能である．制御する仕事の単位としては，例えば，OS の下で動くユーザプログラム，或いはシェルの下で動く一つのジョブであり，小さな規模の仕事を制御するために一般のユーザが利用するための機能ではない．しかし，その機能を良く理解した上で注意して利用する限りにおいては大変便利な機能であるため，本章では荘園機能の概略と注意点について説明する．

14.1 機能の概略

荘園は KL1 言語で規定されている実行の制御，資源管理及び例外処理を行なうための最小単位である．荘園の本来の意味は，奈良時代から室町時代にかけて貴族や社寺の保有する私有地のことである．荘園の領主，即ち地主は小作達に土地を分け与え，小作達は限られた土地を利用して農耕を営み，そして年貢を領主に納めた．領主の権力は偉大であり，小作達は全て領主の統制下にあった．しかし，小作達は土地が足らなくなると領主に直訴し，荘園に余裕があれば新たな土地が与えられた．また小作にトラブルが生じた時には，領主がそれを解決した．

KL1 言語における荘園もこれとほとんど同じで，一固まりの KL1 プログラムを実行する環境を荘園，KL1 プログラムを小作，例外事象をトラブルと読み換えると理解しやすいであろう．例えば，新たに荘園を生成する際には，実行する際に消費しても良い資源として計算時間とメモリ領域の上限値を与える．その荘園の中で KL1 のゴールを実行させたとき、そのゴールから発生した全てのゴールは同じ荘園の中で実行される．また，荘園の中に更に荘園を作ることもでき，それを子荘園と呼ぶ．このように，一般に荘園は階層構造をなす．

ここで，例えば子荘園が与えられた資源を使い切ったとしよう．すると，その情報は荘園に報告され，もし資源に余裕があれば自分の資源を子荘園に分け与える．また，子荘園の実行を一時的に中断したり再開したり，または強制的に終了したりする場合に荘園機能を用いる．更に，荘園内のゴールで例外事象が発生した際には荘園に報告され，あらかじめ荘園に定義した例外処理を行なう．

荘園には，荘園内の実行を制御するためのコマンドを流す制御ストリームと，荘園内の状態が報告される報告ストリームがつながっている．図 14.1 に階層構造をした荘園のイメージを示す．以下に荘園機能とその使い方の概要を示すが，詳細な使い方に関しては [2] 或いは [3] を参照のこと．

14.1.1 実行制御

荘園内のゴールの実行 (以下単に『荘園の実行』と呼ぶ．) の起動 / 停止 / 再起動 / 実行放棄を制御する．これらの制御は制御ストリームにコマンドを流すことによって行なう (図 14.2)．ただし並列処理を効率良く行なうため，これらの操作は必ずしも即座に行なわれるとは限らない．

また，荘園内のゴールの実行が全て終了した時には終了メッセージが報告ストリームに流される．

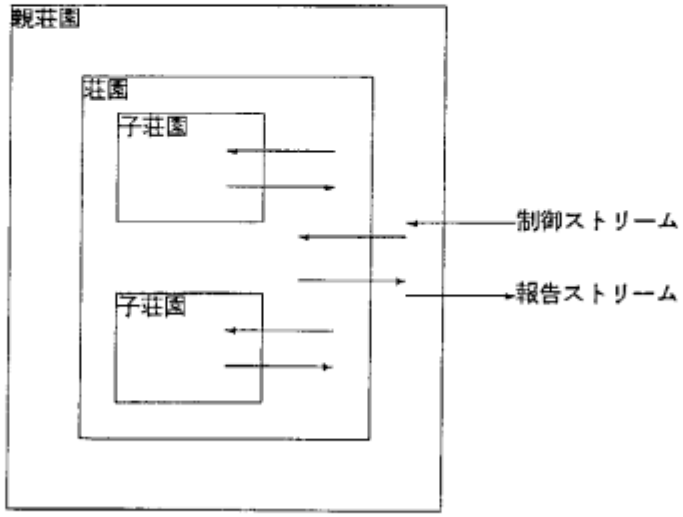


図 14.1: 荘園の階層構造

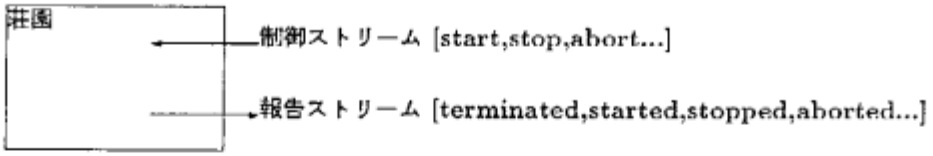


図 14.2: 実行制御

14.1.2 資源消費制御

荘園の実行で消費する計算時間やメモリ領域などの計算機資源 (以下単に『資源』という) を制御する. 消費資源の上限の設定, 設定値の積み増し等の制御は, 制御ストリームにコマンドを流すことによって行なう (図 14.3).

また, 資源消費量が設定した上限に近づくと報告ストリームに資源不足メッセージが流される. なお, 実際に上限を越えると実行を中断する. 制御ストリームを通じて消費資源の上限を増やしてやれば, 荘園の実行は再開する.

また, 制御ストリームに資源消費状況の問い合わせメッセージを流すと, 報告ストリームを通じて資源消費状況が報告される.

なお, マルチ PSI で管理されている資源消費量はリダクション数に対応するものだけである. 将来, 計算時間, メモリ消費量, プロセサ間の通信量なども管理するように改良する可能性がある.

14.1.3 例外処理

荘園の実行中に生じた例外事象の処理を行う. 例外事象には組込み述語の引数のタイプエラー, デッドロック, 実行の失敗 (ゴールに対応する候補節がない場合や, ユニフィケーションの失敗) 等がある. こ

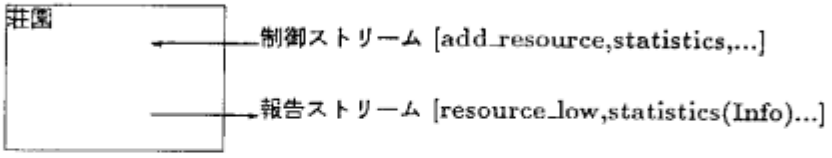


図 14.3: 資源消費制御

れらの例外事象は報告ストリームを通じて報告される (図 14.4).

例えば, divide(2, 0, X) を実行してゼロ除算エラーが発生した場合には, まず報告ストリームにエラーの内容とエラーを起こしたゴールが報告される. その際に, 例外を発生したゴールの代わりに実行するゴールを返す変数 NewGoal を渡す. 例外処理ハンドラは NewGoal を例えば divide(2, 1, X) に決めてやると, エラーを起こしたゴールのかわりに NewGoal が実行される.

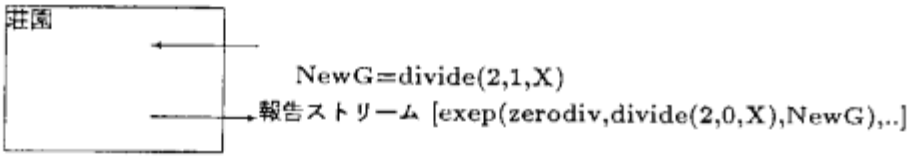


図 14.4: 例外処理

14.2 使い方の注意

先に, 荘園機能は一般のユーザがあまり安易に利用するべきものではないと説明した. というのも, 一般に荘園の外部が荘園の内部と共有変数を持っていると, その使い方は大変難しくなるからである. また, 荘園を作るためのオーバヘッドは大きく, 多用すると非効率的なプログラムになる. 本節では, これら荘園機能を利用する際に注意せねばならない点を, いくつかの例を用いて説明する.

なお, ここでは荘園の使い方をごく簡単な説明にとどめるので, 実際に使う場合にはマニュアル [2], [3] を参照のこと.

14.2.1 失敗するかも知れないユニフィケーション

荘園機能の例外処理機能を, 次のような目的で用いられないかを考えてみよう.

> 『失敗するかも知れないユニフィケーションを荘園の中に入れてしまえば, 失敗するかどうかを荘園を監視していればわかるであろう.』

成功するか失敗するかわからないユニフィケーションがあった場合, これを単純に荘園の例外処理機能を用いて実現するのは危険である. 例えば, 次のようなゴールの実行を考えてみよう. なお, この例で shoen(X = Y) は荘園の中で変数 X と Y のユニフィケーションを行なうことを模式的に示したものである.

```
?- X = a,  Y = b,  shoen( X = Y ).
```

ここで, ユーザの意図するところをまとめてみよう.

- 荘園の外で X と Y の値がそれぞれ a と b に決まる
- 荘園の中で X と Y のユニフィケーションが実行される.
- 実行の失敗を示す例外事象が荘園に報告される.
- 荘園の報告ストリームを監視していれば, ユニフィケーションが成功したか失敗したかがわかる (図 14.5).

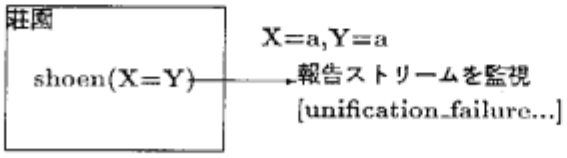


図 14.5: 失敗するかも知れないユニフィケーションを監視

ところが, KL1 ではゴールの実行順序は規定されてないため, 実際には次のように動作するかも知れない.

- 荘園の中で X と Y のユニフィケーションが実行される.
- 荘園の外で X と a のユニフィケーションが実行され, 次に Y と b のユニフィケーションが実行されたところでユニフィケーションが失敗し, 全体の実行が失敗する.

このように, 荘園の外部と内部とで共有変数を持っている場合には, 本来ならば荘園の中のみで発生して欲しい例外事象が, 荘園の外に波及してしまう恐れがある. これがユーザプログラムの荘園の中だけで起こる問題であれば, 全体が失敗する程度でデバッグが難しくなるぐらいの被害ですむ. しかし, オペレーティングシステムの荘園とユーザプログラムの荘園が共有変数を持っていると, ユーザプログラム中のユニフィケーションの失敗が, オペレーティングシステム全体の失敗につながるということがあり, 大変危険である. 従って, PIMOS では保護フィルタと呼ばれる機能を用いてオペレーティングシステムは絶対に失敗しないように作られている. なお, 保護フィルタについては後ほど述べる.

では, 先ほどのゴールの実行をユーザが意図した通りに動くようにするための方法を考えてみよう. 先ほどの問題点は, 荘園の外部で失敗した点であった. 従って, 荘園の内部で失敗するようにするには, 次のように簡単な同期メカニズムを入れてやれば良い. なお, ここでは説明を簡単にするため X と Y はいずれもアトムデータにしか決まらないものとしておく.

プログラム

```
my_unify(X, Y):- atom(X), atom(Y) | X=Y.
```

実行例

```
?- X = a,  Y = b,  shoen(my_unify(X, Y)).
```

このようにすれば, 必ず荘園の外部のゴールが実行されてから荘園内のゴールが実行されるので, ユーザの意図した通りに動くようになる. ただし, 荘園を作るオーバーヘッドは大きく, ユニフィケーション一つ毎に作っていたら大変非効率的なプログラムとなる. 従って, 基本的にはシェルから起動する程度の大きさの仕事のまとまりを荘園にするのが一般的である.

また，この例では荘園の内外で共有している変数がアトムに限定されていたためにこのように簡単な同期メカニズムを用いることで問題は解決された．しかし，共有変数が構造体データに決まる場合には，各要素に含まれる全ての変数の値が決まるまで待つような作りにしなければならないので，実際には大変複雑なプログラムになることが多い．

14.2.2 保護フィルタ

今度は，ある程度の大きさのまとまった処理を荘園の上で実行させることを考えてみよう．その際，荘園の外部と内部で共有する変数がない場合には，プログラミングの際に特別注意することはない．しかし，共有変数を用いる場合には，前節で説明したように，主にプログラムのバグが原因で荘園の内部で起こったエラーが荘園の外部に影響を及ぼさないように注意する必要がある．

そのための対策として，荘園の外部と内部で共有している変数に保護フィルタをつける方法がある．失敗するかも知れないユニフィケーションの例で示した同期メカニズムも，実は保護フィルタの簡単なものである．では，次のような例を考えてみよう．

> ゴール main(X,Y) は X の値を決める．またゴール sub(X,Y) は Y の値を決める．しかし，プログラムにはバグがあるかも知れず，main のバグと sub のバグを切り分けたい．そのために，それぞれ別の荘園の下で実行し，それぞれの報告ストリームを監視することによってどちらのエラーであるか特定したい．

実行例

```
?- shoen1(main(X, Y)),  shoen2(sub(X, Y)).
```

この実行例は，ゴール main(X,Y) を荘園 1 の内部で実行し，sub(X,Y) は荘園 2 の内部で実行するものである．それぞれのゴールは変数 X と Y を共有している．ゴール main(X,Y) のエラーは荘園 1 に，ゴール sub(X,Y) のエラーは荘園 2 に報告して欲しいものとする．

ここで，もしゴール sub(X,Y) が先に実行され，プログラムのバグで変数 X の値を決めてしまった場合，ゴール sub(X, Y) のバグであるにも関わらず，エラーは荘園 1 に報告される．変数 Y に関しても同様に，ゴール main(X,Y) のバグであるにも関わらずエラーは荘園 2 に報告される．

これを回避するために，共有変数 X と Y にフィルタをつけてみよう．なお，ここでも説明を簡単にするため，X と Y はいずれもアトムデータのみに決まるものとする．

プログラム

```
main(X, Y):- true |                          %(1)
        filter(Y, Y0),                       %(2)
        main_1(X, Y0).                       %(3)
sub(X, Y):- true |                           %(4)
        filter(X, X0),                       %(5)
        sub_1(X0, Y).                        %(6)
filter(A, A0):- atom(A) | A=A0.              %(7)
```

実行例

```
?- shoen1(main(X,Y)),  shoen2(sub(X,Y)).%(8)
```

ここで，ゴール filter(A,A0) は保護フィルタと呼ばれ，荘園の内部で用いる変数をそのままの形では外部に見せないためのゴールである．この保護フィルタによって，ゴール main(X,Y) のバグは荘園 1 に，ゴール sub(X,Y) のバグは荘園 2 に報告される．

このように，荘園の内側と外側で共有変数を用いる場合には，保護フィルタをつけた方がデバッグが効率的に行なえるプログラムとなる．しかし，失敗するかも知れないユニフィケーションの例でも説明したように，共有変数が構造体の場合には含まれる全ての変数に対して保護フィルタを設けなければならず，実際には更に複雑なものとなる．

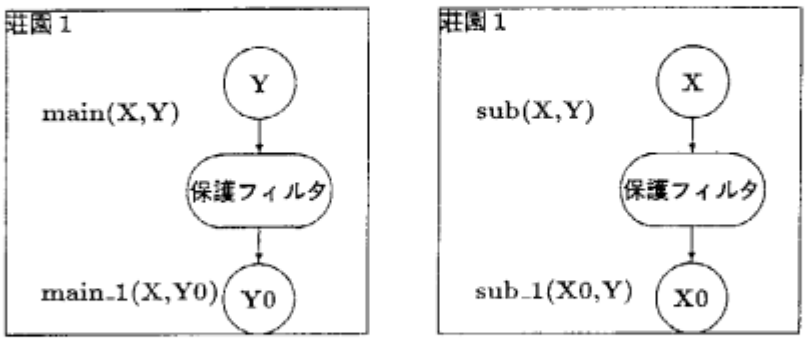


図 14.6: 保護フィルタ

14.2.3 複数の計算方法のうちの一つを利用

荘園機能の実行制御機能のうち, 実行の放棄機能を次のような目的で利用できないかを考えてみよう.

『一つの問題を解くのに複数の計算方法がある時, それぞれ別の荘園の上で並列に実行させ, 一つの荘園から答が出てきたら別の荘園は放棄すれば良いだろう.』

求まった答を取り出す方法としては, 次の 2 通りの方法が考えられる.

[方法 1]: 2 つの荘園に同じ変数を渡して, どちらかが答を返すのを待つ

[方法 2]: 一本にマージするストリームを渡して, どちらかがメッセージを送ってくるのを待つ

では, それぞれに関して問題点と注意する点を説明する. なお, 説明を簡単にするため, ここでは答としては整数のみを返すものとする.

[方法 1]: 2 つの荘園に同じ変数を渡して, どちらかが答を返すのを待つ

以下にプログラム例を示す.

```
solver(X):-                                              %(1)
   terminator(Report1, Report2, Control1, Control2),     %(2)
   shoen1((algorithm1(X), Control1, Report1),            %(3)
   shoen2((algorithm2(X), Control2, Report2).            %(4)

terminator([terminated|_], _, _, Control2):- true |      %(5)
   Control2=[abort].                                     %(6)
terminator(_, [terminated|_], Control1, _):- true |      %(7)
   Control1=[abort].                                     %(8)
```

どちらの計算方法も同じ答を返してくる場合には問題ない. また違う答を返してくる場合にも, algorithm1(X) が終了して荘園 2 を実行放棄するまでに時間が充分あれば良いが, ほとんど同時に終わって答を返そうとした場合には, ゴール solver(X) は失敗する.

これは, 共有変数 X に保護フィルタをつけても同じことであり, 同じ変数の値を別々のゴールが決めようとするような使い方はしてはならない. 次のように共有変数を用いないようにすれば良い.

```
solver(X):-                                                    %(1)
   selector(X1, X2, X),                                        %(2)
   terminator(Report1, Report2, Control1, Control2),           %(3)
   shoen1((algorithm1(X1), Control1, Report1),                 %(4)
   shoen2((algorithm2(X2), Control2, Report2).                 %(5)

selector(X1, X2, X):- integer(X1) | X=X1.                      %(6)
selector(X1, X2, X):- integer(X2) | X=X2.                      %(7)

terminator([terminated|_], _, _, Control2):- true |            %(8)
   Control2=[abort].                                           %(9)
terminator(_, [terminated|_], Control1, _):- true |            %(10)
   Control1=[abort].                                           %(11)
```

ただし，このように複数の解が求まってきてその中から一つ選ぶタイプの問題では，次の方法 2 で示すようなマージャを使うのが一般的である．

[方法 2]: 一本にマージするストリームを渡して，どちらかがメッセージを送ってくるのを待つ

以下にプログラム例を示す．

```
solver(X):-                                                    %(1)
   merge({X1, X2}, Xs),                                        %(2)
   terminator(Report1, Report2, Control1, Control2),           %(3)
   result(Xs, X),                                              %(4)
   shoen1((algorithm1(X1), Control1, Report1),                 %(5)
   shoen2((algorithm2(X2), Control2, Report2).                 %(6)

terminator([terminated|_], _, _, Control2):- true |            %(7)
   Control2=[abort].                                           %(8)
terminator(_, [terminated|_], Control1, _):- true |            %(9)
   Control1=[abort].                                           %(10)

result([H|_], X):- integer(H) | X=H.                           %(11)
```

この方法だと，どちらの計算方法が早く終わっても，或いは同時に終わっても，また別の答を返してきても正しく答えは得られる．しかし，片方の荘園で計算できた時にもう一方の荘園の実行を放棄すると，ストリームは閉じられないのでゴール merge(X1,X2,Xs) はデッドロックする．

このデッドロックを解消するためには，マージャの入り口に異常終了した時にストリームを閉じるためのバルブ機能と呼ばれるフィルタを次のようにして入れねばならない．

```
solver(X):-                                                    %(1)
   merge({X1, X2}, Xs),                                        %(2)
   valve(Knob1, XX1, X1),                                      %(3)
   valve(Knob2, XX2, X2),                                      %(4)
   terminator(Report1,Report2,Control1,Control2,Knob1,Knob2),  %(5)
   result(Xs, X),                                              %(6)
   shoen1((algorithm1(XX1), Control1, Report1),                %(7)
   shoen2((algorithm2(XX2), Control2, Report2).                %(8)

valve(close, XXs, Xs):- true | Xs=□.                           %(9)
```

```
alternatively.                                                    %(10)
  valve(_,    [],   Xs):- true | Xs=[].                           %(11)
  valve(Knob, [H|T], Xs):- true |                                 %(12)
          Xs=[H|Z], valve(Knob, T, Z).                            %(13)

  terminator([terminated|_],_,_,Control2,_,Knob2):- true |%(14)
          Control2=[abort],Knob2=close.                           %(15)
  terminator(_,[terminated|_],Control1,_,Knob1,_):- true |%(16)
          Control1=[abort],Knob1=close.                           %(17)

  result([H|_], X):- integer(H)| X=H.                             %(18)
```

なお, このように荘園機能を用いなくても alternatively を用いて実行の中断機構を各アルゴリズムに持たせることもできるのは, 入門編で説明した通りである.

第 15 章

プログラミングテクニック (1)

本章では, KL1 のプログラミングテクニックとして以下のものを学ぶ.

- 計算の終了の判定
- 優先度制御の応用
- ストリーム通信あれこれ

15.1 計算の終了の判定

本節では, 計算の終了即ちゴールの実行が終了したことを判定するためのいくつかの代表的なプログラミングテクニックを説明する. 計算の終了の判定方法は, 対象とする問題の性質, プログラミングスタイルによって異なる. ここでは代表的ないくつかの例を示すので, 目的に応じて使い分けること.

15.1.1 ストリームが閉じられるのを監視

入力データを与えればその計算結果がストリームを通じて出力データとして返ってくるタイプの KL1 プログラムでは, 出力データが全て返ってきてストリームが閉じられるのを監視する (図 15.1).

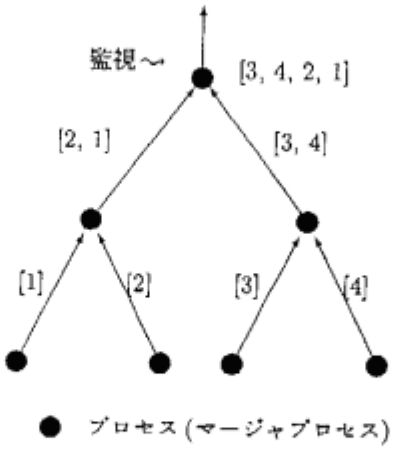


図 15.1: ストリームが閉じられるのを監視

例えば, クイックソートのプログラムにおいては, 次のようにして計算の終了を判定すれば良い.

プログラム例

```
go(End):- true |                                    %(1)
        qsort([3, 6, 1, 4, 2, 8], Ys),              %(2)
        judge(Ys, End).                             %(3)
judge([_|T], End):- true | judge(T, End).           %(4)
judge([],    End):- true | End=end.                 %(5)
```

実行例

```
?- go(End)|End.                                     %(6)
   End=end.                                         %(7)
```

(2) で実行したゴール qsort([3, 6, 1, 4, 2, 8], Ys) は, 変数 Ys に次々とソートされた結果がリストになって返ってくる. 従って, 計算の終了は ストリーム Ys が閉じられたことによってわかる. ここで注意するのは, Ys の値がリストに決まった時点ではまだ計算は終了していない点で, 例えば次のようにして終了を判定してはいけない.

```
judge(Ys, End):- wait(Ys) | End=end.
          或いは
judge([_|_], End):- true  | End=end.
```

15.1.2 プロセスの分岐履歴に基づく終了判定法

プロセスが次々と新しいプロセスを生成するプログラムで, かつ答はストリームで収集しないようなものは, プロセスの分岐履歴に基づいて終了を判定する. 即ち, 親プロセスはその子プロセスの終了のみを監視し, 全ての子プロセスが終了したら更に自分の親に終了を報告するという方法である (図 15.2).

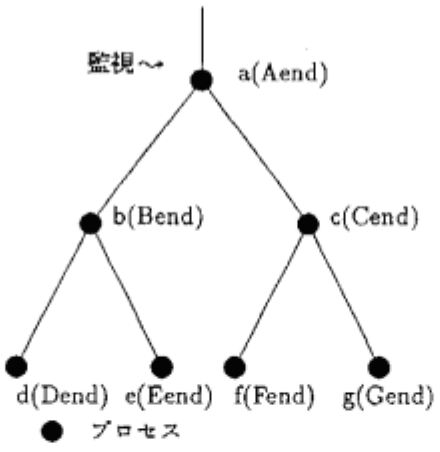


図 15.2: プロセスの分岐履歴に基づく終了判定法

ここで, 末端のプロセス d(Dend) が仕事を終えると, 変数 Dend の値を決める. 同様にプロセス e(Eend) が仕事を終えると, 変数 Eend の値を決める. d と e の仕事が両方とも終わると, 親プロセスである b(Bend) の仕事も終わりである. 即ち, 変数 Dend と Eend の両方の値が決まったら, 変数 Bend の値を決める.

同様にして, Bend と Cend の値が決まったら, プロセス a(Aend) の仕事は全て終わったことになる. このように, 子プロセスを生成する際には終了判定用の別々の変数を渡してやり, この変数を親プロセスが監視してやることによって計算の終了を判定する.

プログラム例

```
fork(0, End):- true | End = end.                %(1)
otherwise.                                      %(2)
fork(L, End):- NL:=L-1 |                        %(3)
        fork(NL, End1),                         %(4)
        fork(NL, End2),                         %(5)
        judge(End1, End2, End).                 %(6)
judge(end, end, End):- true | ENd = end.        %(7)
```

実行例

```
?- fork(2, End)|End.                            %(8)
   End=end.
```

なお, プロセスの分岐履歴に基づく終了判定法はプロセス構造を素直に反映していて簡単ではあるが, 以下の点であまりエレガントとは言えない. 例えば, 子プロセスを 1 つしか生成しないプロセスが, わざわざ子プロセスの終了を待つのは無駄である. またプロセス b(Bend) は d(Dend) と e(Eend) を生成した後は, 単にそれらがが終了するのを待つだけのもので, このような場合には次に示すショートサーキット法を用いた法がエレガントである.

15.1.3 ショートサーキット法 (1)

ショートサーキット法は, プロセスを生成する際に隣同士のプロセスを次々と連結するような回路を構成しておき, プロセスの中に用意したスイッチは初めはオープン状態にしておく. プロセスの実行が終了した時には, そのスイッチをクローズする. すると, 全てのプロセスの実行が終了するとその回路がショートして, 全体の実行が終了したことがわかるという方法である (図 15.3).

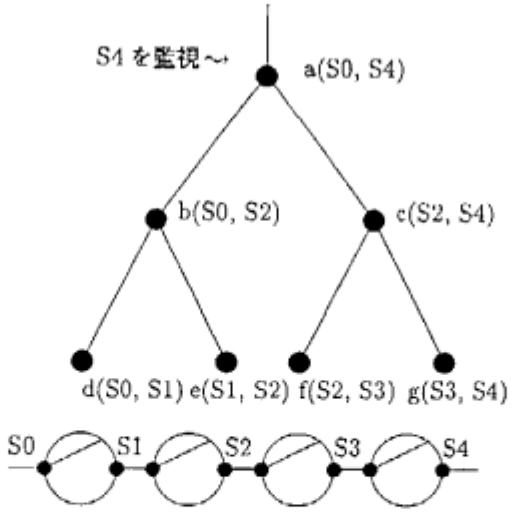


図 15.3: ショートサーキット法 (1)

ゴール a(S0,S4) の引数 S0 はサーキットの入り口を表わす変数で, S4 は出口を表わす変数である. 例えば, S0 の値を short に決めてやって, S4 にその値が返ってきたら, サーキットはショートした, 即ち全てのプロセスは実行を終了したことがわかる. では, プログラム例をみてみよう.

プログラム例

```
fork(0, In, Out):- true | In=Out.               %(1)
```

```
otherwise.                                          %(2)
 fork(L, In, Out):- NL:=L-1 |                       %(3)
        fork(NL, In, Switch),                       %(4)
        fork(NL, Switch, Out).                      %(5)
```

実行例

```
?- fork(2, In, Out), In=short|Out                   %(6)
   Out=short.
```

(1) の節では, 変数 In と Out とのユニフィケーションをしているが, これがこれは実際にはプロセスの実行が終了した時にスイッチをクローズする操作を示す.

(4) と (5) のゴールは, いずれもプロセスを生成する操作を表わしているが, 共有変数 Switch を用いて, 兄弟どうしのプロセスを連結するサーキットを形成している.

実行例の (6) をみると, ゴール fork(2,In,Out) を実行する際には, サーキットの入り口を表わす変数 In の値を short に決めてやる. 実行の終了の判定は, 出口を表わす変数 Out の値が決まるのを監視することにより行なう.

15.1.4 ショートサーキット法 (2)

ショートサーキット法は, プロセスの終了判定に用いられるだけでなく, ストリームに流すメッセージが全てのプロセスに行き渡ったか否かを判定するのにも用いることができる. その場合には, サーキットを構成する変数をメッセージにつけて渡してやり, 全てのメッセージで一つのサーキットを構成し, 各メッセージがプロセスに行き渡った時点でスイッチを閉じてやれば良い (図 15.4).

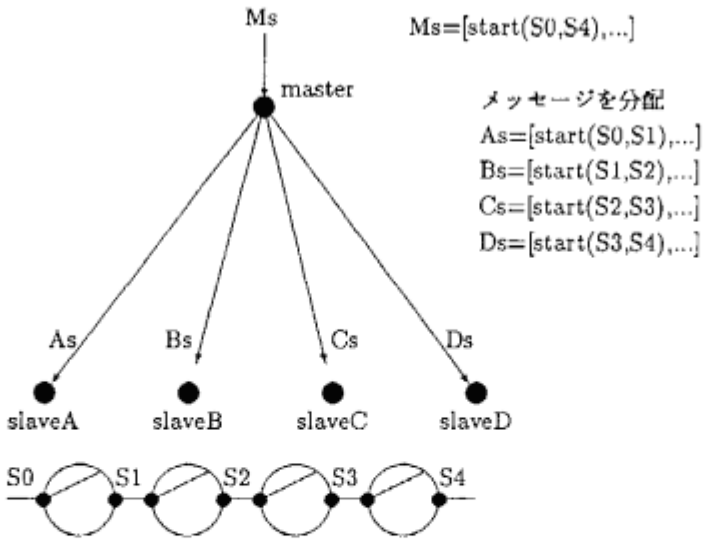


図 15.4: ショートサーキット法 (2)

master プロセスは, Ms ストリームを通じて流れて来るメッセージを待っている. メッセージを受け取ると, slaveA から slaveD までのプロセスにそのメッセージを分配する. その際, メッセージにはサーキットを構成するための変数をつける.

slave プロセスは, master プロセスから流れてきたメッセージを受け取るとサーキットのスイッチを閉じる. 全てのプロセスがメッセージを受け取ったら, サーキットはショートする. では, プログラム例をみてみよう.

プログラム例

```
create_process(Ms):- true |                                          %(1)
        master(Ms,As,Bs,Cs,Ds),                                      %(2)
        slave(As),                                                   %(3)
        slave(Bs),                                                   %(4)
        slave(Cs),                                                   %(5)
        slave(Ds),                                                   %(6)
master([],As,Bs,Cs,Ds):- true | As=[],Bs=[],Cs=[],Ds=[].%(7)
master([start(In,Out)|Cdr],As,Bs,Cs,Ds):- true |                     %(8)
        As=[start(In,Sw1)|NAs],                                      %(9)
        Bs=[start(Sw1,Sw2)|NAs],                                     %(10)
        Cs=[start(Sw2,Sw3)|NAs],                                     %(11)
        Ds=[start(Sw3,Out)|NAs],                                     %(12)
        master(Cdr,NAs,NBs,NCs,NDs).                                 %(13)
slave([]):- true | true .                                            %(14)
slave([start(In,Out)|Cdr]):- true |                                  %(15)
        In=Out,slave(Cdr).                                           %(16)
```

実行例

```
?- create_process(Ms),                                               %(17)
   Ms=[start(end,Short0),start(end,Short1)]|Short0,Short1.%(18)
   Short0=end,Short1=end.                                            %(19)
```

(1) から (6) では, プロセスを生成してメッセージを流すためのストリームを接続している. (7) では, メッセージストリームが閉じられた時に master プロセスは終了することを示している. (8) から (13) では, start メッセージを master プロセスが受け取った時に slave プロセスに start メッセージを送っている. その際にサーキットを構成するための変数をメッセージにつけている. (16) では, slave プロセスが start メッセージを受け取った時にショートサーキットのスイッチを閉じている.

実行例をみてわかるように, ショートサーキットの出口を表わす変数 Short0 と Short1 は, 2 つ流したそれぞれのメッセージが各プロセスに行き渡った時に閉じられていることがわかる.

15.2 優先度制御の応用

本節では, 初級編で学んだ優先度制御を使用する際の注意点と応用例と説明する.

15.2.1 優先度は下げる方向で使おう

一般的に, 優先度は下げる方向で使うよう心掛けるべきである. 例えば, 次に示すプログラムは優先度を上げる方向で使った例で, 思った通りには実行されない悪い優先度の使い方の例である.

```
top:- true |                                    %(1)
   go@priority(*,1000).                         %(2)
go:- true |                                     %(3)
   a,b.                                         %(4)
a :- true | goal_0@priority(*,3000).            %(5)
b :- true | goal_1@priority(*,2000).            %(6)
```

まず, 最低優先度のゴール go の中でゴール a と b を実行する. それぞれのゴールの中では, 自分の優先度を 3000 と 2000 に上げている. ユーザが意図するところはおそらく次のようなものであろう.

- ゴール goal_0 は優先度 3000 で実行する.
- ゴール goal_1 は優先度 2000 で実行する.
- すなわち, ゴール goal_0 は goal_1 より優先的に実行して欲しい.

ところが, 実際には次のように実行されて思い通りには動かないかも知れない.

- ゴール go は優先度 1000 で実行される.
- ゴール a と b も優先度 1000 で実行される.
- ゴール b が先に実行されて, goal_1 が優先度 2000 で実行される.
- この時点で, ゴール a の実行は優先度 1000 で実行されているので, 優先度 2000 の goal_1 の処理が終わるまで, goal_0 の実行は行なわれない.

この例の問題点は, ゴールの優先度を変えるという処理を行なうゴール a と b の優先度が, 変えた後のゴールの優先度よりも低い点であることは明らかであろう. この例はごく単純な例なので, このような間違った優先度制御を行なうことはまずないと思われるが, プログラム中で動的に優先度を変える場合, しばしば優先度を上げる方向で使ってしまうことがあるので注意すること.

優先度制御を行なう場合には, あくまでも優先度を下げる方向で使うように心掛けるべきである.

15.2.2 負荷分散を行なう部分は高優先度

負荷分散を行なうプログラムにおいては, 負荷分散を行なう処理すなわちゴールを投げる処理は, 投げたゴールの優先度よりも低くした方が効率的に実行されることを初級編で学んだ. ここでは再度この意味を考えてみよう.

まず, 4 台のプロセッサにゴール job を 8 個サイクリックに割り付けたプログラム例を次に示す。なお, 投げるゴール job の数は引数 N で指定するものとする. また, 何台のプロセッサに投げるかは組込み述語を用いて求めるものとする.

プログラム例

```
:- module foo.
:- public distribute/1.
        distribute(N):- true |                  %(1)
```

```
                current_processor(_, X, Y),          %(2)
                PEs := X*Y,                          %(3)
                fork(N, PEs, 0).                     %(4)
    fork(0, PEs, PE):- true | true.                  %(5)
otherwise.
    fork(N, PEs, PE):- true |                        %(6)
                PeNo:=PE mod PEs,                    %(7)
                job@processor(PeNo),                 %(8)
                NextPE:=PE+1,                        %(9)
                N1:=N-1,                             %(10)
                fork(N1, PEs, NextPE).               %(11)
    job :- ....                                      %(12)
```

実行例

```
    ?- foo:distribute(8).                            %(13)
```

プロセッサが 4 台の時 (13) のようにゴールを実行するとゴール job はプロセッサの 0, 1, 2, 3, 0, 1, 2, 3 に割り付けられる。ところが, 最初にプロセッサ 0 に job が投げられたところで, ゴール fork と job は同じ優先度で平等にスケジュールされて実行される. 即ち, ゴールを投げる処理は job の処理に邪魔されて, それ以降のゴールを投げる処理が遅れてしまう.

そこで, 一般的には次のようにして負荷分散処理を行なう部分は投げたゴールの優先度よりも高くするよう心掛けねばならない. 次のプログラム例では, ゴールを投げる処理を行なう fork プロセスは優先度 4095 で (4) 実行され, 投げられた job プロセスは優先度 2000 で実行される (12).

プログラム例

```
:- module foo.
:- public distribute/1.
    distribute(N):- true |                           %(1)
            current_processor(_, X, Y),              %(2)
            PEs := X*Y,                              %(3)
            fork(N, PEs, 0 )@priority(*,4095).       %(4)
    fork(0, PEs, PE):- true | true.                  %(5)
otherwise.
    fork(N, PEs, PE):- true |                        %(6)
            PeNo:=PE mod PEs,                        %(7)
            job_ext@processor(PeNo),                 %(8)
            NextPE:=PE+1,                            %(9)
            N1:=N-1,                                 %(10)
            fork(N1, PEs, NextPE).                   %(11)
    job_ext :- job@priority(*,2000).                 %(12)
    job :- ...                                       %(13)
```

実行例

```
    ?- foo:distribute(8)@processor(0).               %(14)
```

15.2.3 アルゴリズムへの適用 (最適経路問題)

優先度制御は, プログラムの実行を効率化する目的で利用するのが一般的であることは既に学んだ通りである. 本節では, アルゴリズムの一部を優先度制御を適用することによって非常に効率化できる例を示す.

例としては初級編で用いた最適経路問題を取り上げる．ここでは，初級編で用いたアルゴリズム (以後従来のアルゴリズムと呼ぶ) の説明を行ない，効率化するためにアルゴリズムを改良し，改良部分を優先度制御機能を用いたプログラム例を説明する．

『図 15.5に示されるようなネットワークにおいて，各辺にはそれぞれ正のコストが割当てられている時，その上の 2 点 (開始点と終了点) 間を結ぶ経路のうちコストが最小の経路を求めよ．』例えば，点 A から点 P への最適経路は A-E-I-J-F-G-K-O-P で，コストは 23 である．

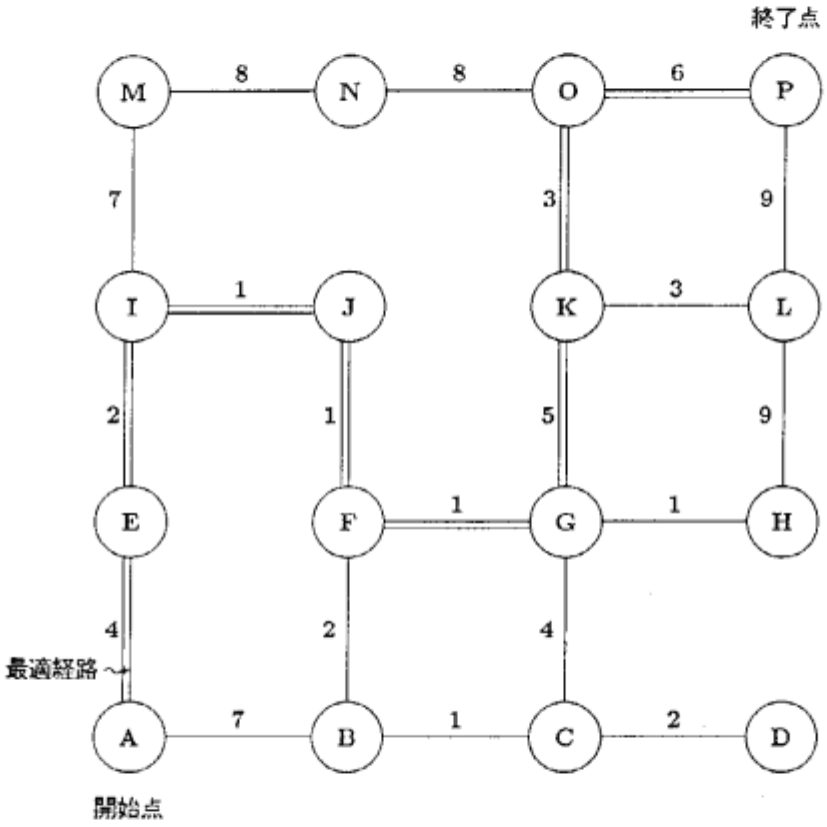


図 15.5: 最適経路問題のネットワーク例 (1)

従来のアルゴリズム

ネットワークの各ノード毎にプロセスを生成し，ノード間のメッセージ通信によって最適解を求める．全ての経路を探索し，一番その中でコストの低いものが最適経路である．

なおここでは説明を簡単にするため，図 15.6に示すような簡単なネットワークを用いて，開始点をノード A，終了点をノード D として説明を行なう．この図はネットワークの構成を表わすと同時に，プロセスの構成も表わしている．

各ノードを表わす円はメッセージを受け取るプロセスを表わしている．また，各プロセスは隣接するプロセスと入出力ストリームで接続されている．例えば，プロセス A はプロセス C と B と接続されており，A の入力ストリームは C の出力ストリームである AIn1 と B の出力ストリーム AIn2 をマージし

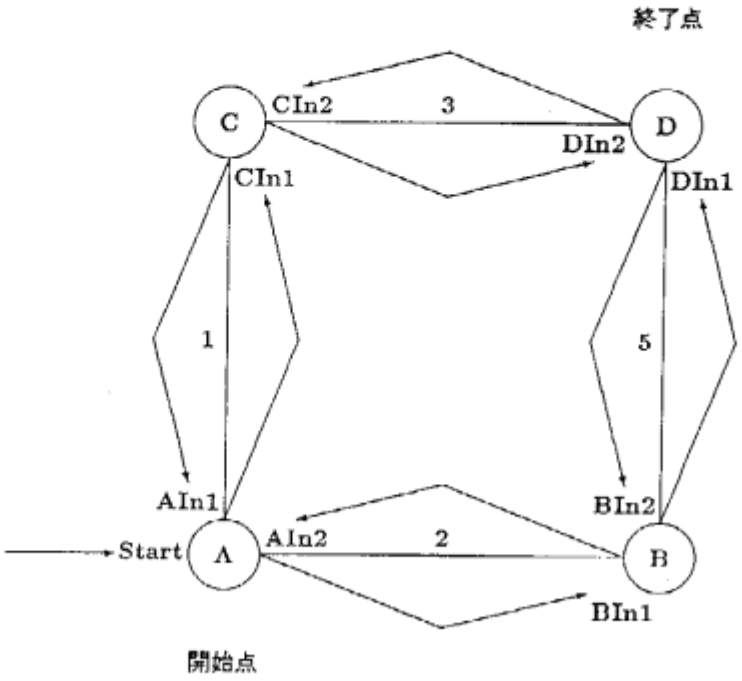


図 15.6: 最適経路問題のネットワーク (2)

たものである. 更に, A は開始点であるため, 実行を開始するためのメッセージを流すストリーム Start も入力ストリームにマージされている.

なお, 最適解の探索を開始する前には, あらかじめこの図に示すようなプロセスが構成されており, 各ノードプロセスはメッセージの到着を待っているものとする. また, 各ノードプロセスは内部状態として現在までの最小コストとそのコストを得た隣接ノードの方向 (東:e, 西:w, 南:s, 北:n) を保持しており, 初期状態としては充分大きいコスト (ここでは 9999) を持っているものとする.

最適経路の探索は, 次のようなアルゴリズムで実行される.

- 実行の開始は, ストリーム Start にコスト 0 のメッセージを送ることにより行なう. メッセージは, 次のような形式をしている.

  reached(Direction,Cost,ForwardTo)

  Direction はそのメッセージを受け取ったノードからみて, 送ったノードが東 (e) 西 (w) 南 (s) 北 (n) のいずれの方向であるかを示す. 実行の開始時には, * で表わす. また, 例えばノード C が A からのメッセージを受け取った際には, 南 (s) で表わす.

  Cost は, そのメッセージを送ったノードが現在保持している最小コストに受け取ったノードまでのコストを加えたものである.

  ForwardTo は, そのメッセージを受け取った後にメッセージを送るべき隣接ノードの情報である. 例えば, 実行の開始時にノード C がメッセージを受け取った際には, ノード C と B が隣接するノードであるので, ForwardTo はこれら複数の隣接ノードに関する情報が, 次のようなリスト形式で決まる.

  [{ 方角, コスト, 出力ストリーム },...]

- ノード n がメッセージreached(Direction,Cost,ForwardTo) を受信した時, 値 Cost がその時点で保持している最小コストと同じか大きい場合には, その経路は最適経路とはなり得ないのでこのメッセージは捨てて (即ち探索枝を刈り込む), 次のメッセージを受信するのを待つ.
- そうでない場合, 即ちその時点で保持している最小コストの値より小さい場合には, それ以前に保持していたコストと方向を捨て, Direction と Cost を新しいコスト及び方向とする. また, そのノードまでの最小コストが変化したことを隣接するノードに伝えるため, まず隣接ノード情報 ForwardTo を得, それらに対して新たにメッセージを送る.
- 実行の開始は, 開始点にコスト 0 のメッセージを送ることで始まる. このメッセージに対して隣接するノードに次々と新たなメッセージが流れ, 全ての経路をメッセージが流れる. ただし, 枝刈りされたメッセージは捨てられる. このようにして, ネットワーク上から全てのメッセージが消滅した時, 各ノードが現在保持している最小コストと方向が, 開始点からそのノードまでの最適経路とコストである.
- なお, 終了判定の方法はシュートサーキット法を用いる.

では, 実際に図 15.6のネットワークを用いて, 本アルゴリズムで最適経路を探索する様子を追ってみよう.

まず, 最初にストリーム Start にコスト 0 のメッセージを送ると, ノード A は自分の保持しているコストの初期値 9999 を捨てて新たに 0 をコストとし, 隣接するノード B にコスト 2, ノード C にコスト 1 のメッセージを送る (図 15.7).

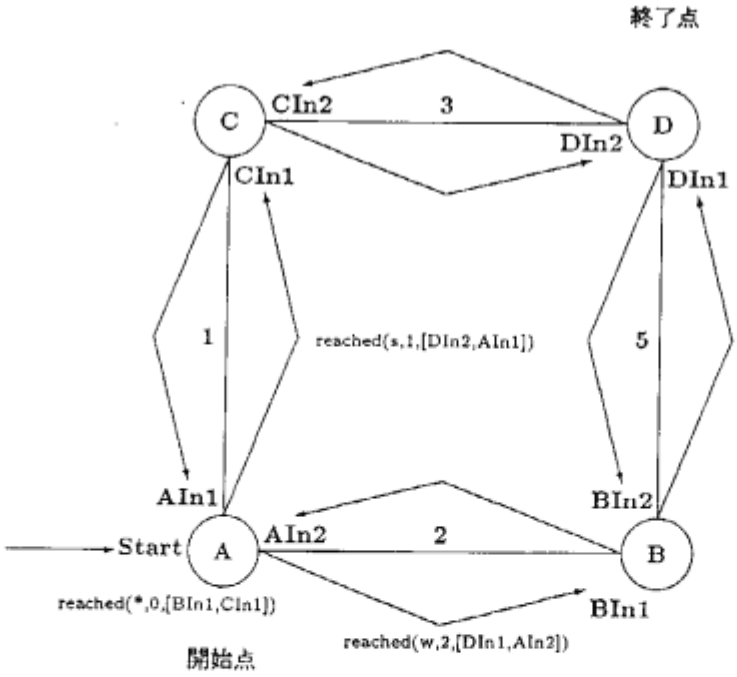


図 15.7: 実行開始直後の状態

メッセージを受け取ったノード B は自分の保持しているコストの初期値 9999 を捨てて新たにコストを 2 とし, 隣接するノード A にコスト 2, D にコスト 3 の新たなメッセージを送る. なお, ここで次にノード A が受け取ったコスト 2 のメッセージは, A が保持している最小コスト 0 よりも大きいので, このメッセージは捨てられる. 即ち, このネットワーク上の各経路には正のコストが与えられているので, ループすることはない.

また, ノード B がメッセージを処理している間にノード C が受け取ったコスト 1 のメッセージの処理は, 並列に行なわれる.

同様にして次々と終了点の方向に向かってメッセージが流れ, ネットワーク上からメッセージが消滅した時に最適経路が得られる.

改良したアルゴリズム

ここでは, 最適経路探索問題の効率化を考えるためにまず従来のアルゴリズムにおける問題点を洗い出してみる. そして, 効率化するためにアルゴリズムを改良し, 優先度制御機能を用いてプログラム化した例を示す.

まず図 15.5を見てみよう. ここで, 例えばノード G に着目して見よう. 開始点から G までの経路は, A-E-I-J-F-G (コスト 8), A-B-F-G (コスト 9), A-B-C-G (コスト 12), A-E-I-J-F B-C-G (コスト 15), A-E-I-M-N-O-K-G (コスト 37) 等全部で 12 通りある. ここでもし探索が開始されてからネットワーク上を流れるメッセージが, このコストの高い順に伝達された場合, これは G までの全ての可能な経路を試みることになる. 即ち, 全てのメッセージが枝刈りされることなくネットワーク上を伝達されることになり, その時探索にかかる計算量は最大となる.

この問題の計算量を低く抑えるには, コストの低い経路情報はできるだけ早く伝達することが重要である. 先にコストの低い経路情報が知らされれば, それより高いコストの経路に対応するメッセージが到着しても, それ以上先には伝達されず, 探索空間を小さくすることができる.

この方針を徹底したのが Dijkstra の算法である. Dijkstra の算法では, 到達コストがわかっているネットワークの枝をコストの順にソートしておき, コストの低い順に隣のノードへの経路のコストを計算していく. こうすると, コストの最適経路以外には計算をしないことになり, ネットワーク中の枝の数 (n) 程度の繰り返しで最適解を求められる. ソーティングに均衡木を用いれば, 一回の繰り返し当たりの手間は最悪 $\log n$ なので, 全体のコストは最悪 $n \log n$ 程度である.

では, Dijkstra の算法アルゴリズムを従来のアルゴリズムに導入してプログラムの効率化を考えてみよう. しかし, 本アルゴリズムはコストの順にソートして低い順に実行すると言う点において, 逐次アルゴリズムである. これをそのまま KL1 で実現した場合には, 並列性が全く失われてしまう. そこで, ソーティングを真面目に行なうことは考えずに, 次のように優先度制御機構を利用して同等の機能を実現して見よう.

『ネットワークを流れるメッセージに優先度をつける』

Dijkstra の算法の要点は, コストの低いメッセージが高いメッセージよりも早く伝達されれば枝刈りの効率が良くなる点であった. しかし, ストリームを流れるメッセージ自身に優先度をつけることはできないので, ストリームにメッセージを流す処理をするプロセス (以後メッセージプロセスと呼ぶ) を設け, そのプロセスの優先度をコストに応じて設定するようにして実現する. なお, メッセージプロセスの優先度は次式により求めるものとする.

$$\text{優先度} = \frac{4095 \times (\text{コストの最大値} - \text{コスト})}{\text{コストの最大値}}$$

コストの最大値は, ネットワーク中の枝が持っているコストの最大値に, ネットワークの直径 (一番遠いノード間の距離で, 格子状のネットワークであれば, X, Y 各方向のサイズから 1 減じたものの和) を乗じたもので抑えられる. 例えば, 図 15.5のようなネットワークにおいては, 直径即ちノード A から P までの距離は 6 である. また, ネットワーク中の最大コストは 9 である. 従って, コストの最大値は 54 となる.

この算式でコスト 0 から 54 までのメッセージに対して付けられる優先度は次の表 15.1のようになる.

このような優先度付けを行なうと, 一台のプロセッサで実行した時には, 図 15.5のネットワークにおいては次表 15.2のような実行順でメッセージが流れる. なお, ここでは説明を簡単にするため, 優先度は 54 を最高とし 0 と最低として説明する.

コスト	優先度
0	4095
1	4019
2	3943
...	...
20	2578
21	2502
...	...
30	1820
31	1744
...	...
40	1061
41	985
...	...
53	75
54	0

表 15.1: コストとメッセージの優先度

優先度	54	50	46	47	46	45	44	41	40	37	35	33	31	28	25
コスト	0	4	6	7	8	9	10	13	14	17	19	21	23	26	29
1	外～A														
2		A～E													
3			E～I												
4				I～J A～B											
5					J～P B～C										
6						B⊗F P～G									
7							C～D G～H								
8								I～M							
9									G～K						
10										K～O K～L					
11											H⊗L				
12												M～N			
13													O～P		
14														L⊗P	
15															N⊗O

表 15.2: メッセージの伝達と枝刈り

表中, 縦軸は時間軸を示す. また, 横軸はメッセージで送られるコストとその優先度を示す. 例えば, 時間 1 の外～A という表記は, 優先度 54 でコスト 0 のメッセージがノード A に送られたことを示す. また, 時間 4 の時にはふたつのメッセージ送出が並列に行なわれていることを示す. 時間 6 の時に B からドに送られたメッセージ (B⊗F) は, F の保持している最小コストよりも大きいので捨てる, 即ち枝刈りが行なわれたことを示す.

表をみればわかるように, コストの低いメッセージは必ず高いメッセージより早く到着しているので, 枝刈りが効率的に行なわれている様子がわかる. 即ち, Dijkstra のアルゴリズムのうちソーティングの部分が優先度制御機構によって実現されている訳である.

ただし, 実際にはコストの差が小さいメッセージにおいては優先度が重なる場合もあり, その場合には不完全なソーティングを行なったと同じことになる. また, 複数プロセッサで実行した場合には, 優先度は同一プロセッサ内でのみ意味をなすので, 暇なプロセッサにとっては優先度の低いメッセージも直ちに処理することになる. しかし, 優先度の低いメッセージであっても, 最終的には他のメッセージと比較すれば優先度が高くなっている場合もあるので, 無駄な処理をしているとは一概に言えない.

以上のように, 優先度制御機構を利用した本アルゴリズムは, Dijkstra のアルゴリズムを並列化した一方式であるといえる.

プログラム例

では, 改良したアルゴリズムをプログラム化した例をみてみよう. まず, プログラムの起動方法とその結果の形式を説明する.

```
?- bestpath:go(R)|R.
R= [a-0-*, b-2-w, c-1-s, d-4-w]
```

上記のように実行すると，その結果はリスト形式で返ってくる．ここで，リストの各要素はネットワーク上の各ノードが保持している解，即ち開始点からそのノードまでの最適経路を次のように表わしている．

(ノード名)-(最適コスト)-(最適経路への方向)

なお最適経路への方向は最適コストを得た隣接ノードの方向であり，実際に経路を得るためには，このデータを追っていけば良い．また，ノード a の最適経路への方向は * となっているが，これは開始点であることを示す．

以下にプログラムを示すが，ゴール go と create_net の定義の部分は，プログラムの起動，及びネットワークを生成する部分である．またネットワークとしては，図 15.6の例を用いている．

```
:- module bestpath.
:- public go/1.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% go(Result)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% トップレベルの呼びだしゴールである．
%
% ボディ部では，ゴール create_net を実行してネットワークを生成すると同時に，
% ゴール message を実行して開始ノードの入力ストリーム Start に初期メッセージ
% を流す．
%
% また，その際にショートサーキットの入り口をアトム end に決め，出口を変数 End
% としておく．
%
% Result : 解を表わすリストで，ノードの数個要素がある．
%          各要素の形式は，(ノード名)-(最適コスト)-(方向)．
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
go(Result):- true |                                                    %(1)
        create_net(Start,Result,End),                                  %(2)
        message(0,54,*,Start,end-End).                                 %(3)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% create_net(Start,R0,End)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ネットワークを生成するためのゴールである．
%
% ボディ部では，各ノード毎に node プロセスを生成する．各ノードに対して複数あ
% る入力ストリームは，マージする．隣接ノード情報 (AOuts,BOuts 等) は，リストデー
% タとして初期設定し，隣接ノードの入力ストリームをそのノードの出力ストリームと
% する．
%
% Start : 開始ノードの入力ストリーム
% R0    : 解
% End   : ショートサーキットの出口を表わす変数
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
create_net(Start,R0,End):- true |                                      %(1)
        node(AIn,_,9999,AOuts,End,R0-R1,a),                            %(2)
```

```
        node(BIn,_,9999,BOuts,End,R1-R2,b),                          %(3)
        node(CIn,_,9999,COuts,End,R2-R3,c),                          %(4)
        node(DIn,_,9999,DOuts,End,R3-[],d),                          %(5)
        merge({AIn1,AIn2,Start},AIn),                                %(6)
        merge({BIn1,BIn2},BIn),                                      %(7)
        merge({CIn1,CIn2},CIn),                                      %(8)
        merge({DIn1,DIn2},DIn),                                      %(9)
        AOuts=[{s,1,CIn1},{w,2,BIn1}],                               %(10)
        BOuts=[{s,5,DIn1},{e,2,AIn2}],                               %(11)
        COuts=[{n,1,AIn1},{w,3,DIn2}],                               %(12)
        DOuts=[{w,3,CIn2},{n,5,BIn2}].                               %(13)
```

ノードプロセスの定義

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% node(In, D, C, Outs, End, RO-R, N)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 入力ストリームにメッセージ reached が来た時に，次に送るべき隣接ノードのリス
% トを ForwardTo に得，次のメッセージの到着を待つゴールである．
%
% メッセージに含まれているコストが現在保持しているコスト以上であれば，それ以
% 上隣接ノードに伝える必要はないので，ForwardTo は [] に決める．また次のメッ
% セージを待つ際には，方向 (D)，コスト (C)，隣接情報 (Outs) は同じものを使う．
%
% 現在保持しているコストより小さければ，ゴール make_forwarding_list を実行して，
% ForwardTo と新しい隣接情報 (NOuts) を得る．　また次のメッセージを待つ際には，
% 新たな方向 (Dir)，新たなコスト (Cost)，新たな隣接情報 (NOuts) を使う．
%
% また実行が終了してショートサーキットが閉じられ，End の値が end に決まったら，
% 隣接ノード情報に含まれる出力ストリームを全て閉じ，　そのノードプロセスが現在
% 保持しているノードの ID，コスト及び方向を解とする
%
% In    : 入力ストリーム
% D     : コスト C が来たパス情報の方角
% C     : 現在の最適コスト
% Outs  : 隣接ノード情報 {Direction,Cost,OutputStream} のリスト
% End   : 終了フラグ
% RO-R  : 解を表わす．開始ノードからそのノードまでの解が RO で，
%         それ以外のノードまでの解が D リスト形式で R に決まる．
% N     : ノードの ID，即ちノード名前を表わす a,b,c...
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
node(_,D,C,Outs,end,RO-R,N) :- true |                                %(1)
        RO=[N-C-D|R],closeOuts(Outs).                                %(2)
alternatively.                                                       %(3)
node([reached(Dir,Cost,ForwardTo)|In],D,C,Outs,End,RO-R,N) :-        %(4)
        Cost >= C |                                                  %(5)
        ForwardTo = [],                                              %(6)
        node(In,D,C,Outs,End,RO-R,N).                                %(7)
node([reached(Dir,Cost,ForwardTo)|In],D,C,Outs,End,RO-R,N) :-        %(8)
        Cost < C |                                                   %(9)
```

```
        make_forwarding_list(Outs,ForwardTo,NOuts),                    %(10)
        node(In,Dir,Cost,NOuts,End,RO-R,N).                            %(11)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% make_forwarding_list(Outs,ForwardTo,NOuts)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 隣接ノード情報リスト (Outs) から，メッセージを送るべき隣接ノード情報リスト
% (ForwardTo) と新しい隣接ノード情報リスト (NOuts) を得る．
%
% 隣接ノード情報リストには，方向 (Dir)，コスト (Cost)，出力ストリーム (Out) が含
% まれている．ForwardTo と NOuts はいずれもこの隣接ノード情報リストをコピーし
% てやれば良い訳であるが，出力ストリームに関しては，マージしてからコピーする．
%
% Outs      : オリジナルの隣接情報リスト
% ForwardTo : メッセージを送るべき隣接情報リスト
% NOuts     : 新しい隣接情報リスト
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
make_forwarding_list([{Dir,Cost,Out}|T],ForwardTo,NOuts) :- true |     %(1)
        Out={Out1,Out2},                                               %(2)
        ForwardTo=[{Dir,Cost,Out1}|ForwardToT],                        %(3)
        NOuts    =[{Dir,Cost,Out2}|NOutsT],                            %(4)
        make_forwarding_list(T,ForwardToT,NOutsT).                     %(5)
make_forwarding_list([],ForwardTo,NOuts) :- true |                     %(6)
        ForwardTo=[],NOuts=[].                                         %(7)
```

メッセージプロセスの定義

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% message(Cost,MaxCost,Dir,Node,TO-T,Priority)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ノードの入力ストリーム (Node) にメッセージを流すためのゴール message/5 を優先度
% (Priority) で実行する．
%
% Priority : 優先度
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
message(Cost,MaxCost,Dir,Node,TO-T,Priority) :- true |                 %(1)
        message(Cost,MaxCost,Dir,Node,TO-T)@priority(*,Priority).      %(2)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% message(Cost,MaxCost,Dir,Node,TO-T)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ノードの入力ストリーム (Node) に，方向 (Dir) とコスト (Cost) をパラメタとして
% reached メッセージを流す．メッセージを受け取ったノードプロセスは，次にメッ
% セージを送るべき隣接情報リスト (Forwarding) を決めて来るので，その情報に従っ
% て次々と隣接ノードプロセスにメッセージを送る (forward_messages)．
%
% Cost    : コスト
% MaxCost : 最大コスト
% Dir     : 方向
% Node    : ノードの入力ストリーム
```

```
% T0-T      : ショートサーキットの入り口と出口を表わす
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
message(Cost,MaxCost,Dir,Node,T0-T) :- true |                        %(1)
        Node=[reached(Dir,Cost,Forwarding)],                         %(2)
        forward_messages(Cost,MaxCost,Forwarding,T0-T).              %(3)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% forward_messaged(Cost,MaxCost,Forwarding,T0-T)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% メッセージを送るべき隣接ノード (Forwarding) に対して，メッセージを送るための
% プロセス message を生成する．その際，最大コスト (MaxCost) とコスト (Cost) から
% message プロセスの優先度 (Prio) を計算し，パラメタとして渡す．
%
% Cost       : コスト
% MaxCost    : 最大コスト
% Forwarding : メッセージを送るべき隣接ノード情報のリスト
% T0-T       : ショートサーキットの入り口と出口を表わす
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
forward_messages(C0,MaxCost,[{Dir,C1,Node}|F],T0-T) :-               %(1)
        C:=C0+C1,C=<MaxCost|                                         %(2)
        Prio := 4095 * (MaxCost - C) / MaxCost ,                     %(3)
        message(C,MaxCost,Dir,Node,T0-T1,Prio),                      %(4)
        forward_messages(C0,MaxCost,F,T1-T).                         %(5)
forward_messages(_,_,[],T0-T) :- true | T0=T.                        %(6)
otherwise.                                                           %(7)
forward_messages(C0,MaxCost,[_|F],T0-T) :- true |                    %(8)
        forward_messages(C0,MaxCost,F,T0-T).                         %(9)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% closeOuts(Outs)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 接続情報リストに含まれている出力ストリームを全て閉じる．
%
% Outs : 接続情報リスト
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
closeOuts([{_,_,Out}|Outs]) :- true |                                %(1)
        Out=[],                                                      %(2)
        closeOuts(Outs).                                             %(3)
closeOuts([]) :- true | true.                                        %(4)
```

15.2.4 動的負荷分散制御への応用

初級編で我々は負荷分散制御の目的と考え方を学び，静的に負荷分散制御を行なったプログラム例を紹介した．しかし，動的にしか個々の仕事の大きさがわからないような問題においては，静的な負荷分散制御だけで各プロセッサに仕事を均一に分散させるのはほとんど不可能に近い．このような問題においては，暇なプロセッサに対して次々と仕事を分散させると言う，動的負荷分散制御を行なうのが一番効率的であろう．

ここでは優先度制御機構を利用することによって暇なプロセッサを見つけ，それらに対してプロセッサ割り付けを行うと言う比較的簡単な動的負荷分散制御を行う方法を説明する．

考え方

プロセッサが暇であるかどうかは，優先度が充分低いゴール (以後 himada ゴールと呼ぶ) をそのプロセッサに投げてみて，それが実行されたことによって判定できる (図 15.8).

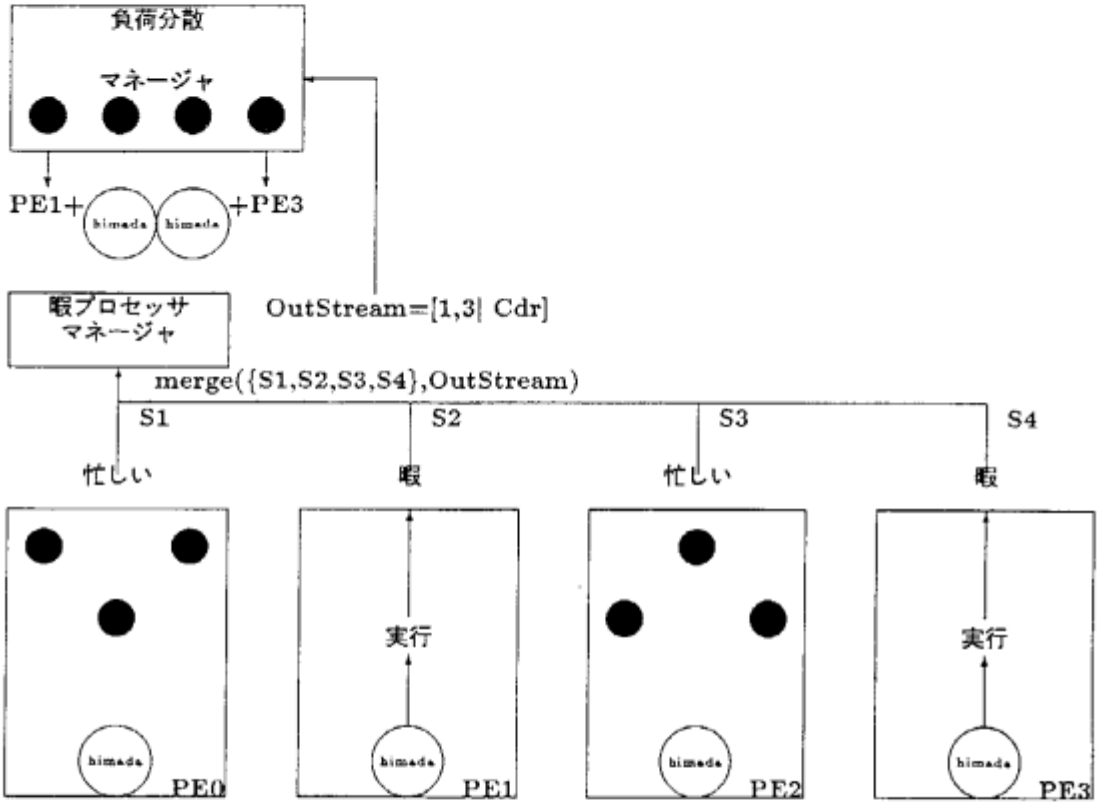


図 15.8: 動的負荷分散方式のプロセス構成

また，現在暇なプロセッサは誰であるかを管理する暇プロセッサマネージャプロセスを，特定のプロセッサ上に用意する (図 15.8). マネージャプロセスの仕事は，次の通りである.

初めに全プロセッサに対して himada ゴールを投げる: himada ゴールを投げる際には，マネージャプロセスに対してゴールが実行されたことを通知するためのストリームを同時に渡してやる. すると，himada ゴールが実行されるとマネージャには実行されたプロセッサの番号が返ってくる.

暇なプロセッサを要求されたら，プロセッサ番号を教える: 負荷分散を行なうプロセスから，暇なプロセッサの番号を教えてくれ，という要求に対して，himada ゴールが実行されたプロセッサの番号を渡す.

また，負荷分散を行なう負荷分散マネージャプロセスも基本的にはマネージャプロセスと同じプロセッサ上においておく. 負荷分散マネージャの仕事は，以下の通りである.

一つの仕事を独立性の高い部分問題に分割する: 部分問題に分けてから仕事を分散させるので，なるべく早くこの処理を行なった方が仕事の分散が早めにできるので効率的である. 特に，全体のプロ

セッサの台数個の仕事に分割するまでの処理は高速に行なった方が効率的であることは明らかであろう.

部分問題に分けたら, 暇なプロセッサにゴールを投げる: 暇なプロセッサ番号は, マネージャに教えてもらう. もし暇なプロセッサがなければ, 部分問題への分割処理を続行する. なお, ゴールを投げる処理は他のいかなる処理よりも高い優先度で行なうようにしなければならないのは既に学んだ通りである.

部分問題を投げる時には, himada ゴールも同時に投げる: すると, 部分問題の実行が終わった時点で優先度の低い himada ゴールが実行され, 再度そのプロセッサが暇になったことがマネージャに通知される.

投げる仕事の粒度は, パラメタにより変えられるようにするのが望ましい: どの程度の粒度の仕事を投げれば一番効率的であるかは, 実際に行してみなければわからないことが多い. そのため, 仕事の粒度はパラメタ等で簡単に変えられるようにしておくのが望ましい.

この方式で負荷分散が動的に行なわれる様子を追ってみよう.

1. 暇プロセッサマネージャは, 最初に himada ゴールを全プロセッサに投げる.
2. 暇プロセッサマネージャは, himada ゴールが実行されたプロセッサの番号, 即ち現在暇なプロセッサ番号を保持しておく.
3. 負荷分散マネージャは問題の分割作業を行う. 部分問題が一つできた時点で, 直ちに暇なプロセッサに部分問題ゴールと himada ゴールを投げる.
4. 次々と部分問題を投げ, 全プロセッサが忙しくなった時には部分問題ゴールを投げる作業を中断する. また, 次の部分問題を作る作業は続行する.
5. 暇なプロセッサが出てきたら, 部分問題を作る作業は中断して, 直ちに部分問題ゴールを投げる.
6. 部分問題を作る作業を終了したら, そのプロセッサは暇になったことになるので, 自分が暇になったことを himada ゴールを実行して暇プロセッサマネージャに伝える. すると, 以降部分問題を分散させる時には自分のプロセッサ上でも実行される.

プログラム例

ここでは, パラメタで与えられた数の仕事を, 動的負荷分散制御を行なって暇なプロセッサに次々と分散させていくプログラム例を示す. なお, 分散する仕事自身の内容はここではあまり意味がないので, ゴール job(Number) の中では, 引数 Number で与えられた仕事を何か実行するものとする.

```
:- module dynamic.
:- with_macro pimos.
:- public go/3.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% go(BootPE,PEs,NofJob)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% トップレベルの呼びだしゴール top を最高の優先度 4095 で実行する.
%
% BootPE : 使用するプロセッサの開始番号
% PEs    : 使用するプロセッサの台数
% NofJob : 負荷分散行う仕事の数
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
go(BootPE, PEs, NofJob):- true |                                    %(1)
    top(BootPE, PEs, NofJob)@priority(*, 4095).                     %(2)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% top(BootPE,PEs,NofJob)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% トップレベルの呼びだしゴールで，NofJob で指定された数の仕事を，PEs で指定
% された台数のプロセッサに投げる．また，システム全体のプロセッサ数は組込み述語
% を用いて得ているため，BootPE で指定したプロセッサから PEs 台数のプロセッサを
% 用いる．また，ゴール who_is_idle_initially を実行して，himada ゴールを全プロ
% セッサに投げる．そして，実際に仕事を投げる負荷分散マネージャ spawn_jobs を
% 実行する．
%
% BootPE : 使用するプロセッサの開始番号
% PEs    : 使用するプロセッサの台数
% NofJob : 負荷分散行う仕事の数
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
top(BootPE, PEs, NofJob):- true |                                    %(1)
    merge(WhoInit, Who, HimaPEs),                                    %(2)
    current_processor(_, CX, CY), Config:=CX*CY,                     %(4)
    who_is_idle_initially(0, PEs, BootPE, Config, WhoInit),          %(5)
    spawn_jobs(NofJob, Who, HimaPEs).                                %(6)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% who_is_idle_initially(Counter,PEs,BootPE,Config,Who)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% himada ゴールを最低の優先度で実行するための himaka  ゴールを，全てのプロセ
% ッサに投げる．その際，himada ゴールが実行されたことを返すストリームをマージ
% インして渡してやる．
%
% Counter : 使用するプロセッサ台数回のループを行うためのカウンタ．
% PEs     : 使用するプロセッサの台数．
% BootPE  : 使用を開始するプロセッサの番号．
% Config  : 実際のプロセッサ台数．
% Who     : himada ゴールが実行された時に，プロセッサ番号を返すためのストリーム
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
who_is_idle_initially(PEs, PEs, _, _, Who):- true | Who=[].          %(1)
otherwise.                                                           %(2)
who_is_idle_initially(C, PEs, BootPE, Config, Who):- true |         %(3)
    NC:=C+1,  Pro:= (BootPE+C) mod Config  ,                         %(4)
    merge(Who0, Who1, Who),                                          %(5)
    himaka(Who0)@processor(Pro),                                     %(6)
    who_is_idle_initially(NC, PEs, BootPE, Config, Who1).           %(7)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% himaka(Who) / himada(Who,MyPE)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% himaka ゴールは，優先度 100 で himada ゴールを呼び出す．その際にそのゴールが
% 実行されているプロセッサ番号を渡す．
% himaka ゴールは，実行されたらストリームにプロセッサ番号を入れてストリームを
% 閉じる．
%
% Who : himada ゴールが実行された時に，プロセッサ番号を返すためのストリーム
```

```
% MyPE: himada ゴールが実行されたプロセッサ番号
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
himaka(Who):- true |                                               %(1)
    himada(Who)@priority(*, 100).                                  %(2)
himada(Who):- true |                                               %(3)
    current_processor(MyPE, _, _),                                 %(4)
    Who=[MyPE].                                                    %(5)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% spawn_jobs(Counter,Who,HimaPEs)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 暇プロセッサマネージャから暇なプロセッサ番号を得，そこへ仕事を投げる．
% 負荷分散マネージャである．
%
% Counter : 分散した仕事の数を数えるためのカウンタ
% Who     : 仕事が暇になったら通知するためのストリーム
% HimaPEs : 暇なプロセッサ番号を得るためのストリーム
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
spawn_jobs(0, Who, HimaPEs):- true | Who=[].                       %(1)
spawn_jobs(C, Who, [PeNo|HimaPEs]):- true |                        %(2)
    NC := C-1 ,                                                    %(3)
    merge(Who0, Who1, Who),                                        %(4)
    spawn(C, Who0)@processor(PeNo),                                %(5)
    spawn_jobs(NC, Who1, HimaPEs).                                 %(6)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% spawn(Number,Who)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 負荷分散マネージャによって投げられた仕事の本体ゴール job 優先度 2000 で実行
% する．優先度は，負荷分散マネージャよりも小さくしなければならない．
% また，同時に himaka ゴールを実行し，その中で優先度 100 で himada ゴールを
% 実行する．
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
spawn(Number, Who):- true |                                        %(1)
    job(Number)@priority(*,2000),                                  %(2)
    himaka(Who).                                                   %(3)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% merge(Xs,Ys,Zs)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% マージャである．組込みマージャを用いても良い．
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
merge(Xs, [], Zs):- true | Zs=Xs.                                  %(1)
merge([], Ys, Zs):- true | Zs=Ys.                                  %(2)
merge([X|Xs], Ys, Zs):- true | Zs=[X|Zs0], merge(Xs, Ys, Zs0).     %(3)
merge(Xs, [Y|Ys], Zs):- true | Zs=[Y|Zs0], merge(Xs, Ys, Zs0).     %(4)
```

15.3 ストリーム通信あれこれ

本節ではストリーム通信に関する注意点ならびに代表的なプログラミングテクニックについて説明する.

15.3.1 Incomplete Message

Incomplete message とはメッセージ中に論理変数を含む場合を言い, KL1 における通信手段の基本である.

KL1 のような incomplete message を使ったストリーム通信の利点は, 相手との通信を行う順序が実行の順序で決まるのではなくメッセージをストリームに入れた順序で決まる点である. このような特徴によって, プログラマはプロセス間の同期にそれほど神経過敏になる必要がなくなった.

スタックプロセスに対して push(X) コマンドを送る場合で考えてみると, スタックにプッシュしようとする要素がまだ実際には生成されていない状況でも論理変数を使えば将来作られるデータをプッシュすることができる. プッシュされたデータを読み出す側は, もしその時点でまだデータがセットされていなければ, 自動的に中断されるので特に問題はない. 下の例では (2) のゴールの実行によって変数 X の値が決まるのであるが, (1) の push(X) はなんら問題は無い.

```
      ......,
Stack=[push(X)|NewStack],        % (1)
generate(X),                     % (2)
      ......,
```

このような性質は, 先にも述べたように, プログラマにたいして実行順序と同期の複雑な関係を考えることから開放するとともに, 自然な並列性をも引き出すことにつながる.

このスタックの例ではメッセージを送る側もその中の論理変数を具体化するのも同じプロセスと考えられるが, メッセージ中にそのメッセージに対する返信用の論理変数を埋め込んでおき, 受け取った側がその変数を具体化するという手法も非常に良く使われる. このような手法を back communication と呼ぶことがある. 通信のモデルとしてはメッセージを送るためのストリームと, 受けるためのストリームを 2 本張るのが自然な気がするかもしれないが, プロセス間のインタラクションにこのような独立した 2 本のストリームを張るのは実際にはあまり使われない. この理由は, 送り出したメッセージと受け取ったメッセージの間の依存関係が分かり難いからである.

次のプログラムは back communication の簡単な例である. (1) のユニフィケーションで他のプロセスにメッセージを送っているわけであるが、変数 Reply はこのメッセージに対する返信先を指定した変数であり, (2) のゴールでその返信を待っている.

```
process(Other) :- true |
    Other=[message(Reply)|NewOther],    % (1)
    wait_reply(Reply, NewOther).        % (2)
```

15.3.2 共有プロセスとマージ

KL1 では, 非常に大きなテーブルなど複数のプロセスから参照 (あるいは更新) されるデータ構造をプロセスとして実現し, 他の多くのプロセスからストリームを介して共有するといったプログラミングは非常に良く使われる. 勿論共有プロセスへのメッセージの入り口には多入力マージャが使われる. (図-15.9参照)

このようなマージャを介した共有プロセスに対するアクセスではマージャの持つ非決定性により, あるプロセスから送り出したメッセージの順序が必ずしも連続的に目的プロセスに到着しない点に注意する必要がある.

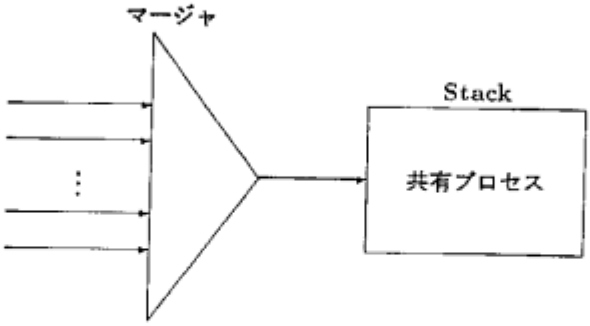


図 15.9: 共有プロセス

先のスタックの例で考えてみよう．例えば，あるプロセスがスタックの先頭の要素を取り出しその要素の値によって適当な別の値をプッシュする場合を考える．このプログラムを次のように書くと:

```
    ......,
Stack=[pop(Top)|Stack0],
check(Top, Stack0, NewStack),
    ......,

check(ready,   Stack,NewStack) :- true |
    Stack=[push(run)|NewStack].
check(suspend, Stack,NewStack) :- true |
    Stack=[push(idle)|NewStack].
```

メッセージを受け取るスタックプロセスには，このプロセスが送るメッセージ pop(Top) と push(run) (あるいは push(idle)) の間に他のプロセスからのメッセージが到着するかもしれないので，意図したとうりの結果が得られないかもしれない．このような取り出しと更新の操作を不可分にするためには 2 つのメッセージを合わせた

```
pop_and_push(Old,New)
```

のようなメッセージを用意する必要がある．ここでもメッセージを送る時点で変数 New の値は必ずしも定まっている必要がない点に注意して欲しい．

15.3.3 入力ストリームのマージ

多くのマージャは，次の例のように一本の出力ストリームを複数のゴールに分け与える場合に使われるが，ここでは別の使用例として入力ストリームを分岐させる例を示そう．

出力ストリームの分岐:

```
p(Stream) :- true |
```

```
Stream={S1,S2,S3},     % 出力ストリームを３本に増やす.
q(S1), r(S2), s(S3).   % それぞれのストリームを渡す.
```

例としてクライアント/サーバモデルで, サーバがクライアントからの要請によって新たなクライアントプロセスを生成する場面を考えて見よう. 生成されたクライアントは引数として自身の名前とサーバへのストリームを持つものとし, サーバは新たに生成されたクライアントへのストリームを自身のテーブルに登録するものとする. サーバへのクライアント生成依頼には new(Name) というようなメッセージを送るとする. ここで Name は生成されるクライアントの名前である. 次のプログラムはその非常に簡単なプログラム例である.

```
server([new(Name)|Command], Clients) :-
    true |
    merge(ToServer,Command,NewCommand),          % (1)
    client(Name,InStream,ToServer),
    server(NewCommand, [(Name,InStream)|Clients]).
```

生成されたクライアントが持つサーバへのストリームは, その時点でのサーバへの入力ストリーム Command とマージしたものになれば良いので, (1) で示したようなマージャを生成し, その入力の片方 ToServer を引き数として持てば良い. サーバ自身はそのマージャの出力 (すなわち NewCommand) が自身への入力ストリームとなり, これを引き数として持てば良い.

第 16 章

プログラミングテクニック (2)

16.1 要求駆動型プログラミング

要求駆動型プログラミングとは本来連続的に実行可能な処理を，その処理主体が無制限に続けるのではなく，実行することによって生成されるデータの受け手が新たなデータを生成する旨の要求を出すことによって間欠的に行うようなプログラミング技法である．

フィボナッッチ数列生成プログラムを例に取って説明しよう．例で示したプログラムは，ある与えた数よりも小さな数のフィボナッチ数が何個有るかを求めるプログラムである．

```
go(Target,N) :- true |
    fibonacci(0,1,Fs,Abort),
    consume(Fs,Target,0,N,Abort).

fibonacci(N1,N2,Ns0,abort) :- true | true.
alternatively.
fibonacci(N1,N2,Ns0,Abort) :- true |
    Ns = [N2|Ns1],
    N3 := N1 + N2,
    fibonacci(N2,N3,Ns1,Abort).

consume([X|Xs],Target,K,N,Abort) :- X<Target |
    K1 := K+1,
    consume(Xs,Target,K1,N,Abort).
consume([X|Xs],Target,K,N,Abort) :- X>=Target |
    N=K, Abort=abort.
```

このプログラムは，述語fibonacci/4が1から順次フィボナッチ数列を生成し，consume/5がその数列を生成順に消費するプログラムであるが，fibonacci/4の実行条件がconsume/5の実行とは無関係である点に注意して頂きたい．ここでKL1のゴールの実行順序に関する仕様を思い出すと，KL1では同一のプライオリティを持つ述語[1]同士の間の実行順序は特に規定していないので，例えば述語fibonacci/4の実行が終了するまでconsume/5の実行は行われないかも知れない．このことが何を意味するかと言うと，fibonacci/4が数を要素とするリストを生成し続け，最後には利用可能なメモリを全て使い尽くしてしまうかも知れないということである．[2]つまり上記のプログラムはフィボナッチ数を

[1] プライオリティに関してなにも指定していない場合その節のボディのゴールは全て同じプライオリティと見做される．

[2] 実際のマシン上でも十分このようなスケジューリングが起こる可能性はあることを特に付記しておく．

生成するという意味では正しいかもしれないが, 停止性が保障できないという意味では誤ったプログラムであると言えよう.

このような問題を避けるための方法として要求駆動型プログラミングが使える. 例えば数列の 1 要素を受け取るごとに次の数の要求を出すようなプログラムに書き換えてみると次のようになる.

```
go(Target,N) :- true |
    Buffer=[Box|Tail],
    finbonacci_lazy(0,1,Buffer),
    consume(Buffer,Target,0,N).

fibonacci_lazy(N1,N2,[N|Ns]) :- true |
    N = N2,
    N3 := N1 + N2,
    fibonacci_lazy(N2,N3,Ns).
fibonacci_lazy(N1,N2,[]) :- true | true.

consume([X|Xs],Target,K,N) :- X<Target |
    K1 := K+1, Xs=[Box|Tail],
    consume(Xs,Target,K1,N).
consume([X|Xs],Target,K,N) :- X>=Target | N=K, Xs=[].
```

このプログラムでは先のプログラムと違い, リストの生成者はフィボナッチ数を生成する側 (すなわち fibonacci_lazy/3) ではなく数列の消費者側に変わっており, 生成する側は次の数を入れるリストが容易されるまで実行を続けることができない. このように変更することによって必要なメモリ量も少なく抑えることができるわけである. またバッファを使って fibonacci_lazy/3 をコントロールしているため先のプログラムのような停止用の引き数も必要無くなった.

フィボナッチ数列の問題では例えばあらかじめ生成する数の最大値を与えておくなどして使用するメモリ量を抑えることも可能であるが, 一般にデータを生成する側のゴールの実行で最終的にどの程度のデータが生成されるか分からない場合にはこのような要求駆動型のプログラミングを行うことは必須であると言えよう.

16.2 バッファリング

要求駆動型プログラミングが重要であることは前章で述べたが, 本章ではこれに関連してバッファリングについて少し触れたい.

16.2.1 有限長バッファ

KL1 のプログラムはストリーム通信が基本となることはこれまでの説明で十分お分かり頂けたと思うが, この通信を考える上でメッセージやデータのバッファリングは, 処理効率を考えるなら忘れてはならないテクニックの一つである.

先の要求駆動で動くフィボナッチ数列生成プログラムは, 述語 consume がコンスセルを一つ与えると fibonacci_lazy が次の数を一つ生成し car 部にセットするというプログラムになっており, バッファ長が 1 であるようなストリームで通信していると考えることができる. しかしこれではせっかく並列言語を使っていても fibonacci_lazy と consume は交互にしか動けず速度の向上があまり期待できない.

そこで先のプログラムをバッファ長 2 に変えてみよう. 取り敢えずの解としては, 差分リストを使った次のようなプログラムが考えられよう.

```
go(Target,N) :-
    true |
    Buffer=[Box1,Box2|Tail],                    % (1)
    finbonacci_lazy(0,1,Buffer),
    consume(Buffer,Tail,Target,0,N).

fibonacci_lazy(N1,N2,[N|Ns]) :- true |
    N = N2,
    N3 := N1 + N2,
    fibonacci_lazy(N2,N3,Ns).
fibonacci_lazy(N1,N2,[]) :- true | true.

consume([X|Xs],Tail,Target,K,N) :- X<Target |
    K1 := K+1, Tail=[Box|NewTail],
    consume(Xs,Target,K1,N).
consume([X|Xs],Tail,Target,K,N) :- X>=Target |
    N=K, Tail=[].
```

こうすれば一応はバッファ長を 2 にすることができる. このプログラムでは最初に Buffer に与える Box の数 (プログラム中の (1) のユニフィケーション) を増やすことによってバッファ長は任意に大きくすることも可能である. このようにストリーム中に最大限溜め得るデータの数を制限したような通信方式をしばしば有限長バッファ通信と呼ぶ.

しかし実はここで示した有限長バッファのプログラムスタイルには幾つかの問題点がある. まずその一つはバッファに対する参照数に関する問題であるが, これについては次の単一参照の章で述べることにし, この章ではもう一方の問題点である通信量に関して考えてみよう. KL1 が実行される主な並列マシンとしてはマルチ PSI のような共有メモリを持たない疎結合マルチプロセッサが考えられるが[3], ここでいう通信量とはこのような結合形態でのプロセッサ間通信におけるデータ流量のことである.

まず現在のマルチ PSI での KL1 の実現方式に沿ってバッファ長 2 のフィボナッチ数生成プログラムを解析してみよう. 説明の都合により, ゴール fibonacci_lazy とゴール consume は異なったプロセッ

[3] PIM にはクラスタと呼ばれる共有メモリで結合された密結合のマルチプロセッサもあるが, クラスタ間についてはやはりマルチ PSI と同様の性質があるためここでは特に考えない.

サ上で実行されるものとする. 図-16.1は特にバッファの部分を図示したものである. Buffer の実体であるコンスセルは consume が実行されているプロセッサ (PE_j) のローカルなメモリに生成されており, fibonacci_lazy からはプロセッサにまたがるポインタによって参照されている.

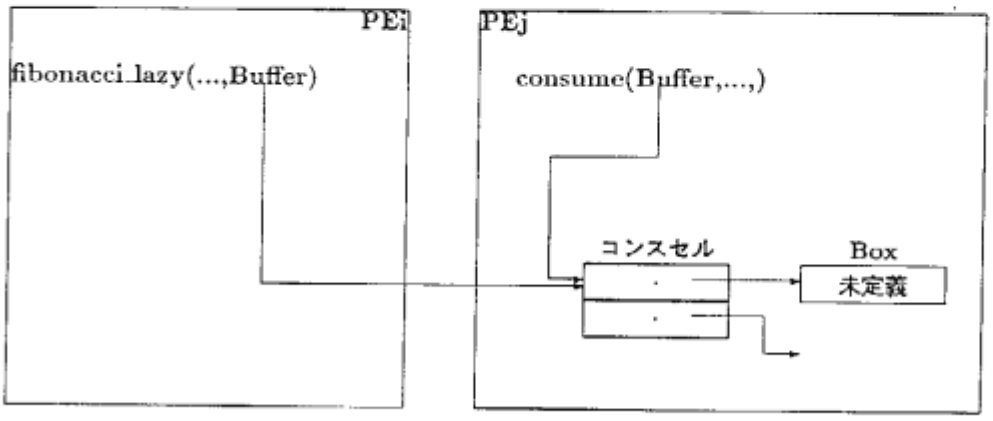


図 16.1: プロセッサ間における有限長バッファ (1)

このような状態で fibonacci_lazy が次のフィボナッチ数をバッファに入れようとすると:

1. まずコンスセルを一つ自身のプロセッサ (PE_i) にコピーし (図-16.2参照),
2. その car 部から指される変数セル Box1 と整数をユニファイする.

というような操作が行われる. ここで問題となるのは 1 の操作である. 本来プロセッサ間通信においても論理的にはストリーム中に流れるフィボナッチ数列のみが転送されることが望ましいはずであるが, 先に示したようなプログラムではストリームとして使われるコンスセルまでもがデータの流れとは逆方向に流れてしまうことになる. もちろんこのような事態は言語の実現方式にも依存するわけではあるが, どのような実現方式であれ何らかの望ましくないオーバヘッドが入ってしまうことは確かであろう. また fibonacci_lazy が必要とする時点でコンスセルを一つずつしかコピーしないため折角複数要素が入るバッファを用意したつもりでも, 結局は 1 要素ごとに通信が行われ, 通信のオーバヘッドが大きくなるとともに並列性をも損なう結果となってしまう.

さて, ここで示したようなオーバヘッドを避ける為にはどのようなプログラムを書けば良いのであろうか. 有限長バッファでの問題点はデータを要求する側のゴールがそのデータを収めるためのバッファをも用意している点だと言える. そこで今度はデータを要求する側では必要とするデータの個数のみを伝え, 要求された側で入れ物を用意するプログラムに書き変えてみよう.

```
go(Target,N) :- true |
    finonacci_lazy(0,1,Cntrl),
    consume(Cntrl,Target,0,N).

fibonacci_lazy(N1,N2,[gen(Size,Reply)|Con]) :- true |
    new_vector(Buffer,Size),
    fibonacci_lazy_loop(N1,N2,0,Size,Buffer,Reply,Con).
fibonacci_lazy(N1,N2,[]) :- true | true.
```

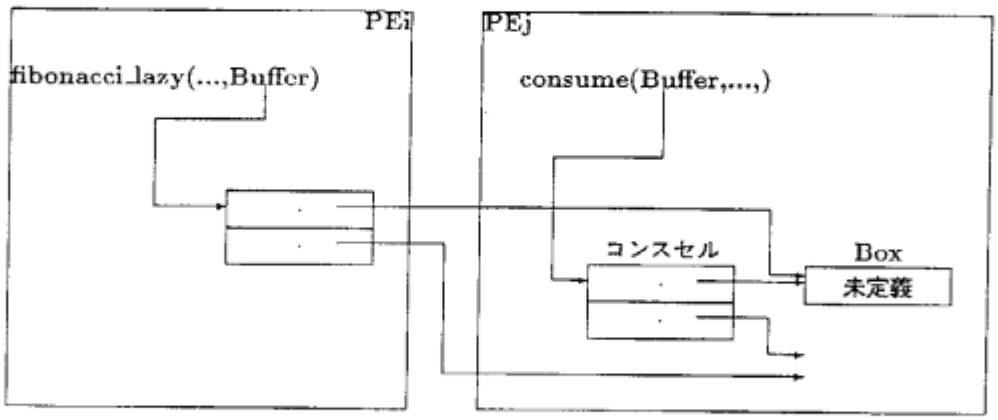


図 16.2: プロセッサ間における有限長バッファ (2)

```
fibonacci_lazy_loop(N1,N2,K,Size,Buffer,Reply,Con) :- K<Size |
    set_vector_element(Buffer,K,_,N2,NewBuffer),
    N3 := N1 + N2,
    K1 := K + 1,
    fibonacci_lazy_loop(N2,N3,K1,Size,NewBuffer,Reply,Con).
fibonacci_lazy_loop(N1,N2,K,Size,Buffer,Reply,Con) :- K>=Size |
    Reply=Buffer,
    fibonacci_lazy(N1,N2,Con).

consume(Fib0,Target,K,N) :- true |
    BufferSize=100,
    Fib0=[gen(BufferSize,FibVct)|Fib1],
    consume_loop(0,BufferSize,FibVct,Target,K,N,Fib1).

consume_loop(Bpt,BufferSize,FibVct,Target,K,N,FibReq) :-
    Bpt<BufferSize |
    set_vector_element(FibVct,Bpt,_,FibN,NewFibVct),
    Bpt1 := Bpt + 1,
    check(Bpt1,BufferSize,NewFibVct,Target,K,N,FibReq,FibN).
consume_loop(Bpt,BufferSize,_,      Target,K,N,FibReq) :-
    Bpt>=BufferSize |
    consume(FibReq,Target,K,N).

check(Bpt,BufferSize,FibVct,Target,K,N,FibReq,FibN) :-
    FibN<Target |
    K1 := K+1,
    consume_loop(Bpt,BufferSize,FibVct,Target,K1,N,FibReq).
check(_,  _,          _,      Target,K,N,FibReq,FibN) :-
    FibN>=Target |
    K=N, FibReq=[].
```

このプログラムでもバッファ自身がプロセッサ間でコピーされる点では変わらないが, 一度にベクタ全体がコピーされるため通信オーバヘッドは軽減され, 通信後の実行もスムーズになる.

次にマージャを使った例を示そう. この例では厳密な意味でバッファ領域を確保していることにはなっていないが, 要求駆動型のプログラミングであり, コンテキストスイッチを抑えるよう配慮されているという意味でバッファリングとは無関係ではないので比較のため付けることにする. ちなみにマルチPSIでは将来処理系レベルの最適化によりプログラム中の (1) で生成されるコンスセルがプロセッサ間でコピーされず, 要素であるフィボナッチ数のみが転送されるようになる予定である.

```
go(Target,N) :- true |
    merge(BufferIn, BufferOut),
    finonacci_lazy(0,1,Cntrl,BufferIn),
    consume(Cntrl,BufferOut,Target,0,N).

fibonacci_lazy(N1,N2,[gen(Size)|Cntrl],Buffer) :- true |
    fibonacci_lazy_loop(N1,N2,0,Size,Buffer,Cntrl).
fibonacci_lazy(_, _, [],              Buffer) :- true | Buffer=[].

fibonacci_lazy_loop(N1,N2,K,Size,Buffer,Cntrl) :- K<Size |
    Buffer=[N2|NewBuffer],                          % <---- (1)
    N3 := N1 + N2,
    K1 := K + 1,
    fibonacci_lazy_loop(N2,N3,K1,Size,NewBuffer,Cntrl).
fibonacci_lazy_loop(N1,N2,K,Size,Buffer,Cntrl) :- K>=Size |
    fibonacci_lazy(N1,N2,Cntrl,Buffer).

consume(Fib0,Buffer,Target,K,N) :- true |
    BufferSize=100,
    Fib0=[gen(BufferSize)|Fib1],
    consume_loop(BufferSize,Buffer,Target,K,N,Fib1).

consume_loop(Size,[FibN|Buffer],Target,K,N,FibReq) :- Size>=0 |
    Size1 := Size - 1,
    check(Size1,Buffer,Target,K,N,FibReq,FibN).
consume_loop(Size,Buffer,       Target,K,N,FibReq) :- Size<0 |
    consume(FibReq,Buffer,Target,K,N).

check(Size,Buffer,Target,K,N,FibReq,FibN) :- FibN<Target |
    K1 := K+1,
    consume_loop(Size,Buffer,Target,K1,N,FibReq).
check(_,   _,      Target,K,N,FibReq,FibN) :- FibN>=Target |
    K=N, FibReq=[].
```

16.2.2 ダブルバッファリング

一般に並列性を考えた場合バッファを一つだけ用意するのでは不十分である. データの消費者が一つのバッファを受け取って処理をしている間データの供給元がidle状態になってしまうし, 逆に供給元が次のデータを用意している間消費者側がidle状態になってしまう. このような状況を避けるためには,

バッファを受け取った時点で前もって次のデータを要求する旨を供給元に伝えておけば，消費者側も供給元も平行して実行することができるようになり，速度の向上が期待できる．

この様に，バッファ中のデータを全て使い切る前に次のバッファを用意しデータを生成しておく方式はダブルバッファリングと呼ばれ，OS の内部などでは良く使われているテクニックである．勿論，データの供給元や消費者側の実行速度の変化の具合によって適当にバッファのサイズや或いは前もって用意するバッファの数などを調節する必要はあり，場合によってはＮ段バッファリングが良いということもあろう．

```
go(Target,N) :- true |
    BufferSize=100,
    finonacci_lazy(0,1,[gen(BufferSize)|Cntrl],FibStr),
    consume(Cntrl,BufferSize,FibStr,Target,0,N).

fibonacci_lazy(N1,N2,[gen(Size)|Cntrl],FibStr) :- true |
    new_vector(Buffer,Size),
    fibonacci_lazy_loop(N1,N2,0,Size,Buffer,Cntrl,FibStr).
fibonacci_lazy(N1,N2,[],FibStr) :- true | FibStr=[].

fibonacci_lazy_loop(N1,N2,K,Size,Buffer,Cntrl,FibStr) :- K<Size |
    set_vector_element(Buffer,K,_,N2,NewBuffer),
    N3 := N1 + N2,
    K1 := K + 1,
    fibonacci_lazy_loop(N2,N3,K1,Size,NewBuffer,Cntrl,FibStr).
fibonacci_lazy_loop(N1,N2,K,Size,Buffer,Cntrl,FibStr) :- K>=Size |
    FibStr=[Buffer|NewFibStr],
    fibonacci_lazy(N1,N2,Cntrl,NewFibStr).

consume(Cntrl,BufferSize,FibStr,Target,K,N) :- true |
    Cntrl=[gen(BufferSize)|NewCntrl],
    consume_sub(0,BufferSize,FibStr,Target,K,N,NewCntrl).

consume_sub(Bpt,BufferSize,[FibVct|FibStr],Target,K,N,Cntrl) :-
    consume_loop(Bpt,BufferSize,FibVct,Target,K,N,Cntrl,FibStr).

consume_loop(Bpt,BufferSize,FibVct,Target,K,N,Cntrl,FibStr) :-
    Bpt<BufferSize |
    set_vector_element(FibVct,Bpt,_,FibN,NewFibVct),
    Bpt1 := Bpt + 1,
    check(Bpt1,BufferSize,NewFibVct,Target,K,N,Cntrl,FibStr,FibN).
consume_loop(Bpt,BufferSize,_,     Target,K,N,Cntrl,FibStr) :-
    Bpt>=BufferSize |
    consume(Cntrl,BufferSize,FibStr,Target,K,N).

check(Bpt,BufferSize,FibVct,Target,K,N,Cntrl,FibStr,FibN) :-
    FibN<Target |
    K1 := K+1,
    consume_loop(Bpt,BufferSize,FibVct,Target,K1,N,Cntrl,FibStr).
check(_,  _,          _,     Target,K,N,Cntrl,_,    FibN) :-
```

```
FibN>=Target |
K=N, Cntrl=□.
```

16.3 単一参照

16.3.1 MRB 方式とは

MRB とは Multiple Reference Bit の略であり．ここで言う MRB 方式とはこの 1 ビットのフラグを使った一種の参照カウント方式のことである．MRB 方式を簡単に説明すると:

> 参照がひとつしかないこと (すなわち単一参照であること) が判っているデータと，複数の参照があるかもしれないデータを区別し，前者についてそのデータに対する処理の効率化を行う方式

であると言えよう．言い換えるとデータに対する参照がただ一つかどうかを表すのが MRB であり，この MRB は参照されるデータや参照するためのポインタに付加されている．

説明の都合上，以下ではあるデータへの参照数が複数の可能性がある場合そのデータを MRB 黒と呼び，そのようなポインタを黒ポインタと呼ぶ．また同様に単一参照である場合のデータを MRB 白と言いポインタを白ポインタと呼ぶことにする．

例を示そう．まず次のようなゴールを考える．

ゴール: `?- p(X), q(X).`

ここで変数 X はゴール p(X) 及び p(X) の引き数として使われており，参照数が 2 の状態である．述語 p/1 の定義として節-1 のようなプログラムがあると，節-1 が選択された後 (すなわちコミットした後) ではボディ部の 2 つのゴールに変数 X へのポインタが渡されるため参照数が 3 に増えることになる．

節-1: `p(X) :- true | r(X), s(X).`

上記までの説明に従えば，ゴール p(X) が指す X へのポインタも，節-1 の実行後に生成されたゴール r(X) と s(X) が指すポインタも黒ポインタということになるが，実は未定義変数へのポインタだけは特別に扱われなければならない．次の例を考えて見よう．

```
?- generator(X), consumer(X).

generator(X) :- true | X=[tick|Y], generator(Y).
consumer([tick|X]) :- true | consumer(X).
```

このプログラムは generator(X) が生成したリストを consumer(X) が文字どうり消費するプログラムであるが，この例のように一般には一つの未定義変数を参照するゴールは，データを書き込むゴールと読み出すゴールの 2 つが存在するのが普通である．このことから，未定義変数へのポインタだけは，その変数セルへのポインタが 2 つ以下であることが明白な場合に MRB 白 (白ポインタ) であるとし，3 つ以上のポインタが有るかも知れない場合には MRB 黒 (黒ポインタ) であるとする．まとめると:

CASE-1: 未定義変数へのポインタは，その変数への参照が 2 つ以下であることが明白な場合に MRB 白．3 つ以上かも知れない場合には MRB 黒．

CASE-2: 具体値へのポインタは，そのデータへの参照が 1 以下であることが明白な場合は MRB 白とし，2 以上かも知れない場合は MRB 黒とする．

『明白な』とか『かも知れない』といった婉曲な表現が奇異に感じられるかも知れないが，それは MRB 方式という参照数管理方式が，基本的に一度でも MRB 黒になった参照に関して，たとえその後の実行で参照数が減少しても MRB 白に戻すことをしていないためである．ただし，この MRB の白から黒への変更はリダクションの完了時点 (すなわちコミットバーを越えた時点) での参照数の増減で決定される点を注意する必要がある．もっとも KL1 の実行がリダクションを基本としており，ガード部の実行が逐次的に実行されることを考えればこの増減の考え方は自然である．

もう一度先の例で考えて見よう．ゴールの起動時には変数 X は未定義変数であり，参照数が 2 であるので p(X) 及び q(X) からのポインタは白ポインタである．この後，節 -1 の実行によって参照数が 3 に増え，r(X) 及び s(X) からのポインタは黒ポインタに変わる．節 -2 の実行 (正確にはボディユニフィケーション X = a の実行) によって X へのポインタは消費され参照数が実際には 2 に減少するが減少の場合の管理を行っていないため，ゴール s(X) と q(X) の変数 X へのポインタは相変わらず黒ポインタのままである．

```
ゴール:  ?- p(X), q(X).

節-1:        p(X) :- trupe | r(X), s(X).
節-2:        r(X) :- true | true.
```

構造体を扱ったもう少し複雑な例を考えて見よう．次に示すプログラムでは，引き数として 3 要素のベクタを持つゴールを与えるとする．上記までの説明から，ベクタ中に現れる変数 X,Y,Z はいずれも参照数 2 であり，MRB 白であることは容易に理解できるであろう．

```
ゴール: ?- p({{a,b},{1,2}}

節-3:       p(Vect) :-
                vector_element(Vect,1,Ve1) | q(Ve1).
節-4:       p(Vect) :-
                vector_element(Vect,1,Ve1) | r(Ve1,Vect).
```

節 -3 及び節 -4 のガード部にある組み込み述語 vector_element(V,1,Ve1) の実行直後の内部的な構造を図に表すと図 16.3のようになる．ベクタの第 2 要素であるベクタ {1,2} へのポインタがこの時点で 2 本であることに注意されたい．

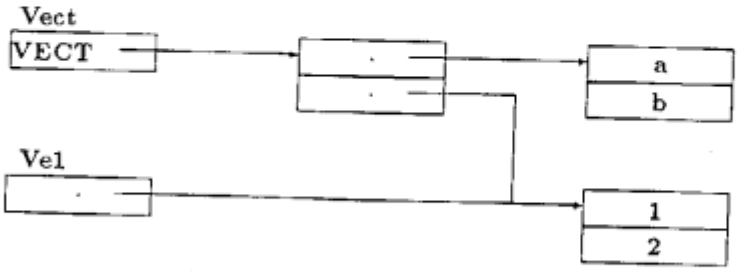


図 16.3: 構造体と要素への参照

この後，もし節 -3 がコミットされてゴール q(Ve1) が実行されるとすると，変数 Ve1 から指す {1,2} へのポインタは，Ve1 からのポインタとゴール q(Ve1) が持つベクタからのポインタの計 2 本であり，結

局 {1,2} への白ポインタを持って q(Ve1) が実行されることになる.[4]

一方, 節-4 がコミットされてゴール r(Ve1,Vect) が実行されるとするとベクタへのポインタがコミット後も残り, 結局変数 Ve1 は {1,2} への黒ポインタを持って実行されることになる.

16.3.2 set_vector_element の真実

ボディ部で使える組み込み述語に set_vector_element というのがある. この組み込み述語は 5 引数で次のような仕様であった.

```
set_vector_element(Vect,Position,OldElement,NewElement,NewVector)
```

ベクタ Vect の第 Position 番目の要素を OldElement とユニファイし, かつ Vector の第 Position 番目の要素を NewElement で置き換えた新しいベクタを生成し NewVector とユニファイする.

もう既に初級編でも述べられているが, set_vector_element を使ってベクタの要素を更新する度に新たなベクタを作っていると, メモリの消費速度も早過ぎるが, それにも増して実行速度が問題になってしまう. そこで MRB を利用した処理系では, set_vector_element の実行時に更新前のベクタを参照しているゴールが当の set_vector_element のみである場合には (すなわち白ポインタで参照している場合), 新たなベクタを生成せずに元のベクタの指定されたポジションを直接書き換えることによってメモリ使用効率及び速度効率を上げている.

参照と更新操作を一つの組み込み述語にまとめてあるのも実は参照数の問題と強く関係している. これは, 要素を参照する操作と更新する操作が別の組み込み述語に分かれている場合を考えれば容易に理解することができる. 例えば, 参照用の組み込み述語を get_element/4 として以下の例を考えてみよう.

```
?- p({X,Y,Z}), q({X,Y,Z}).

   p(Vect) :- true |
        get_element(Vect,2,Elem,NewVect),
            ......
```

このプログラムで get_element(Vect,2,Elem,Vect1) が実行されると Elem が指す変数 Z への参照は 3 本になり所謂黒ポインタになってしまう.

16.3.3 その他の効用

前章では, MRB を使った参照数の管理を行うことによってベクタの更新操作が効率良くできることを述べたが, 本章ではそれ以外の MRB の効用について少しだけ説明する.

まず入門編で説明したスタックプログラムを思い出して欲しい. 以下の節定義はそのスタックプログラムの一部である.

```
stack([push(X)|S], Stack) :- true |
    stack(S, [X|Stack]).
```

このプログラムをメモリの消費及び参照数の増減という観点で眺めてみると幾つかのことが分かる. まず, ヘッドにある [push(X)|S] というパターンはヘッドユニフィケーションによって対応するゴールの第一引き数のコンスセルへのポインタと push というファンクタを持つ構造体へのポインタをそれぞれ 1 本消費していると読める. またボディ部の [X|Stack] なるパターンはコンスセルを一つ生成しその

[4] ベクタへのポインタが消費され, ベクタ経由の {1,2} へのポインタが 1 本無くなっている点に注意. 先にも述べたように, 白から黒への変更はコミット前とコミット後の参照数の増減で決まる.

セルへの白ポインタをゴールに渡していると考えられる．ここで，ヘッドユニフィケーションで消費したコンスセルへのポインタが白ポインタであった場合は，消費することによってもう誰もそのセルを参照することがないのでボディ部で必要なコンスセルとして再利用できることになる．一方，同様に push(_) へのポインタが白ポインタである場合はこの領域は後で再利用するためにフリーリストと呼ばれる未使用領域のチェインに繋がれ，同様の大きさの領域が必要になった時点でこのリストから外され再利用される．このように不必要になった時点でその領域を回収することによってそのプログラムを実行するために必要なワーキングセットを小さくすることができるわけである．

16.3.4 単一参照性を保つには

ここまでの説明で，データへの参照を単一にしておくことによって多くのメリットが得られることがお分かり頂けたと思う．ではどのようなプログラムを書けば単一参照性を保つことができるのであろうか．以下ではこのための幾つかの注意点と代表的なプログラミングスタイルについて説明する．

単一参照性を保つための基本的な考え方は:

> 同じ構造体を使用したいサブモジュールが複数ある場合，一つのモジュールで読み終わってから次に渡すようにする．

であろう．このようなプログラムの構造はおおよそ次のような形をしている．

```
p(Table, ...) :- true |
    q(Table,Table1, ...),
    r(Table1,Table2, ...).
    s(Table2, ...).
```

Table で受け取ったデータを，不必要になった時点で変数 Table1 にユニファイし次のゴールに渡すといったことを繰り返すプログラムであるが，このように 2 つの変数を使って次から次へと渡してゆく形式を取り敢えずバケツリレーと呼ぶことにしよう．

このバケツリレー的なプログラミングの欠点は処理が逐次的になりがちであるということである．例えば上の例で，Table を必要とするそれぞれのゴールが参照しかせず，たとえ更新処理をしても互いに更新されたデータには影響されないような場合，各ゴールの実行がリレーされてくる Table の到着するまで待たされるとするといかにもばからしい．バケツリレーで単一参照を保つか，複数参照にはなるが並列性を重視するかはプログラムによって大きく変わるので，ここでは特にどちらが良いとは言えないが，とりあえず次の点に注意して適宜決めて行けば良かろう．

- 構造体の要素を更新するか．するとすればその頻度はどの程度か．また大きさはどの程度か．
- 共有しようとするゴール間で構造体に対する依存関係はあるか．
- 各ゴールの処理量はどの程度か．またその処理量と構造体のコピーする手間との比はどの程度か．
- 構造体をどの程度ダイナミックに生成 / 放棄するか．

16.3.5 プログラム例

変数を下のような構造体で表現した場合に，そのような表現の変数を含むような項 (このような項を *frozen term* と呼ぶ) を入力として，含まれる全ての変数表現を KL1 におけるいわゆる論理変数に変換した項 (これを *melted term* と呼ぶ) を生成するプログラムを示す．

`'$VAR'(VarNumber)` :

識別番号が VarNumber であるような変数を表す．ただし，その項の中で変数名が同一であるような変数については同じ識別番号が与えられる．

例えば, このプログラムに次のような *frozen term* を入力すると:

入力: f('$VAR'(0),'$VAR'(1),'$VAR'(0))

出力として以下のような *melted term* が得られる。

出力: f(A,B,A)

このプログラムの基本的なアルゴリズムは次に示すとうりである.

1. 対象となる項を外側から順々にコピーする.
2. もしコピーしようとする項が変数表現であった場合には新たな変数を一つ生成し,
3. その変数をコピー結果とするとともに識別番号を使って変数表に登録する.
4. 項を全てコピーし終わったら, 変数表に登録された各変数の同じ識別番号で登録された変数を全てユニファイする.

変数表に登録する場合には一旦別の変数として登録しておき, コピー終了時点で全体をユニファイしているのは, 同一の変数が 2 個までのものについて MRB を白に保つよう配慮しているためである.

```
%%%  melt/3  %%%
%
%       第1引数: 与えられた frozen term
%       第2引数: 与えられた frozen term 中に含まれる変数の種類数
%       第3引数: 生成された melted term
%
  melt(Ft, 0, Mt) :- true | Mt=Ft.
  melt(Ft, Vn, Mt) :- Vn>0 |
      new_vector(VarTbl, Vn),
      set_empty(0,Vn, VarTbl,VarTbl1),
      melt_term(Ft,Mt, VarTbl1,VarTbl2, Vn),
      unify_variables_table_entry(0,Vn, VarTbl2).

  set_empty(N, N, V, VV) :- true | VV=V.
  set_empty(K, N, V, VV) :- K<N |
      K1 := K+1,
      set_vector_element(V, K, _, [], V1),
      set_empty(K1, N, V1, VV).

  melt_term(X,Y, Vt,Vt1, Vn) :- atomic(X) | Y=X, Vt1=Vt.
  melt_term([Car|Cdr],List, Vt,Vt2, Vn) :- true |
      List=[Ncar|Ncdr],
      melt_term(Car,Ncar, Vt,Vt1, Vn),
      melt_term(Cdr,Ncdr, Vt1,Vt2, Vn).
  melt_term('$VAR'(N),Y, Vt,Vt1, Vn) :- N>=0, N<Vn |
      enter(N,Y, Vt,Vt1).
otherwise.
  melt_term(X,Y, Vt,Vt2, Vn) :- vector(X,Size) |
      new_vector(V, Size),
```

```
        melt_args(0,Size, X,V, Vt,Vt2, Vn, Y).

    melt_args(N,N, _,V, Vt,Vt2, Vn, Y) :- true | Y=V, Vt2=Vt.
    melt_args(K,N, X,V, Vt,Vt2, Vn, Y) :- K<N |
        K1 := K+1,
        set_vector_element(X,K,Xk,_,X1),
        melt_term(Xk,Vk, Vt,Vt1, Vn),
        set_vector_element(V,K,_,Vk,V1),
        melt_args(K1,N, X1,V1, Vt1,Vt2, Vn, Y).

    enter(P,V, Vt,Vt1) :- true | set_vector_element(Vt,P,E,[V|E],Vt1).

    unify_variables_table_entry(N,N, _) :- true | true.
  otherwise.
    unify_variables_table_entry(K,N, Vt) :- true |
        K1 := K+1,
        set_vector_element(Vt,K,Vk,_,Vt1),
        unify_elements(Vk),
        unify_variables_table_entry(K1,N, Vt1).

    unify_elements([X]) :- true | true.
    unify_elements([X,Y]) :- true | X=Y.
  otherwise.
    unify_elements([X,Y|Z]) :- true | X=Y, unify_elements([Y|Z]).
```

付録 A

PDSS の使い方

この章では, PDSS の基本的な使い方を説明する. 詳細な使い方については, マニュアル [2] を参照の事. なお, ここでは *icot21* 或いは *icot22* 上での操作を説明する.

A.1 起動の仕方

PDSS は, エディタから起動する. 以下, エディタを起動する所から説明する.

1. エディタを起動する. *shell* から次のようなコマンドを入力する.

```
daresore@icot21[0]% nemacs
```

2. するとエディタの画面が表示される.

3. 次のようなエディタのコマンドを入力して PDSS を起動する.

Meta-X pdss *return*

4. するとウィンドウが 2 つに分割される. それぞれのバッファ名を覚えておくと, 後々役立つ.

   - PDSS-SHELL
     PDSS のトップレベルのウィンドウである. プログラムのロードや, プログラムの実行を行なうためのコマンドを入力する.
   - PDSS-CONSOLE
     プログラムの実行中のエラーを表示したり, トレース出力を行なうウィンドウである.

これで PDSS が起動された.

A.2 プログラムの作成とその実行

まず, プログラムの作成とその実行の一般的なサイクルを説明する.

プログラムの作成とコンパイル
プログラムの作成は, エディタで行なう. 作成したプログラムはまずコンパイルしなければならない. コンパイルは, PDSS を起動した時と同様にエディタののコマンドを入力して行なう.

プログラムのロード
コンパイルを終えると, 次にプログラムをロードする. PDSS-SHELL 上でコマンド (ゴール) を入力する事により行なう.

実行
ロードを終えたら, 今度はゴールを呼び出す. PDSS-SHELL 上で行なう.

デバッグ

プログラムのデバッグは，トレーサを用いて行なう．PDSS-SHELL 上で，各種のデバッグモード等を切り替える事によって起動する．トレース出力は PDSS-CONSOLE にされる．トレーサのコマンドもこのウィンドウから入力する．バグを発見したら，プログラムをエディットしてコンパイルからやり直す．

次に，各処理における操作方法の説明を行なう．

1. プログラムの作成とコンパイル
   エディター上でプログラムを作成する．なお，ファイル名には拡張子.kl1 を付けておく事．このプログラムをコンパイルするには，そのウィンドウ上で次のようなコマンドを入力する．

   Control-c Control-c

   すると PDSS-COMPILER ウィンドウが現れて，そのプログラムがコンパイルされている事を表わすメッセージが表示される．エラー等があった場合には，このウィンドウにエラーメッセージが表示される．その場合はプログラムを修正した後に再度コンパイルをする．正常にコンパイルされた時には，次のようなメッセージが表示される．

   ```
   PDSS KL1 Compiler:  Success
   ```

2. プログラムのロード
   プログラムのロードは，PDSS-SHELL からのゴール入力にて行なう．その際，プログラムのファイル名を入力するが，ファイル拡張子の部分は入力してはいけない．

   ?-load(test).*return*

   すると，次のようなメッセージが表示される．

   ```
   test :  loading...  done
    module name :  reidai
   ```

3. 実行
   実行は，PDSS-SHELL ウィンドウでゴールを入力する事により行なう．

   ?-reidai:append([1,2],[2,3],V) | V.*return*

   実行中に発生したエラーは，PDSS-CONSOLE に表示される．

4. デバッグ
   プログラムのデバッグを行なうには，トレーサを用いる．PDSS-SHELL から次のようなゴールを実行する事によって，トレーサを起動する．
   次の例は，モジュール reidai のトレースを行なう時の例である．

   ?-trace(reidai).*return*

   この後，このモジュールのゴールが実行された時にはトレースされる．その表示は PDSS-CONSOLE ウィンドウに出力される．

   また，特定のモジュールの特定のゴールだけトレースしたい場合は，次のようなゴールを実行する．

   ?-spy(example,go,0).*return*

ここでexampleはモジュール名である．goは述語名である．0は引数の数である．

トレースをやめるには，次のようなゴールを入力する．

?-nodebug.*return*

なお，トレーサの操作方法についてはマニュアル[2]を参照の事．

5. 終了

PDSSを終了するには，次のようなゴールをPDSS-SHELLから入力する．

?-halt.*return*

また，操作していて何か変な状態になった場合は，エディタのコマンドでエディタ自身を終了させても良い．

Control-x Control-c

参考文献

[1] 淵一博監修, 並列論理型言語 GHC とその応用 共立出版

[2] ICOT 第四研究室, 1988 年 1 月 5 日, PDSS 使用手引き (第 1. 64 版)

[3] ICOT PIMOS 開発グループ, 1989 年 4 月 19 日, PIMOS 使用の手引き