

KLIC 講習会テキスト

KL1 言語編

平成 5 年 12 月 財団法人 新世代コンピュータ技術開発機構 作成
平成 7 年 9 月 財団法人 日本情報処理開発協会開発研究室 改訂

はじめに

本編は KLIC 講習会のために作成した KL1 言語の入門テキストである。

KL1 言語について、まったくの白紙状態からの入門から、若干複雑なプログラムを組めるようになるための基本テクニックまでを解説した。

講習会テキストとしては、別に「KLIC システム編」がある。こちらは KLIC システムの具体的な使い方を解説したものである。本編と合わせて御利用願いたい。

種々のプログラム・テクニックの解説には演習問題を付け加えるようにした。巻末に解答例を掲げたが、そこに与えた解答がベストとは限らない。理解を深め、また確認するために、これらの演習問題はぜひ自分で解いてみた上で、掲載した解答例と比較検討されることをお奨めしたい。

なお一部の内容は 1989 年に作成された新世代コンピュータ技術開発機構による”KL1 プログラミング入門編 / 初級編 / 中級編”より引用した。

1993 年 12 月

ICOT KLIC 開発グループ

1995 年 9 月

JIPDEC KLIC 保守普及グループ

目次

1	KL1 入門	1
1.1	はじめに	1
1.2	構文と実行機構の基本	2
1.2.1	プログラムの形式と基本実行機構	2
1.2.2	他のプログラム言語と比べて	6
1.3	データとシンタクス	7
1.3.1	論理変数	7
1.3.2	アトミックなデータ	8
1.3.3	構造を持ったデータ	9
1.3.4	略記法	10
1.4	プロセスとストリーム通信	11
1.4.1	プロセス	11
1.4.2	ストリーム通信	13
1.5	プロセス・ネットワーク	15
1.5.1	フィルタ	16
1.5.2	ストリームの連結	17
1.5.3	マージャ	18
1.5.4	ディスパッチャ	19
1.5.5	サーバ	20
1.6	プログラム動作の指定	20
1.6.1	並列実行指定の方針	20
1.6.2	ゴール分散プラグマ	22
1.6.3	ゴール優先度指定プラグマ	22
1.6.4	節優先度指定プラグマ	22
1.7	一階述語論理と KL1	22
1.7.1	プログラムは公理の集まり	23
1.7.2	実行は証明過程, 計算結果は反例	24
1.7.3	証明系としての Prolog と KL1	24
2	プロセス・ネットワーク	26
2.1	基本的なプロセスとプロセス・ネットワークの復習	26
2.1.1	木構造	26
2.1.2	要求駆動とサーバプロセス	28
2.2	解の回収と計算終了判定	30
2.3	プロセス・ネットワーク構築上の注意点	30
2.3.1	入出力を分ける	30
2.3.2	各ノードの役割は単純に	31
2.3.3	部分木は直接通信させない	32
2.3.4	ループは避ける	33

2.3.5	カオスなネットワークはやめよう	33
2.3.6	マージャ	33
2.4	やや高度なネットワークの組み方	34
2.5	演習問題	35
3	差分リストの使い方	39
3.1	クイックソートの原理	39
3.2	差分リスト	42
3.3	ショートサーキット	43
3.4	出力ストリーム	44
4	優先度と負荷分散	47
4.1	優先度制御の利用目的	47
4.2	基本プログラミング	48
4.2.1	プログラムの実行とスケジューリング	48
4.3	KL1 のゴール優先度指定方法	51
4.3.1	再び生産者と消費者の問題	51
4.3.2	ゴール優先度指定の文法	53
4.3.3	優先度指定使用上の注意	53
4.4	もうひとつの優先度	54
4.5	応用: 幅優先評価の実現	55
4.5.1	すべてのノードを評価する方法	56
4.5.2	不要なノード評価を始めない方法	56
4.5.3	不要なノード評価を打ち切る方法	56
4.5.4	優先度を用いて幅優先に評価を行なう方法	62
4.6	応用: 簡単な探索問題	62
4.7	負荷分散制御の利用目的	68
4.8	ゴール分散の指定方法	69
4.9	応用: 簡単な探索問題	71
5	総合演習問題	75
A	演習問題の解答例	78

第 1 章

KL1 入門

KL1 は並列論理型言語 Guarded Horn Clauses^[5] に基づいて設計された、並列処理の記述に適したプログラム言語で、第五世代コンピュータプロジェクトがその目標としてきた並列推論システムの中核となる並列推論マシンの共通核言語として、オペレーティング・システム^[1] から種々の応用プログラムの記述にまで、広く用いられてきた。^[6] この章では KL1 の言語仕様の概略と、基本的なプログラミング技法について解説する。

1.1 はじめに

KL1 という言語の特徴をひとことで言うならば記号処理を並列に行なうための言語ということになる。

数値処理にくらべて記号処理では、複雑に絡み合うデータを取り扱う必要があることが多い。このため、こうしたデータをどのようなデータ構造で表現するかが重要になる。また、そうしたデータ構造をどのようにメモリ上（あるいはディスク装置上）で管理するかも問題である。こうした管理にプログラマが多くの労力を割かずに済むように、言語システムで基本的な機能を用意して、もっと本質的なプログラミングに集中できるようにしよう、というのが記号処理言語の考え方である。具体的には、一意性のあるものに名前をつけると自動的に一意な表現を与えてくれる記号アトムの機構や、標準的なデータ構造についてそのためのメモリ割り付けや解放の労力を減らしてくれる自動メモリ管理機構などが、代表的な特徴である。この点、KL1 は代表的な記号処理言語である Lisp 一族と同様の機能を提供している。

並列処理のためには、全体の処理を複数の部分処理に分割して、必要なところでは部分処理間の同期を取りながら計算を進める必要がある。このために、逐次処理言語に並列実行を指定する機構と同期のための機構を追加し、並列処理にも使えるようにした言語（あるいは OS の機能まで含めたシステム）は数多い。

こうしたアプローチにはふたつの大きな問題点がある。ひとつは、もともとの言語の設計の原則は逐次処理のままなので、並列に実行できる部分をいちいち指定しなければならないことである。このため、同じプログラムをプロセッサ台数の大きく異なるハードウェアで共通に使い、しかも効率良く動かすようにすることが難しく、ハードウェアごとにプログラム全体をかなり書き直す必要が生じる。まして、どのような並列処理が適当なのか良くわからない問題について、実験を積み重ねながら並列処理の仕方を模索していくような場合、そのたびにプログラムを書き直してデバッグし直すことになり、多大な労力が必要になる。もうひとつは同期処理の面倒さである。同期の必要性を意識しながらのプログラミングは非常に厄介であるし、同期にバグが入ると、そのバグがどのように表面化するかに再現性がない（実行するたびに違う現象が起きる）ことが多いので、デバッグは非常に困難になる。

KL1 は逐次処理に並列実行のための機構を付加するという方法で設計した言語ではない。最初からすべてが並行動作することを前提とした言語である。こうすると非常に頻繁な同期が必要になるのだが、データフロー同期機構を導入して同期を自動化することによって、プログラマによる同期の誤りが入り込む可能性を排除している。物理的な並列実行の指定は別途プラグマと呼ばれる記述によって行なう。プラ

グマはプログラムの正当性¹を変えないように設計してあるので、並列処理の仕方を変えるたびにデバッグし直す必要がない。

こうした KL1 の特徴のおかげで、並列処理ソフトウェアの研究開発の労力は非常に軽減される。同じプログラムのプラグマ部分を変更するだけでさまざまな並列実行の仕方を指定できるので、いったんバグを取ってしまえば並列実行指定を変えるたびにバグに悩まされる必要はない。このため第五世代コンピュータプロジェクトでは、並列処理のもっとも困難な課題である負荷分散方式の研究などを、数多くの実験を行ないながら実証的に進めることができたわけである。

1.2 構文と実行機構の基本

まず KL1 のプログラム言語としての機構の概要を説明することにしよう。

1.2.1 プログラムの形式と基本実行機構

最初に KL1 のプログラムの形式と、基本的な実行機構について、簡単な例を用いて解説する。

簡単なプログラム

もっとも簡単な KL1 プログラムの例として、入力の 0, 1 を反転して出力するインバータを考えてみる。プログラムは以下のように書ける。

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

この二行のプログラムは、それぞれ:

- もし not の第一引数 In が 0 だったら、第二引数 Out は 1 にする
- もし not の第一引数 In が 1 だったら、第二引数 Out は 0 にする

と読む。このふたつの行をそれぞれ節 (clause) と呼び、KL1 のプログラムの基本単位である。

それぞれの節の冒頭部分にある “not(In, Out)” は、この節が述語 not に関する定義であること、引数はふたつで、この節の中ではそれぞれ In, Out と呼ぶことを宣言している。この部分を節のヘッド (head) と呼ぶ。述語という名前は KL1 の論理型言語としての生い立ちから来るのだが、ここでは単に手続き、あるいはサブルーチンのことだと思ってさしつかえない。

ヘッドに続く “:-” の後、縦棒 (“|”) の前までをガード (guard) と呼ぶ。ガードはその節を選んで良いかどうかの条件を指定する条件部である。

縦棒の後、フルストップ (“.”) をボディ (body) と呼ぶ。ボディはその節を選んだときに何をするかを指定する部分である。ボディにはいろいろなものを書けるが、この例では出力用の引数の値を決める操作である “Out = 1” などを行なっている。

プログラムの実行

KLIC で前掲のインバータのプログラムを実行するには、メイン・プログラムとしての体裁を整えなければならない。たとえば以下のようなものを用意する。

```
:- module main.  
  
main :- not(1, X), io:ostream([print(X),nl]).  
  
not(In, Out):- In = 0 | Out = 1.  
not(In, Out):- In = 1 | Out = 0.
```

¹ 正確には停止性を除いた部分正当性。

最初の行で、これがメインプログラムのモジュールだということを宣言している。次の main から始まる行がメインプログラムの本体である。そこには not という先ほど見たプログラムを呼び、その結果を書き出す、ということが書いてある。書き出すための io:outstream 云々というのは、ここでは無意味なおまじないだと思っておいてよい。

ここでは not の引数に 1 と X を与えた。整数 1 は普通の整数値 1 を表す。KL1 では大文字で始まる名前は変数を表す。ここでは他にはまだどこにも出てこなかったまっさらの変数 X を第 2 引数に渡したわけである。

このプログラムのうち、not の呼び出しの部分の実行は以下ようになる。

1. 第 1 引数の In は会話的な呼び出しの時に与えた引数 1 に対応する。これは、ふたつあった節のうち、前の方の節のガードに書いた選択条件 “In = 0” は見たさないが、後の節の条件 “In = 1” は満たしている。そこで、この後の方の節を選ぶ。
2. 節が選ばれたので、そのボディを実行する。ボディでは第 2 引数 Out (ここでは会話的に呼び出した時に書いた X に対応づけられている) の値を 0 と決めている (“Out = 0”)。
3. 他にすることはないので、これで実行を終わる。

この実行の結果、引数に与えた X の値が 0 に決まったので、それが出力され、プログラムはつつがなく終了することになる。²

KL1 の変数は C のような手続き型言語でいう『変数』とは大きく異なる。手続き型言語では『変数』は値の格納場所であって、計算の進行に伴って格納されている値は 0 になったり 1 になったりと変化する。KL1 の変数はいっと数学でいう変数に近く、値は決まっていなかったり決まっているかのどちらかで、いったん値を決めたら後で変わることはない。

最初に与える第一引数を 0 に変えれば、当然出力されるのは 1 になる。

ユニフィケーション

前掲のインパータのプログラムの中の “In = 0” や “Out = 1” のような、等号 (“=”) の両辺に値を書いた形のゴールをユニフィケーション (unification) と呼ぶ。等号を用いているように、これは両辺の値が等しいという意味だが、縦棒の左のガードにある “=” は条件指定、右のボディにある “=” は値の決定と、同じ記号でも出てくる場所で意味が違う。C 言語ならそれぞれ “==” と “=” にあたるようなものである。

ガード・ユニフィケーション 節の選択条件になるガードでのユニフィケーションは、両辺が等しいことが条件の一部であることを示すものである。これは単に等しいかどうかを試すだけなので受動的なユニフィケーション (passive unification) とも呼ばれる。前掲の例の “In = 0” では In が 0 かどうかを調べているわけである。

ボディ・ユニフィケーション 実行を指定するボディでのユニフィケーションは、両辺を等しいものにするという積極的な操作で、能動的なユニフィケーション (active unification) とも呼ばれる。前掲の例の “Out = 1” では、まだ値の決まっていなかった Out の値を 1 に決めてしまうことによって、両辺を積極的に等しいものにしてしまっているわけである。

節の形式と基本実行機構

プログラムを定義する節は、一般には以下のような形をしている。

述語名 (引数, ...) :- ガード | ボディ.

それぞれの部分の役割は以下の通りである。

述語名: 節で定義 (の一部) を与える述語 (手続き) の名前.

²後に述べるが、プログラムの実行はつつがなく終了するとは限らない。

引数: 述語の引数と節の中で使う変数名の対応をつけるための仮引数の並び。KL1 では変数名の有効範囲はひとつの節の中だけで、同じ名前でも別の節にあればまったく別のものとみなされる。同じ述語の再帰的な呼び出しについても、そのたびごとに同じ名前の変数が別の変数を意味する。これは C の auto 変数と同様である。

ガード: 節の適用条件を指定する部分。カンマで区切ってゴールをいくつでも書け、すべてを満足した場合にだけ適用条件を満たしたものとする。ガードにはユニフィケーションの他に、大小比較など、言語で定義するある決まった種類の述語の呼び出しだけが書ける。

ボディ: 節を選んだときに実行すべき部分。やはりカンマで区切ってゴールをいくつでも書け、その節が選ばれたらすべてを実行する。ボディにはユニフィケーションの他に、他の述語、あるいは自分自身を呼び出すゴールが書ける。

KL1 で呼び出せる述語には、あらかじめ言語で定義する組込述語 (built-in predicates) と、プログラム中に定義するユーザ定義述語 (user-defined predicates) の二種類がある。

組込述語の多くは種々のデータについての基本操作や基本的な値の検査のためのもので、それらがどのように動作するかは KL1 言語の仕様の一部として定められている。特殊な組込述語として、引数のない述語 “true” がある。これはガードに現れれば常に真 (つまり無条件)、ボディに現れれば何もしないこと (no operation) を意味する。

ユーザ定義述語のゴールの実行では、その述語を定義する節がいくつかあるとすると、まず各節の適用条件であるガードの真偽を試す。次にガードが真であるとわかった節をひとつだけ選び、³ そのボディのゴール (ユニフィケーションを含む) をすべて今後実行すべきゴールとする。これを繰り返すのが KL1 プログラムの実行過程である。

並列実行, 通信, 同期の機構

節はある条件 (ガード) を満たすゴールを、ボディのゴールに書き換える書換ルールであると見ることもできる。図 1.1 に示すように書き換えるべきゴールの集合⁴ からひとつのゴールを取り出し、この書換えを施した結果をゴールの集合に戻すのが実行の 1 ステップになる。書換えを繰り返すことによって最終的にはすべてのゴールが何もしないゴール “true” にまで書き換えられると実行は終了する。⁵ この過程は、最初に与えられたゴールを “true” にまで書換えていく簡約化 (reduction) であるとも考えられる。

この書換えはかならずしも逐次的にひとつずつ行なわなくとも良い。複数のゴールについて同時に行なえば、実行は並列になる。

KL1 では複数のゴールが同じ変数を共有することがある。あるゴールが変数の値を決めれば、それを共有する他のゴールは、その値を使ってどの節を使って簡約化するかを決めることもできる。これが KL1 の通信機構である。

前掲のインパータのプログラムについて、メインプログラムを以下のように書換えた場合を考えてみよう。

```
main :- not(1, X), not(X, Y), io:outstream([print(Y),nl]).
```

このように実行すると、X で示す変数は述語 not に関するふたつのゴールで共有することになる。

KL1 では、複数のゴールがあるときにはどの順番に実行するかは特に決めておらず、処理系の都合で決めて良いことになっている。⁶

仮に、まず最初の “not(1,X)” から先に実行するものとしてみよう。第 1 引数は 1 だから、第 2 引数 X の値は 0 に決まる。この X はもうひとつのゴール “not(X,Y)” の第 1 引数にもなっている。だから

³ 複数あれば、どれが選ばれるかはわからない。

⁴ 同じものがあってもよいので、正確にはマルチ集合。

⁵ KL1 ではいつまで待っても実行が終らないが有用なプログラムというものも考えられる。

⁶ 通常は実行は左から右に行なわれるが、必ずそうなるとは限らない。また、実行順序を指定するために後述の優先度指定機構もあるが、やはり絶対的なものではない。

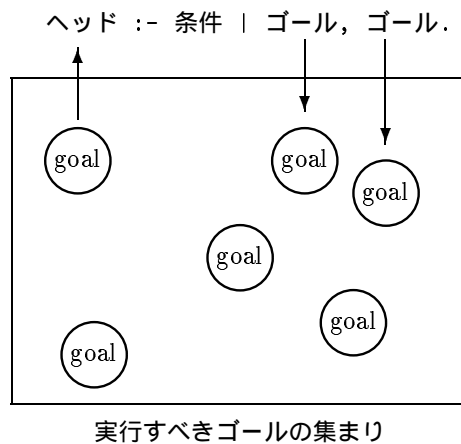


図 1.1: KL1 の実行機構

このゴールの第 1 引数は 0 になったわけである。そこで、このゴールを実行すると Y は 1 に決まる。したがって、このゴール列の実行の結果、X は 0 に、Y は 1 になる。

実行の順序が変わったらどうなるだろう。最初に二番目のゴール “not(X,Y)” を実行しようとしたとする。すると、第 1 引数 X の値はまだ決まっていないから、述語 not のふたつの節のガードに書いてある適用条件 In=0 と In=1 は、どちらも In の値がまだ決まっていないことになる。これではどの節を使って良いのか、条件を試すことができない。このような場合、当面の間このゴールの実行は見合わせる。これを実行の中断 (suspension) と呼ぶ。

このゴールがすぐ実行できないとなると、他にはもうひとつのゴールしかすることはしない。そこでそのゴールを先に実行すると X の値が 0 に決まる。これで先程実行を中断していたゴールの第 1 引数の値は決まり、実行を中断し続ける理由はなくなった。そこで、このゴールの実行を再開する。こんどはガードの真偽を調べるのに障害はなく、無事 In=0 の節が選ばれ、Y の値は 1 に決まる。これだけが KL1 の通信と同期の機構で、これ以外には何も無い。

要約すると以下のようなになる。

- 通信は共有変数を用いて行なう
- 同期はガードで必要な値を検査すると自動的に行なわれる

KL1 では選択実行 (if-then-else のようなもの) の条件はすべてガードの真偽を調べることによって決める。その判断に必要な値について、まだ値が決まっていない (計算できていない) 場合は、実行を中断して値が決まるのを待ち合わせる自動的な同期が起きる。KL1 の変数はいったん値が決まってしまうと、後からそれが別の値に変わるということはなく、ずっと同じ値を持続させる。だから、KL1 プログラムでは単純な計算順序の誤りによって誤った選択をしてしまうという危険性は皆無なのである。

実行機構についての補足

以上で KL1 の基本的な実行機構のすべてなのだが、ここまでの内容でいくつかまだよく説明していないことがあるので、ここで少し補足しておこう。

複数の節が使えたら？ もしふたつ以上の節のガードが真だったら、どの節を選ぶのだろう。KL1 の言語仕様では、このような場合にはガードが真であるような節のうちのどれを選ぶかあえて決めていない。⁷ 処理系の自由でどう決めても良いことになっている。⁸ したがって、正しい KL1 プログラムでは、ひとつ

⁷ 節間の優先度を与える機構もあるが、これは絶対的なものではない。

⁸ 実際、非共有メモリのマルチプロセッサシステム上の処理系では、近くにデータが揃っていてプロセッサ間通信をしなくてもガードの真偽を判定できるような節があれば、そちらを選ぶように実現することがあるので、毎回の節を選ぶかが変わったりする。

の述語について排他的になるようにガード条件を書くか、排他的でない場合にはどれを選んでも正しく動くように書かなければならない。

どの節も使えなかったら？ もしどの節の選択条件も、みな成り立たないとわかったら、どうするのだろう。たとえば前掲の述語 `not` を “`not(3, X)`” のような引数で呼んだらどうなるのだろう。この場合はゴールの実行は失敗 (failure) になり、プログラムは異常終了する。

ボディ・ユニフィケーションで、既に値の決まっている変数の値をまた決めようとしたら？ 既に決まっている値と同じ値なら、何もしなかったのと同じになる。違う値にしようとしたのなら、ユニフィケーションの失敗になり、やはり異常終了になる。

1.2.2 他のプログラム言語と比べて

ここまで述べてきた KL1 の特徴を、他の言語と比べて振り返ってみよう。

手続き型言語と KL1

変数とその値を変えていくという手続き型言語に見られる考え方は、KL1 には存在しない。KL1 の変数は値が決まっていなかったり、決まっているかのいずれかで、いったん値が決まったらそれは決して変わらない。時々刻々変化する計算状態が条件判断に影響を与えることはないのだから、計算順序の自由度がたいへん大きくなる。この特徴は並列処理にとっては大きな利点である。

関数型言語と KL1

時間変化する計算状態が問題にならないという意味で、KL1 は純粋な関数型言語に近い。⁹ 並列処理に対する利点も共通している。

関数型言語と KL1 の最大の違いは非決定性 (nondeterminacy) にある。非決定性とは、まったく同じ計算を複数回行なっても違う結果が得られることがある、という性質である。この非決定性は複数の節のガードが真である場合に生じる。これはデバッグを考えると欠点なのだが、効率良く問題を解かせる上では利点になる。関数型言語のプログラムはどんな計算機システム上でも (順序は違っても、結局は) 同じ計算をするが、KL1 のプログラムは計算機システムの構成 (プロセッサの台数や、その間の通信の速度など) に応じて、自在の動きをするように書くことができる。

他の論理型言語と KL1

Prolog と KL1 一見 KL1 のプログラムは、同じホーン論理¹⁰に基づく論理型言語である Prolog に非常に良く似ている。プログラムを論理式として解釈できる点も同様で、その際にどのように対応づけて解釈するかまでもまったく同じである。しかし、プログラム言語として見た時、KL1 と Prolog はまったく異なる言語であるといって良い。Prolog は、原則逐次処理の範囲内で、ホーン論理の自動証明系としての完全度をできるだけ上げるように設計されている。¹¹ 一方、KL1 は完全性の追求はあきらめ、並列処理言語としての能力を追求した言語である。両者は共にホーン論理に基づきながら、まったく違う方向に発展してきた言語で、KL1 を並列 Prolog と呼ぶのは間違いである。

他の並列論理型言語と KL1 Concurrent Prolog[3], Parlog[2], Guarded Horn Clauses (GHC) など、いわば同じ穴のムジナで、ほぼ同じ方向を目指した言語である。KL1 はこれら committed choice 型などとも呼ばれる並列論理型言語のうちでもっともシンプルな言語である。GHC を元に、言語仕様をさらに簡素にした Flat GHC と呼ばれる言語をその基礎としている。

⁹ ここでいう意味では、広く使われている Lisp は関数型ではなく、手続き型言語である。

¹⁰ 機械的な定理証明が比較的容易で、表現力もかなり大きい一階述語論理のサブセット。Prolog (カットなどを含まないピュアなもの) も KL1 も、この論理についての不完全だが健全な自動証明系とみなせる。

¹¹ しかし、本当に完全ではない。

オブジェクト指向言語と KL1

KL1 自身はオブジェクト指向言語ではない。だが、KL1 や他の並列論理型言語で多用されるプログラミング・スタイルは、オブジェクトをプロセスとして実現する、オブジェクト指向のプログラミング・スタイルである。オブジェクト指向プログラミング・パラダイムのかなりの部分は、ごく素直に KL1 で記述できるのである。このプログラミング・スタイルについては後に解説する。

1.3 データとシンタクス

ここまでの例では、具体的なデータとしては 0, 1 などの整数値しか出てこなかった。KL1 では他にもさまざまな型のデータが扱える。本章では KL1 の基本的なデータ型とそれらに対する操作について、簡単に述べる。

1.3.1 論理変数

KL1 などの論理型言語が C などの手続き型言語と最も大きく異なるのは、変数の扱いである。手続き型言語の変数とはっきり区別するために論理変数などと呼ぶこともある。論理変数は手続き型言語の変数よりも数学でいう「変数」にずっと近いものである。

変数はメモリ番地ではない

普通の手続き型言語の変数は、特定の番地から始まるある大きさのメモリ領域につけた名前である。計算機のメモリを表すものだから、中身を読み出したり書き込んだりすることができる。KL1 の変数はメモリのような値の格納場所を表すものではなく、値につけた名前である。だからその値を参照することはできても、値を変更するなどということはできない。

たとえば、多くのプログラム言語ではサブルーチンの引数を用いる時に名前で参照できるように、引数に名前をつける。C 言語の場合、引数名は引数の値が格納してある場所の名前である。したがって、その内容を書換えることもできる。ところが論理型言語では、引数名が指し示すのは渡された値そのものであって、引数値を格納してある場所ではない。だからそれを書換えるなどということはできない。

論理変数で特徴的なのは、値が決まっていないという状態があることである。値を書換えることはできなくても、決まっていない値を決めることはできる。これによって変数を介した値の受渡しができる。プログラム中のふたつの部分が同じ変数を共有すれば、片方ではその値を決め、もう一方ではその値を用いることができる。

変数に型はない

KL1 では変数についてどんな型のデータが入るかは宣言しない。ソース・プログラム中の同じ節の同じ変数が、ある呼ばれ方をしたときはひとつの型のデータ、別のときには別の型のデータを値とすることがある。このあたりは Prolog や Lisp と同様である。

このことには利点も欠点もある。プログラム中の変数にはどんな種類の値が入るかが、原則としては入力データを与えるまでわからないので、処理系が最適化しにくいことは欠点のひとつである。¹² また、人間がプログラムを読むときにも、変数の型という理解の助けになる情報が欠けている分だけ、強い型付けをする言語に比べると不便だろう。

一方利点としては、C の union のような特別の記法を用いなくてもさまざまな型のデータを混在させることができること、Ada の generic package のような面倒な宣言をしなくても、いろいろな型のデータに対して共用できるプログラム・モジュールを簡単に作れることなどがある。

全体として、変数に型がないことは、プログラムを読むのにはちょっと不便だが、書くのには便利である。このことは、実験的なプログラムを何度も書き直しながら、アルゴリズムを開発していくような場合には有利になる。これは AI 研究に変数に型をつけなくて良い Lisp が広く使われてきた理由のひとつで

¹² この欠点はコンパイル時の解析によって型を推論すれば、ある程度はカバーできる。

あろう。一方、既に確立されたアルゴリズムに基づいたプログラムを組み、それを長年に渡って保守していくような場合には不利もある。

1.3.2 アトミックなデータ

アトミックなデータとは、内部構造を持たず、その値自身だけに意味があるようなデータである。KLIC で扱えるアトミックなデータ型としては、特に何の値を表すわけでもなく、自己同一性だけに意味がある識別子である記号アトムと、整数値を表す整数アトムがある。

記号アトム

KL1 の記号アトムは Lisp などと異なり、属性などの情報を持つことはない、純粋の識別子である。したがって、記号アトムについてできる演算は、ガードでの同一性の判断だけである。プログラム中の諸概念を一意に表すのに用いる。

アトムの表記は Edinburgh Prolog のアトムの表記と同様で、以下のいずれかである。

- 英小文字から始まり、英数字および下線 “_” の任意個の並び。たとえば “bit”, “pim”, “an_Atom” など。
- 特殊文字 (“+”, “-”, “:”, “=” など) の任意個の並び。たとえば “+”, “:-” など。
- ふたつの引用符 “'” でくくられた任意の文字列。ただし、引用符を含む場合は引用符を二回続ける。たとえば “'Hello world'”, “'quoted'” など。
- 特殊なアトム。具体的には “!”, “;”, “[” の三種類。

数値アトム

整数値を表し、表記も通常は普通の十進記法 (に必要な符号がついたもの) を用いる (“3”, “-15” など)。1 語が 32 ビットのシステム上の KLIC の整数は 28 ビットの値を持つので、表せる範囲は -2^{27} から $2^{27} - 1$ まで、1 語が 64 ビットなら値は 60 ビットなので -2^{59} から $2^{59} - 1$ までである。

整数に対する演算や比較は、言語のプリミティブとして用意した述語である組み込み述語 (built-in predicates) を用いて行なう。整数演算としては加減乗除の四則演算があり、ガードの条件として大小比較ができる。通常、これらの演算、比較には、組み込み述語を直接用いるよりもマクロ記法を用いるのが便利である。

整数についてのガードでの大小比較には “<”, “>”, “=<”, “>=” を用いる。たとえば “X =< Y” は「X は Y と等しいか、または Y よりも小さい」を表す。相等、不等の条件には “=:” と “=\=” を用いる。

たとえば第一引数と第二引数のどちらか大きい方を第三引数の値とするようなプログラムは、以下のよう書ける。

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

```
max(X, Y, Z) :- X < Y | Z = Y.
```

このプログラムでは X と Y が等しい場合、どちらの節のガードも真である。前にも述べたように、このような場合にどちらの節が選ばれるかは、言語仕様としてはまったく規定しない。このプログラムではどちらの場合でも結果は変わらないので問題ない。ふたつの節を非対称的に書いて X と Y が等しい場合は片方しか選ばれないように書くよりも、素直な書き方だともいえよう。

整数演算のためのマクロは通常の加減乗除などの演算子 (“+”, “-”, “*”, “/”, “mod”) と括弧を用いた算術式を “:=” の右辺に、結果を値にしたい変数を左辺に書く。¹³

たとえば第一引数と第二引数の和を第三引数の値とするような述語は以下のように定義できる。

```
sum(X, Y, Z) :- true | Z := X + Y.
```

前述のように、ガードに現れる “true” は常に真であるような条件、つまり無条件を表す。

なお、KLIC では整数演算のオーバーフローチェックは行っていない。

¹³ この他にビットごとの論理和、論理積、論理排他和、反転などもあるが、ここでは詳しく述べない。

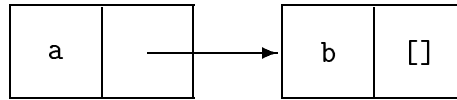


図 1.2: コンスによるリストの表現

1.3.3 構造を持ったデータ

構造を持ったデータとは、要素となるデータを集めてできているデータのことである。KL1 で扱える構造を持つデータの代表的なものには、以下のものがある。¹⁴

ファンクタ ファンクタ構造の記法は、その名前であるアトム、開括弧 (“(”) 最初のもの以外の要素をカンマで区切って並べたもの、そして閉括弧 (“)”) という順に記述する。たとえば “a(b, c)” は、名前が a、引数が b と c というファンクタ構造を表す。要素はどんな型でも良く、入れ子構造をしていても良い。たとえば “f(g(a, b), h(c))” は名前が f、ふたつの引数がまたファンクタ構造であるようなファンクタ構造である。

コンス コンスは任意の型のふたつの要素を持つ構造である。ふたつの要素は Lisp に習って car, cdr と呼ぶ。表記にはカギ括弧対 (“[” と “]”) の間に car, cdr の要素を縦棒 (“|”) で区切って書き並べる。たとえば “[a|b]” は car がアトム a、cdr がアトム b であるようなコンスを意味する。

コンスそれ自体はふたつの要素だけを持つ構造データだが、これを組み合わせてさまざまなデータ構造を作る。その代表がリスト構造である。コンスの car がひとつの要素、cdr がリストの残り、要素がひとつもないリストはアトム “[]” 表すものと取り決めれば、ふたつの要素 a, b を持つリストは “[a|b|[]]” と、ネストしたふたつのコンスを用いて表すことができる (図 1.2)。このままではリストが長くなると多数の括弧が必要になるので、これを “[a, b]” のように書き表す。

未完成データ構造

ファンクタやコンスのように任意の型の要素を持てる構造データの場合、要素の値がまだ決まっていないことがあっても良い。このような構造を未完成データ構造¹⁵ (incomplete data structure) と呼ぶ。

未完成データ構造の値の決まっていない要素は、変数として書き表わす。たとえば “[a|X]” は car がアトム a、cdr が X という名前を持つ変数であるようなコンスを表す。

未完成データ構造の中の変数も、構造体全体を持っているゴールと別のゴールで共有することができる。変数の値を決めるのは別のゴールの方であっても良い。つまり、誰か他の人が詳細を決めることになっているデータ構造、ということになる。こうしたデータ構造は KL1 のプログラミング・テクニックのさまざまな局面で重要な働きをつとめ、柔軟なプログラミング・スタイルの源となっている。

構造データのユニフィケーション

ここまででは、アトミックなデータ同士のユニフィケーションしか出てこなかったが、構造を持つデータ同士もユニフィケーションできる。

ガード・ユニフィケーション ガードでのユニフィケーションは、両辺の値の一致を調べるものだが、構造体同士の場合は以下のような再帰的な規則になる。

1. まず両辺が全体として同じ型の構造体で、ファンクタ同士なら同じ名前で、要素個数も同じであることを確かめる。そうでなければ値は不一致である。

¹⁴ KLIC ではここに述べる構造データは基本的なデータ型として用意されている。それ以外の構造データは generic object と呼ばれる拡張機能を用いて実現される。

¹⁵ 不完全データ構造と呼ぶ場合もある。

2. 次に両者の対応する (同じ要素番号の) 各要素について、この規則を再帰的に適用する。すべてについて一致するときのみ、全体が一致するものとする。

要素のユニフィケーションにあたって、相手が新しく出てきた変数なら、その変数には対応する要素が渡される。たとえば、以下のプログラムは第 1 引数に渡されたコンスの `car` と `cdr` を、第 2, 第 3 の引数に返すような述語の定義になっている。

```
carcdr(Cons, Car, Cdr) :- Cons=[X|Y] | Car=X, Cdr=Y.
```

ボディ・ユニフィケーション 一辺が変数のボディ・ユニフィケーションは、もう片方が構造を持っているいなくても同じで、その変数の値を他辺の値に決めるものである。両方ともが構造体の場合のユニフィケーションの規則は、やはり以下のような再帰的なものである。

1. まず両辺が全体として同じ型の構造体で、ファンクタ同士なら同じ名前で、要素個数も同じであることを確かめる。そうでなければユニフィケーションは失敗である。
2. 次に両者の対応する (同じ要素番号の) 各要素について、この規則を再帰的に適用する。

いずれにせよ、両辺とも値が決まっているようなユニフィケーションをボディで行なうのは、推奨するコーディング・スタイルではない。

ここでは KL1 で扱えるデータについて概説した。中でも要素が何なのかまだ決まっていない未完成データ構造を扱えるのが KL1 の大きな特徴である。次章ではこの未完成データ構造を活用するプログラミング手法である、プロセスとストリームについて述べる。

1.3.4 略記法

以下の説明を続ける前に、いくつかの便利な略記法を紹介しよう。

ガード・ユニフィケーションのヘッドでの表記

ガードで引数として渡された変数そのものとのユニフィケーションを行なう場合、ヘッドの対応する引数の位置に、直接ユニフィケーションの相手を書いてしまうような略記ができる。たとえば、前のインバータの例にあった

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

という節は

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

と略記できる。

整数 1 のような具体的な値でなくても、引数同士のユニフィケーション (引数の値が同じ) をガードで行なう場合、たとえば

```
same(X, Y, R) :- X = Y | R = same.
```

という節は

```
same(X, X, R) :- true | R = same.
```

と、同じ変数名をヘッドに 2 回使って表現することもできる。

ゴール true の省略

ガードが無条件の場合、つまりガードが引数のない述語 `true` の呼び出しだけの場合は、ガード部全体をボディとの区切りである縦棒ごと省略することができる。たとえば、前述のインバータの例は

```
not(1, Out) :- Out = 0.
```

と書いても良い。

ガードだけでなく、ボディも `true` だけならば、ヘッドとの区切りである “:-” ごとボディ全体を省略しても良い。たとえば:

```
one(1) :- true.
```

という節は

```
one(1).
```

と書いても良い。

1.4 プロセスとストリーム通信

これまでに KL1 の基本的な言語仕様を解説した。ここでは、KL1 で良く使われるプロセス (process) とプロセス間をつなぐストリーム (stream) を用いたプログラミング・スタイルについて解説する。このプログラミング・スタイルは Shapiro と竹内によって最初に提案されたもので [4]、第五世代プロジェクトで開発されたオペレーティング・システムや種々の実験的な並列応用プログラムは、多くがこのスタイルに従って書かれている。

1.4.1 プロセス

前述の通り、KL1 の基本的な実行機構は簡約化の (並列な) 繰り返しである。しかし、単純な簡約化の繰り返しというだけでは、複雑な計算をわかりやすく記述するには役不足で、数多くの簡約化操作をうまくまとめあげるような概念が有用である。そのような役割を果たすのがプロセスという考え方である。

KL1 のプロセスとは

ボディに同じ述語を呼び出すゴールをひとつ含むような節は、(引数は変わるが) 同じことを繰り返し行なうループと見ることができる。たとえば、与えられた引数から始めて、0 になるまでカウントダウンしていくような述語は

```
count_down(0).
count_down(N) :- N>0 | M:=N-1, count_down(M).
```

というふたつの節で定義できる。

この例で “`M:=N-1`” とはなっていないことには気をつけていただきたい。前にも説明したが、KL1 の変数は値の置き場所を示すものではなく、むしろ値そのものにつけた名前なのである。だから `N-1` の演算結果である新しい値には、`M` という `N` とは異なる名前をつけているのである。

この例のボディには “`M:=N-1`”, つまり引き算を行なうゴールと, “`count_down(M)`” という再帰呼び出しのゴールのふたつがある。KL1 ではボディ中のゴールの記述順序には特に意味がないので、両者の実行順はどうなるかわからない。引き算よりも前に再帰呼び出しの実行を始めることもあるし、両方同時に実行することもある。しかし、述語 `count_down` を定義するふたつの節は、両方とも選択条件として引数の値を調べているので、その値が決まるまで実行は待たされる。この引数の値は引き算の結果なのだから、実際には必ず引き算の方が先に実行されることになる。

基本的な実行機構であるひとつひとつの簡約化は細切れの操作なのだが、この繰り返し全体を見ると、ある程度の大きさを持った連続性のある計算過程と考えることができる。このような計算過程をプロセ

スと呼ぶ。¹⁶ プロセスとみなせるような、まとまった計算を行なう述語の例をいくつかあげて、KL1 のプロセスとはどんな概念なのか、どういう特徴を持つのかを、もう少し詳しく説明していこう。

リスト要素の和

整数を要素とするリストに対して、要素の総和を計算するような述語は以下のように書ける。

```
sum([], PSum, Sum) :- Sum=PSum.  
sum([One|Rest], PSum, Sum) :- NewPSum:=PSum+One, sum(Rest, NewPSum, Sum).  
  
sum(List, Sum) :- sum(List, 0, Sum).
```

このプログラムはふたつの述語の定義からなる。両者は同じ `sum` という名前だが、引数個数が異なる。KL1 では同じ名前でも引数個数が異なれば違う述語として扱う。この例でもそうだが、同じ名前でも引数個数が異なる述語は補助的な述語の名前として使うことが多い。

最初のふたつの節で定義する 3 引数の述語が、実際の計算をする述語である。この述語を単独に見ると、第 1 引数に与えられるリストの要素すべてと、第 2 引数に渡される数とを足し合わせ、第 3 引数にその結果を返すものになっている。最初の節は、空のリストについては要素がないのだから第 2 引数をそのまま返せば良い、ということを表している。もうひとつの節は、リストが空でない場合、最初の要素が `One` なら、この `One` と第 2 引数 `PSum` との和 `NewPSum` を求め、これとリストの残り部分 `Rest` の各要素との和が、求める総和である、ということを表す。

最後の節で定義する 2 引数の述語がもともと定義したかった述語で、リストの要素の総和と 0 の和を計算すれば、リストの要素の総和そのものを計算することになる、という意味になる。

注意されたいのは、2 番目の節のボディにある、足し算と再帰呼び出しのふたつのゴールは、並列に動いてもまったくかまわないということである。再帰呼び出しの実行では、節の選択条件は第 1 引数のリストが空かどうかだけに依っており、足し算の結果は節の選択に関係しない。だから、再帰呼び出しだけ先にどんどん実行してしまい、足し算は後からやっても構わないのである。¹⁷ ただし、次々に呼ばれる足し算の引数のひとつ (`PSum`) は、一段前の呼び出し時に呼んだ足し算の結果になっている。だから、要素個数と同じ回数行なわれる足し算については、実行順が自動的に決まってしまう。

このように KL1 でプロセスと呼び慣わしている計算過程は必ずしも逐次的な過程ではなく、それ自体の中に並列性を持っていることも少なくない。プロセスという概念は、プログラムを読む側の解釈に過ぎないのである。

自然数のリスト

ある数未満の自然数¹⁸すべてを要素とするリストを作るような述語は、以下のように書ける。

```
naturals(N, M, List) :- N>=M | List=[].  
naturals(N, M, List) :- N<M | List=[N|Rest], N1:=N+1, naturals(N1, M, Rest).  
  
naturals(M, List) :- naturals(0, M, List).
```

このプログラムも同様に 2 引数の本来定義したかった述語と、補助的な役割を果たす 3 引数の述語からなっている。

最初のふたつの節で定義する述語は、第 1 引数の値から始めて、第 2 引数未満の整数すべてを小さい順に要素に持つリストを、第 3 引数に返すものである。最初の節は、第 1 引数が第 2 引数より小さくなく

¹⁶ KL1 ではプロセスという概念は言語仕様の一部ではなく、あるプログラミング・スタイルにつけた名前に過ぎないことに注意されたい。

¹⁷ 並列に行なって良いということは必ずしも実際に行なえば速くなるということではない。通信のためのコストなどがあるので、普通は足し算のような簡単な操作を並列に行なうメリットはない。実際の KL1 の処理系ではこのような足し算を並列に行なおうとしたりはしない。この例は整数の足し算という簡単な操作だったのだが、足し算の代わりにもっと複雑な操作を各要素について行なう必要があれば、実際に並列に行なうメリットがでてくる。

¹⁸ ここでは 0 も自然数だとしている。

れば、空のリストを返せば良い、という意味である。次の節は、第 1 引数の方が小さければ、第 1 引数を先頭要素とするリストを返せば良く、リストの残り部分は第 1 引数よりひとつだけ大きい値から始めて第 2 引数未満の整数すべてを小さい順に要素に持つリストにすれば良い、という意味である。

最後の節で本来定義したかった述語を定義していて、3 引数の方の述語を 0 から始めるように呼べば、与えられた引数未満の自然数すべてを要素にするリストが作れる、ということを書いてあるわけだ。

この例でも、ふたつめの節のボディにある、ユニフィケーション、足し算、再帰呼び出しの三者は、どんな順で実行しても（並列に実行しても）構わない。

1.4.2 ストリーム通信

上に述べたプロセスをモジュールとして結び合わせ、より複雑な計算を記述する方法を説明しよう。

プロセスの複合

ひとつのプロセスの計算結果を用いて、別のプロセスが計算をするように、プロセスを組み合わせるプログラムを組むことができる。たとえば、与えられた数未満の自然数の総和を求めるには、前掲のふたつのプロセスを組み合わせ、以下のようなプログラムを書けば良い。

```
sum_up_to(N,Sum) :- naturals(N,List), sum(List,Sum).
```

このプログラムでは変数 `List` がふたつのプロセスから共有されており、`naturals` が作ったリストを `sum` に渡す手段になっている。

このふたつのプロセスは同じボディの中に書かれているのだから、どんな順序で実行しても良い。2 引数の述語 `sum` はガードの条件がないから、`List` の値が決まっていなくても実行でき、3 引数の方の述語 `sum` を呼び出す。こちらの方は引数の値によってどの節を選ぶか決めなければならないので、値が決まるまで待たされるわけである。

さて、ここで先ほどの自然数のリストを作る述語の定義をもう一度見てみよう。

```
naturals(N,M,List) :- N>=M | List=[].
naturals(N,M,List) :- N<M | List=[N|Rest], N1:=N+1, naturals(N1,M,Rest).
```

前にも書いたように、ふたつ目の節のボディのユニフィケーション、足し算、再帰呼び出しの三者はどんな順でも実行できる。ユニフィケーションが他のゴールと並列に実行できるということは、まだ計算が終わらないのに答を返せるということである。もちろん答は全部求まったわけではなく、結果を返す引数とユニフィケーションしているコンス構造の `cdr` である `Rest` は、再帰呼び出しゴールの実行結果が入るまでは変数のままだから、前述の未完成データ構造である。しかし、計算結果が全体としてはコンスであることはこのユニフィケーションで既に決まってしまう。また、その `car`、つまりリストとして見たときの最初の要素は、最初の呼び出しの第 1 引数が 0 なのだから、もうそれに決まっている。

今度は総和を求める方のプロセスの本体である 3 引数の `sum` の方を、もう一度振り返ってみよう。

```
sum_up_to(N,Sum) :- naturals(N,List), sum(List,Sum).
```

述語 `naturals` の結果がコンスであることまでもう決まっているのだから、ガードの条件（実際にはヘッド中に略記されている）の判定はでき、ふたつめの節を選ぶ。そしてその `car`（ここでは変数 `One` で受けている）が 0 であることも決まっている。第 2 引数の `PSum` は最初は 0 にして呼んでいるのだから、もう足し算の引数はふたつとも決まっているわけである。だから $0+0$ という足し算は、`naturals` の実行が少し進んだだけでもうすぐにでも実行できるようになっているわけだ。

再帰呼び出しの方はそうはいかない。まだ `naturals` が最初の 1 段階しか進んでいないとすると `Rest` の値は決まっていない。つまり、再帰的に呼び出された `sum` では第一引数がまだ決まっていないわけで、どの節を選べば良いかの判定ができない。この `Rest` は `naturals` と `sum` のふたつのプロセスの間で共有する変数になっていて、`naturals` がその値を決めることになる。だから `sum` の再帰呼び出しの実行は `naturals` の方の計算がもっと進むまで待たされる。この後に `naturals` の方がもう一段進めば、`sum` の方ももう一段進めるようになる。

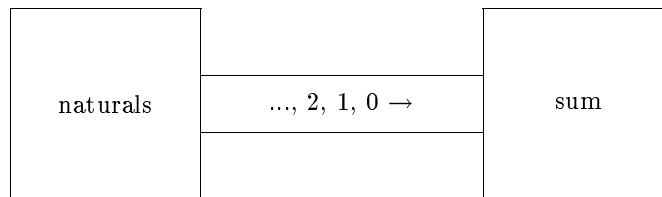


図 1.3: プロセス間の通信路

このように、要素がまだ決まっていない構造体を結果として返し、後からその要素を確定していくことができる、という仕組は、KL1 の大きな特徴のひとつである。要素がすべて決まるまで結果を返せないのだとしたら、`naturals` の実行がすべて終わるまで `sum` の実行を始めることができない。それだけ並列度が低くなってしまうわけである。¹⁹

ストリーム通信

前節に述べたふたつのプロセスは、述語 `naturals` で表されるプロセスが次々に作り出す値を、述語 `sum` で表されるプロセスが次々に使っていく、という関係にある。その仲介役を果たすのが要素が段々に具体的な値に決まっていくようなリストという未完成データ構造だった。

このリスト構造を、データが次々に流れていくような通信路だとも考えることもできる。プロセス `naturals` はこの通信路に次々に自然数を流して行き、プロセス `sum` がその通信路の出口からデータを読み取っては計算を進める (図 1.3)。データがまだ到着していなければ、読み取り側のプロセスは待たされる。

通信路を実現しているのはデータ構造コンスである。その `car` にはストリームを流れるデータが入り、`cdr` はストリームの続きを表す。空のリスト `[]` は、必要な通信が終りまで来てしまい、もうこれ以上メッセージがないことに対応する。このように通信路がデータ構造になっているので、データの流れる順序はどのようなデータ構造を作るかで完全に決まってしまうので、計算の実行順序や通信遅延の違いが、そのままデータの順序に影響することはない。このような通信路をストリーム (stream) という。²⁰

ストリームを流れるデータはプロセス間で受け渡されるメッセージ (message) であるとも考えられる。つまり、`naturals` というプロセスは `sum` というプロセスに次々に、この数も足してくれ、この数も、というメッセージを送っているわけである。

プロセスの状態

メッセージを受けてそれに従って仕事をする、というふうにプロセスを書き表せることを説明したが、もし仕事の内容がメッセージだけで完全に決まるのならわざわざそんな書き方をする必要はない。プロセスを作ってそれにメッセージを送る代わりに、必要な仕事をする述語を定義してそれを呼び出してしまえば良い。

メッセージ駆動のプロセスとして記述する意義は、プロセスには状態 (state) を持たせることができるということにある。前述の和を求める例では、`PSum` という部分和を保持する引数が、その状態を表すものになっていた。

プロセスの状態がもっと明確に現れる例を示そう。整数値を保持し、メッセージによってそれを増減するようなカウンタの機能を持つプロセスは、以下のように書ける。

```

counter(Stream):- counter(Stream,0).

counter([],Count).
counter([_|Stream],Count) :- New:=Count+1, counter(Stream,New).

```

¹⁹ 繰り返しになるが、プログラムに並列性があるということと、その並列性を実際の並列実行として実現するかどうかは別の問題である。実際に並列に実行するか逐次に実行するかは通信コストなどの要因を考えて決めるべきである。

²⁰ この例の場合は合計を求めるのが目的なので、データの到着順は問題ではないのだが、一般には順序が問題になることが多い。

```
counter([down|Stream],Count) :- New:=Count-1, counter(Stream,New).
```

このプログラムで定義するプロセスは、第2引数としてその時々のカウンタの値を保持している。初期値は0で、後から up, down というメッセージが来るたびにそれを増減して再帰呼び出しの引数に渡すことによって、状態を更新しているわけである。このように KL1 のプロセスは状態を引数値として保持する。

このプロセスを呼び出す時には

```
..., counter(Stream), some_other_process(Stream), ...
```

のようにして、メッセージ・ストリームを媒介して他のプロセスと通信できるようにするわけだが、このストリームがカウンタのプロセスとそれ以外のプロセスとを結ぶ唯一の通信手段である。状態として保持しているカウンタの値を直接他のプロセスが操作することはできない。つまり、プロセスの状態として値を保持することは情報を隠蔽していることになる。

この隠された情報にはどうやってアクセスしたら良いのだろう。上のプログラムではメッセージを送ってカウンタを上下することはできるが、カウンタの値が何なのかを読むことができない。これについては次節で説明しよう。

複雑なメッセージ

これまでの例では、ストリームを流れるメッセージはすべてアトミックなデータ（整数値や up, down などのアトム）だけだった。もちろんメッセージとしてもっと複雑な構造を持つデータを流しても構わない。

よく使われるメッセージとしてファンクタ構造がある。ファンクタをメッセージとして用いて、ファンクタ名と引数個数でメッセージの種類を、要素でメッセージの詳細を表すのが便利である。

前述のカウンタをひとつずつではなく、一度にいくつでも増減できるようにするには、増減のためのメッセージを引数を持つファンクタにして、以下のように書けば良い。

```
counter(Stream):- counter(Stream,0).
```

```
counter([],Count).
```

```
counter([up(N)|Stream],Count) :- New:=Count+N, counter(Stream,New).
```

```
counter([down(N)|Stream],Count) :- New:=Count-N, counter(Stream,New).
```

カウンタの値を読み取る機能は、以下のようなファンクタをメッセージとする節を追加すれば実現できる。

```
counter([show(V)|Stream],Count) :- V=Count, counter(Stream,Count).
```

値を読み取るためのメッセージ show は、カウンタの値を返してもらうための引数 V を未定義のままを送る。受けとったプロセスは内部状態に応じてその値を決めてやるわけである。このような未定義の部分を含んだメッセージを未完成メッセージ (incomplete message) という。未完成メッセージも未完成データ構造の一種である。このようにすれば、一本のストリームを介して双方向の通信ができるわけである。

以上、ここではプロセスとストリームという概念を用いる KL1 のプログラミング・スタイルについて述べた。このプログラミング・スタイルの実現には未完成データ構造が大切な役割を果たしている。

1.5 プロセス・ネットワーク

前章では KL1 のプログラミング・スタイルとしてのプロセスと、その間の通信に用いるメッセージ・ストリームについて述べた。本節では、このプロセスとストリームを組み合わせる複雑なプログラムを組み上げていくための、基本的な手法を紹介する。

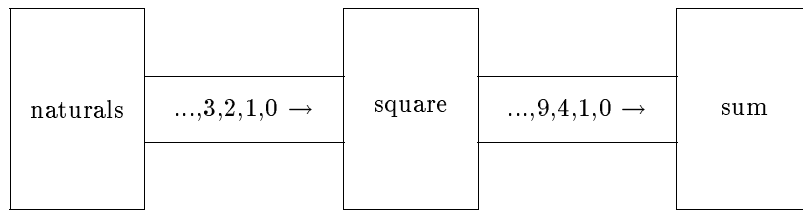


図 1.4: フィルタ

1.5.1 フィルタ

与えられた数未満の自然数の和を計算するプログラムの例を 1.4.2 で紹介したが、これは自然数のリストを作るプロセスと、その要素の総和を計算するプロセスのふたつからなっていた。

では少し問題を変えて、与えられた数未満の自然数の平方の総和を計算するプログラムを考えてみよう。前章のプログラムを利用して作るとすると、すぐに思いつく方法は以下のふたつだろう。

- 自然数のリストを作るプロセスを直して、自然数の平方のリストを作るようにする。
- リスト要素の総和を計算するプロセスを直して、リスト要素の平方の総和を求めるようにする。

これはいずれも、ふたつのプロセスのどちらかを改修して、必要な機能を作ろうという方針である。もちろんこのどちらの方法でもプログラムは作れるし、この問題に限っていえばそれで十分である。しかし、もし両方のプロセスともずっと複雑で、簡単には改修できないとしたら、他にどのような方法があるだろう。

与えられた数未満の自然数のリストを作るプログラムはもうある。リスト要素の総和を求めるプログラムもある。とすると、整数のリストをもらって、各要素の平方を要素とするようなリストを作るプログラムをその間に入れてやれば解決である。たとえば以下のようなプログラムを作れば良い。

```

square([],Out) :- Out=[].
square([One|Rest],Out) :-
    Square:=One*One, Out=[Square|OutTail], square(Rest,OutTail).
  
```

この述語を使えば、プログラム全体は以下のように書ける。

```

square_sum_up_to(N, Sum) :-
    naturals(N,Naturals), square(Naturals,Squares), sum(Squares,Sum).
  
```

述語 `square` はリストを入力としてリストを出力するような述語を定義している。この述語の計算過程をプロセスととらえ、入出力のリストをストリームと解釈するとどうなるだろう。このような見方をすると、このプログラムは一本のストリームから整数値のメッセージを受け、もう一本のストリームにその平方をメッセージとして送り出すプロセスを表している。このように、あるストリームから受けとったメッセージになんらかの変換を施して、別のストリームに出力するようなプロセスをフィルタ (filter) と呼ぶ。全体のプログラムはプロセス `naturals` とプロセス `sum` が、フィルタ `square` を通るストリームを使って通信する、という構成になる (図 1.4)。

この例ではフィルタとなったプロセス `square` は状態を持たず、出力は入力メッセージだけに依存している。一般には、フィルタの動作は入力メッセージだけではなく、フィルタ・プロセスの状態に依存しても良い。フィルタ・プロセスは入力メッセージに応じて状態を変えられるから、入力の履歴に応じてフィルタの仕方を変えることもできる。

フィルタは 1 本の入力ストリームと 1 本の出力ストリームを持つ。ふたつのフィルタの片方の出力をもう一方の入力につなげれば、全体としてはやはり 1 入力 1 出力のフィルタができる。この場合、最初のフィルタがメッセージを次のフィルタに送ってしまった後、次のフィルタがそのメッセージの処理をするのと並列に、最初のフィルタは次のメッセージの処理を行なえる。このようにフィルタをたくさんつなげていけば、パイプライン処理ができるわけである (図 1.5)。

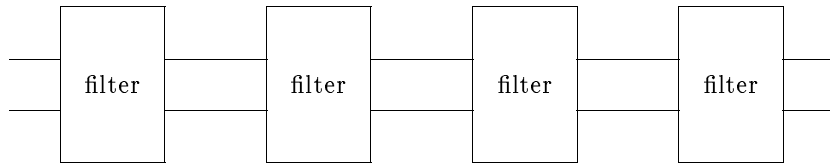


図 1.5: パイプライン処理

1.5.2 ストリームの連結

また問題を少し変えて、こんどは与えられた数未満の自然数の平方と立方の両方の総和を計算するプログラムを考えてみよう。

前節の場合と同様の方法で、入力 n に対して $n^2 + n^3$ を出力とするようなフィルタを作るという方法はある。しかし、それではせっかく作った `square` 述語は使わずに、別に定義しなければならない。そこで、前述の `square` はそのままにして、別に入力メッセージの立方を出力するようなフィルタを作って、これを使うことを考える。²¹

このようなフィルタ自体をどう定義すれば良いかは、もうわかりだろう。

```

cube([],Out) :- Out=[].
cube([One|Rest],Out) :-
    Cube:=One*One*One, Out=[Cube|OutTail], cube(Rest,OutTail).
  
```

このふたつのフィルタと残りの `naturals`, `sum` をどうストリームで結合するのだが、入力には両方ともプロセス `naturals` の出力を共通に与えれば良い。これはごく簡単に

```

square_sum_up_to(N,Sum) :-
    naturals(N,Naturals), square(Naturals,Squares), cube(Naturals,Cubes),
    ...
  
```

と、同じ変数を両方に書けば良い。問題は `Squares`, `Cubes` の 2 本の出力ストリームに流れるメッセージすべてを、どうやってプロセス `sum` に渡すかである。

ひとつの方法は、まず片方のストリームのメッセージを送り込み、それが終わったらもう一方のストリームのメッセージを送るようにすることである。そのための交通整理をする述語は以下のように書ける。

```

append([],In2,Out) :- Out=In2.
append([Msg|In1],In2,Out) :- Out=[Msg|OutTail], append(In1,In2,OutTail).
  
```

この述語は第 1, 第 2 のふたつの引数が入力ストリームに、第 3 引数が出力ストリームになっている。全体としては、まず第 1 引数にやってくるメッセージを次々に出力し、それが終わったら第 2 引数の方のメッセージを流すようにしている。

最初の節は、第 1 引数である一方のストリームが終りまで来たら、後は第 2 引数であるもう一方のストリームをそのままとめて出力ストリームとしてしまえば良い、という意味である。メッセージをひとつひとつ取り出しては中継する必要はないわけである。もうひとつの節は、第 1 引数のストリームの方にメッセージが来たら、それをそのまま出力することを意味する。²²

この述語を使って全体のプログラムを書くと、以下のようになる。

²¹ もちろんこの例のような簡単な問題なら全部書き直してしまった方が早いぐらいである。ここに説明するような方法は、本当は必要な計算がもっと複雑で、`square` のような述語を書き直す手間が大きい場合にこそ有効である。

²² 引数をストリームとしてではなく、単なるリストとして解釈すれば、この述語はふたつのリストをつなぎ合わせたようなリストを作る、という述語になっている。

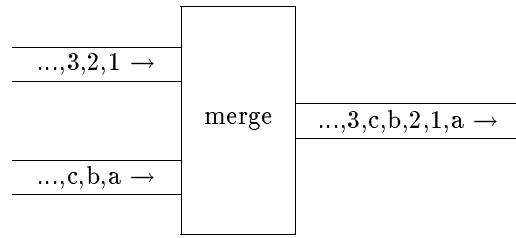


図 1.6: マージ

```
queer_sum(N,Sum) :-
    naturals(N,Naturals), square(Naturals,Squares), cube(Naturals,Cubes),
    append(Squares,Cubes,Both), sum(Both,Sum).
```

メッセージを交通整理する `append` では、メッセージの中身はまったく見ていない。単にメッセージが来たかどうかだけを見て、来たメッセージをそのまま中継しているだけである。そのため、どんなメッセージが流れるストリームに対しても、同じ `append` を使うことができる。KL1 の変数に型がない (どんな型の値も入れられる) ことの利点が現れる例である。

1.5.3 マージ

前節で紹介した `append` を使うやり方では、ちょっと不満が残る。このやり方では、ストリーム `Squares` の出力が終るまで、プロセス `sum` にはストリーム `Cubes` に流れるメッセージがひとつも届かない。もし `square` の処理にかなり時間がかかるとすると、並列に動いている `sum` は暇になってしまう。すでに `cube` の方の処理がある程度進んでいれば、その出力もどんどん足し込んでいけるのに、メッセージが届かなくては何もできない。

そこで、片方のストリームが終りまで来なくても、もう片方のストリームの出力も中継してやれるようなやり方を考えよう。それには、どちらの入力ストリームにでも良いから、メッセージが来たらどんどん出力に流していくようなプロセスを作れば良い。そのような述語の定義は以下になる。

```
merge([],In2,Out) :- Out=In2.
merge(In1,[],Out) :- Out=In1.
merge([Msg|In1],In2,Out) :- Out=[Msg|OutTail], merge(In1,In2,OutTail).
merge(In1,[Msg|In2],Out) :- Out=[Msg|OutTail], merge(In1,In2,OutTail).
```

最初のふたつの節は、どちらか一方のストリームが終りまで来たら、もう一方のストリームをそのまま出力につないでしまえば良い、という意味である。残りのふたつの節が、どちらか一方にメッセージが来た時にそれを中継して出力する、ということをするためのものである。

このような、複数のストリームからの入力すべてをひとつのストリームにまとめるプロセスをマージ (merger) という。前節に述べた `append` もマージの一種ともいえるが、まず片方のストリームへのメッセージを流し、それが終わってからもう一方のストリーム、となっているので、より制限が強い。

両方のストリームともにメッセージが来ている時に、この述語 `merge` がどのような動作をするかに注目しよう。この場合、3 番目、4 番目の節は両方とも適用条件を満たしている。このようなときにどちらが選ばれるかは、言語仕様としては決めていない。同じプログラムを同じ入力データで動かしても、マージが実際にどのように動くかによって、流れていくメッセージの順番は毎回違うかも知れない (図 1.6)。これは KL1 の特質のひとつである非決定性が現れている例である。²³ この非決定性はプログラムのデバッグには厄介な性質だが、この例のように並列性を上げるために必要な場合がある。なお、別々のストリームからのメッセージがどんな順で出力されるかは非決定的だが、もともと一本のストリームの中で前後関係があったメッセージは出力中でもその前後関係を保っている。

²³ もちろん前節の `append` のような非決定性のない書き方をすることもできる。

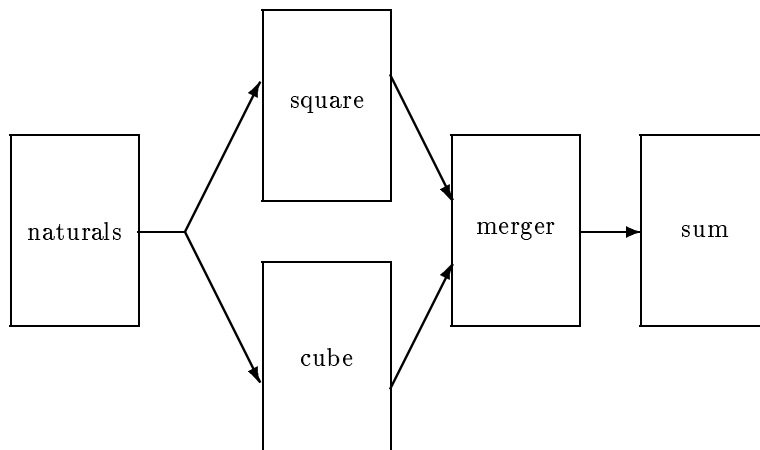


図 1.7: マージャを用いたネットワーク

この述語を使えば、全体のプログラムは以下のようになる。

```

queer_sum(N,Sum) :-
    naturals(N,Naturals), square(Naturals,Squares), cube(Naturals,Cubes),
    merge(Squares,Cubes,Both), sum(Both,Sum).
  
```

これなら両方のストリームのどちらからでも、メッセージが来るごとにどんどんプロセス `sum` に送りつけることができ、並列性の面で `append` を使ったプログラムよりも有利になっている。プロセスの全体像は図 1.7 に示すようになっている。

このように KL1 プログラムで定義するマージャでは、入力数を動的に増減するなどの処理が容易でないこと、その際の効率を良くしにくいことなどの問題がある。そこで、入力ストリーム数を増減でき、入力数の多寡にかかわらず一定の手間でマージできるような、組込機能としてのマージャも用意されている（ここでは詳しく述べない）。

1.5.4 ディスパッチャ

平方と立方の総和を求めるプログラムでは、平方を作るフィルタ、立方を作るフィルタの両者に同じ入力メッセージを与えれば良かった。では、そうはいかない場合 — 入力メッセージの内、あるメッセージはひとつのフィルタを、他のメッセージは別のフィルタを通したい場合はどうしたら良いだろう。

例として、こんどは与えられた数未満の自然数について、偶数は平方し、奇数は立方したものの総和を求めることを考えよう。まず、入力を偶数か奇数かに応じて、別々のストリームに振り分けて出力するようなプロセスを作れば良い。そのプログラムは以下のようになる。

```

dispatch([],Odd,Even) :- Odd=[], Even=[].
dispatch([One|Rest],Odd,Even) :- One mod 2=\=0 |
    Odd=[One|OddTail], dispatch(Rest,OddTail,Even).
dispatch([One|Rest],Odd,Even) :- One mod 2:=0 |
    Even=[One|EvenTail], dispatch(Rest,Odd,EvenTail).
  
```

もうプログラムの細かい説明をするまでもあるまい。この述語で実現されるプロセスは、入力ストリームが 1 本、出力ストリームが 2 本あり、入力メッセージの内容に応じてどちらのストリームに出力するか決めている。このようなプロセスをディスパッチャ (dispatcher) と呼ぶ。

これを用いると、プログラムの全体は以下のようになる。

```

queer_sum(N,Sum) :-
  
```

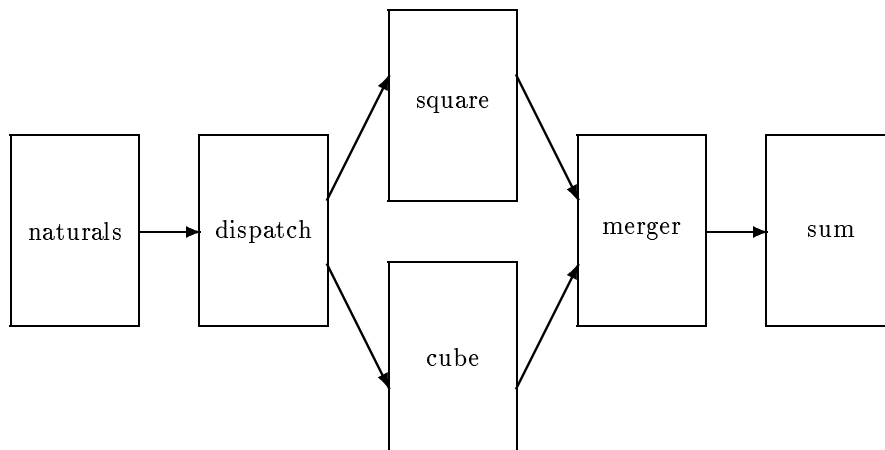


図 1.8: ディスパッチャを用いたネットワーク

```

naturals(N,Naturals), dispatch(Naturals,Odd,Even),
square(Even,Squares), cube(Odd,Cubes),
merge(Squares,Cubes,Both), sum(Both,Sum).

```

プロセス構成を図 1.8 に示す。

1.5.5 サーバ

もうひとつの重要なプロセス・ネットワークの構成要素となるプロセスに、複数のプロセスからアクセスされる共通のデータを貯めておくサーバ (server) がある。サーバに対して、ストリーム経由でそこにアクセスするプロセスをクライアント (client) と呼ぶ。

サーバプロセスの典型的な述語構造は以下のようなものになる。

```

p([message(In,Out)|S],State) :-
    compute(In,State,Out,NewState), p(S,NewState).

```

既に表示したカウンタはサーバの一例である。

ここでは KL1 のプロセスをストリームで結合していく基本的な手法を紹介した。実際のプログラムは、こうしたさまざまな手法を組み合わせることで構成していくことになる。

1.6 プログラム動作の指定

ここまで、並列に動作できる KL1 プログラムをどのようにして組んでいくかについてひと通り説明してきた。こんどは、ここまで述べた並列実行の可能性を、実際にどのようにして物理的な並列実行に結び付けていくかについて説明しよう。並列動作できるように書いたプログラムを実際に並列に動作させるための指定を、KL1 ではプラグマと呼んでいる。

1.6.1 並列実行指定の方針

まず KL1 では並列実行指定 (プラグマ) についてどのような方針を取ったのか、それはなぜなのかについて述べる。

負荷分散はプログラムで指定する

実際の並列計算機に計算を効率良く行なわせるためには、ふたつのことを考えなくてはならない。

負荷の分散 行なわなければならない計算が一台のプロセサに集中してしまうと、そのプロセサの処理が
終るまで計算が終らなくなってしまう、並列に計算している意味がない。したがって、全体の問題
をいくつかの部分問題に分割し、多くのプロセサに均等に分散して、どのプロセサにも同程度の負
荷がかかるようにしなければならない。

通信の削減 全体の問題をいくつかの部分問題に分割して分散させるには、まず問題を配るための通信が
必要で、最後には結果を集めるための通信が必要になる。また、効率の良いアルゴリズムを使うに
は部分問題が完全に独立にできず、計算途中でも通信が必要になることも多い。通信が多くなると
そのコストのために返って効率が落ちてしまうこともある。したがって、なるべく部分問題を解く
プロセサ間の通信が少なくなるような問題の分割が重要である。

この負荷分散と通信削減は二律排反関係にあり、あまり分散し過ぎると通信が多くなり過ぎ、かといって
あまり通信を減らそうとすると分散できない。この問題が起きないように問題の分割法を見つけ、また、
両者のトレードオフ点を見つけることは、並列処理ソフトウェアの研究の最重要課題といえよう。

特に問題の部分問題への分割の仕方は、解くべき問題が何なのか、どのようなアルゴリズムを使うの
かに大きく依存する。現在のソフトウェア技術では、部分問題への分割とその分散についての指針を明記
していないようなプログラムが与えられても、それに対していつでも自動的に効率的な負荷分散を行える
ようにすることは、まず不可能である。もちろん、たとえば行列のかけ算を多数行なうなど、あらかじめ
行なうべき計算の量を予測しやすい問題領域に限定すれば、かなり効率的な自動並列化を行なうこともで
きる。しかし、いわゆる知識情報処理のような、計算結果によって次に行なうべき計算が動的に大きく変
わることがあたりまえの分野では、自動負荷分散は非常に難しい。

現在の技術がこういう水準にあるので、KL1 はこの困難な自動負荷分散の方式の研究ツールとして役
に立てるものと位置付けている。したがって、負荷分散は言語処理系で自動的にには行おうとせず、プログ
ラムが指定するものとした。ただし、プログラムがさまざまな負荷分散指定を行なうときに、できるだけ
簡単にできるように工夫している。それについては次節に述べる。

ふたつの並列性を分離する

KL1 では二種類の並列性を別々に指定するものとしている。二種類の並列性とは、以下のようなもの
である。

論理的並列性 プログラムのどの部分が並列に動作しても良いかを指定するもの。次に述べる物理的な
並列性と区別して、並行性と呼んだりもする。これはプログラムの正しさに関わるもので、指定が
誤っていればプログラムは正しく動作しないかも知れない。KL1 ではガードで節の選択条件を指
定すると、選択条件が判定できるデータがそろうまで自動的に実行を遅らせるので、並列に動作し
て良いかどうかは暗黙に指定することになる。

物理的並列性 論理的には並列に動作して良い部分の内、どれを実際に並列に動かすかを指定するもの。
この指定による並列動作は、もともと並列動作して良いものの内から指定するのだから、プログラ
ムの正しさには影響しないが、プログラムの実行効率は大きく左右する。KL1 ではこれはプラグマ
(pragma) と呼ばれる機能を用いて指定する。

論理的並列性と物理的並列性の指定を分離すれば、プログラムの正しさに影響を与えることなく負荷分散
の仕方だけを変えることができる。分散方式の研究を行なうためにはさまざまな分散方式を実験する必
要が生じるが、その際にただでさえ困難な並列プログラムのデバッグを、毎回改めてやり直す必要がなけ
れば、実験の効率を著しく高めるだろう。

なお、プラグマはプログラムの正しさとは分離された効率のためだけの指定なので、言語処理系はこ
れに従わない方が効率が良いと判断できれば、必ずしも従わない場合もあってよい。

以下、このプラグマの指定方法について概説する。

1.6.2 ゴール分散プラグマ

計算の分散には実行ノード指定のためのプラグマを用いる。指定には

$$Goal@node(Node)$$

のような形式でボディ・ゴールにプラグマを付加する。ここで指定するノード (node) とは、個別のプロセサ、または内部では自動負荷分散するようなプロセサの集まりである。現在のところ、指定にはノードについて一連番号を用いている。

ゴール分散を用いる具体例をあげると、たとえば以下になる。

$$p([One|Rest], N, State) :- q(One, State, New)@node(N), N1:=N+1, p(Rest, N1, New).$$

この例では、次々に到着するメッセージについての処理 (q) を、順番に各ノードに分散しているわけである。何も指定のないゴール (この場合なら足し算と再帰呼び出しのゴール) は、元のゴールを実行したのと同じノードで実行する。

1.6.3 ゴール優先度指定プラグマ

並列に実行できるゴールが複数あり、プロセサの数がそれよりも少ない時は、どのゴールから順に実行するかが効率を大きく左右する場合が少なくない。そこで、どのゴールを先に実行した方が効率上有利かの示唆を与えるためのプラグマが、ゴール優先度指定プラグマである。指定はボディ・ゴールに

$$Goal@priority(Priority)$$

のようなプラグマを付加して行なう。この *Priority* が具体的な優先度の値を指定する部分だが、ここでは詳細は述べない。

優先度の絶対的な値はどうでも良いが、とにかく現在実行中のゴールの優先度よりも低い優先度で実行したい、という場合は多い。そのような場合は:

$$Goal@lower_priority$$

のように指定すればよい。

優先度指定プラグマは必ずしも守られるとは限らない。必ず守ろうとすると効率的な実装が難しくなり、返って効率低下の原因となるからである。また、優先度の指定はひとつのノード内でだけ有効で、他のノードにもっと優先度の高いゴールがあっても、それを移動して来て先に実行したりはしない。ノード間に渡りような優先度管理を常に行なおうとすると、優先度管理がシステム全体のボトルネックになってしまうからである。

1.6.4 節優先度指定プラグマ

複数の節の選択条件が真である場合、どの節を選ぶかは言語仕様としては定めていない。どれを選んでも正しく動くようにプログラムを書くべきであることは前に述べた。しかし、どの節を選ぶかによって実行効率に影響を与える場合もある。そこで、どちらの節も選べる場合に、どの節を選ぶと有利かについての示唆を与えるためのプラグマが、節優先度指定プラグマである。節の優先度指定には、まず優先して欲しい節 (複数でも良い) を先に書き、他の節との間に “alternatively.” という節のようなものを書く。

節の優先度も必ずしも守られるとは限らない。やはり必ず守ろうとすると効率的な実装が難しくなるからである。

1.7 一階述語論理と KL1

論理型言語というのは「問題を述語論理で記述すればそのままプログラムとして動く」という宣伝がよくなされていた。この影響が、論理型言語は人工知能研究向きの特種なプログラム言語で、一般のプ

プログラミングとは無縁であるという誤解が、まだ一部に根強くあるようだ。そうした偏見に染まることなく KL1 の並列プログラム言語としての言語仕様を理解していただきたかったので、ここまでの説明では KL1 と一階述語論理との関係についてあえて触れずに来た。ここまでの説明を理解していただければ、KL1 はそれほど風変わりな言語ではないということも、理解していただけたのではないだろうか。

実際、KL1 は (Prolog とは違って) 「問題を述語論理で記述すればそのままプログラムとして動く」といったことは最初から考えていない。並列処理の記述に必要な仕様を素直に採り入れたら、いや、むしろ従来のプログラム言語に意識的・無意識的に採用されてきた逐次処理特有の機能を排除したら、ごく自然にこのような言語になるのである。

そうはいても、KL1 の設計は論理型言語として出発し、その仕様の根幹には論理型言語としての血が脈々と流れている。ここでは、一階述語論理と KL1 の関係について、ごく簡単に触れておきたいと思う。なお、以下の説明の大部分は KL1 だけでなく、Prolog などの他の論理型言語にもあてはまるものである。

1.7.1 プログラムは公理の集まり

KL1 プログラムを構成する節は

述語名 (引数, ...) :- ガード | ボディ.

という形式をしている。ここで縦棒 (“|”) を取り除いて、ガードとボディの区別をなくしてしまうと、節の形は

述語名 (引数, ...) :- ゴール,

という形式である。

前に述べたように、ガードにあるユニフィケーションを、ヘッド中に直接相手を書くように略記してしまうことができた。ガードとボディの区別をなくしてしまったので、もとはボディにあったユニフィケーションにも、この規則を適用することにする。たとえばストリームを結合するプログラム

```
append([], In2, Out) :- Out = In2.
append([Msg|In1], In2, Out) :-
    Out = [Msg|OutTail],
    append(In1, In2, OutTail).
```

について、このような書き換えをし、変数名や述語名を少し変えると

```
a([], 0, 0).
a([M|I1], I2, [M|O]) :- a(I1, I2, O).
```

となる。

このような変換をした結果は、一階述語論理の公理して解釈することができる。その意味づけは、“:-” の右辺が成り立つならば、左辺が成り立つ (右辺が空なら、左辺は無条件に成り立つ) ということで、上の append の例のふたつの節を論理式として書けば、それぞれ

$$\forall o A([], o, o) \tag{1.1}$$

$$\forall m, i_1, i_2, o A(i_1, i_2, o) \rightarrow A([m|i_1], i_2, [m|o]) \tag{1.2}$$

ということになる。日本語で書き下せば、(1.1) は「どんな o についても、 $A([], o, o)$ が成り立つ」、もうちょっと解釈を入れると「空リストと何かを結合したものは、その何かである」という意味である。(1.2) は「どんな m, i_1, i_2, o についても、 $A(i_1, i_2, o)$ が成り立てば $A([m|i_1], i_2, [m|o])$ が成り立つ」、解釈を入れて「 i_1 と i_2 を結合したものが o になっているのなら、 $[m|i_1]$ と i_2 を結合したものは $[m|o]$ になっている」ということになる。

1.7.2 実行は証明過程、計算結果は反例

このように節を公理に対応づけると、KL1 のプログラムの実行過程は、トップレベルに与えられた命題の証明過程であると考えることができる。たとえば、以下のようなメインプログラムを考える。

```
main :- a([1,2], [3,4,5], L).
```

これを論理として解釈すると

$$\forall l \neg A([1, 2], [3, 4, 5], l) \quad (1.3)$$

つまり「どんな l をもってきてても、それに対して $A([1, 2], [3, 4, 5], l)$ は成り立たない」、いい換えれば「 $[1, 2]$ と $[3, 4, 5]$ を結合したようなものは存在しない」という意味である。プログラムの実行は、この命題が成り立たないことを証明する過程である。

ある命題を成り立たないことを証明するには、反例をあげれば良い。論理型言語による計算とは、この反例を見つけることである。その反例自身が計算の結果になっている。上記の `append` の場合なら、 $\forall l \neg A([1, 2], [3, 4, 5], l)$ の反例、すなわち $A([1, 2], [3, 4, 5], l)$ を満たす l を見つけてやれば良い。これは、 $[1, 2]$ と $[3, 4, 5]$ を結合したらどんなリストになるかを計算していることに他ならない。

KL1 や Prolog が属するホーン論理に基づく論理型言語では、反証したい命題から始めてトップダウンに、公理を用いてブレークダウンしていく方法でこの証明を行なう。このようなやり方を後向き推論 (backward reasoning) などとも呼ぶ。²⁴ 実際の証明手続きをちょっと追ってみると、以下ようになる。

1. 公理である (1.2) から

$$A([2], [3, 4, 5], x) \quad (1.4)$$

が成り立つような x があれば、 $l = [1|x]$ が元の命題 (1.3) の反例になっていることがわかる。そこで、命題 (1.4) を満たすような x がないか探すことにする。

2. ステップ 1) とまったく同様に (1.2) から

$$A([], [3, 4, 5], y) \quad (1.5)$$

を満たすような y があれば、 $x = [2|y]$ が命題 (1.4) を成立させるような例になっていることがわかる。そこで、こんどは (1.5) を満たすような y がないか探すことにする。

3. 公理 (1.1) によって、 $y = [3, 4, 5]$ としたら、(1.5) が成り立つことがわかる。

以上から、 $l = [1|x] = [1, 2|y] = [1, 2, 3, 4, 5]$ が元の命題 (1.3) の反例になっていることがわかった。

1.7.3 証明系としての Prolog と KL1

ここまでの説明は、原則としては KL1 であっても Prolog であっても同じことである。上に述べた証明のステップでは、証明がうまく進むように、適当な公理 (プログラム上では節) を選んでいったので、一本道の簡単な証明ができた。しかし、証明したい命題に適用できる公理が数多くあるもっと複雑な問題になると、自動的に適切な公理の選択をすることは不可能である。そのような場合にどうするかが問題になる。

Prolog は、最初の公理をまず選んで証明を進め、それで行き詰まったら公理を選ぶところまで後戻りして次の公理を選んでみるという、バックトラッキング機構を採り入れた。これで公理の選び方の問題は半分は解決したのだが、とりあえず選んだ公理を用いた証明が、行き詰まることなく無限ループに入ってしまうことがあるので、すべての場合でうまく行くわけではない。Prolog のセマンティクスを変えずに並列処理するシステムの研究も行なわれているが、このあたりの方式については基本的には同じである。

バックトラッキングの処理は並列処理との相性があまり良くない。何人かのグループで、ある方針にしたがって共同作業をしている時に、行き詰まったからこれまでやってきた方法は捨てて別の方法でやる

²⁴ 自動定理証明では、逆に公理から始めて証明できる命題を生成して行く、ボトムアップの方法もある。エキスパート・システムなどで良く使われる前向き推論 (forward reasoning) というのも同じである。

う、などと各人が勝手に言っているようでは、到底効率のよい作業はできない。誰か管理者が全体の作業状況を見ていて、一斉に方針変更するように指示を出す、という方針ならもう少しましになるだろうが、そうやって管理者が把握できる人数は知れている。計算機による並列処理の場合にも、これと同じような状況が起きる。

そこで KL1 では、どの公理を選ぶかを決めたら決して後戻りしない方針をとった。そのかわり、どの公理を選ぶかの条件となる部分を「ガード」として分離し、その条件を正確に判断できる情報が集まるまでは選択を待つことによって、誤った選択をしないようにしたわけである。このような方針をとる論理型プログラム言語は、自らの選択に賭けるというニュアンスから committed choice 型の並列論理型言語などと呼ばれることもある。

Prolog も KL1 も、ホーン論理についての健全な証明系になっている。つまり、どちらでも実行がうまく終了すれば、出てきた結果がプログラムを公理系と解釈した場合の正しい定理になっているのである。このことは、プログラムの理論的解析などにとって便利な性質である。

一方、どちらも、完全な証明系ではない。Prolog でも KL1 でもうまく証明できない定理がある。カットや組込述語などの付加機能のない純粋な Prolog では、この不完全性は、実行が終らないことがあるという形で現れる。KL1 では同様に実行が終らないことがあるのに加えて、失敗という形で終ることもある点が異なっている。このため、プログラム言語処理系をそのままホーン論理の自動証明器としてみると、Prolog に比べて証明できる範囲がやや狭まっているわけだ。KL1 はその部分をあきらめることによって、並列プログラム言語として必要な機能を充実しているのである。

第 2 章

プロセス・ネットワーク

プロセス・ネットワークとして KL1 プログラムを組むことが推奨される理由には次の二つがある。

- KL1 の実行機構, 特にデータフロー同期機構からいって自然であること。
- プロセス・ネットワークを意識したプログラムは, データの流れの明確化と処理の抽象化が成されているので, バグが少ないこと。

本章では, 前章の復習も兼ねて, 基本的なプロセスとプロセス・ネットワークについて概観する。そして計算終了判定と解の回収について簡単に述べた後, プロセス・ネットワーク構築上の注意点について説明する。そして演習問題を行う。

2.1 基本的なプロセスとプロセス・ネットワークの復習

2.1.1 木構造

プロセス・ネットワークの基本の一つは木構造である。

前章では次のようなプログラムが出てきた。この中にも木構造が存在する。

```
queer_sum(N,Sum) :-  
    naturals(N,Naturals), dispatch(Naturals,Odd,Even),  
    square(Even,Squares), cube(Odd,Cubes),  
    merge(Squares,Cubes,Both), sum(Both,Sum).
```

これは ‘N より小さい偶数の平方と N より小さい奇数の立方の総和を求める’ という問題を解くプログラムであった。自然数を偶数と奇数に分ける, すなわち入力分割する (述語 `dispatch/3` によって実現される) ディスパッチャプロセス, 入力の平方を作って出力する (述語 `square/2` によって実現される) フィルタプロセス, 入力の立方を作って出力する (述語 `cube/2` によって実現される) フィルタプロセス,

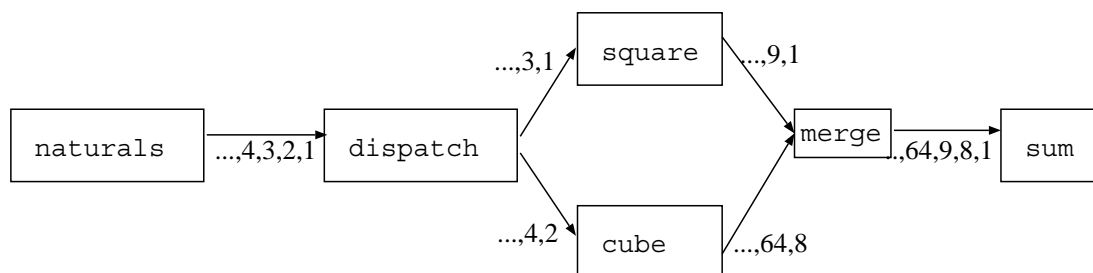


図 2.1: 木構造プロセス・ネットワーク

そして (述語 `merge/3` によって実現される) マージプロセス, といったプロセスがこのプログラム中に現れている。

このプログラムは, (`naturals/2` によって実現される) 自然数生成プロセスから (`square/2`, `cube/2` によって実現される) フィルタプロセスまでの, 入力を分割していく木構造, それらフィルタプロセスから (`sum/2` によって実現される) 総和計算プロセスまでの, マージプロセスを使って部分解を集めて全体の解を作り出す木構造, この二つの木構造プロセス・ネットワークを実現する。このような木構造を持ったものがプロセス・ネットワークの基本の一つである。

前章の復習も兼ねて, 上のプログラム中の各プロセスを実現する述語についてみていこう。

まず, ディスパッチャプロセスを実現する述語 `dispatch/3` である。

```
dispatch([],Odd,Even) :- Odd=[], Even=[].
dispatch([One|Rest],Odd,Even) :- One mod 2 == 0 |
    Odd=[One|OddTail], dispatch(Rest,OddTail,Even).
dispatch([One|Rest],Odd,Even) :- One mod 2 \= 0 |
    Even=[One|EvenTail], dispatch(Rest,Odd,EvenTail).
```

第 1 引数が入力ストリーム, 第 2 及び第 3 引数が出力ストリームである。入力ストリームから流れてきた入力がある条件を満たしたら出力ストリームのどちらかにそれを流す, その条件と出力ストリームの組ごとに節が用意されている。第 1 節は, 入力ストリームが閉じられたら出力ストリームを全て閉じて (再帰的に自らを呼ばないこと) 自ら消滅するために用意されている。この第 1 節は後述するようにプログラムの終了判定にとって重要である。

次にフィルタプロセスを実現する述語 `square/2`, `cube/2` である。

```
square([],Out) :- Out=[].
square([One|Rest],Out) :-
    Square:=One*One, Out=[Square|OutTail], square(Rest,OutTail).

cube([],Out) :- Out=[].
cube([One|Rest],Out) :-
    Cube:=One*One*One, Out=[Cube|OutTail], cube(Rest,OutTail).
```

第 1 引数が入力ストリーム, 第 2 引数が出力ストリームである。入力ストリームに流れてきた入力に処理を施し (ここでは平方や立方をとり) その結果を出力ストリームに流している。入力によって施す処理を変えたい時にはディスパッチャと同様に節を幾つか用意してやれば良い。第 1 節は, ディスパッチャの第 1 節と同様に, 入力ストリームが閉じられたら出力ストリームを全て閉じて自ら消滅するために用意されている。

フィルタプロセスを幾つか直列につなげばパイプライン・プロセス・ネットワークが出来る。入力 N に対して N^6 を求めるパイプラインは, 次のような述語で実現できる。

```
pipe(In,Out) :- true | square(In,Mid), cube(Mid,Out).
```

そして, マージプロセスを実現する述語 `merge/3` である。

```
merge([],In2,Out) :- Out=In2.
merge(In1,[],Out) :- Out=In1.
merge([Msg|In1],In2,Out) :- Out=[Msg|OutTail], merge(In1,In2,OutTail).
merge(In1,[Msg|In2],Out) :- Out=[Msg|OutTail], merge(In1,In2,OutTail).
```

入力ストリームである第 2 引数と第 3 引数のどちらかに入力の流れが来たら, それをそのまま出力ストリームに流す, ということをしているのが第 3 節, 及び第 4 節である。第 1 節, 第 2 節は, 入力ストリームのどちらかが閉じられたら, もう一方を出力ストリームに直結し, 自ら消滅するために用意されている。

2本の入力ストリームに同時に入力が出てきた場合の非決定性(どちらが先に出力ストリームに流されるかは決められていないこと)はマージの問題点ではなく、KL1の特質であることは前章で述べた。マージには入力ストリームの本数の問題があるが、これについては後に詳しく述べる。

プログラムが大規模になってくると全体を一つのフラットな木構造にすることは難しくなる。そのような場合にはネストした木構造にするのが良い。

探索問題では、問題を部分問題に分割し、各部分問題を各プロセスに解かせ、その部分問題の解を回収して全体の解を得るという分割統治法がよく用いられるが、この木構造プロセス・ネットワークはその解法によくマッチする。探索問題の多くはこの木構造を基本としたプロセス・ネットワークで解ける。

ところで、上述の `square/2`、`cube/2` などの述語によって実現されるプロセスは、第1引数の入力ストリームに入力が出てきた時になって始めて動き出す。流れてくる前は中断している。つまりこれらのプロセスはプロセス・ネットワークの中でデータ駆動的に動いている。しかし、このデータ駆動だけで全てがうまくいくわけではない。

2.1.2 要求駆動とサーバプロセス

ある与えられた数よりも小さいフィボナッチ数が何個あるかを求めるプログラムを組んでみよう。もし、与えられた数までのフィボナッチ数を生成するプロセスが既にあったとしたら、今までのデータ駆動という考え方に基けば、それとカウンタプロセス(入力ストリームを持ち、そこに何個の入力が出てきたかをカウントすることを基本動作とするプロセス)をストリームで結ぶ、というのが方針の一つとして立つ。これに基づいたプログラムが次のものである。

```
go(Target, Answer) :- true |
    fibonacci(0, 1, Stream, Target),
    consume(Stream, 0, Answer).

fibonacci(N1, N2, Stream, Target) :- N2 >= Target |
    Stream = [].
fibonacci(N1, N2, Stream, Target) :- N2 < Target |
    Stream = [N2 | StreamN],
    N3 := N1 + N2,
    fibonacci(N2, N3, StreamN, Target).

consume([], Count, Answer) :- true |
    Answer = Count.
consume([X | StreamN], Count, Answer) :- true |
    CountN := Count + 1,
    consume(StreamN, CountN, Answer).
```

与えられた数までのフィボナッチ数を生成し出力ストリームに流すことを基本動作とするプロセスをフィボナッチプロセスとする。`fibonacci/4`がそのフィボナッチプロセスを実現する述語である。第4引数がその与えられた数、第3引数が出力ストリームである。

カウンタプロセスを実現するのが述語 `consume/3` である。第1引数が入力ストリームであり、カウント結果は第3引数の変数に具体化される。ストリームを介してカウンタプロセスがデータ駆動的に動くのが分かる。ところで、ここでKL1のゴール実行順序に関する仕様を思い出すと、KL1では同一の優先度を持つゴール間の実行順序は特に規定していなかった。すると、このプログラムの場合、フィボナッチプロセスがカウンタプロセスより早く動く可能性がある。そうすると、フィボナッチプロセスが“Target”までのフィボナッチ数を生成して終了してから初めてカウンタプロセスが動き出す、つまり、“Target”が大きい値だった場合はそのフィボナッチプロセスが大量のメモリを消費してしまう、という可能性がある。このフィボナッチプロセスとカウンタプロセスの間で同期を取ってない点で、上のプログラムには欠陥がある。

そこで、このプログラムを要求駆動的¹ に書き換え、プロセス間で同期を取るようにする。つまり、カウンタプロセスからの要求がある度にフィボナッチプロセスが一つだけフィボナッチ数を生成してカウンタプロセスに渡すようにする。KL1 実行機構を思い出せば、要求がくるのを待つようなフィボナッチプロセスはすぐに書けるだろう。カウンタプロセスは単なるカウントだけではなく、フィボナッチプロセスにフィボナッチ数生成を要求し、“Target” より小さいかどうかのチェックをする働きをする。プログラムを次に示す。

```

go(Target,Answer) :- true |
    fibonacci_lazy(0,1,Stream),
    consume(Stream,Target,Answer).

fibonacci_lazy(_,_,[]) :- true | true.
fibonacci_lazy(N1,N2,[Box|Stream]) :- true |
    N3 := N1 + N2,
    Box = N3,
    fibonacci_lazy(N2,N3,Stream).

consume(Stream,Target,Answer) :-
    NewStream = [Box|Stream],
    consume(Box,Target,0,NewStream,Answer).

consume(Box,Target,Count,Stream,Answer) :- Target =< Box |
    Stream = [],
    Answer = Count.
consume(Box,Target,Count,Stream,Answer) :- Target > Box |
    NewCount := Count + 1,
    Stream = [NewBox|NewStream],
    consume(NewBox,Target,NewCount,NewStream,Answer).

```

ストリームの流れが逆であることに注意されたい。fibonacci_lazy/3 は、第 3 引数のストリームから結果を与える変数 “Box” が到着したら、その変数にフィボナッチ数を入れ (“Box” にフィボナッチ数を具体化し) 次の結果を与える変数の到着を待つ。この例題では変数 “Box” の到着、即ち第 3 引数がリストセルにユニファイされることがフィボナッチ数の計算要求を意味する。逆に言えばこのストリームが閉じられることがプロセスの終了を意味する。すなわち fibonacci_lazy/3 は要求駆動的にフィボナッチ数の計算を行なう述語であり、また答えの容器の到着を待つという意味でデータ駆動でもある。一方 consume/5 によって実現されるカウンタプロセスは、第 2 節の第 2 ボディゴールで “Stream = [NewBox|NewStream]” つまりストリームに変数 “NewBox” を流している。これがフィボナッチプロセスに ‘次のフィボナッチ数を生成して “NewBox” に入れろ’ と要求を出していることになるのである。そしてその要求の結果がそこに入れられる (具体化されてくる) のをガードで待っている。尚、プログラムの都合上、consume/3 の第 1 ボディゴールで最初の要求を出している。

未完成メッセージの技法を使って書くことも出来る。この方式の方が意味を表現し易く、特にメッセージ駆動と呼ぶことがある。次がメッセージ駆動的に書き直したプログラムである。

```

go(Target,Answer) :- true |
    fibonacci_lazy(0,1,Stream),
    consume(Stream,Target,0,Answer).

fibonacci_lazy(N1,N2,[]) :- true | true.

```

¹ 現在のところ KL1 には要求駆動を明示する枠組はないので、あくまで記述されたプログラムの意味として解釈されたい。

```

fibonacci_lazy(N1,N2,[make(X)|Stream]) :- true |
    X = N2,
    N3 := N1 + N2,
    fibonacci_lazy(N2,N3,Stream).

consume(Stream,Target,Count,Answer) :- true |
    Stream = [make(X)|StreamN],
    consume(StreamN,Target,Count,Answer,X).

consume(Stream,Target,Count,Answer,X) :- Target <= X |
    Stream = [],
    Answer = Count.

consume(Stream,Target,Count,Answer,X) :- Target > X |
    CountN := Count + 1,
    consume(Stream,Target,CountN,Answer,X).

```

consume/4 と consume/5 の二つの述語でカウンタプロセスが実現される。カウンタプロセスは‘次のフィボナッチ数を生成して “X” に入れろ’ という要求メッセージである “make(X)” という未完成メッセージをストリーム流す。そして、“X” がフィボナッチプロセスによってフィボナッチ数に具体化されるのを待ち、カウントしたのち、次の要求を出す。“X” に具体化される値が “Target” 以上になったら、ストリームを閉じ、答えを “Answer” に具体化して、消滅する。

フィボナッチプロセスは、要求を受けて計算を行ない結果を要求者に返す、という点でサーバと見ることも出来る。このサーバも KL1 プロセスの重要なプロセスの一つである。

2.2 解の回収と計算終了判定

図 2.1のプロセス・ネットワークのように、マージャを使って解を収集する木構造を構成することが解回収の基本的な方法である。この他に差分リストという方法があるが、これは次章で説明される。

通常の自然なプロセスは入力ストリームが閉じられると出力ストリームを閉じて自分から消滅する。マージャも入力ストリームが全て閉じられるとその出力ストリームを閉じて消滅する。逆に言えば、出力ストリームが閉じられたということがそのストリームが継っていた部分の計算が終了したことを意味する。計算終了の判定はこの出力ストリームが閉じられることを監視することで行うことが多い。しかし、このような自然なプロセスやマージャのみを使ってプロセス・ネットワークを組めない時は、それぞれのネットワークに応じた計算終了判定法を工夫する必要がある。その中の一つにショートサーキットがあるがこれは次章で説明される。

2.3 プロセス・ネットワーク構築上の注意点

この節ではプロセス・ネットワーク構築上の注意点について述べると共に、やや高度なプロセス・ネットワークについての説明も行なう。

2.3.1 入出力を分ける

どの引数が問題記述すなわち入力で、どれがそれに対する解すなわち出力であるかをはっきり分け、それを意識しながらプログラミングする。

```

a([E|X1],Y):-integer(E) |
    E1:=E+1, Y=[E1|Y1], a(X1,Y1).
a(X,[E|Y1]):-integer(E) |
    E1:=E-1, X=[E1|X1], a(X1,Y1).

```

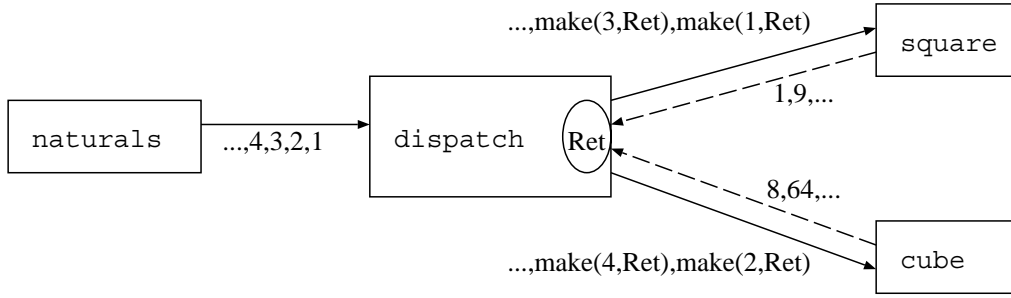


図 2.2: 木構造プロセス・ネットワーク 2

このようなストリームの向きが決まっていないプロセスも作れることは作れるが、このようなプロセスをプロセス・ネットワークに入れておくことはバグの元凶であり、またそもそもプロセス・ネットワークの各ノードの役割や結合形態を云々すること自体、難しくなる。また、KLIC 処理系の性能アップの切札として導入予定の KL1 プログラム静的解析は、入出力のはっきりしないプログラムでは効果的に働かないことも付け加えておく。

2.3.2 各ノードの役割は単純に

未完成メッセージの使い方を覚えてしまうと、章の最初に出てきたプログラムで、ディスパッチャとマージャ以降の部分と一緒にして次のように書きたくなる。

```
queer_sum(N,Ans) :-
    naturals(N,Naturals), dispatch_sum(Naturals,Odd,Even,0,Ans),
    square(Even), cube(Odd).

dispatch_sum([],Odd,Even,Sum,Ans) :- Odd=[], Even=[], Ans=Sum.
dispatch_sum([One|Rest],Odd,Even,Sum,Ans) :- One mod 2 =\= 0 |
    Odd=[make(One,Ret)|OddTail],
    dispatch_sum(Rest,OddTail,Even,Sum,Ans,Ret).
dispatch_sum([One|Rest],Odd,Even,Sum,Ans) :- One mod 2 =:= 0 |
    Even=[make(One,Ret)|EvenTail],
    dispatch_sum(Rest,Odd,EvenTail,Sum,Ans,Ret).

dispatch_sum(Rest,Odd,Even,Sum,Ans,Ret) :- wait(Ret) |
    SumN := Sum+Ret,
    dispatch_sum(Rest,Odd,Even,SumN,Ans).

square([]) :- true | true.
square([make(One,Square)|Rest]) :- true |
    Square:=One*One, square(Rest).

cube([]) :- true | true.
cube([make(One,Cube)|Rest]) :- true |
    Cube:=One*One*One, cube(Rest).
```

このプログラムによって実現されるプロセス・ネットワークは図 2.2 のようになる。

dispatch_sum/5 及び dispatch_sum_6 は、二つの述語で一つのプロセスを実現し、入力分割と、square/1 で実現されるプロセス及び cube/1 で実現されるプロセスへの要求メッセージ送出、そして平

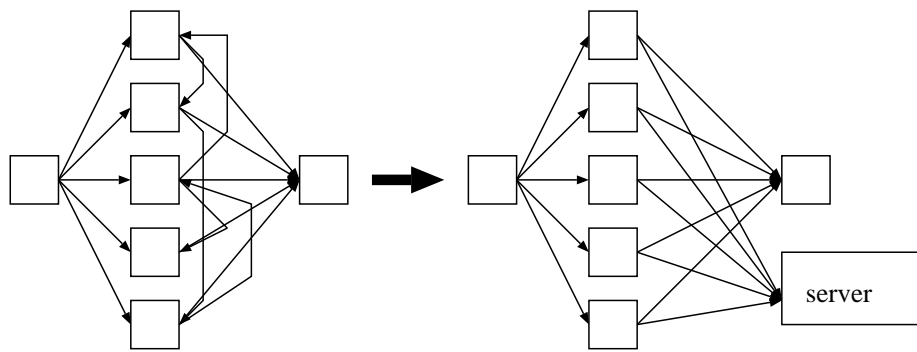


図 2.3: サーバプロセスを通した通信へ

方と立方の合計の保持も行なっている。この場合はプログラム全体の規模が小さいので問題ないが、一般に規模の大きなプログラムでこのような一人何役も兼ねたプロセスを多用すると、プログラム全体の構造が見えにくくなる。後に述べるようなサーバプロセスを作る必要性のある場合はいざ知らず、プログラム全体を素直な木構造で表現出来る場合は、できる限りマージャ、フィルタ、ディスパッチャといった単純な構成要素から考えた方がよい。

ネットワーク構造が大きくなってくると、当然モジュール化が必要になる。木構造のサブネットワークをモジュールとしてまとめ、メインネットワークのひとつのノードと考えるのが便利である。このときも各サブネットワークがマージャ、フィルタ、ディスパッチャなどの役割を果たすようにすると、プログラム全体がネストした木構造になり、構造がわかりやすくなる。

2.3.3 部分木は直接通信させない

部分問題を解く計算を通して得られたデータを、別の部分問題を解くところでも活用したくなる場合は少なくない。このような場合も、サブネットワークすなわち部分木どうしが直接通信するようにプログラムすると、プログラムの全体の木構造が崩れて、構造が把握しにくくなる。

プログラム全体は木構造のまま、部分木間でデータを共有したい場合は、共有データのためのサーバプロセスを置き、ここを経由して通信するようにすると、構造を把握しやすくなる。すなわち、図 2.3 の左のようなプロセス・ネットワークでなく、右のようなプロセス・ネットワークを構成するのである。プログラムは以下になるだろう。

```
p(I, 0) :-
    decompose(I, I1, ..., In),
    merge({S1, ..., Sn}, S),
    shared_data_server(S),
    p1(I1, S1, 01),
    ...,
    pn(In, Sn, 0n),
    compose(01, ..., 0n, 0).
```

決してとるべきでない通信形態は‘部分木のうち最初に解を見つけたものが他の部分木の計算をやめさせる’というような方法である。このような形態をとると複数の部分木がほぼ同時に解を見つけた場合に、いつでも正しく動作させるようにするのが難しくなる。解が見つかった、という事実は共有データである。解を見つけた部分木は共有データのサーバに連絡し、このサーバが他の部分木を終了させるようにするのが良い。一般に、競争条件は原則としてストリームのマージ機構に一元化すべきである。

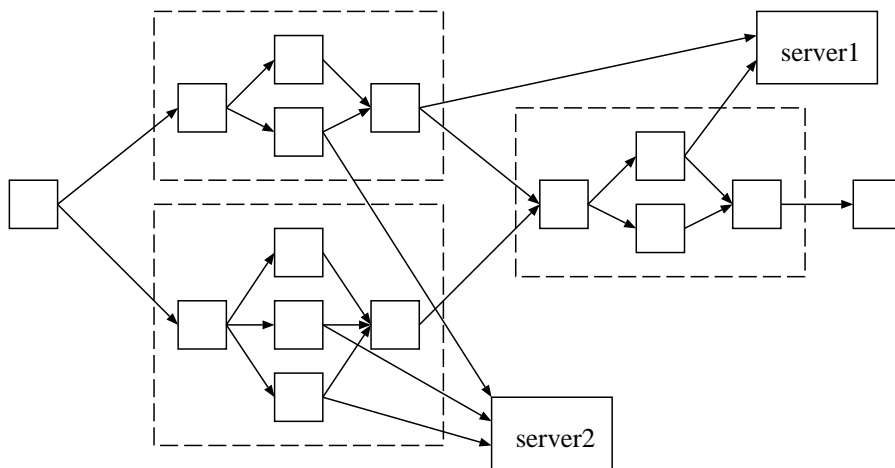


図 2.4: プロセス・ネットワーク

2.3.4 ループは避ける

ストリーム通信を使えばループ構造の正しいプログラムも書けるが、ループ構造はデッドロックの温床になるので乱用すべきではない。ループがなければデッドロックはあり得ない。

直接的なループ構造を避けるためには、やはりサーバ方式が有効である。パイプライン構造の下流から結果を直接上流にフィードバックするのではなく、データベースであるサーバプロセスに送ることとし、上流からはこれにアクセスして情報を得るようにすると、基本的なプロセス構造がすっきりして把握しやすくなるのが少なくない。

2.3.5 カオスなネットワークはやめよう

今まで述べたプロセス・ネットワークの構築に関する事柄を一つにまとめると、主構造は(パイプラインを含んだ)ネストした木構造で、サーバを通して副次ラインが張ってある図 2.4 のようなプロセス・ネットワーク、ということになる。

もちろん、この形に固執することはない。アルゴリズムに応じて、なるべく明確かつ単純な構造の KL1 プログラムを書くことを心がけるべきである。プログラムを書き始める前に、プロセス・ネットワークのデザインをさぼって成りゆきで書き始めてしまうと、カオスなネットワークが出来ることになってしまう。

2.3.6 マージャ

プロセス・ネットワークの重要な要素であるマージャには、入力ストリームの本数という問題点がある。次のマージャプログラムは入力ストリーム 2 本用のものである。

```
merge([], In2, Out) :- Out=In2.
merge(In1, [], Out) :- Out=In1.
merge([Msg|In1], In2, Out) :- Out=[Msg|OutTail], merge(In1, In2, OutTail).
merge(In1, [Msg|In2], Out) :- Out=[Msg|OutTail], merge(In1, In2, OutTail).
```

入力ストリームを 3 本、4 本と増やしたい場合には、3 本用マージャ、4 本用マージャと一つ一つプログラムを用意してやるか、2 本用マージャでもって木構造をつくる(図 2.5 参照) かしなければならない。前者の方法では、プログラムサイズが大きくなるという問題点がある。後者の方法では、入力ストリームが N 本あった場合、最低でも深さ $\log N$ の木構造を構成してやる必要があり、実行時の効率も悪くなる²。

² 一つのメッセージがマージャ木構造を通り抜けるのには、最悪、その木の深さに比例する時間と計算量がかかる

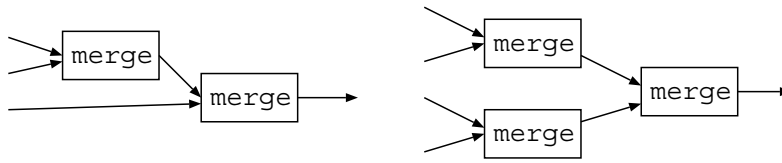


図 2.5: マージャのツリー構造

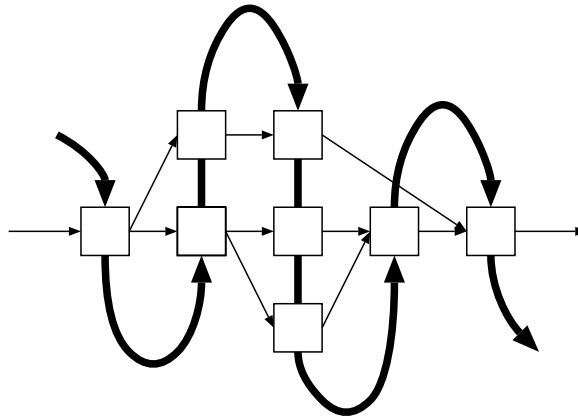


図 2.6: 状態問い合わせ用ストリーム

この問題を解決するためにKLIC 処理系では組み込み述語としてマージャをサポートする。

これは, “merge(In,Out)” という形で呼び出し, “In” に入力ストリームを要素とするベクトルを unify してやることで利用出来る。例えば, 以下の二つのボディゴールによって, “In1, In2, In3” を入力ストリーム, “Out” を出力ストリームとする入力ストリーム 3 本用マージャが用意出来る。

```
merge(In,Out),
In = { In1, In2, In3 },
```

2.4 やや高度なネットワークの組み方

デバッグのための状態問い合わせストリームというものがある。これは全てのプロセスを串刺しにしたストリーム (図 2.6) で, これをプロセス・ネットワークに組み込んでおき, デバッグ時にこのストリームを通して各プロセスの状態をかき集めてデバッグに役立てるものである。

次に, プロセス・ネットワークの使い方の面白い例を一つ挙げよう。

最適経路問題とは, 図 2.7 のようなノード間をつないだコストつき経路網が存在する時に, 開始点から終了点までの経路のうち, コスト最小のものを求めるという, 探索問題の一つである。探索問題であるから, これを部分問題に分割して, 各部分問題にプロセスを割り当てて, というように今まで述べてきた考え方に沿って問題を解いていくこともできるのだが, もっと直接的かつ大胆に, 経路上の各ノードをプロセス, ノード間のアークを双方向ストリーム³, として解くことも出来る。各ノードプロセスは自分から伸びているアークのコストを知っている。また, 各ノードはそれまでに判明した, 開始点から自分までの最適 (コスト最小) 経路とそのコストの対を保持している。そして, それよりも低コストな, 新たな自分までの最適経路を見つけた場合はその新たな最適経路とコストの対でもって保持している対を更新し, 自分に

³実際には向きが逆の 2 本の単方向ストリームとして実現する

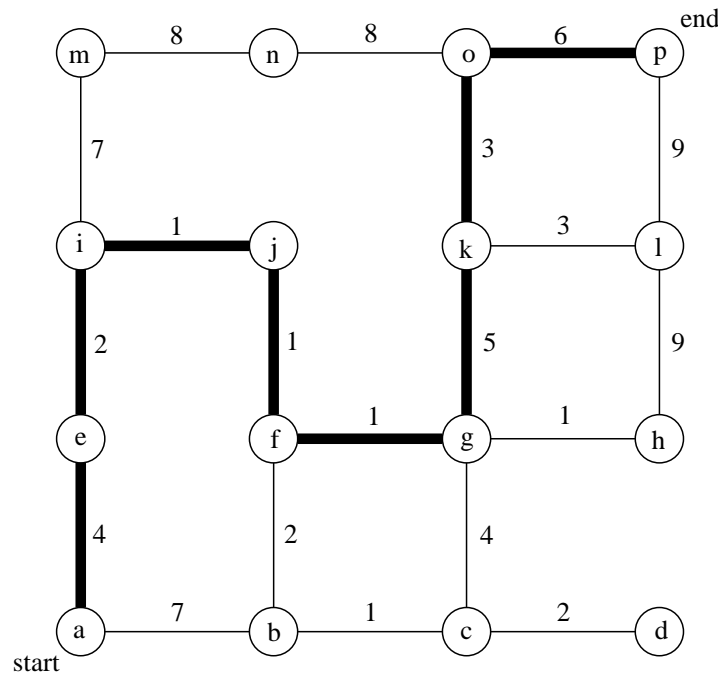


図 2.7: コスト付き経路網

直接継っているノードにその新たな対をメッセージとして送る。例えば、ノード f がある時点で、自分までの最適経路がコスト $7 + 2 = 9$ の経路 $a \rightarrow b \rightarrow f$ だと思っている (その対を保持している) とする。そこへ、ノード j から ”ノード j までの最適経路は今のところ $a \rightarrow e \rightarrow i \rightarrow j$ で、そのコストは 7 だと分かった ” という意味のメッセージ $\{a \rightarrow e \rightarrow i \rightarrow j, 7\}$ を受けとったら、自分までの最適経路が今まで思っていた b 回りの経路でなく、j 回りの経路であることが分かる。そこで、保持している対をその j 回りのものに更新し、ノード b, g, j にメッセージ $\{a \rightarrow e \rightarrow i \rightarrow j \rightarrow f, 8\}$ を送る。開始点 a がノード b, e にメッセージを送ることに始まって、このようなメッセージがプロセス・ネットワーク上を流れ回った後には、各ノードプロセスには開始点からそのノードまでの最適経路とそのコストの対が保持されている。計算の終了は全てのメッセージがプロセス・ネットワークから消滅したことで判定する (この判定は次章で述べるショートサーキットという技法が使われるがここでは述べない)。

2.5 演習問題

演習問題の前に画面への出力方法を示す。プログラムの実行結果を出力するためのストリームを確保する組み込み述語は `io:ostream/1` である。そのストリームに “`print(X)`” というメッセージを流せば、変数 “X” に具体化されている値が画面に出力される。改行するには、 “`nl`” というメッセージを送る。例えば、以下のゴールで、変数 “X” に具体化されている値を出力し、改行する。

```
io:ostream([print(X),nl]),
```

【演習問題 1】 フィボナッチ数の平方と立方の総和

与えられた整数 Target より小さな全てのフィボナッチ数について、平方と立方の総和を求めるプログラムを作成し、Target=100 とした時の実行結果を出力せよ。

【演習問題 2】 $2^l \times 3^m \times 5^n$

与えられた整数 Target より小さい正の整数で、 $2^l \times 3^m \times 5^n$ (l, m, n は負でない整数) という形で表すことの出来るものが幾つあるか求めるプログラムを作成し、Target=10000 とした時の実行結果を出力せよ。但し、パイプライン型のプロセス・ネットワークを使ってデータ駆動で書くこと。X の Y に対する剰余が Z に等しいことの判定には “ $X \bmod Y =: Z$ ”, 等しくないことの判定には “ $X \bmod Y \neq Z$ ” とガードに書けば良い。X の Y に対する商を Z に具体化するには、 “ $Z := X / Y$ ” というボディゴールを書けば良い。

【演習問題解答 3】 フィボナッチ数の平方と立方の総和

データ駆動で書いたものが次のプログラムである。本文中に出てきた fibonacci/4, square/2, cube/2, merge/3, sum/2 をストリームでつなげば良い。

```
:- module main.

main :- true |
    main(100, Answer),
    io:ostream([print(Answer), nl]).

main(Target, Answer) :- true |
    fibonacci(0, 1, Fibo, Target),
    square(Fibo, Squares), cube(Fibo, Cubes),
    merge(Squares, Cubes, Both),
    sum(Both, Answer).
```

Target=100 とした時の答えは、935596 である。

要求駆動、あるいはメッセージ駆動で書くのなら、フィボナッチプロセスに要求を出し、その結果であるフィボナッチ数を square/2, cube/2 で実現されるフィルタプロセスに配るプロセスを組み込むのが良い。以下のプログラムはメッセージ駆動で書いたものである。square/2, cube/2, merge/3, sum/2 は本文中と同じである。

```
:- module main.

main :- true |
    main(100, Answer),
    io:ostream([print(Answer), nl]).

main(Target, Answer) :- true |
    fibonacci_lazy(0, 1, Stream),
    dispatch(Stream, Target, Fibo),
    square(Fibo, Squares), cube(Fibo, Cubes),
    merge(Squares, Cubes, Both),
    sum(Both, Answer).

fibonacci_lazy(N1, N2, []) :- true | true.
fibonacci_lazy(N1, N2, [make(X) | Stream]) :- true |
    X = N2,
    N3 := N1 + N2,
    fibonacci_lazy(N2, N3, Stream).
```

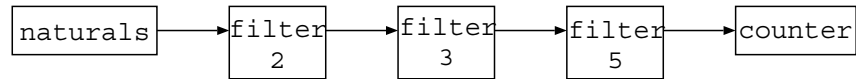



図 2.8:

```

dispatch(Stream,Target,Fibo) :- true |
    Stream = [make(X)|StreamN],
    dispatch(StreamN,Target,Fibo,X).

dispatch(Stream,Target,Fibo,X) :- X >= Target |
    Stream = [],
    Fibo = [].
dispatch(Stream,Target,Fibo,X) :- X < Target |
    Fibo = [X|FiboN],
    dispatch(Stream,Target,FiboN).

```

【演習問題解答 4】 $2^l \times 3^m \times 5^n$

入力を正の整数Nで割れるだけ割った数を出力するフィルタを用意する。例えば、N=2としたものに12を通すと3を出力するようなフィルタである。また、入力のうち1が何個あったかをカウントするカウンタプロセスを用意する。そして、N=2としたフィルタ、N=3としたフィルタ、N=5としたフィルタ、これらをつないだパイプラインの前に自然数生成プロセス、後ろにカウンタプロセスをつないで、図 2.8のようなプロセス・ネットワークを作れば、問題の式で表せる数が幾つあるかを計算するプロセス・ネットワークが出来る。

```

:- module main.

main :- true |
    main(10000,Answer),
    io:outstream([print(Answer),nl]).

main(Target,Answer) :- true |
    naturals(1,Target,Stream0),
    filter(2,Stream0,Stream1),
    filter(3,Stream1,Stream2),
    filter(5,Stream2,Stream3),
    counter(0,Stream3,Answer).

naturals(N,M,List) :- N>=M |
    List=[].
naturals(N,M,List) :- N<M |
    List=[N|Rest],
    N1:=N+1,
    naturals(N1,M,Rest).

filter(_,[],Out) :- true |
    Out = [].

```

```

filter(X,[E|InN],Out) :- true |
    filter(X,E,InN,Out).

filter(X,E,InN,Out) :- E mod X == 0 |
    EN := E / X,
    filter(X,EN,InN,Out).
filter(X,E,InN,Out) :- E mod X \== 0 |
    Out = [E|OutN],
    filter(X,InN,OutN).

counter(C,[],Res) :- true |
    Res = C.
counter(C,[1|InN],Res) :- true |
    CN := C + 1,
    counter(CN,InN,Res).
counter(C,[E|InN],Res) :- E \== 1 |
    counter(C,InN,Res).

```

Target=10000 とした時の答えは,174 である.

第 3 章

差分リストの使い方

3.1 クイックソートの原理

クイックソートは、ソートすべき数値の集合の中から一つを取りだし、その数よりも小さい集合と大きい集合との2つに分け、それぞれの集合の中で、さらに一つを取りだし、その中で大きい集合と小さい集合とに再帰的に分類していく方法である。クイックソートを使えば、 $N \log N$ 回の比較回数で済みますことが期待できる。

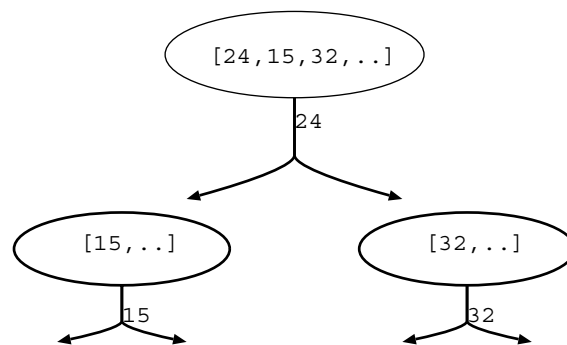


図 3.1: クイックソート

まず、リストで与えられた数列の中から数値を1つ取り出して、2つの集合に分けることを考える。(とりあえず、数値が同じだった場合のことは考えない)

```
qsort([H|T]) :- true | compare(H,T).
```

```
compare(V,[H|T]) :- V > H | compare(V,T), 小さい方の集合に H を加える.
```

```
compare(V,[H|T]) :- V < H | compare(V,T), 大きい方の集合に H を加える.
```

とすればよい。しかし、グローバル変数という概念は、KL1の世界にはないため、集合を蓄える場所として KL1 ではゴールの引数を使用(利用)する。

```
qsort([H|T]) :- true | compare(H,T,L_small,R_small).
```

```
compare(V,[H|T],L_small,R_large) :- V > H |  
    compare(V,T,L_smallにHを加えたもの,R_large).
```

```
compare(V,[H|T],L_small,R_large) :- V < H |  
    compare(V,T,L_small,R_largeにHを加えたもの).
```

では, L_{small} に H を加えたものをどう表現すればよいだろう. 長さがわからない場合はリストで表現するとよい. (長さがわかっているような場合には, 複合項やベクタを利用する方法もある.) 例えば, 1 と 2 と 3 の集合は $[1,2,3]$ と長さ 3 のリストで表現すればよい.

リストで情報をどのような順番で記録して行けばよいのであろうか. この記録の仕方には, 2 つの方法がある. スタックとキューである.

- LIFO 法 (スタック)

データを次々に積み上げて行く方法である.

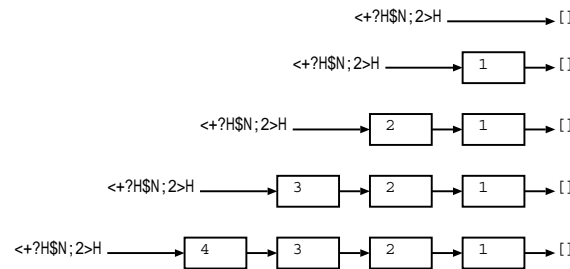


図 3.2: LIFO

述語 `compare` の場合には, L_{small} に H を加えたものは $[H|L_{small}]$ になる. 集合の初期値は, 空集合であるので, $[]$ (空リスト) にしなければならない.

```
qsort([H|T]) :- true | compare(H,T,[],[]).
```

```
compare(V,[H|T],L_small,R_large) :- V > H | New_L = [H|L_small],
    compare(V,T,New_L,R_large).
compare(V,[H|T],L_small,R_large) :- V < H | New_R = [H|R_large],
    compare(V,T,L_small,New_R).
```

- FIFO 法 (キュー)

データを次々に後尾に挿入して行く方法である.

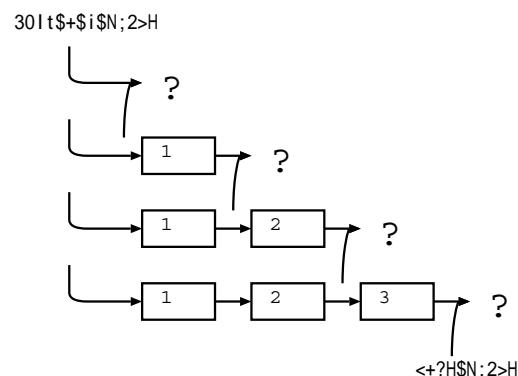


図 3.3: FIFO

L_{small} に H を加えたものは, L_{small} を $[H|$ 次に加える変数] にする. また最後は, $[]$ (空リスト) で集合を閉じるようにしなければならない.

```

qsort([H|T]) :- true | compare(H,T,ExL_small,ExR_large).

compare(V,[],L_small,R_large) :- true | L_small = [], R_large = [].
compare(V,[H|T],L_small,R_large) :- V > H | L_small = [H|New_L],
    compare(V,T,New_L,R_large).
compare(V,[H|T],L_small,R_large) :- V < H | R_large = [H|New_R],
    compare(V,T,L_small,New_R).

```

これらの記憶方法には、共に異なる特徴がある。LIFO 法では、実行中のゴールが、生成したデータの集合を常に参照でき、集合内のデータを検査しながら仕事ができる。またデータの入ってきた順とは、逆にデータが並ぶので、データの並びを逆転するのにも向いている。一方、FIFO 法では、実行中のゴールは、生成したデータの集合を参照できないが、外部のゴールから、生成したデータが見え、並列処理できる（並列性がある）。

どちらの方法がよいかは、プログラマがアプリケーションを見て判断する。この場合、LIFO 法での利点はないので、並列性のある FIFO 法の方が好ましいようである。（LIFO 法でもできないわけではない。）

これで、数を 1 つ取り出して、それよりも小さいものの集合と大きいものの集合が得られた。次に小さいものの集合の中で、さらに数を 1 つを取り出して、同様に分類するようにしなければならない。再分類する処理方法は全く同じなので、もう一度最初から呼ぶようにすればよい。

```

qsort([H|T]) :- true | compare(H,T,L_small,R_large),
    qsort(L_small), qsort(R_large).

compare(V,[],L_small,R_large) :- true | L_small = [], R_large = [].
compare(V,[H|T],L_small,R_large) :- V > H | L_small = [H|New_L],
    compare(V,T,New_L,R_large).
compare(V,[H|T],L_small,R_large) :- V < H | R_large = [H|New_R],
    compare(V,T,L_small,New_R).

```

このプログラムは、動作中は図のように “compare” が木のノードとして、足（ゴール同士がつながるストリーム）を 2 本持ち、末端に “qsort” つながるような形になる。

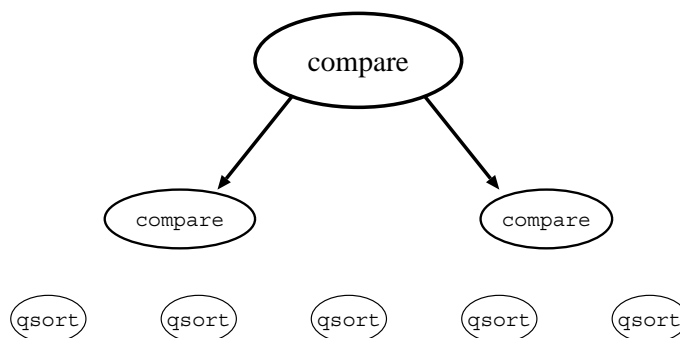


図 3.4: Binary Tree

このプログラムを述語名を木の名前にして、値が同一だった場合の処理やデータの終わりの処理を付け加えれば次のように完成する。

```

node([],V,L,R) :- true | L = [], R = [].
node([H|T],V,L,R) :- H > V | R = [H|NR], node(T,V,L,NR).
node([H|T],V,L,R) :- H < V | L = [H|NL], node(T,V,NL,R).

```

```
node([H|T],V,L,R) :- H == V | node(T,V,L,R).
```

```
terminal([])      :- true | true.
```

```
terminal([H|T]) :- true | node(T,H,L,R), terminal(L), terminal(R).
```

3.2 差分リスト

先程の例では、ソートされた木を生成することはできたが、ソートされた結果を外に出していない。結果を外に出す時には、先程の FIFO 法があるが、それだけでは結果を出力する窓口が一つしかない。すでにプログラムは、並列性を持ったプログラムであり、並列に結果を並べられるような工夫が必要となる。

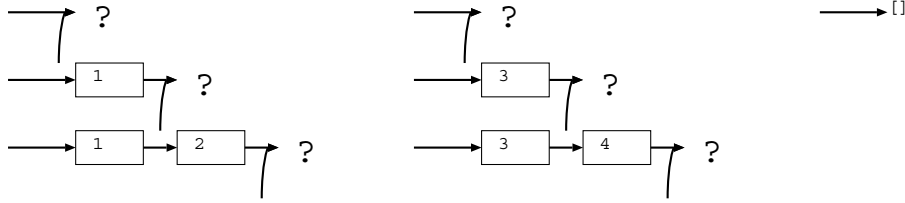


図 3.5: 並列に解が出力

ここで、出力のための引数とは別に、他のゴールの出力をつなぐための新たな変数を 1 つ設ける。これにより、複数の結果出力を並行させ、それらをつなぐようにできる。このように出力のための変数を 2 つ (直接出力のための変数と、以降のゴールの出力をつなぐための変数) を持ち回って、並列処理を損なうことなく結果を収集する技法を差分リストという。

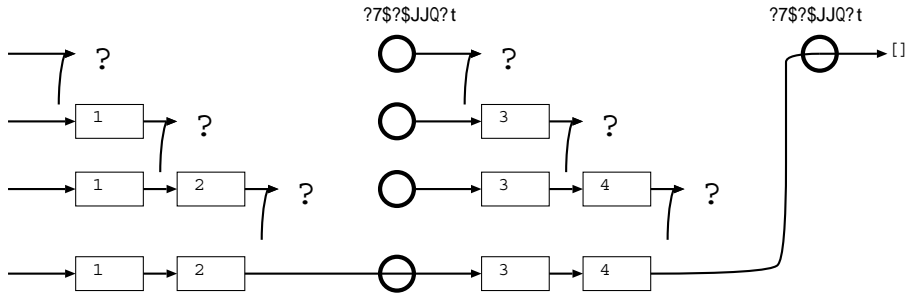


図 3.6: 解の連結

差分リストは、ノード間の解を集めるのに有効な技法である。クイックソートの解を出力するため、差分リストの 2 引数 (解を出力するための引数と解同士をつなぐ新たな変数の引数) を加える。プログラムは次のようになる。

```
qsort(In,Out) :- true | terminal(In,Out,[]).
```

```
node([],V,L,R,Xs,Ys) :- true | L = [], R = [], Xs = [V|Ys].
```

```
node([H|T],V,L,R,Xs,Ys) :- H > V | R = [H|NR], node(T,V,L,NR,Xs,Ys).
```

```
node([H|T],V,L,R,Xs,Ys) :- H < V | L = [H|NL], node(T,V,NL,R,Xs,Ys).
```

```
node([H|T],V,L,R,Xs,Ys) :- H == V | node(T,V,L,R,Xs,Ys).
```

```
terminal([],Xs,Ys) :- true | Xs = Ys.
```

```
terminal([H|T],Xs0,Ys) :- true |
    terminal(L,Xs0,Xs1), node(T,H,L,R,Xs1,Xs2), terminal(R,Xs2,Ys).
```

“qsort” から “terminal” に渡された差分リストは最終的に “node” に渡り、入力ストリームが閉じられた時に解を出力するようにしている。ここで “node” は、差分リストを持ち回っているが、その差分リストを処理しているのは、入力ストリームが閉じられた時にのみ差分リストの単一化が実行されている。単一化を処理するのはいつでもよいから、“node” を呼び出している “terminal” の中で処理することができ、“node” への引数として渡す必要がないことがわかる。

このようにクイックソートの原理を素直に書き下し、整形していくと、次のような (例題プログラムとしてよく出される) クイックソートのプログラムになる。(例題では node → split or partition, terminal → qsort になっていることが多い。)

```
qsort(In,Out) :- true | terminal(In,Out,[]).

node([],V,L,R) :- true | L = [], R = [].
node([H|T],V,L,R) :- H > V | R = [H|NR], node(T,V,L,NR).
node([H|T],V,L,R) :- H < V | L = [H|NL], node(T,V,NL,R).
node([H|T],V,L,R) :- H == V | node(T,V,L,R).

terminal([],Xs,Ys) :- true | Xs = Ys.
terminal([H|T],Xs0,Ys) :- true |
    terminal(L,Xs0,Xs1), node(T,H,L,R), Xs1 = [H|Xs2], terminal(R,Xs2,Ys).
```

3.3 ショートサーキット

差分リストは、結果を出力するための有力な方法だが、出力された結果が全て求まったことは、結果が求まったタイミングを得ることができる方法であると言える。すなわち、差分リストの考え方は (結果を見たいのではなく、) 処理終了のタイミングを得たい時にも応用が効く。各ゴールは、ある (通常、処理が終了した) タイミングで差分リストの 2 引数を閉じるように (単一化) すれば、外から差分リストの先頭を参照しているゴールが、差分リストの終端が見えることで、すべてのゴールの処理が終了したタイミング得ることができる。

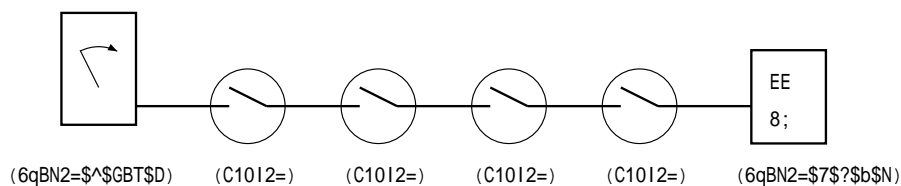


図 3.7: ショートサーキット

プログラムでは、下記のようにして監視ゴールが、仕事の終了のタイミングをつかむ。

```
job :- inspector(C0),
    worker1( ... ,C0,C1),
    worker2( ... ,C1,C2),
    .
    .
    workerN( ... ,CN,[]).

inspector(W) :- wait(X) | next_job.
```

こうして、差分リストのようにゴール間に 2 引数を付加してタイミングを得るための技法をショートサーキットと呼ぶ。

3.4 出力ストリーム

KL1 は並列言語であり、一般の手続型言語のように処理順が明確に規定されているわけではない。(KL1 ではゴールの実行順を規定するのは、ゴール間の引数の同期の機構だけである。)

このような並列言語でデータを出力するような場合に、単純にデータを出力するような機能を用意してプログラムを書いてしまうと、一つしかない出力装置を取り合うことになってしまう。

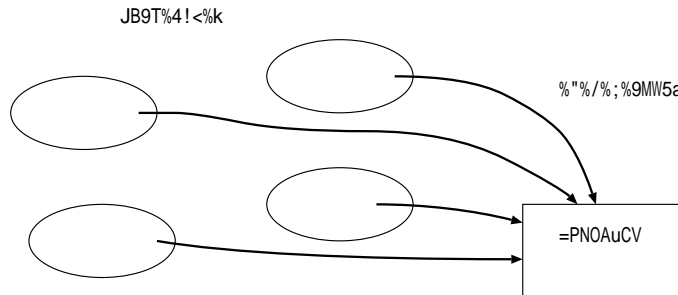


図 3.8: アクセスの競合

一つしかない共有資源を、複数の処理単位 (ゴール) が並列にアクセスするような場合には、あらかじめ一本のストリームを用意して各ゴールからのアクセスを差分リストで渡すようにするか、アクセスを取りまとめるような仲介者 (merger) を介在させて、最終的に一本のストリームにまとめるかしなければならない。

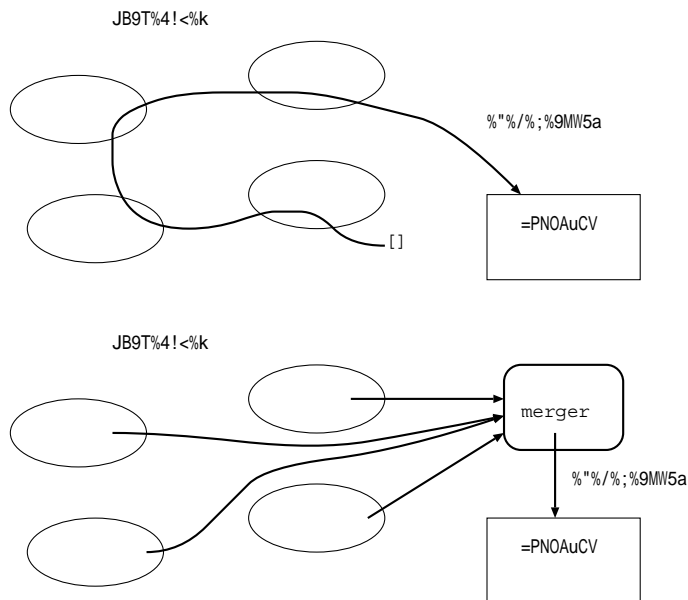


図 3.9: 共有資源へのアクセス

このような事情から klic では、直接出力装置にアクセスする手段はなく、一本のアクセス要求ストリームを処理するデータ出力のための組込み述語 (outstream/1) を提供している。


```
job :- true | work(0), io:outstream(0).
```

```
work(0) :- true |
    worker1( ... ,0,01),
    worker2( ... ,01,02),
    .
    .
    workerN( ... ,0n,[ ]).
```

outstream が受け入れるコマンドは print/1(引数の内容出力要求) と nl(改行要求) がある。なお、print/1 で引数の内容が具体化していない場合には同期の機構が働いて待ち合わせするようになっており、具体化されるまで実際の出力はされない。

【演習問題 5】 データ記録方法の練習

バイナリソートのプログラムを LIFO 法で解を出力するプログラムを書いて、FIFO 法の場合と比較せよ。

【演習問題 6】 ソートの練習

下記のような重複のある数列を入力として、重複なしの昇順のソートをして、数と出現回数を出力せよ。

```
[3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3,8,4,6,2,6,4,3,3,8,3,2,7,9,5,0,2,
8,8,4,1,9,7,1,6,9,3,9,9,3,7,5,1,0,5,8,2,0,9,7,4,9,4,4,5,9,2,3,0,7,
8,1,6,4,0,6,2,8,6,2,0,8,9,9,8,6,2,8,0,3,4,8,2,5,3,4,2,1,1,7,0,6,9]
```

なおデータは、下図のようにソース上でゴール data:sort(X) と指定し、data.kl1 を一緒にコンパイルすることで得られる。）

ソース上で、

```
:- module main.
main :- data:sort(X), my_sort(X).
my_sort(X) :- ... .
```

にて、次のようにコンパイル実行する。

```
% klic my_prog.kl1 data.kl1
```

なお、data.kl1 の中身は次のようになっている。

```
:- module data.
sort(X) :- true |
    X = [3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3,8,4,6,2,6,4,3,3,8,3,2,7,9,5,0,2,
        8,8,4,1,9,7,1,6,9,3,9,9,3,7,5,1,0,5,8,2,0,9,7,4,9,4,4,5,9,2,3,0,7,
        8,1,6,4,0,6,2,8,6,2,0,8,9,9,8,6,2,8,0,3,4,8,2,5,3,4,2,1,1,7,0,6,9].
```

【演習問題 7】 2 段ソートの練習

前問題と同じデータを入力として, 出現回数の大きいものの順に, 出現回数と数を入力せよ.

第 4 章

優先度と負荷分散

実際に KL1 プログラムを計算機上で効率的に動作させるために有効である、優先度制御の使い方について説明する。同時に計算を分散させるための負荷分散制御の使い方についても説明する。

4.1 優先度制御の利用目的

逐次型計算機では、処理順序がプログラムに記述した順序で決まる。一方 KL1 では、リダクションされたサブゴールの実行がプログラム中に記述された順番に従って実行されるとは限らない。順番に関係なく実行されることは、次のように言い換えることができる。

KL1 ではリダクションにより生成されたサブゴールを並列に実行しても構わない。

そのため KL1 言語を利用すると並列性を余り意識せずに自然に並列プログラミングできるわけである。

前章までで述べてきたように、KL1 には暗黙の同期機構があるので、論理的には実行順序の制御を記述する必要はない。しかし実際に KL1 プログラムを計算機上でより効率的に動作させるには、実行順序を制御する機構を利用した方が有効な場合がある。そのために KL1 言語の優先度制御を利用することになる。

優先度制御を利用する理由を挙げてみよう。

1. 領域計算量 (計算に必要な記憶領域) を減らす効率化を図る

例えば生産者 - 消費者問題のプログラムを作成した場合、生産者側と消費者側をどのような順序で実行すべきであろうか。生産者側を優先的に実行すれば、生産者が一時的に巨大なデータを生成し、メモリ不足を引き起こす恐れがある。消費者の優先度を上げることで、メモリ不足を防げる可能性がある¹。

2. 時間計算量 (ゴールリダクション数) を減らす効率化を図る

例えば木探索問題において、探索中の枝に評価値設定し、この評価値をそのまま優先度として利用すれば、枝の探索順序を自然に決定できる。その結果としてプログラムが最良優先探索プログラムとなり、計算量を減らせる可能性がある。

3. メタプログラミングの代用処理をする

非決定性プログラム、例えば OR 型探索プログラムのある枝で解が見つかった場合、他の枝に探索中止通知を他の処理より優先的に送る必要があるし、その通知を優先的に受け取る必要がある。またマルチプロセッサ上で負荷分散処理を行う場合、管理ゴールを優先的に処理する場合にも利用する (本処理の説明は今回割愛する)。

以下文法および基本的プログラミングについて説明する。

¹ 生産者を優先的に実行すれば、消費者がデータを消費するたびに起こるサスペンドの回数を減らすことができるのも事実である。実際には注意が必要である。

4.2 基本プログラミング

4.2.1 プログラムの実行とスケジューリング

簡単なプログラム実行

まず次のプログラムの実行に関して、リダクションを試みる順番を考える。

```
main :- true | q, r.  
q  :- true | s, t.  
r  :- true | true.  
s  :- true | true.  
t  :- true | true.
```

あるゴールがリダクションし、そこで生成されたサブゴールが左優先でリダクションが試みられると仮定する。ゴール呼び出しが入れ子になっている場合、ゴールリダクションによって生成されたサブゴールが LIFO でリダクションが試みられるとすると図 4.1 の順に実行されるであろう²。

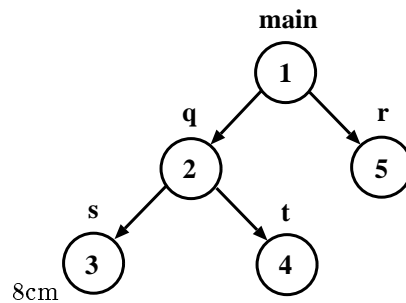


図 4.1: リダクション順の例

非決定性がないこのようなサスペンドしない実行が、最も効率良いものと言える³。しかしデータ処理が絡んでくるとそうとも言えない場合がある。生産者 - 消費者の問題を使って次項で説明しよう。

過剰生産とその抑制

非常に単純な次のプログラムを用意する。

```
% gencon.kl1  
:- module main.  
  
main:- true |  
    generator(100,Stream), consumer(Stream).  
  
generator(0,Stream) :- true | Stream = [].  
generator(N,Stream) :- N >= 0 |  
    Stream = [N | NextStream],  
    N1 := N - 1,  
    generator(N1,NextStream).  
  
consumer([]):- true | true.  
consumer([N|Stream]):- integer(N) | consumer(Stream).
```

²現在の逐次版 KLIC 処理系はこのスケジューリングを採用している

³この例は各述語の引数がないので効率的とも言える。

このプログラムは図 4.2 の通り整数列が次々に消費者に送られる。

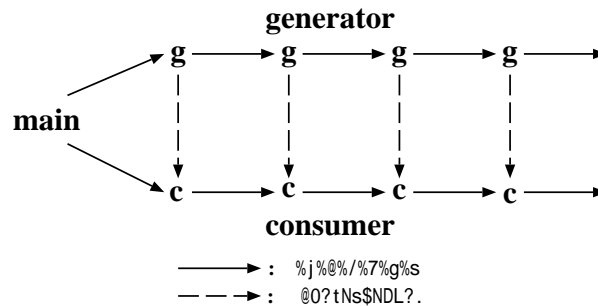


図 4.2: 生産者 - 消費者問題プログラムの実行

前項で取り上げた LIFO スケジューリングを仮定し、プログラムを実行する。

```
% klic -o gencon gencon.kl1
% gencon
%
```

無事動いたとしよう。プログラムを書き換えて生成データを増やす。

```
main :- true | generator(20000,Stream), consumer(Stream).
```

実行する。最大ヒープサイズの違いによって次のような実行結果が得られる可能性がある⁴。

```
% klic -o gencon gencon.kl1
% gencon -H30k
Fatal Error: Maximum heap size specified (30720 words) has been used up
% gencon -H40k
%
```

最大ヒープサイズ 30k ワードで実行するとメモリが足りなくなるのである。何故であろうか？

述語定義を眺めると述語 `generator` が自分自身をサブゴールとしてリダクションするように記述されている。LIFO スケジューリングに従うと仮定すると、述語 `generator` だけが次々に走り、長さ N のリストを生成して述語 `generator` の実行が終了する。次に述語 `consumer` の実行が開始することになる。長さ 100 程度 of リストを生成するのは問題なかったが、長さ 20,000 のリストの生成を試みると過剰生産でメモリが飽和してしまうことがあり得るのである。

さてどうするべきか。ひとつの解決方法として生産者プロセスと消費者プロセスの間に同期をとる、要求駆動的なプログラミングをすることが考えられる。

要求駆動型プログラミングによる解決

次の方針でプログラムを作成する。

1. 生産者は生産する必要がなくなったら、その旨を消費者に伝え終了する。さもなければ生産者はひとつ整数を消費者に送る。
2. 消費者は生産終了の旨を受け取ったら終了する。消費者は整数をひとつ受け取ったら、消費した旨を生産者に伝える。
3. 生産者は消費した旨を受け取ったら上述の手順でさらに生産処理を続ける。

⁴実行時オプション `-H30k` は、最大ヒープサイズを 30k word として実行することを意味する。

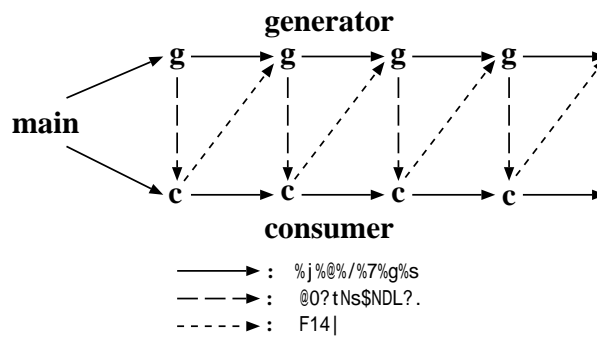


図 4.3: 同期付生産者 - 消費者問題プログラムの実行

プログラム例を以下に示す.

```
% request.kl1
:- module main.

main :- true | generator(next,20000,Stream), consumer(Stream).

generator(_,0,Stream)      :- true | Stream = [].
generator(Next,N,Stream) :- N =\= 0, Next=next |
    Stream = [send(N,NextN)|NextStream],
    N1 := N - 1,
    generator(NextN,N1,NextStream).

consumer([]) :- true | true.
consumer([send(N,Next)|Stream]) :- integer(N) |
    Next = next,
    consumer(Stream).
```

さすがにこのプログラムは

```
% klic -o request request.kl1
% request -H30k
%
```

と、メモリ不足にならずに動く.

このプログラムは図 4.3 の通り生産者に対して、消費者が整数を受け取った後に `next` を送る. 生産者は消費者から送られてきた `next` で同期をとりながら次の整数を消費者に送る (最初の 1 回を除く).

実行トレースを見てみよう.

```

1 CALL: main:main?
1 REDU: main:main :-
2   0:+generator(next,20000,_6)
3   1:+consumer(_6)?
2 CALL: main:generator(next,20000,_6)?
2 REDU: main:generator(next,20000,[send(20000,_E)|_F]) :-
4   0:+generator(_E,19999,_F)?
4 CALL: main:generator(_E,19999,_F)?
4 SUSP: main:generator(_E,19999,_F)?
3 CALL: main:consumer([send(20000,_1B)|_F])?
3 REDU: main:consumer([send(20000,next)|_F]) :-
4   0!+generator(next,19999,_F)
5   1:+consumer(_F)?
4 CALL: main:generator(next,19999,_F)?
4 REDU: main:generator(next,19999,_F) :-
6   0:+generator(next,19999,_F)?
6 CALL: main:generator(next,19999,_F)?
6 REDU: main:generator(next,19999,[send(19999,_2B)|_2C]) :-
7   0:+generator(_2B,19998,_2C)?
7 CALL: main:generator(_2B,19998,_2C)?
7 SUSP: main:generator(_2B,19998,_2C)?
5 CALL: main:consumer([send(19999,_38)|_2C])?
5 REDU: main:consumer([send(19999,next)|_2C]) :-
7   0!+generator(next,19998,_2C)
8   1:+consumer(_2C)?
7 CALL: main:generator(next,19998,_2C)?
7 REDU: main:generator(next,19998,_2C) :-
9   0:+generator(next,19998,_2C)?
9 CALL: main:generator(next,19998,_2C)?
9 REDU: main:generator(next,19998,[send(19998,_48)|_49]) :-
10  0:+generator(_48,19997,_49)?
10 CALL: main:generator(_48,19997,_49)?
10 SUSP: main:generator(_48,19997,_49)? a

```

残念ながらサスペンドするゴールが増えていることがわかる。要求駆動型にもそれなりの欠点があり、前項のようなプログラムと併せて時と場合に応じてプログラミングするのが望ましいと言えよう。

前項のプログラムを要求駆動型にせずに、KL1 の優先度制御記述を利用して簡単に実現する方法について次節で説明しよう。

4.3 KL1 のゴール優先度指定方法

4.3.1 再び生産者と消費者の問題

優先度指定付きのプログラムをまず見せよう。

```
% genconwp.kl1
:- module main.

main:- true |
    generator(20000,Stream)@lower_priority, consumer(Stream).

generator(0,Stream) :- true | Stream = [].
generator(N,Stream) :- N =\= 0 |
    Stream = [N | NextStream],
    N1 := N - 1,
    generator(N1,NextStream).

consumer([]):- true | true.
consumer([N|Stream]):- integer(N) | consumer(Stream).
```

先ほどの要求駆動型プログラムより簡単であり、48 ページのプログラムとほとんど変わらない。
次の通りメモリ不足に陥らず、計算できる。

```
% klic -o genconwp genconwp.kl1
% genconwp -H30k
%
```

実行トレースを見てみよう。

```
1 CALL: main:main?
1 REDU: main:main :-
2 0:+consumer(_6)
3 1*+generator(20000,_6)@prio(67108862)?
2 CALL: main:consumer(_6)?
2 SUSP: main:consumer(_6)?
3 CALL: main:generator(20000,_E)?
3 REDU: main:generator(20000,[20000|_14]) :-
4 0:+generator(19999,_14)
2 1#+consumer([20000|_14])@prio(67108863)?
2 CALL: main:consumer([20000|_14])?
2 REDU: main:consumer([20000|_14]) :-
5 0:+consumer([20000|_14])?
5 CALL: main:consumer([20000|_14])?
5 REDU: main:consumer([20000|_14]) :-
6 0:+consumer(_14)?
6 CALL: main:consumer(_14)?
6 SUSP: main:consumer(_14)?
4 CALL: main:generator(19999,_23)?
4 REDU: main:generator(19999,[19999|_29]) :-
7 0:+generator(19998,_29)
6 1#+consumer([19999|_29])@prio(67108863)?
6 CALL: main:consumer([19999|_29])? a
```

実行順序が制御されていることが分かる。

ソースプログラム中の見慣れない記号を説明しよう。今までの記述と異なるのは次の箇所である。

```
generator(20000,Stream)@lower_priority
```


ゴールの後ろについた@lower_priority は、ゴール優先度指定プラグマと呼ばれる。この記述は‘現在よりひとつ低い優先度でこのゴールリダクションを試みよ’を意味する。この優先度に関しては、次の規則に従って KL1 プログラムが実行されている。

- ゴール優先度指定プラグマが付いてない限り、サブゴールは親ゴールと同じ優先度となる。

この規則に従うと上述のプログラムは、次の意味付けされていると考えることができる。

- 消費者は生産者より優先的に実行される。

そんな訳でメモリの大量消費を防ぐことができたのである。さて文法について次項で説明しよう。

4.3.2 ゴール優先度指定の文法

KLIC の優先度は KLIC の整数の表現範囲分だけ存在する。大きいほど優先度が高くなる。優先度指定の方法は以下の 2 通りある。

1. 絶対指定

次の通りである。

```
Goal@priority(優先度)
```

指定されたゴールを指定された優先度でリダクションを試みる。

2. 相対指定

次の通りである。

```
Goal@lower_priority
```

指定されたゴールの親ゴールより 1 つ低い優先度でリダクションを試みる。

4.3.3 優先度指定使用上の注意

優先度指定を使うにあたり、次のことに注意すべきである。

優先度は下げる方向で使うべきである

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 |
    go@priority(100000)
go:- true |
    a,b.
a :- true | goal_0@priority(200000).
b :- true | goal_1@priority(300000).
```

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 200000 と 300000 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 200000 で実行する。
- ゴール goal_1 は優先度 300000 で実行する。
- すなわち、ゴール goal_1 は goal_0 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 100000 で実行される。
- ゴール a と b も優先度 100000 で実行される。
- ゴール a が先に実行されて、goal_0 が優先度 200000 で実行される。
- この時点で、ゴール b の実行は優先度 100000 で実行されているので、優先度 200000 の goal_0 の処理が終わるまで、goal_1 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。さらにもうひとつ重要な注意点がある。次のことである。

ゴール優先度は、同一プロセッサ内でのみ有効である。

マルチプロセッサ環境下で将来 KLIC を動作させる時の注意である。例えば生産者と消費者を別々のプロセッサで動作させた場合、前述のゴール優先度を利用したプログラムでは過剰生産を防ぐことはできないのである。

4.4 もうひとつの優先度

ゴールの優先度について述べてきたが、ここではもうひとつの節優先度指定プラグマ `alternatively` について説明する。

再度生産者 - 消費者問題のプログラムを考えてみよう。今度は以下プログラムのように生産者が処理を中止するかどうかを判断するために変数をひとつもち、この変数に何か値が設定された時に処理を中止するように定義する。

```
generator(stop,Stream) :- true | Stream = [].
generator(Stop,Stream) :- true |
    Stream = [1 | NextStream],
    generator(Stop,NextStream).
```

説明を簡単にするため整数列を 1 の列で表現した。

このプログラムにはバグがある。述語 `generator` は 2 つの節をもち、次の意味で実行したい。

- 第 1 引数に `stop` が到着済みならば、整数列の生成を止める。
- 第 1 引数にまだ `stop` が到着済みではないみたいならば、整数を 1 つ生成する。

しかしこのプログラムをコンパイルすると

```
Redundant clause
Clause deleted: generator/2
```

といった警告メッセージが出力される。つまりこのままではコンパイラが最適なコードを生成するにあたり、述語 `generator` 定義中に冗長節があると指摘してしまう。このような場合、リダクションを試行する節に優先順序を記述することで `generator` 定義にある両節へのリダクションチャンスを与えることができる。それが `alternatively` である。alternatively を使ったプログラムを示す。

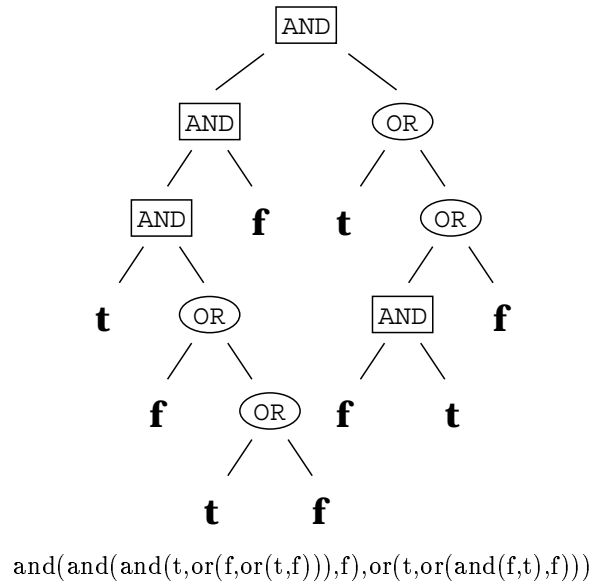


図 4.4: 二進木の例

```
generator(stop,Stream) :- true | Stream = [].
alternatively.
generator(Stop,Stream) :- true |
    Stream = [1 | NextStream],
    generator(Stop,NextStream).
```

述語 `generator` の第 1 節のリダクション、つまり第 1 引数へのメッセージ到着済みであることが確認できれば第 1 節を選択し、確認できなければ第 2 節を実行することになる。

注意点

次の点に注意すべきである。

第 2 節が選択されたからといって、第 1 引数にメッセージが到着してないとは限らない。

優先度はプログラムの意味を変えないものである。第 2 節は第 1 引数に変数でないことを示していないのである。KL1 は変数が具体化されたことを確認できるが、その逆はできない。

4.5 応用: 幅優先評価の実現

ここでは論理式の評価を例にとり、優先度を用いた幅優先評価の実現方法を述べる。一般に、論理式は、AND, OR, `t(true)`, `f(false)` をノードとする二進木で表現することができる。例を図 4.4 に示す。二進木で表わした場合、論理式の評価は二進木の評価になる。

論理式を表わした二進木の評価について、以下では、まず、

- すべてのノードを評価する方法
- 不要なノード評価を始めない方法
- 不要なノード評価を打ち切る方法

を述べ、そして優先度を用いて

- 幅優先に評価を行なう方法

を述べる。

```

% naive.kl1
:- module main. % 1

main :- true | % 2
    eval(T,E), io:outstream([print(E),nl]), % 3
    T = and(and(and(t,or(f,or(t,f))),f),or(t,or(and(f,t),f))). % 4

eval(and(L,R),E) :- true | eval(L,EL), eval(R,ER), and_node(EL,ER,E). % 5
eval(or(L,R),E) :- true | eval(L,EL), eval(R,ER), or_node(EL,ER,E). % 6
eval(t,E) :- true | E = t. % 7
eval(f,E) :- true | E = f. % 8

and_node(t,t,E) :- true | E = t. % 9
and_node(f,_,E) :- true | E = f. % 10
and_node(_,f,E) :- true | E = f. % 11

or_node(t,_,E) :- true | E = t. % 12
or_node(_,t,E) :- true | E = t. % 13
or_node(f,f,E) :- true | E = f. % 14

```

図 4.5: すべてのノードを評価するプログラム (naive.kl1)

4.5.1 すべてのノードを評価する方法

プログラム (naive.kl1) を図 4.5 に示す. `eval/2` は, 下につながるノードがあれば `eval/2` をフォークし, フォークによって求まった 2 つの評価値から自ノードの評価値を決定する (5,6 行目). 下につながるノードのない場合 (自分がリーフの場合) は評価値をすぐに返す (7,8 行目). `and_node/3` 及び `or_node/3` は 2 つの値の AND あるいは OR 演算を行なう.

`eval/2` のリダクションの様子を図 4.6 に示す. このプログラムはすべてのノードを評価してしまうので図にはすべてのノード (17 ノード) が登場した.

4.5.2 不要なノード評価を始めない方法

AND ノードでは, 下につながる 2 つのノードのうち一方のノードの評価結果が `f` ならばもう一方のノードは評価する必要がない. また OR ノードでは, 一方が `t` ならばもう一方のノードは評価する必要がない. このことを利用して不要なノード評価を行なわくしたのが図 4.7 のプログラム (seq.kl1) である.

`eval/2` は, まず `eval/2` を呼んで左のノードの評価を行ない, `and_node/3` あるいは `or_node/3` を呼ぶ (5,6 行目). `and_node/3` 及び `or_node/3` は左のノードの評価結果によって, `eval/2` を呼んで右のノードを評価する (10,11 行目), あるいは右のノードを評価を始めることなく評価値を返す (9,12 行目).

seq.kl1 の実行の様子を図 4.8 に示す. 不要なノード評価を始めないことにより 17 ノード中 9 ノードについてのみ `eval/2` が実行された.

このプログラムでは naive.kl1 に比べて評価対象のノードが少なくすむが (計算量が少なくすむが) 逐次にしか動かない. `eval/2` は 2 つのゴール (`eval/2` と `and_node/3` あるいは `or_node/3`) をフォークするが (5,6 行目), フォークされた 2 つのゴールは並列には動かない. `and_node/3` 及び `or_node/3` は `eval/2` の結果が求まるまで動けないからである.

4.5.3 不要なノード評価を打ち切る方法

seq.kl1 では不要なノード評価を行なわないことを

```

2 REDU:main:eval(and(and(and(t,or(...)),f),or(t,or(and(...),f))),_3) :-      % 1
4 REDU:main:eval(and(and(t,or(f,or(...)),f),_F) :-                          % 2
7 REDU:main:eval(and(t,or(f,or(t,f))),_1C) :-                               % 3
10 REDU:main:eval(t,t)                                                         % 4
11 REDU:main:eval(or(f,or(t,f)),_2A) :-                                         % 5
13 REDU:main:eval(f,f)                                                         % 6
14 REDU:main:eval(or(t,f),_37) :-                                              % 7
16 REDU:main:eval(t,t)                                                         % 8
17 REDU:main:eval(f,f)                                                         % 9
8 REDU:main:eval(f,f)                                                         % 10
5 REDU:main:eval(or(t,or(and(f,t),f)),_10) :-                                  % 11
19 REDU:main:eval(t,t)                                                         % 12
20 REDU:main:eval(or(and(f,t),f),_51) :-                                       % 13
22 REDU:main:eval(and(f,t),_5D) :-                                           % 14
25 REDU:main:eval(f,f)                                                         % 15
26 REDU:main:eval(t,t)                                                         % 16
23 REDU:main:eval(f,f)                                                         % 17

```

図 4.6: naive.kl1 の実行トレース (eval/2 のリダクション)

```

% seq.kl1
:- module main.                                                                % 1

main :- true |                                                                  % 2
    eval(T,E), io:outstream([print(E),nl]), % 3
    T = and(and(and(t,or(f,or(t,f))),f),or(t,or(and(f,t),f))). % 4

eval(and(L,R),E) :- true | eval(L,EL), and_node(EL,R,E). % 5
eval(or(L,R),E) :- true | eval(L,EL), or_node(EL,R,E). % 6
eval(t,E) :- true | E = t. % 7
eval(f,E) :- true | E = f. % 8

and_node(f,_,E) :- true | E = f. % 9
and_node(t,R,E) :- true | eval(R,E). % 10

or_node(f,R,E) :- true | eval(R,E). % 11
or_node(t,_,E) :- true | E = t. % 12

```

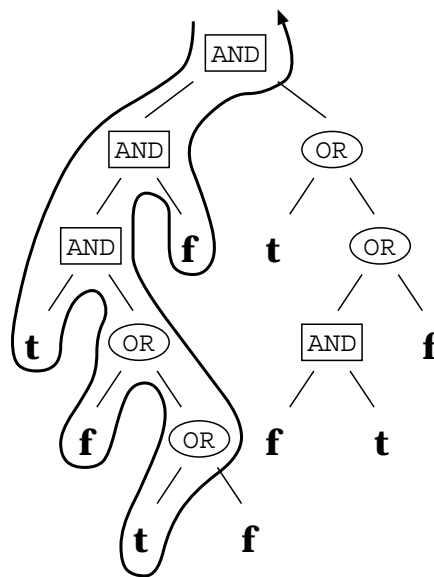
図 4.7: 不要なノード評価を始めないプログラム (seq.kl1)

```

2 REDU:main:eval(and(and(and(t,or(...)),f),or(t,or(and(...),f))),_3) :-      % 1
4 REDU:main:eval(and(and(t,or(f,or(...)),f),_F) :-                          % 2
6 REDU:main:eval(and(t,or(f,or(t,f))),_18) :-                               % 3
8 REDU:main:eval(t,t)? 1                                                    % 4
10 REDU:main:eval(or(f,or(t,f)),_18) :-                                     % 5
11 REDU:main:eval(f,f)? 1                                                  % 6
13 REDU:main:eval(or(t,f),_18) :-                                           % 7
14 REDU:main:eval(t,t)? 1                                                  % 8
16 REDU:main:eval(f,f)? 1                                                  % 9

```

(a) eval/2 のリダクション



(b) ノードの評価

図 4.8: seq.kl1 実行

まず一方のノードを評価し、その結果によってはもう一方の評価を始めない。

という逐次的な方式で実現したが、ここでは

両方のノードとも評価を始め、たまたま先に求まった一方の評価結果によっては、もう一方の評価を途中で打ち切る。

という並列実行が可能な方式で実現する。この方式では各ノードが並列に評価されうるわけである。プログラム (cont.kl1) を図 4.9 に示す。

eval/3 は、下のノードを評価する 2 つの eval/3 と、それらを制御する and_node/6 あるいは or_node/6 をフォークする (7~12 行目)。フォークされた 2 つの eval/3 は並列に実行されうる。

eval/3, and_node/6, or_node/6 の第 1 引数はノードの評価を途中で打ち切るための引数である。この引数を以下では『打ち切り引数』と呼ぶことにする。3 つの述語とも打ち切り引数をチェックする節 (5 及び 15, 20 行目) の優先度が alternatively によって高くなっている。このため打ち切り引数が abort に具体化されていたならば、他の引数の具体化状況に依らず打ち切り引数をチェックする節が選択されノードの評価は行なわれない。

```

% cont.kl1
:- module main.                                     % 1

main :- true |                                     % 2
    eval(_,T,E), io:outstream([print(E),nl]),      % 3
    T = and(and(and(t,or(f,or(t,f))),f),or(t,or(and(f,t),f))). % 4

eval(abort,_,_) :- true | true.                    % 5
alternatively.                                     % 6
eval(AB,and(L,R),E) :- true |                      % 7
    and_node(AB,EL,ER,ABL,ABR,E),                 % 8
    eval(ABL,L,EL), eval(ABR,R,ER).               % 9
eval(AB,or(L,R),E) :- true |                      % 10
    or_node(AB,EL,ER,ABL,ABR,E),                 % 11
    eval(ABL,L,EL), eval(ABR,R,ER).              % 12
eval(_,t,E) :- true | E = t.                      % 13
eval(_,f,E) :- true | E = f.                      % 14

and_node(abort,_,_,ABL,ABR,_) :- true | ABL = abort, ABR = abort. % 15
alternatively.                                     % 16
and_node(_,t,t,_,_,E) :- true | E = t.            % 17
and_node(_,f,_,_,ABR,E) :- true | E = f, ABR = abort. % 18
and_node(_,_,f,ABL,_,E) :- true | E = f, ABL = abort. % 19

or_node(abort,_,_,ABL,ABR,_) :- true | ABL = abort, ABR = abort. % 20
alternatively.                                     % 21
or_node(_,f,f,_,_,E) :- true | E = f.            % 22
or_node(_,t,_,_,ABR,E) :- true | E = t, ABR = abort. % 23
or_node(_,_,t,ABL,_,E) :- true | E = t, ABL = abort. % 24

```

図 4.9: 不要なノード評価を打ち切るプログラム (cont.kl1)

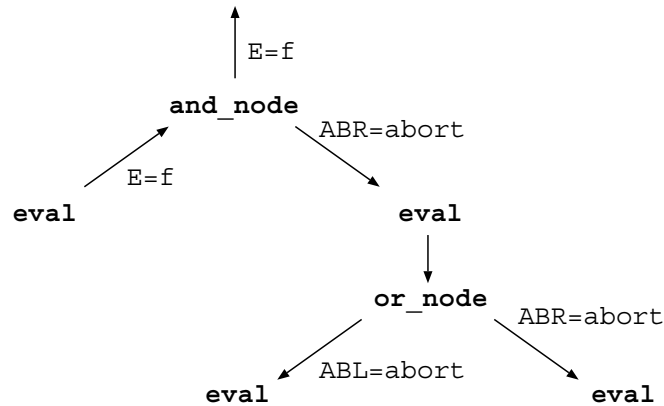


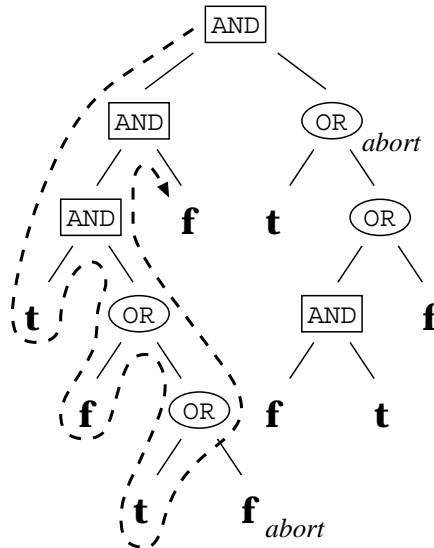
図 4.10: abort の伝播

```

2 REDU:main:eval(_9,and(and(and(t,or(...)),f),or(t,or(and(...),f))),_3) :- % 1
5 REDU:main:eval(_16,and(and(t,or(f,or(...)),f),_63E5) :- % 2
8 REDU:main:eval(_30,and(t,or(f,or(t,f))),_63D4) :- % 3
11 REDU:main:eval(_4A,t,t) :- % 4
12 REDU:main:eval(_45,or(f,or(t,f)),_63C0) :- % 5
15 REDU:main:eval(_6C,f,f) :- % 6
16 REDU:main:eval(_67,or(t,f),_63AD) :- % 7
19 REDU:main:eval(_8E,t,t) :- % 8
20 REDU:main:eval(abort,f,_639C) % 9
9 REDU:main:eval(_2B,f,f) :- % 10
6 REDU:main:eval(abort,or(t,or(and(f,t),f)),_63E4) % 11

```

(a) eval/3 のリダクション



(b) ノードの評価

図 4.11: cont.kl1 の実行

and_node/6 は、一方のノードの評価結果が *f* であることがわかった段階で評価値 *E* を *f* に決定し、打ち切り引数を *abort* に具体化してもう一方のノードの評価を打ち切る (18,19 行目). or_node/6 は、一方の評価結果が *t* であることがわかった段階で同様の処理を行なう (23,24 行目). 打ち切り引数の具体化は、and_node あるいは or_node から eval へ、そして and_node あるいは or_node を経由してさらに下の eval へと伝播して行き (図 4.10 参照)、ノードの評価が打ち切られる。

cont.kl1 の実行の様子を図 4.11 に示す。2 つのノードが評価を打ち切られ (図 (a) 9,11 行目, 図 (b) *abort*)、17 ノード中 9 ノードについてのみ eval/3 の通常実行が行なわれた。図 4.11 を図 4.8 と比較すると seq.kl1 と同じノードが同じ順序で評価されたことがわかる。しかしこれは「たまたまそのように評価された」だけである。seq.kl1 は評価順序を「まず左のノードを評価し、次に右へ行く」と決めていたが、cont.kl1 は「どちらもいつでも評価を試みてよい」プログラムであり、評価順序は定めていない。

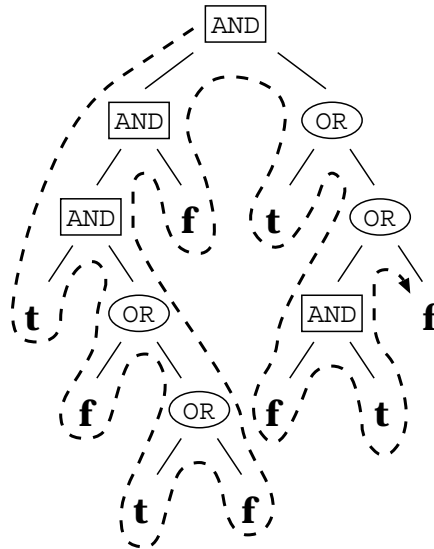
前述したようにこのプログラムは並列実行が可能である。即ち、ある時刻において実行可能なゴールが一般に複数存在する。一方このプログラムではゴールに対して優先度を指定していないのですべてのゴールは同じ優先度をもち実行可能なゴールはどれでも実行されうる。これは言い換えると「あるゴールが実行可能であっても、一般に、同じ優先度をもつ実行可能なゴールが他にも存在するので、そのゴール


```

2 REDU:main:eval(_9,and(and(and(t,or(...)),f),or(t,or(and(...),f))),_3) :- % 1
4 REDU:main:eval(_14,and(and(t,or(f,or(...))),f),_12) :- % 2
7 REDU:main:eval(_26,and(t,or(f,or(t,f))),_24) :- % 3
10 REDU:main:eval(_38,t,t) % 4
11 REDU:main:eval(_39,or(f,or(t,f)),_37) :- % 5
13 REDU:main:eval(_4A,f,f) % 6
14 REDU:main:eval(_4B,or(t,f),_49) :- % 7
16 REDU:main:eval(_5C,t,t) % 8
17 REDU:main:eval(_5D,f,f) % 9
8 REDU:main:eval(_27,f,f) % 10
5 REDU:main:eval(_15,or(t,or(and(f,t),f)),_13) :- % 11
19 REDU:main:eval(_6E,t,t) % 12
20 REDU:main:eval(_6F,or(and(f,t),f),_6D) :- % 13
22 REDU:main:eval(_80,and(f,t),_7E) :- % 14
25 REDU:main:eval(_92,f,f) % 15
26 REDU:main:eval(_93,t,t) % 16
23 REDU:main:eval(_81,f,f) % 17

```

(a) eval/3 のリダクション



(b) ノードの評価

図 4.12: ゴールの順序を入れ換えた cont.kl1 の実行

ルが実行される保証はない」ということである。

評価の打ち切りを効率よく行なうには、評価する必要のないノードに対応する eval/3 の実行が始まる前に and_node/6 及び or_node/6 の実行が開始されなければならない。実行の終わった eval/3 の打ち切り引数を具体化しても意味がないからである。しかし、上述したように、そのように実行される保証はない。最悪の場合はすべてのノードについて eval/3 の実行が終了してから and_node/6 及び or_node/6 が実行されノード評価の打ち切りは全く行なわれない。

図 4.9 の cont.kl1 では、ボディの左側のゴールが右側より先に実行が試みられたので、and_node/6 及び or_node/6 が eval/3 よりも先に実行されノード評価の打ち切りが行なわれた。これは「たまたま期待通りに実行された」と考えるべきである。ひとつのボディ内の優先度を指定していないゴール群はすべて同じ優先度を持つので、何らかの都合により eval/3 が and_node/6 あるいは or_node/6 より先に実行されても文句は言えない。

cont.kl1 の 8 行目と 9 行目、及び 11 行目と 12 行目を入れ換えると and_node/6 及び or_node/6 の実行は eval/3 の後になる。この時の eval/3 の実行の様子を図 4.12 に示す。図からわかるようにノード評価の打ち切りは全く行なわれない。17 ノードすべてについて eval/3 が実行されてしまった。

4.5.4 優先度を用いて幅優先に評価を行なう方法

前述した cont.kl1 はゴールの優先度制御を行なわないためノード評価打ち切りの機能の働く保証がなかった。ここでは優先度を用いて幅優先評価を行ないノード評価の打ち切りを行なう方法を示す。

プログラム (prio.kl1) を図 4.13 に示す。このプログラムは 9,12 行目の eval/3 に優先度のプラグマが付いていることを除いて図 4.9 の cont.kl1 と全く同じである。

eval/3 は、2 つの eval/3 と and_node/6 あるいは or_node/6 をフォークするが (7~12 行目)、2 つの eval/3 の優先度は and_node/6 及び or_node/6 よりも低い。この結果 eval/3 よりも and_node/6 及び or_node/6 が優先的に実行され不要なノード評価の打ち切られることが期待される。またルートに近いノードに関するゴールほど高い優先度で実行されるので全体としては幅優先評価が行なわれる。

図 4.14 にゴールの優先度を、図 4.15 に eval/3 のリダクションの様子を示す。3 つのノードが評価を打ち切られ (5,6,7 行目)、17 ノード中 4 ノードについてのみ eval/3 の通常実行が行なわれた。

4.6 応用: 簡単な探索問題

最後に次節に述べる負荷分散制御への理解を深めるために、非常に良く似た探索問題を優先度制御を利用してプログラム化してみよう。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 4.16 に示されるような 2 分探索木があった時、アトムを含むリーフを 1 つ見つけるプログラムを書け。

図 4.16 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 の名前が付いてあり、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、ヒューリスティッ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、アトム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

```

% prio.kl1
:- module main.                                     % 1

main :- true |                                     % 2
    eval(_,T,E), io:outstream([print(E),nl]),      % 3
    T = and(and(and(t,or(f,or(t,f))),f),or(t,or(and(f,t),f))). % 4

eval(abort,_,_) :- true | true.                    % 5
alternatively.                                     % 6
eval(AB,and(L,R),E) :- true |                      % 7
    and_node(AB,EL,ER,ABL,ABR,E),                 % 8
    eval(ABL,L,EL)@lower_priority, eval(ABR,R,ER)@lower_priority. % 9
eval(AB,or(L,R),E) :- true |                      % 10
    or_node(AB,EL,ER,ABL,ABR,E),                  % 11
    eval(ABL,L,EL)@lower_priority, eval(ABR,R,ER)@lower_priority. % 12
eval(_,t,E) :- true | E = t.                       % 13
eval(_,f,E) :- true | E = f.                       % 14

and_node(abort,_,_,ABL,ABR,_) :- true | ABL = abort, ABR = abort. % 15
alternatively.                                     % 16
and_node(_,t,t,_,_,E) :- true | E = t.             % 17
and_node(_,f,_,_,ABR,E) :- true | E = f, ABR = abort. % 18
and_node(_,_,f,ABL,_,E) :- true | E = f, ABL = abort. % 19

or_node(abort,_,_,ABL,ABR,_) :- true | ABL = abort, ABR = abort. % 20
alternatively.                                     % 21
or_node(_,f,f,_,_,E) :- true | E = f.              % 22
or_node(_,t,_,_,ABR,E) :- true | E = t, ABR = abort. % 23
or_node(_,_,t,ABL,_,E) :- true | E = t, ABL = abort. % 24

```

図 4.13: 優先度を用いて幅優先評価を行なうプログラム (prio.kl1)

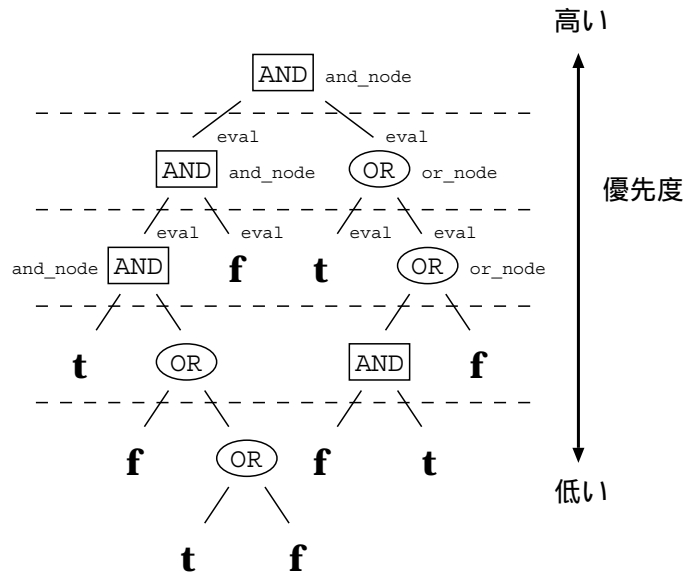


図 4.14: prio.kl1 における優先度

```

2 REDU:main:eval(_9,and(and(and(t,or(...)),f),or(t,or(and(...),f))),_3) :- % 1
6 REDU:main:eval(_11,or(t,or(and(f,t),f)),_63E4) :- % 2
5 REDU:main:eval(_16,and(and(t,or(f,or(...)),f),_63E5) :- % 3
12 REDU:main:eval(_48,f,f) :- % 4
11 REDU:main:eval(abort,and(t,or(f,or(t,f))),_63C2) % 5
9 REDU:main:eval(abort,or(and(f,t),f),_63D2) % 6
8 REDU:main:eval(abort,t,_63D3) % 7

```

図 4.15: prio.kl1 の実行トレース (eval/3 のリダクション)

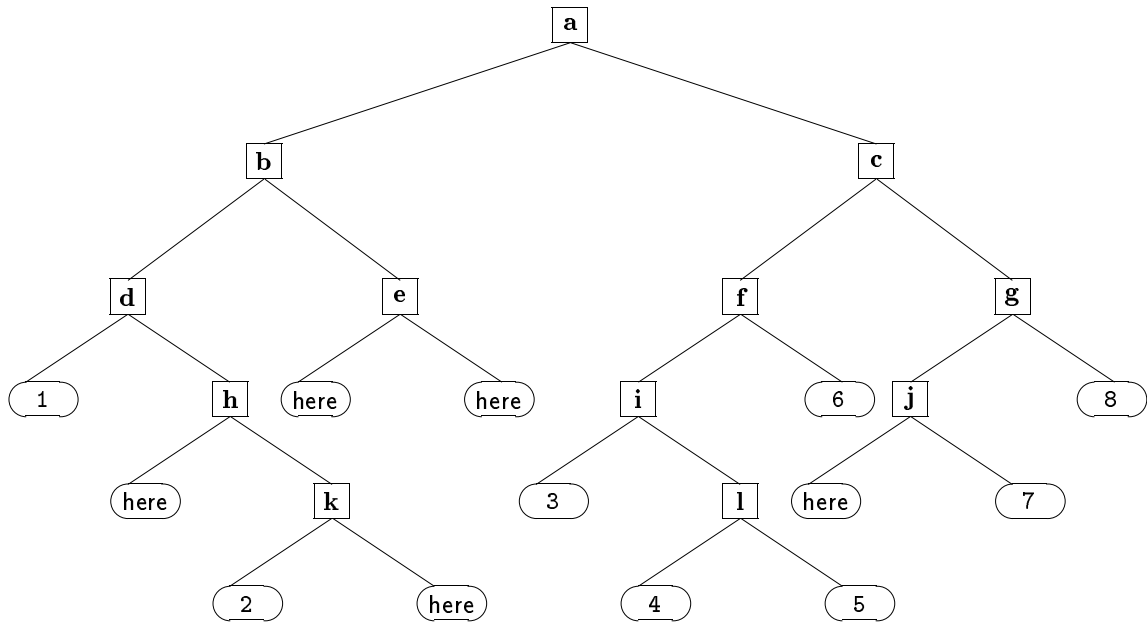


図 4.16: 探索木

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.

    search(Tree,Output):- true |                               %(1)
        control(Result,Output,Cont),                          %(2)
        fork(Tree,Result,Cont).                                %(3)
    fork(_,R,stop):- true | R=[].                               %(4)
alternatively.                                                 %(5)
    fork(H,R,Cont):- integer(H) | R=[].                         %(6)
    fork(H,R,Cont):- atom(H) |                                  %(7)
        R=[H].                                                  %(8)
    fork([TL,TR],R,Cont):-                                       %(9)
        merge:in(R0,R1,R),                                       %(10)
        fork(TL,R0,Cont),                                       %(11)
        fork(TR,R1,Cont).                                       %(12)
    control([H|_],Output,Cont):- wait(H) |                      %(13)
        Output = [H], Cont=stop.                                %(14)
    control([], Output, Cont):- true |                           %(15)
        Output = [] .                                           %(16)

:- module merge.

```

```

in([E|In1],In2,Out) :-                                %(17)
    Out = [E|OutT], in(In1,In2,OutT).                %(18)
in(In1,[E|In2],Out) :-                                %(19)
    Out = [E|OutT], in(In1,In2,OutT).                %(20)
in([],In2,Out) :- Out = In2.                          %(21)
in(In1,[],Out) :- Out = In1.                          %(22)

```

メインプログラム例

```
:- module main.
```

```

main :-                                                %(23)
    Tree=[[[1,[here,[2,here]]],[here,here]],
           [[3,[4,5]],6],[[here,7],8]]],
    simple_search:search(Tree,Out),                    %(24)
    klicio:klicio([stdout(Result)]),                  %(25)
    wait_open(Result, Out).                            %(26)
wait_open(normal(STDOUT), Out) :-                      %(27)
    STDOUT=[ putt(Out), nl ].                          %(28)
wait_open(abnormal, _) :- unix:exit(1).               %(29)

```

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。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 (4) 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、(6) 以降のクローズの選択を行なう。

fork クローズ (6) 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` がアトムなら、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そのアトム自身を流す。

fork クローズ (9) 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 (11) と (12) で生成している。なお、(10) 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 (13) 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。

in クローズ (17), (19), (21),(22) で標準的な 2 入力のマージプロセスを実現している。

main クローズ (23) にて計算を行なっている。`[here]` が印刷される。

ここで、`fork` プロセスを生成する部分 (11) と (12) をみてみよう。(11) はノードの左を探索するプロセスを生成しており、(12) 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、幅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、探索木の探索するゴール (11) および (12) を呼び出す枝より優先度を低くする。すると別の枝の同じ深さのゴールがあった場合は、そちらが優先的に実行される。

例えば探索ルート of 優先度を 4000 とし、このアルゴリズムに従って優先度をみると、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 4.17 のようになる。图中、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よう。

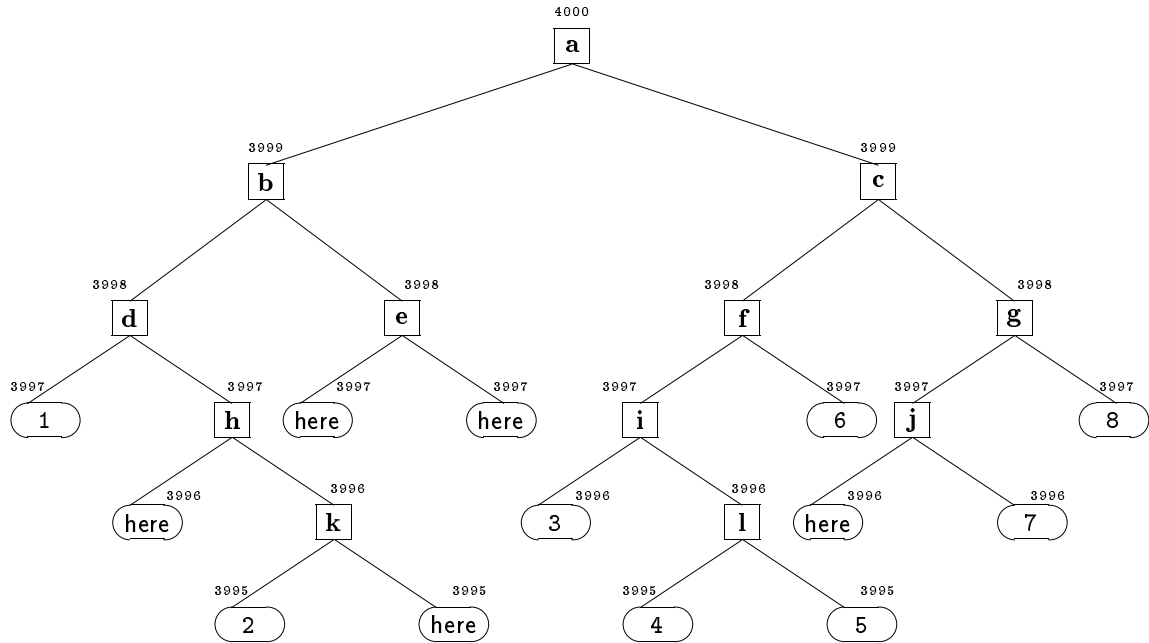


図 4.17: 探索木の各ノードプロセスの優先度

```
:- module search_pri.

search(Tree,Output):- true |                               %(1)
    control(Results,Output,Cont),                          %(2)
    fork(Tree,Results,Cont)@priority(4000).                %(3)
fork(_,R,stop):- true | R=[].                               %(4)
alternatively.                                             %(5)
fork(H,R,_):- integer(H) | R=[].                           %(6)
fork(H,R,Cont):- atom(H) |                                 %(7)
    R=[H].                                                  %(8)
fork([TL,TR],R,Cont):-true |                               %(9)
    merge:in(R0,R1,R),                                     %(10)
    fork(TL,R0,Cont)@lower_priority,                       %(11)
    fork(TR,R1,Cont)@lower_priority.                       %(12)
control([H|_],Output,Cont):- wait(H) |                    %(13)
    Output=[H], Cont=stop.                                  %(14)
control([],Output,Cont):- true | Output=[].               %(15)
```

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う最高優先度を指定した。

fork (11) と (12) の部分で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低くできなくなるが、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。

4.7 負荷分散制御の利用目的

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行うしか方法がない。しかし、並列計算機では逐次に行う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ことができる。

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、KLIC 逐次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、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 KLIC 並列処理系上で動かす必要がある。

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 4.18)。指定の方法についての詳細は次節で説明する。

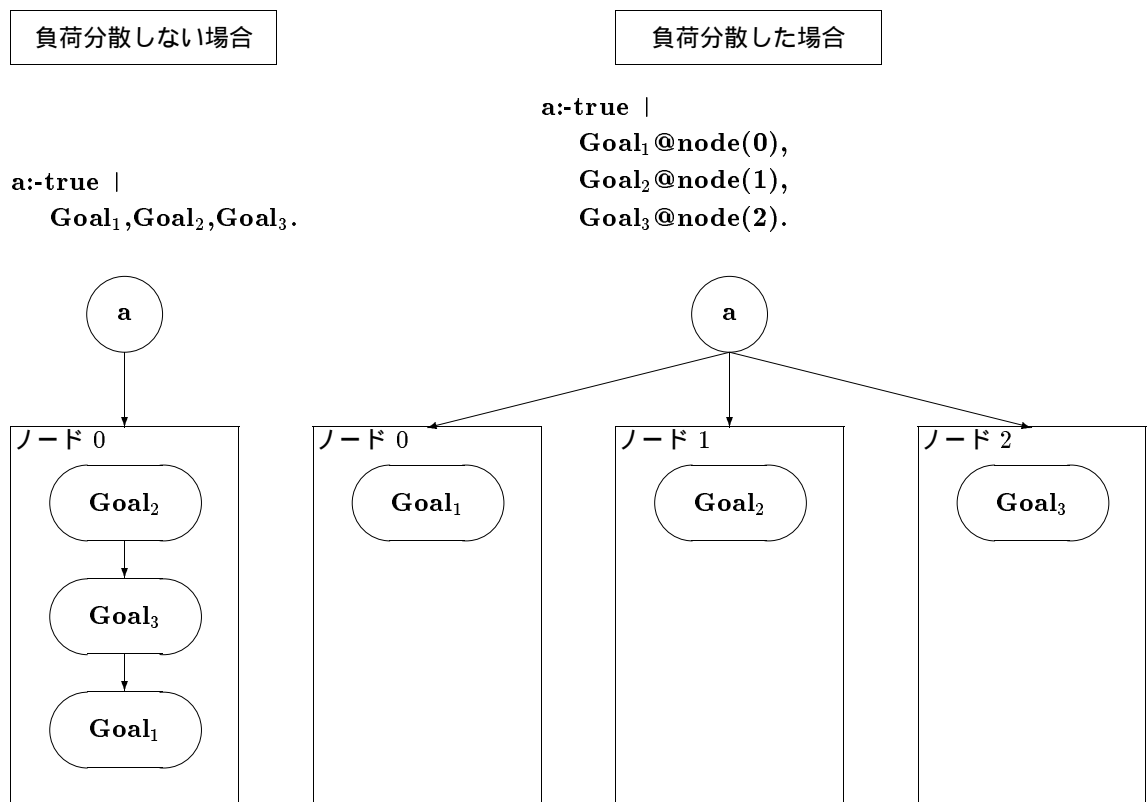


図 4.18: KL1 における負荷分散

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ノード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ノード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、通信時間のオーバー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いる。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ー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よい。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ノード上で実行させる。 プロセスとデータが所在するノードが異なると、その間の通信のオーバーヘッドが増大する。 プロセスとデータは同じノード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ノードにする方が効率が良い場合には、通信の多いものを同じノード上で処理するようにする事。
- (3) アクセスの多いデータベースは、ノード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ノードに通信が集中する。 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ーヘッドが増大する。 また、その結果もコンパクトにまとめている必要がある。

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。

4.8 ゴール分散の指定方法

ゴールの負荷分散の指定は、ボディの各ゴール毎にどのノードで実行するかを次のようにして指定する。

`goal@node(ノード番号)`

なお、ノード番号は 0 以上の整数であり、ノード数は実行する環境により異なる。そのノード数を得るには次のような組込み述語を用いる。

`current_node(PE,N)`

なお、ここで `PE` はこの組込み述語が実行されたノードの番号で、`N` はノード数のことである。

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` をノード 0 と 1 と 2 に投げたもので、ノード番号は整数で指定する。

```
:- module foo,

    foo:- true |                                %(1)
        a@node(0),                             %(2)
        a@node(1),                             %(3)
        a@node(2).                             %(4)
```

また、変数を用いてノード番号を指定することもできる。

```
:- module foo.
```

```
    foo(P0,P1,P2):- true |           %(1)
                      a@node(P0),      %(2)
                      a@node(P1),      %(3)
                      a@node(P2).      %(4)
```

なお、ゴールを別ノードに投げた後には、そのゴールは元と同じ優先度で実行される。従って、例えばノード0で優先度1000で動いていたゴールはノード1でも優先度1000で実行される。その時ノード0では優先度1000以上のゴールがなかった場合には、このゴールはノード1では直ちに実行される。しかし、投げた先のノード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、ノード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。

サイクリック割り付け

ノード数4の構成でゴール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、ノード数も組み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.
    distribute(N):- true |           %(1)
                      current_node(_,PEs),      %(2)
                      fork(N,PEs,0).            %(3)
    fork(0,PEs,PE):- true | true.      %(4)
    otherwise.
    fork(N,PEs,PE):- PeNo:=PE mod PEs |      %(5)
                      spawn@node(PeNo),        %(6)
                      NextPE:=PE+1,            %(7)
                      N1:=N-1,                 %(8)
                      fork(N1,PEs,NextPE).      %(9)
```

“foo:distribute(8)”を実行すると、spawnプロセスはノードの0, 1, 2, 3, 0, 1, 2, 3に割り付けられる。

ランダム割り付け

ノード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ノード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。割り付けるノード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ノードに割り付けた例である。

```
:- module foo.
```

```
    distribute(N):- true |           %(1)
                      current_node(_,PEs),      %(2)
                      random(Rs,23,PEs),        %(3)
```

```

fork(N,Rs).                                %(4)
fork(0,Rs):- true | Rs=[].                 %(5)
otherwise.
fork(N,Rs):- true |                         %(6)
    Rs=[get(PeNo)|NewRs],                   %(7)
    spawn@node(PeNo),                       %(8)
    N1:=N-1,                               %(9)
    fork(N1,NewRs).                         %(10)
random([get(Rnd)|RS],Seed,R):- true |      %(11)
    Rnd:= Seed mod R,                       %(12)
    NewSeed:=(125*Seed+1) mod 4096,         %(13)
    random(RS,NewSeed,R).                   %(14)
random([],_,_):- true | true.              %(15)

```

(16) のようにゴールを呼び出すと、100 ノードの環境であれば、(7) 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、(8) の部分でノードの割り付けを行なう。この例の場合では、ノードは 23, 76, 49, 10, 99, 32, ... と割り付けられる。

暇なノードへの動的割り付け

この方式は、暇なノードが誰であるかを OS なり処理系なりに尋ねて、ゴールを投げようとした時に一番暇だと思われるノードに割り付ける方式である。ただし、残念ながら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、処理系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⁵を用意する。また、あるノードには全ノード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、全てのノード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、各ノード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、ノードが忙しければ小さなカウンタ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、ノードが暇であれば大きなカウンタ値が通知される。マネージャゴールには最高優先度を与えられているので、各ノードからカウンタ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、カウンタ値を元にして、どのノードが暇であるかが大体わかる。

このようにして暇そうなノードを見つけることは可能であるが、一般的に本当に暇なノードを見つけるのは難しい。例えば、処理系に尋ねた瞬間には本当に暇であっても、そのノード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。

4.9 応用: 簡単な探索問題

本節では、負荷分散制御を応用したプログラム例を示す。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。

⁵ 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

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ー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ノードで実行し、レベルに達したところでサイクリックにノード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台のノード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（67 ページ参照）のゴールをただ呼び出しているだけである。また、ノード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ノード番号を問い合わせるものとする。

では、まず図 4.19 に各ノードのプロセスがどのノードで実行されるかを示す。また、各ノード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ノードを表わす。続けてプログラムをみてみよう。

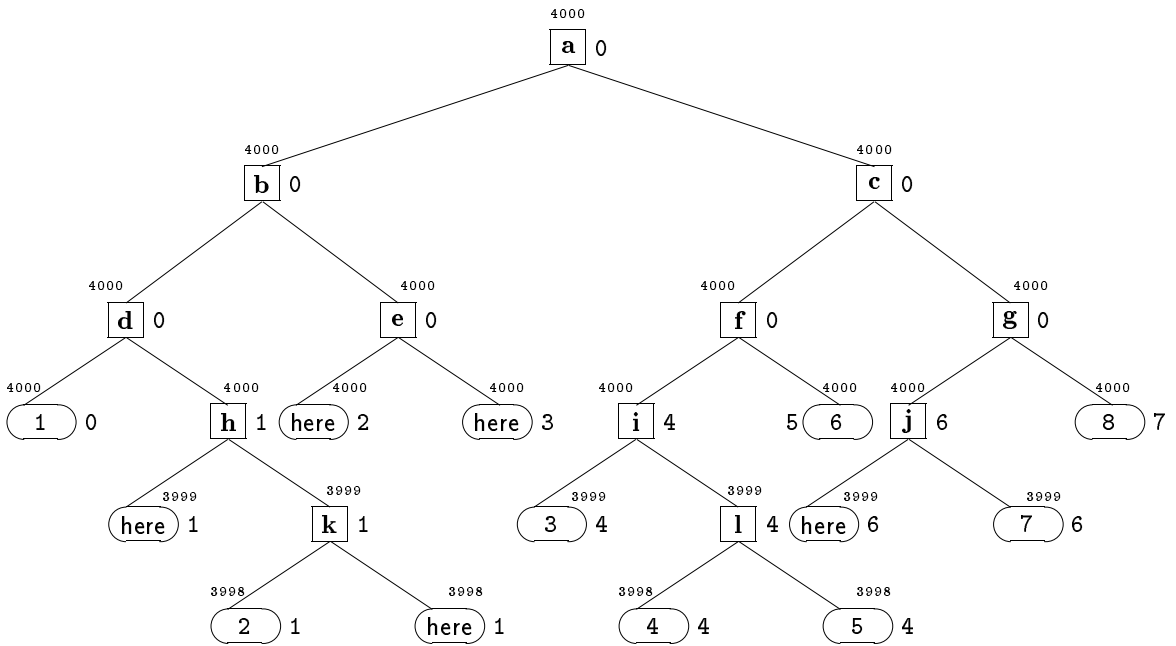


図 4.19: 探索木の各ノードプロセスの実行されるノードと優先度

```
:- module search_load_balancing.

search(Tree,Output):- true |                                     %(1)
    current_node(PE,PEs),                                       %(2)
    control(Results,Output,Cont),                               %(3)
    fork(Tree,Results,0,PeServer,Cont)
        @priority(4000),                                        %(4)
        next_pe(PeServer,PE,PEs).                               %(5)
fork(_,R,_,Next,stop):- true | R=[], Next=[].                 %(6)
alternatively.                                                  %(7)
fork(H,R,_,Next,_):- integer(H) |                               %(8)
    R=[], Next=[].                                              %(9)
fork(H,R,_,Next,Cont):- atom(H) |                               %(10)
    R=[H], Next=[].                                            %(11)
```

```

fork([TL,TR],R,Level,Next,Cont):-                %(12)
    Level =< 1 |                                    %(13)
    L1 := Level+1,                                %(14)
    merge:in(R0,R1,R),                             %(15)
    merge:in(Next0,Next1,Next),                    %(16)
    fork(TL,R0,L1,Next0,Cont),                     %(17)
    fork(TR,R1,L1,Next1,Cont).                     %(18)
fork([TL,TR],R,Level,Next,Cont):-                %(19)
    Level := 2 |                                    %(20)
    Next=[get(PE0),get(PE1)],                      %(21)
    merge:in(R0,R1,R),                             %(22)
    search_pri:fork(TL,R0,Cont)@node(PE0),         %(23)
    search_pri:fork(TR,R1,Cont)@node(PE1).         %(24)

control([H|_],Output,Cont):- wait(H) |            %(25)
    Output=[H], Cont=stop.                         %(26)
control([],Output,Cont):- true | Output=[].        %(27)

next_pe([],_,_):- true | true .                  %(28)
next_pe([get(PeNo)|Next],PE,PEs):-true |          %(29)
    PE1 := PE+1,                                    %(30)
    PeNo := PE mod PEs,                             %(31)
    next_pe(Next,PE1,PEs).                         %(32)

```

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ノード番号と全体のノード数を計算している (2)。 (3)、 (4) では探索を行なうプロセスを生成している。 (5) ではノード番号を得るサーバプロセスを生成している。

fork クローズ (12) 探索のレベルが 1 以下の時には同じノードで実行する。なお、ここで (17)、 (18) 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、ノード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注意しなければならない。

fork クローズ (19) 探索のレベルが 2 に達した時、左右の探索を別ノードで実行させるようゴールを投げている (23)、 (24)。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、 (21) の部分で動作させるべきノード番号を得ている。

next_pe ノード番号をサイクリックに得るためのサーバである。

【演習問題 8】 二進木の探索

二進木を幅優先に探索するプログラムを書け。二進木の形式を図 4.20に示す。

リーフがアトムならばそれが解であり、整数ならばそこには解がないものとする。解が求まったら解に至るルートを [left,right,..., アトム名] の形式で表示する。例えば図 4.20の例でアトム a が見つかった場合は [left,right,left,a] と表示する。例として以下の二進木 T について実行してみよ。

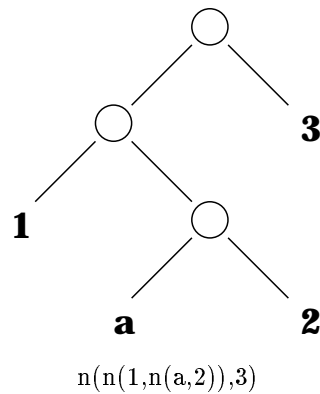
$$T = n(n(n(n(1,n(2,3)),n(4,n(n(a,5),6))),n(b,7)),n(n(n(8,n(9,c)),10),n(n(d,11),12)))$$


図 4.20: 二進木の形式

第 5 章

総合演習問題

【演習問題 9】 ストリームの要素を除去するプログラム

与えられたストリームから指定された要素を全て除去したストリームを作るプログラムを作成しなさい。例えば、ストリームとして $[1, 2, 1, 3, 1]$ を与え、除去する要素を 1 とすると、結果として $[2, 3]$ が返る (図 5.1)。

なお、KL1 のアトミックな項 A と B が等しくないことを調べるには、ガード部で “ $A \neq B$ ” を実行すればよい。または、“ \neq ” の代わりに “otherwise” を用いてもよい。

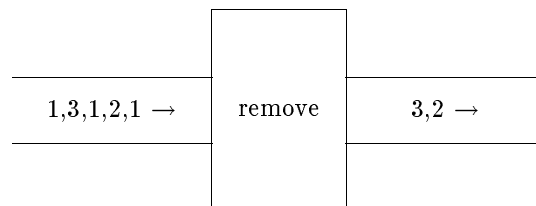


図 5.1: ストリームの要素の除去

【演習問題 10】 ストリームを圧縮するプログラム

与えられたストリームから同一要素をすべて除去したストリームを作るプログラムを作成しなさい。例えば、ストリームとして $[1, 2, 3, 4, 5, 4, 3, 2, 1]$ を与えると、結果として $[1, 2, 3, 4, 5]$ が返る (図 5.2)。

なお、ストリームから要素を除去するには、前の問題で作成したプログラムを利用するとよい。

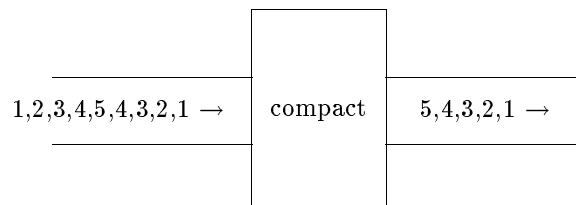


図 5.2: ストリームの圧縮

【演習問題 11】 1,000 以下の素数列を生成するプログラム

1,000 以下の素数を生成し, 生成させた素数の個数を数えて表示するプログラムを作成しなさい.

ヒント 図 5.3, 図 5.4 のプロセス構造を参考にせよ. 素数フィルタは, 新しい素数がみつかった度に生成される.

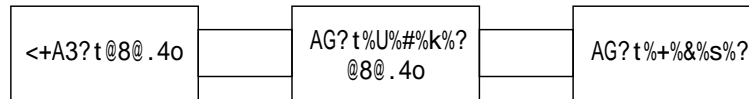


図 5.3: プロセス初期状態

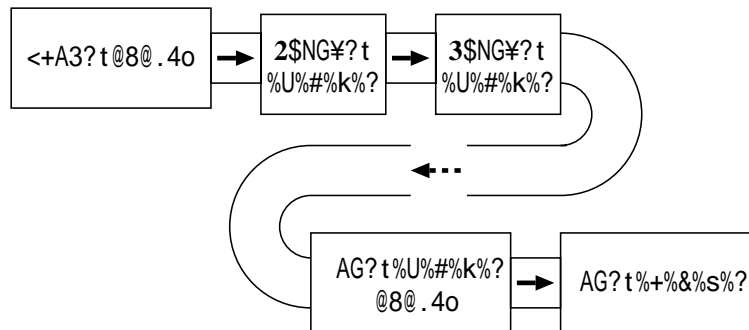


図 5.4: プログラム実行中

【演習問題 12】 20,000 以下の素数列を生成するプログラム

20,000 以下の素数を生成し、生成させた素数の個数を数えて表示するプログラムを作成しなさい。

ヒント 前問と同様な方法を用いると自然数生成器が走り過ぎメモリを大量に消費してしまう。このことを防ぐために要求駆動的なプロセス構造を設定する。図 5.5 のプロセス構造を参考にせよ。同期をとる方法は次の通り。自然数生成器は自然数生成単位 (例えば 100 個毎) にすべての素数フィルタを通りなおかつフィルタプロセスを生成しないデータをひとつ送り、そしてカウンタから次の生成要求が送られてくるのを待つ。カウンタはこのデータを受け取った後、次のデータ生成要求を自然数生成器に送る。自然数生成器は、データ生成要求に従い次の自然数列を生成する。

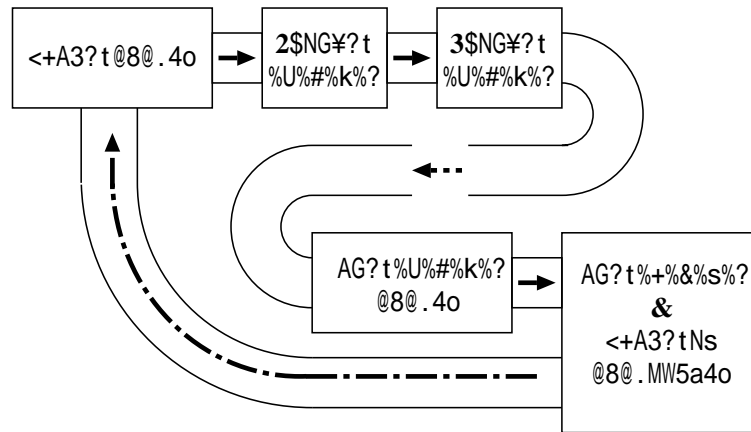


図 5.5: 同期機構付きプロセス構造

もちろん優先度制御を用いるプログラムもある。

【演習問題 13】 1,000 番目の素数を求めるプログラム

小さい方から数えて 1,000 番目の素数を求め、それを表示するプログラムを作成しなさい。

付録 A

演習問題の解答例

【演習問題解答 1】 フィボナッチ数の平方と立方の総和

データ駆動で書いたものが次のプログラムである。本文中に出てきた `fibonacci/4`, `square/2`, `cube/2`, `merge/3`, `sum/2` をストリームでつなげば良い。

```
:- module main.

main :- true |
    main(100,Answer),
    io:ostream([print(Answer),nl]).

main(Target,Answer) :- true |
    fibonacci(0,1,Fibo,Target),
    square(Fibo,Squares), cube(Fibo,Cubes),
    merge(Squares,Cubes,Both),
    sum(Both,Answer).
```

Target=100 とした時の答えは, 935596 である。

要求駆動, あるいはメッセージ駆動で書くのなら, フィボナッチプロセスに要求を出し, その結果であるフィボナッチ数を `square/2`, `cube/2` で実現されるフィルタプロセスに配るプロセスを組み込むのが良い。以下のプログラムはメッセージ駆動で書いたものである。 `square/2`, `cube/2`, `merge/3`, `sum/2` は本文中と同じである。

```
:- module main.

main :- true |
    main(100,Answer),
    io:ostream([print(Answer),nl]).

main(Target,Answer) :- true |
    fibonacci_lazy(0,1,Stream),
    dispatch(Stream,Target,Fibo),
    square(Fibo,Squares), cube(Fibo,Cubes),
    merge(Squares,Cubes,Both),
    sum(Both,Answer).

fibonacci_lazy(N1,N2,[]) :- true | true.
```



図 A.1:

```

fibonacci_lazy(N1,N2,[make(X)|Stream]) :- true |
    X = N2,
    N3 := N1 + N2,
    fibonacci_lazy(N2,N3,Stream).

dispatch(Stream,Target,Fibo) :- true |
    Stream = [make(X)|StreamN],
    dispatch(StreamN,Target,Fibo,X).

dispatch(Stream,Target,Fibo,X) :- X >= Target |
    Stream = [],
    Fibo = [].
dispatch(Stream,Target,Fibo,X) :- X < Target |
    Fibo = [X|FiboN],
    dispatch(Stream,Target,FiboN).

```

【演習問題解答 2】 $2^l \times 3^m \times 5^n$

入力を正の整数Nで割れるだけ割った数を出力するフィルタを用意する。例えば、N=2としたものに12を通すと3を出力するようなフィルタである。また、入力のうち1が何個あったかをカウントするカウンタプロセスを用意する。そして、N=2としたフィルタ、N=3としたフィルタ、N=5としたフィルタ、これらをつないだパイプラインの前に自然数生成プロセス、後ろにカウンタプロセスをつないで、図 A.1のようなプロセス・ネットワークを作れば、問題の式で表せる数が幾つあるかを計算するプロセス・ネットワークが出来る。

```

:- module main.

main :- true |
    main(10000,Answer),
    io:ostream([print(Answer),nl]).

main(Target,Answer) :- true |
    naturals(1,Target,Stream0),
    filter(2,Stream0,Stream1),
    filter(3,Stream1,Stream2),
    filter(5,Stream2,Stream3),
    counter(0,Stream3,Answer).

naturals(N,M,List) :- N>=M |
    List=[].
naturals(N,M,List) :- N<M |
    List=[N|Rest],
    N1:=N+1,

```

```

    naturals(N1,M,Rest).

filter(_,[],Out) :- true |
    Out = [].
filter(X,[E|InN],Out) :- true |
    filter(X,E,InN,Out).

filter(X,E,InN,Out) :- E mod X == 0 |
    EN := E / X,
    filter(X,EN,InN,Out).
filter(X,E,InN,Out) :- E mod X \= 0 |
    Out = [E|OutN],
    filter(X,InN,OutN).

counter(C,[],Res) :- true |
    Res = C.
counter(C,[1|InN],Res) :- true |
    CN := C + 1,
    counter(CN,InN,Res).
counter(C,[E|InN],Res) :- E \= 1 |
    counter(C,InN,Res).

```

Target=10000 とした時の答えは,174 である.

【演習問題解答 3】 データ記録方法の練習

```

sort(I, 0) :- true | sort(I, 0, []).

sort([], CO,C) :- true | CO = C.
sort([H|T], CO,C) :- true | compare(T,H,[],[],L,R),
    sort(L, CO,C1), C1 = [H|C2], sort(R, C2,C).

compare([], V,L,R, OL,OR) :- true | OL = L, OR = R.
compare([H|T],V,L,R, OL,OR) :- H > V | NL = [H|L], compare(T,V,NL,R, OL,OR).
compare([H|T],V,L,R, OL,OR) :- H <= V | NR = [H|R], compare(T,V,L,NR, OL,OR).

```

LIFO 法では, データの終端をもらうまで次の分類に移れない. また出力のための引数を持ち回らなければならず,FIFO 法より不利になる.

【演習問題解答 4】 ソートの練習

```

:- module main.

main :- data:sort(X), qsort(X).

qsort(In) :- true | terminal(In,Out,[]), io:outstream(Out).

node([], V,N,L,R,Xs,Ys) :- true | L = [], R = [], Xs=[print(hist(V,N)),nl|Ys].

```

```

node([H|T],V,N,L,R,Xs,Ys) :- H < V | L = [H|NL], node(T,V,N,NL,R,Xs,Ys).
node([H|T],V,N,L,R,Xs,Ys) :- H > V | R = [H|NR], node(T,V,N,L,NR,Xs,Ys).
node([H|T],V,N,L,R,Xs,Ys) :- H == V | M := N+1, node(T,V,M,L,R,Xs,Ys).

terminal([],Xs,Ys) :- true | Xs = Ys.
terminal([H|T],Xs0,Ys) :- true |
    terminal(L,Xs0,Xs1), node(T,H,1,L,R,Xs1,Xs2), terminal(R,Xs2,Ys).

```

実行結果:

```

hist(0,8)
hist(1,8)
hist(2,12)
hist(3,12)
hist(4,10)
hist(5,8)
hist(6,9)
hist(7,7)
hist(8,12)
hist(9,14)

```

【演習問題解答 5】 2 段ソートの練習

```

:- module(main).

main :- data:sort(X), qsort(X).

qsort(In) :- true | term1(In,Mid,[]), term2(Mid,Out,[]), io:outstream(Out).

node1([],V,N,L,R,Xs,Ys) :- true | L = [], R = [], Xs=[(N-V)|Ys].
node1([H|T],V,N,L,R,Xs,Ys) :- H > V | R = [H|NR], node1(T,V,N,L,NR,Xs,Ys).
node1([H|T],V,N,L,R,Xs,Ys) :- H < V | L = [H|NL], node1(T,V,N,NL,R,Xs,Ys).
node1([H|T],V,N,L,R,Xs,Ys) :- H == V | M := N+1, node1(T,V,M,L,R,Xs,Ys).

term1([],Xs,Ys) :- true | Xs = Ys.
term1([H|T],Xs0,Ys) :- true |
    term1(L,Xs0,Xs1), node1(T,H,1,L,R,Xs1,Xs2), term1(R,Xs2,Ys).

node2([],V,L,R) :- true | L = [], R = [].
node2([(K-M)|T],V,L,R) :- K <= V | R = [(K-M)|NR], node2(T,V,L,NR).
node2([(K-M)|T],V,L,R) :- K > V | L = [(K-M)|NL], node2(T,V,NL,R).

term2([],Xs,Ys) :- true | Xs = Ys.
term2([(K-M)|T],Xs0,Ys) :- true |
    term2(L,Xs0,Xs1), Xs1 = [print(hist(M,K),nl|Xs2],
    node2(T,K,L,R), term2(R,Xs2,Ys).

```

実行結果:

```

hist(9,14)
hist(2,12)
hist(3,12)
hist(8,12)
hist(4,10)
hist(6,9)
hist(0,8)
hist(1,8)
hist(5,8)
hist(7,7)

```

【演習問題解答 6】 二進木の探索

優先度制御により二進木を幅優先に探索するプログラムとその実行結果を以下に示す。また、このプログラムを実行した時の search/3 のリダクションの様子を図 A.2に、探索の様子を図 A.3に示す。

```

:- module main. % 1

main :- true | % 2
      ex:search_tree2(T), search(_,T,Ps), io:outstream([print(Ps),nl]). % 3

search(abort,_,_) :- true | true. % 4
alternatively. % 5
search(AB,n(L,R),Ps) :- true | % 6
      cont(AB,LPs,RPs,ABL,ABR,Ps), % 7
      search(ABL,L,LPs)@lower_priority, search(ABR,R,RPs)@lower_priority. % 8
search(_,Atom,Ps) :- atom(Atom) | Ps = [Atom]. % 9
search(_,Int,Ps) :- integer(Int) | Ps = none. %10

cont(abort,_,_,ABL,ABR,_) :- true | ABL = abort, ABR = abort. %11
alternatively. %12
cont(AB,LPs,_,_,ABR,Ps) :- list(LPs) | Ps = [left|LPs], ABR = abort. %13
cont(AB,_,RPs,ABL,_,Ps) :- list(RPs) | Ps = [right|RPs], ABL = abort. %14
cont(_,none,none,_,_,Ps) :- true | Ps = none. %15

%-----

:- module ex. %16

search_tree2(T) :- T = n(n(n(n(1,n(2,3)),n(4,n(n(a,5),6))),n(b,7)), %17
      n(n(n(8,n(9,c)),10),n(n(d,11),12))). %18

```

実行結果:

```
[left,right,left,b]
```

```

3 REDU:main:search(_B,n(n(n(n(...),n(...)),n(b,7)),          % 1
                    n(n(n(...),10),n(n(...),12))),_3) :-      % 2
7 REDU:main:search(_13,n(n(n(8,n(...)),10),n(n(d,11),12)),_63DF) :- % 3
6 REDU:main:search(_18,n(n(n(1,n(...)),n(4,n(...))),n(b,7)),_63E0) :- % 4
13 REDU:main:search(_4A,n(b,7),_63BC) :-                       % 5
12 REDU:main:search(_4F,n(n(1,n(2,3)),n(4,n(n(...),6))),_63BD) :- % 6
10 REDU:main:search(_30,n(n(d,11),12),_63CD) :-               % 7
9 REDU:main:search(_35,n(n(8,n(9,c)),10),_63CE) :-            % 8
25 REDU:main:search(_B2,10,none) :-                             % 9
24 REDU:main:search(_B7,n(8,n(9,c)),_6376) :-                  % 10
22 REDU:main:search(_98,12,none) :-                             % 11
21 REDU:main:search(_9D,n(d,11),_6363) :-                     % 12
19 REDU:main:search(_7E,n(4,n(n(a,5),6)),_639A) :-            % 13
18 REDU:main:search(_83,n(1,n(2,3)),_639B) :-                  % 14
16 REDU:main:search(_64,7,none) :-                             % 15
15 REDU:main:search(_69,b,[b]) :-                             % 16
39 REDU:main:search(abort,n(2,3),_6330)                       % 17
38 REDU:main:search(abort,1,_6331)                             % 18
36 REDU:main:search(abort,n(n(a,5),6),_6341)                  % 19
35 REDU:main:search(abort,4,_6342)                             % 20
33 REDU:main:search(abort,11,_6352)                           % 21
32 REDU:main:search(abort,d,_6353)                             % 22
29 REDU:main:search(abort,n(9,c),_6365)                       % 23
28 REDU:main:search(abort,8,_6366)                             % 24

```

図 A.2: bf.kl1 の実行トレース (search/3 のリダクション)

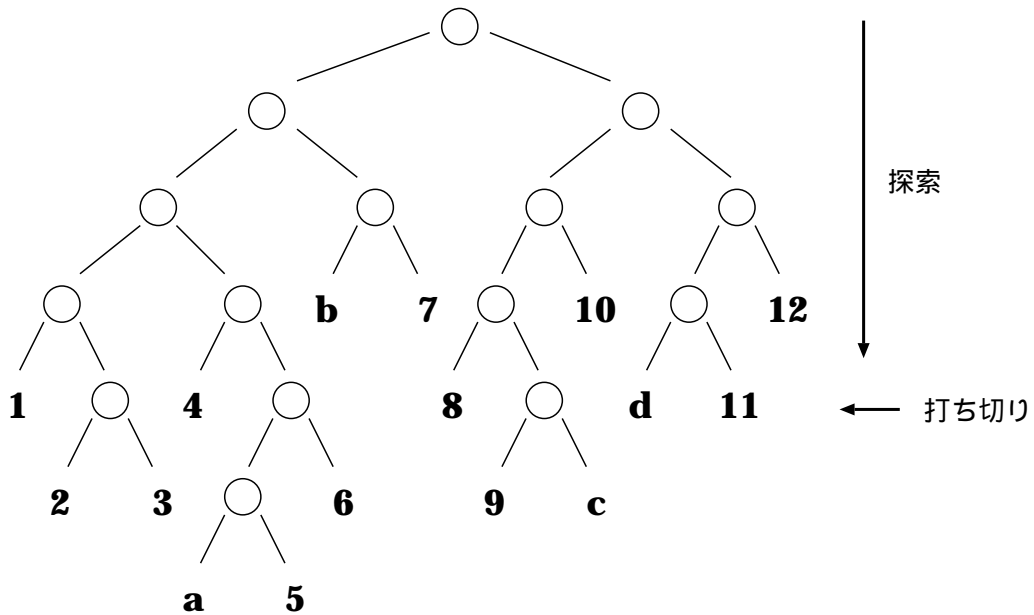


図 A.3: bf.kl1 実行におけるノードの探索

【演習問題解答 7】 ストリームの要素を除去するプログラム

```
:- module main.

main :- true|
    data(List), remove(1,List,Out), io:outstream([print(Out),nl]).

remove(_,[], Out) :- true | Out=[].
remove(X,[X|List],Out) :- true | remove(X,List,Out).
remove(X,[Y|List],Out) :- X \= Y | Out=[Y|OutN], remove(X,List,OutN).

data(List) :- true| List=[1,2,1,3,1].
```

% ‘\=’ の代わりに ‘otherwise’ を用いると remove/3 は次のようになる.

```
remove(_,[], Out) :- true | Out=[].
remove(X,[X|List],Out) :- true | remove(X,List,Out).
otherwise.
remove(X,[Y|List],Out) :- true | Out=[Y|OutN], remove(X,List,OutN).
```

remove/3 は入力ストリームに対して部分的な透過性を持つフィルタである. 第2引数の入力ストリームから第1引数の要素と同じものをすべて除いたストリームを第3引数に返す.

【演習問題解答 8】 ストリームを圧縮するプログラム

```
:- module main.

main :- true|
    data(List), compact(List,Out), io:outstream([print(Out),nl]).

compact([X|List],Out) :- true|
    Out=[X|OutN], remove(X,List,ListN), compact(ListN,OutN).
compact([], Out) :- true| Out=[].

remove(_,[], Out) :- true | Out=[].
remove(X,[X|List],Out) :- true | remove(X,List,Out).
remove(X,[Y|List],Out) :- X \= Y | Out=[Y|OutN], remove(X,List,OutN).

data(List) :- true| List=[1,2,3,4,5,4,3,2,1].
```

compact/2 は第1引数に圧縮すべき入力ストリームをとり, 結果を第2引数に返す. compact/2 は要素が到着するのを待ち, 要素が来たらそれを出力ストリームに流し, その要素と同じものをそれ以降のストリームから除去するプロセス (remove/3) と, その結果を更に圧縮するプロセス (compact/2) とに分岐する.

【演習問題解答 9】 1,000 以下の素数列を生成するプログラム

```
:- module main.                                     % 1
main :- primes:primes(1000, C), io:outstream([print(C),nl]). % 2
```



```

:- module primes.                                     % 3
primes(Max,C) :- gen_primes(Max,Ps), count(Ps,C).      % 4

% Max 以下の素数列を生成する.
gen_primes(Max,Ps) :- gen(2,Max,Ns), sift(Ns,Ps).      % 5

% N0 ~ Max の自然数を生成する. Ns0=[N0,N0+1,N0+2,...,Max]
gen(N0,Max,Ns0) :- N0=<Max | Ns0=[N0|Ns1], N1:=N0+1, gen(N1,Max,Ns1). % 6
gen(N0,Max,Ns0) :- N0 >Max | Ns0=[].                  % 7

% 新しい素数が入力ストリームより流れてきたならば, その素数を
% 用いたフィルタ・プロセスを生成する. そして素数カウンタに
% 新しい素数を送る.
sift([], Zs0) :- Zs0=[].                               % 8
sift([P|Xs1],Zs0) :- Zs0=[P|Zs1], filter(P,Xs1,Ys), sift(Ys,Zs1). % 9

% 入力 X が素数 P で割り切れない場合, 素数候補として X を次の
% フィルタに送る.
filter(_,[], Ys0) :- Ys0=[].                           % 10
filter(P,[X|Xs1],Ys0) :- X mod P=\=0 | Ys0=[X|Ys1], filter(P,Xs1,Ys1). % 11
filter(P,[X|Xs1],Ys0) :- X mod P=:0 | filter(P,Xs1,Ys0). % 12

% 素数をカウントする.
count(L,C) :- count(L,0,C).                             % 13

count([],C0,C) :- C=C0.                                  % 14
count([_|T],C0,C) :- C1:=C0+1, count(T,C1,C).           % 15

```

実行結果(個数の表示)

168

【演習問題解答 10】 20,000 以下の素数列を生成するプログラム

要求駆動のプログラムと優先度制御を利用したプログラムを示す.

要求駆動のプログラム

```
:- module main. % 1
main :- primes:primes(20000, 100, C), io:ostream([print(C),nl]). % 2

:- module primes. % 3
% Sync: メッセージ next の流れるストリームとなる変数である.
% gen_primes はメッセージ next を最初の 1 回を除いて count から受け取る.
% Max 以下 (Max1 未満) の素数列を生成する.
primes(Max,Unit,C) :- Max1:=Max+1, % 4
    gen_primes(Max1,Unit,Ps,[next|Sync]), count(Ps,C,Sync). % 5

% Max 未満の素数列を生成する.
gen_primes(Max,Unit,Ps,Sync) :- gen(Sync,2,Max,Unit,Ns), sift(Ns,Ps). % 6

% 2 以上の自然数を Unit 個ずつ生成する.
% 自然数列の区切り記号として 1 を利用する.
gen([next|_], N0,Max,Unit,Ns0) :- N1:=N0+Unit, Max<N1 | % 7
    gen_unit(N0,Max,Ns0,[]). % 8
gen([next|Sync],N0,Max,Unit,Ns0) :- N1:=N0+Unit, Max >N1 | % 9
    gen_unit(N0,N1,Ns0,[1|Ns]), gen(Sync,N1,Max,Unit,Ns). % 10

% N0 ~ Max-1 の自然数を生成する. Ns0=[N0,N0+1,N0+2,...,Max-1|Ns]
gen_unit(N0,Max,Ns0,Ns) :- N0< Max | % 11
    Ns0=[N0|Ns1], N1:=N0+1, gen_unit(N1,Max,Ns1,Ns). % 12
gen_unit(N0,Max,Ns0,Ns) :- N0>=Max | Ns0=Ns. % 13

% 新しい素数が入力ストリームより流れてきたならば, その素数を
% 用いたフィルタ・プロセスを生成する. そして素数カウンタに
% 新しい素数を送る.
% 素数 P として 1 が来た場合は, 区切り記号が到着したことを
% 意味するので 次の自然数列の発生を要求するためにメッセージ
% next を送る.
sift([], Zs0) :- Zs0=[]. % 14
sift([P|Xs1],Zs0) :- P=\=1 |Zs0=[P|Zs1], filter(P,Xs1,Ys), sift(Ys,Zs1). % 15
sift([1|Xs1],Zs0) :- Zs0=[1|Zs1], sift(Xs1,Zs1). % 16

% 入力 X が素数 P で割り切れない場合, 素数候補として X を次の
% フィルタに送る.
filter(_,[], Ys0) :- Ys0=[]. % 17
filter(P,[X|Xs1],Ys0) :- X mod P=\=0 | Ys0=[X|Ys1], filter(P,Xs1,Ys1). % 18
filter(P,[X|Xs1],Ys0) :- X mod P=:0 | filter(P,Xs1,Ys0). % 19

% 素数をカウントする. 1 を受け取った場合は, 次の自然数列の発生を
% 促すためにメッセージ send を Sync に送る.
```

```

count(L,C,Sync) :- count(L,0,C,Sync). % 20

count([],C0,C,_) :- C=C0. % 21
count([P|T],C0,C,Sync) :- P =\= 1 | C1:=C0+1, count(T,C1,C,Sync). % 22
count([1|T],C0,C,Sync) :- Sync=[next|Sync1], count(T,C0,C,Sync1). % 23

```

実行結果（個数の表示）

2262

優先度を利用したプログラム

```

:- module main. % 1
main :- primes(20000,NP), io:outstream([print(NP),nl]). % 2

primes(Max,NP) :- gen_primes(Max,Ps), count(Ps,NP). % 3

gen_primes(Max,Ps) :- gen(2,Max,Ns)@lower_priority, sift(Ns,Ps). % 4

gen(N,Max,Ns) :- N >Max | Ns=[]. % 5
gen(N,Max,Ns) :- N=<Max | Ns=[N|Ns2], N2:=N+1, gen(N2,Max,Ns2). % 6

sift([], Zs) :- Zs=[]. % 7
sift([P|Xs],Zs) :- Zs=[P|Zs2], filter(P,Xs,Ys), sift(Ys,Zs2). % 8

filter(_,[], Ys) :- Ys=[]. % 9
filter(P,[X|Xs],Ys) :- X mod P=\=0 | Ys=[X|Ys2], filter(P,Xs,Ys2). % 10
filter(P,[X|Xs],Ys) :- X mod P=:0 | filter(P,Xs,Ys). % 11

count(Ps,NP) :- count(Ps,0,NP). % 12

count([], WNP,NP) :- WNP=NP. % 13
count([P|Ps],WNP,NP) :- WNP2:=WNP+1, count(Ps,WNP2,NP). % 14

```

実行結果（個数の表示）

2262

【演習問題解答 11】 1,000 番目の素数を求めるプログラム

```

:- module main.                                     % 1
main :- primes(1000,LP), io:outstream([print(LP),nl]). % 2

primes(Nth,LP) :- gen_primes(AB,Ps), pkup(Ps,Nth,AB,LP). % 3

gen_primes(AB,Ps) :- gen(AB,2,Ns)@lower_priority, sift(Ns,Ps). % 4

gen(abort,_,Ns) :- Ns=[]. % 5
alternatively. % 6
gen(AB, N,Ns) :- Ns=[N|Ns2], N2:=N+1, gen(AB,N2,Ns2). % 7

sift([], Zs) :- Zs=[]. % 8
sift([P|Xs],Zs) :- Zs=[P|Zs2], filter(P,Xs,Ys), sift(Ys,Zs2). % 9

filter(_,[], Ys) :- Ys=[]. % 10
filter(P,[X|Xs],Ys) :- X mod P=\=0 | Ys=[X|Ys2], filter(P,Xs,Ys2). % 11
filter(P,[X|Xs],Ys) :- X mod P=:0 | filter(P,Xs,Ys). % 12

pkup([P|_], Nth,AB,LP) :- Nth=:1 | LP=P, AB=abort. % 13
pkup([_|Ps],Nth,AB,LP) :- Nth=\=1 | Nth2:=Nth-1, pkup(Ps,Nth2,AB,LP). % 14

```

実行結果 (1000 個目の素数の表示)

7919

参考文献

- [1] Takashi Chikayama, Hiroyuki Sato, and Toshihiko Miyazaki. Overview of the parallel inference machine operating system (PIMOS). In **Proceedings of FGCS'88**, pages 230–251, Tokyo, Japan, 1988.
- [2] Keith L. Clark and Steve Gregory. Parlog: A parallel logic programming language. **ACM Transaction on Programming Languages and Systems**, 8(1), 1986.
- [3] Ehud Shapiro. A subset of Concurrent Prolog and its interpreter. ICOT Technical Report TR-003, ICOT, 1983.
- [4] Ehud Shapiro and Akikazu Takeuchi. Object oriented programming in Concurrent Prolog. ICOT Technical Report TR-004, ICOT, 1983. Also in New Generation Computing, Springer-Verlag Vol.1 No.1,1983.
- [5] Kazunori Ueda. Guarded Horn Clauses. ICOT Technical Report TR-103, ICOT, 1985.
- [6] Kazunori Ueda and Takashi Chikayama. Design of the kernel language for the parallel inference machine. **The Computer Journal**, December 1990.
- [7] Takashi Chikayama, Tetsuro Fujise and Daigo Sekita. A Portable and Efficient Implementation of KL1. **Proceedings of PRILP'94**, Lecture Notes in Computer Science #884, Springer-Verlag, 1994.