KLIC ユーザーズ　マニュアル

1995 年 3 月
改訂 1997 年 6 月
このマニュアルは、KLIC 3.1 版に相当します。

近山　隆 (東京大学)
藤瀬 哲郎 (三菱総合研究所)
関田 大吾 (三菱総合研究所)

ICOT 無償公開ソフトウェアの利用条件

1. ICOT 無償公開ソフトウェアの目的

財団法人新世代コンピュータ技術開発機構 (以下、*ICOT* という) は、日本国通商産業省より委託され、第五世代コンピュータ・プロジェクトを推進してきた。また、平成５年度からは、このプロジェクトの後継プロジェクトとして、第五世代コンピュータの研究基盤化プロジェクトを推進している。第五世代コンピュータ・プロジェクトおよびその後継プロジェクト (以下、これらの一連のプロジェクトを本プロジェクトという) は、並列推論処理を中核メカニズムとする新しいコンピュータの基礎技術を創出し、その知見と技術を世界の研究者と共有することによって、コンピュータ科学の発展に貢献することを目的としている。

本プロジェクトによって、並列推論マシン、並列推論ソフトウェア技術といった新しい技術が開発され、また、こうした技術開発に伴い、多くの先進的なソフトウェアが試作されている。これらのソフトウェアは、基礎的な研究開発段階にあるため、多くの研究者に広め発展させていくべきものである。

そこで、 ICOT は、本プロジェクトの国際貢献の目的に鑑み、著作権が国ではなく ICOT に帰属することとなるこれらの研究開発段階のソフトウェアを、「 ICOT 無償公開ソフトウェア」として公開してきた。これらのソフトウェアについては、研究開発のための障害となるいっさいの制約をはずすことによって、多くの研究者の方々に自由に利用してもらい、新しいコンピュータ科学への貢献を実践したいと考えている。

本プログラム及びドキュメント (以下、本プログラムという) は、 *ICOT* 無償公開ソフトウェアの 1 つとして、 ICOT において無償で配布しているものである。

2. 使用、変更、複製、配布の自由

本プログラムの利用者は、その使用、変更、複製を自由に行うことができる。ここでいう変更には、本プログラムの機能、性能、品質を向上させるために改良、拡張を行うこと、もしくは自ら開発したプログラムやドキュメントを本プログラムに追加することが含まれるが、それだけには限定されない。

本プログラムの利用者は、本「 ICOT 無償公開ソフトウェアの利用条件」第 3 項 (無保証) が記されていることを条件として、関連法令に違反しない限り、本プログラムそのもの、または本プログラムの変更版を第三者へ自由に配布することができる。

3. 無保証

本プログラムは、本プロジェクトの研究開発の試作物を『あるがまま』の状態で提供するものである。このため、明示的であるか黙示的であるか、または法令の規定により生ずるものであるか否かを問わず、一切の保証をつけないで提供されるものである。ここでいう保証とは、プログラムの品質、性能、市場性、特定目的適合性、および他の第三者の権利への無侵害についての保証を含むが、それに限定されるものではない。

本プログラムの利用者は、本プログラムが無保証であることを承諾し、本プログラムが無保証であることによるすべてのリスクを利用者自身で負うものとする。

従って、利用者が本プログラムを利用したこと、または利用できないこと、もしくは本プログラムを利用して得られた結果に起因する一切の損害について、著作権者である ICOT および本プログラムの開発に関与した関連機関並びにそれらの役職員及び従業員は、そのような損害の発生する可能性について、知っていたか否かにかかわらず、何らの責任も負わない。本プログラムの利用者は、本プログラムの利用を開始したことによりこれを承諾しているものとみなされる。ここでいう利用とは、本プログラムの使用、変更、複製、配布、二次的著作物の作成を含むがこれらに限定されない。

利用者が本プログラムそのもの、または本プログラムの変更版を、 ICOT 以外の第三者から配布を受けた場合においても、配布を行った第三者が独自に特別な保証を文書で行わない限り、配布を行っ

た第三者は、その利用者に対して、本プログラムに関係する限りにおいて同様に何らの責任を負わないものとする。

JIPDEC 無償公開ソフトウェアの利用条件

ICOT で開発されてきた KLIC は、その後 JIPDEC の AITEC により開発が引き継がれ、配布されています。よって、1996 年以降の著作権は JIPDEC にある旨記載しています。

JIPDEC のコピーライトが附属されている部分についての改変、配布等の条件は「ICOT 無償公開ソフトウェア」のそれと同じとします。つまり、「ICOT 無償公開ソフトウェアの利用条件」の “ICOT” とある部分を JIPDEC として読み替え適用するようにしてください。

なお、JIPDEC の著作ではないソースコードも存在しますが、それらについても同様に「ICOT 無償公開ソフトウェアの利用条件」の著作者、配布者を読み替え適用してください。

1 はじめに

このマニュアルでは、第五世代コンピュータの日本における国家プロジェクトに関与した新世代コンピュータ開発機構と、その後継プロジェクトにおいて開発された KLIC と呼ばれる移植性の高い KL1 の処理系について説明します。

KL1 は Garded Horn Clauses(略して GHC) に基づく、並行論理型プログラミング言語です。KL1 の構文と意味は非常に単純で簡潔ですが、並行計算向けの非常に強力な機能を提供しています。

KLIC は KL1 プログラムを C プログラムにコンパイルします。その後、ホスト・システムの C コンパイラが、 C プログラムを再配置可能なオブジェクトにコンパイルします。続いて、再配置可能なオブジェクトは、 KLIC の実行時ライブラリとともに一括してリンクされます (see Section 4.1.3 [KLIC コンパイラの動作内容], page 49)。したがって、システムはホスト・システムのハードウェア・アーキテクチャに依存しません。さらに、移植性を確保するため、システムは Unix の最小機能だけを使用するように書かれています。

1.1 述語とメソッドの説明について

1.1.1 述語とメソッド

他の論理型プログラミング言語システムと異なり、KLIC は述語とジェネリック・メソッドの 2 種類の手続きを提供します。述語はそれらの引数の関係を定義し、意味は固定です。ジェネリック・メソッド (または、単にメソッド) は、適用するオブジェクトが定義します。したがって、意味は適用されるオブジェクトに依存します。

述語とメソッドの両方で同じ操作を提供するものがあります。例えば、文字列の要素は、次の 2 つのどちらかで操作できます。

string_element *+String +Index -Element* ボディ述語 on builtin
element *+String +Index -Element* ボディ・メソッド on string

前者は、システムの組込み述語で、述語の呼出しは次のように書きます。

ModuleName:*PredicateName*(*Arguments*, ...)

上記の述語 `string_element` の場合は、組込み述語として定義されているので、呼出しにモジュール名は必要ありません。したがって、呼出しは次のように書きます。

`string_element`(*String*, *Index*, *Element*)

一般には、モジュール名がコロン (:) とともに先に書かれ、その後に述語名が書かれます。述語によっては引数を全く持たないものもあります。そのような場合、引数を囲む括弧も省略します。

後者は、文字列クラスのオブジェクトに対して定義されるジェネリック・メソッドで、メソッドの呼出しは、次のように書きます。

`generic:`*MethodName*(*Object*, *OtherArguments*, ...)

上記のメソッド `element` の呼出しは次のように書きます。

`generic:element`(*String*, *Index*, *Element*)

述語とメソッドのどちらでも同じ操作が行えます。例えば、文字列 S の 3 番目の要素 (要素番号 2) を E にとる場合、次に示すどちらの呼出しでもできます。

```
string_element(S, 2, E)
generic:element(S, 2, E)
```

述語 string_element は、文字列の要素をとることだけのものですが、一方、ジェネリック・メソッドの呼出しは、類似のオブジェクトの要素をとるためにも使えますので注意してください。例えば、ベクタ (1 次元配列) の要素も、同じ呼出しでとることができます。

1.1.2 メッセージ

KL1 プログラムは、多数のプロセスで構成することがよくあります。プロセスは、頻繁にストリームを使ってお互いに交信することがあります。ストリームは、実際にはメッセージのリストです。リストは、 Lisp や類似の言語のように、 *car* と *cdr* の 2 個のフィールドを持つコンス・セルと呼ばれるセルからなっています。したがって、メッセージ・ストリームとして使う場合、コンス・セルの car にはメッセージが、cdr にはストリームの残りが入っています。

KLIC システムのいくつかの標準機能も、メッセージ・ストリームのインタフェースを持つプロセスとして提供されます。このマニュアルで説明するそのようなメッセージの例を次に示します。

putc *+C* メッセージ on C 風の入出力

これは、 putc という名前のメッセージが、C 風の I/O プロセスのインタフェース・ストリームへのメッセージであることを意味します。この場合、メッセージは *C* という名前の引数を 1 個持ちます。

メッセージ・ストリームにメッセージを送るために、メッセージ・ストリームを参照する変数を、コンス・セルで具体化します。例えば、 S が C 風の I/O へのストリームで、コード 10 を持つ文字を出力する場合、次のような単一化を行います。

```
S = [putc(10)|T]
```

ここで、変数 T にはストリームの残りが入るので、これ以後のストリームに対するメッセージはすべて、この変数に送ることになります。

1.1.3 引数モード

述語、メソッドまたはメッセージの引数は、特定の入出力モードを持つことがあります。入力引数は、述語やメソッドの呼出しで読み込まれるので、入力引数に未定義のものがあれば中断されます。出力引数には呼出しによって値が与えられます。

述語とメソッドの説明では、入力引数には + を、出力引数には - を付けてあります。引数の中には、読み込まれないものや値が与えられないものがあります。そのような引数には ? が付いています。

1.2 バグ報告とコメントの送付

KLIC システムとこのドキュメントに関する、バグやコメントは、次のメイル・アドレスに報告してください。

```
klic-bugs@icot.or.jp.
```

KLIC のユーザのためのメイル・リストがあります。このメイル・リストは、既知のバグ、バグ改修、新しいリリースなどについて、開発者からのアナウンスに使います。同じメイル・リストをユーザ間のコミュニケーションにも使うことができます。メイル・リストへの申込みは、次のアドレスに送ってください。

```
klic-requests@icot.or.jp
```

2 KL1 とは

KL1 とは、 Guarded Horn Clauses(GHC) に基づいた、並行計算を記述するためのプログラミング言語です。GHC は、並行論理プログラミング言語、またはコミッティッド・チョイス言語と呼ばれる言語の系統に属しています。

他にこの系統に属する言語としては、例えば、 Concurrent Prolog、Parlog、Fleng、Strand、Janus 等があります。これらの言語は、単純で簡潔な構文と意味を持ちながら、並行計算向けに非常に強力な機能を提供しています。

ここでは、 KL1 言語の概略を紹介します。将来 (できれば)、より正確で詳細な記述を補う予定です。

2.1 基本的な実行の仕組み

次に、クイックソート・プログラムの一部を記述した、小さな KL1 プログラムの例を示します。

例 1:クイックソート

```
:- module quicksort.

sort(X, Y) :- sort(X, Y, []).

sort([], Y, Z) :- Y = Z.
sort([P|X], Y, Z) :-
    partition(P, X, X1, X2),
    sort(X1, Y, [P|Y1]),
    sort(X2, Y1, Z).

partition(_, [], S, L) :-
    S = [],
    L = [].
partition(P, [W|X], S, L) :- W =< P |
    S = [W|S1],
    partition(P, X, S1, L).
partition(P, [W|X], S, L) :- W >= P |
    L = [W|L1],
    partition(P, X, S, L1).
```

最初の行 `:- module quicksort.` は、このプログラムモジュールが `quicksort` というモジュールであることを宣言しています (see Section 2.3 [モジュール], page 7) 。

KL1 プログラムの実行とは、(可能ならば並列に行われる) リダクションの繰返しによって、与えられたゴールがプログラム節で繰り返され、リダクションして行くことです。各節は次のような形をしています。

PredicateName(*Argument pattern* ...) :- *Guard* | *Body*.

ゴールがリダクションされるときには、ゴール中の述語に対応するプログラム中の節が調べられ、引数パターンに一致する節が見つかると、そのガード部が調べられます。引数が一致し、ガード部の条件が満たされるような節は、リダクションに使用される候補となります。それらの候補のうちの任意の 1 つが、リダクションに用いられ、元のゴールは、今選ばれた候補のボディ部の (0 個以上の) ゴールによって置き換えられます。

ガード部のテストが必要ない場合、ガード部は、縦棒 (|) と共に省略可能です。

2.2 述語

KL1 の述語は、 Fortran のサブルーチンや、 C の関数に相当します。述語は、ヘッドが同じ述語名で同じ引数個数であるような節の集まりとして定義できます。他のいくつかの言語と異なり、述語は名前だけでなく、アリティ(引数個数) によっても区別されます。同じ述語名でアリティが異なる述語を区別するために、このマニュアルでは、述語名/アリティという表記法を使用します。

先ほどのクイックソートのプログラムでは、`sort` という名前の 2 引数と 3 引数の 2 つの述語が定義されていました (see Section 2.1 [基本的な実行の仕組み], page 6)。このような述語を、 `sort/2` や `sort/3` と表します。

述語を定義する節の順序は、プログラムの意味には関係ありません。例えば、 2 つの整数の大きい方の値を出力する述語は、次のように定義できます。

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

また、同じ述語を次のように節の順序を逆にして定義することもできます。

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

ある節の集まりが適用できないときに限り、他の節の集合を適用したいという場合には、otherwise というキーワードを 2 つの節の集合の間に書く必要があります。例えば、上記の述語 'max' を次のように定義することもできます。

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

この述語の意味は、先ほどの例とほぼ同じですが、 'X' や 'Y' が整数でない場合には、先ほどの 2 つの例では、実行が失敗するのに対し、この述語では、 'M' を 'Y' と単一化するという点が異なっています。

`otherwise` 指示より後ろにある節は、それより前のすべての節が適用できないことが分かるまでは調べられません。このことは、後で何らかの情報 (変数の束縛) が追加された場合でも変わりません。otherwise 指示と alternatively 指示を混同しないようにしてください。`alternatively` 指示は、その指示よりも前にある節が、後ろにある節に比べて優先的に扱われることを示すものです (see Section 2.8 [節の優先関係], page 10)。

2.3 モジュール

KL1 には、大きなプログラムをいくつかのモジュールに分割するためのモジュール構造があります。1 つのモジュールは、 1 つ以上の述語から構成されます。モジュールは、 `:- module` モジュール名`.` の形のモジュール宣言で始まり、その後に述語定義の節が続きます。1 つのモジュール定義は、ファイルの終わり、または別のモジュール宣言で終了します。先ほどのクイックソートの例で、次に示す最初の行

```
:- module quicksort.
```

は、このプログラムモジュールが quicksort であることを宣言しています (see Section 2.1 [基本的な実行の仕組み], page 6)。

同じ名前、同じ引数個数でも、異なったモジュールで定義された述語は、異なる述語とみなされます。したがって、モジュール名を明示することが必要な場合は、モジュール:述語/アリティという表記法が用いられます。

2.4 ゴール

ゴールは、KL1 の実行の単位です。ゴールは述語と結び付けられています。ゴールは、述語を定義する節の 1 つを用いて、0 個以上のより単純なゴールにリダクションされます。

ゴールは以下のように記述されます。

Predicate(*Arguments*, ...)

または、引数を持たない場合には、さらに簡単に

Predicate

のように記述できます。

述語が同じモジュール内にない場合には、

Module:*Predicate*(*Arguments*, ...)

または

Module:*Predicate*

という構文になります。

quicksort モジュールを使う `main` モジュールは、例えば、次のようになります (see Section 2.1 [基本的な実行の仕組み], page 6)。

例 2 : `quicksort` モジュールを使用

```
:- module main.

main :-
    X = [9,2,8,3,6,7,4,1,5],
    builtin:print(X),
    quicksort:sort(X, Y),
    builtin:print(Y).
```

上の例の `quicksort:sort(X, Y)` というボディ・ゴールは、 `quicksort` モジュールの `sort/2` という述語を参照しています。

2.5 初期ゴール

すべての KLIC プログラムは、`main:main` という初期ゴールから実行されます。つまり、`main` モジュールで定義された、引数を持たない `main` という述語です。モジュール `main` の例 (see Section 2.4 [ゴール], page 8) は、メイン・プログラムの例です。コマンド行の引数は、初期ゴールには渡されません。コマンド行引数を得るための述語は、別途用意されています (see Section 3.5.7 [述語インタフェース], page 38)。

2.6 ジェネリック・オブジェクト

ジェネリック・オブジェクトは、 KL1 に新しいデータ型やそれらの操作を追加するための枠組みを提供します。ジェネリック・オブジェクトには、データ・オブジェクト、コンシューマ・オブジェクト、ジェネレータ・オブジェクトという 3 種類があります。

ジェネリック・オブジェクトは、疑似述語 `generic:new` によって作成されます。ジェネリック・データ・オブジェクトは、通常の KL1 のデータと似ています。データ・オブジェクトに対する操作は、それらに対するジェネリック・メソッドという形で定義できます。メソッドは、疑似述語 `generic:`

メソッドによって呼び出されます。コンシューマ・オブジェクトやジェネレータ・オブジェクトは、通常の KL1 プログラムの変数と同じように見えるものであり、それらに対する操作は、単一化によって暗黙のうちに行われます。

KLIC が提供する標準データ型のいくつか、例えば、ベクタや文字列等は、実際にはジェネリック・データ･オブジェクトとして実装されています。これらに関するジェネリック･メソッドは、組込み述語としても呼び出せるようになっています。例えば、`set_vector_element(`*Original*`,`*Index*`,`*NewElement*`,`*New*`)` は、`generic:set_element(`*Original*`,`*Index*`,` *NewElement*`,` *New*`)` と同じことを意味します。

2.6.1 ジェネリック・オブジェクトの生成

ジェネリック・オブジェクトは、次に示す疑似述語によって生成されます。

`generic:new(`*ClassName*`,` *Object*`,` *Args*`,` `...)`

ClassName は、オブジェクトのクラス名を示す記号アトムです。この疑似述語の呼出しによって、新しくジェネリック・オブジェクトが生成され、 *Object* で指定された変数に結び付けられます。ジェネリック・オブジェクト生成時に必要なパラメタは、 *Args* で与えます。*Args* の意味は、それぞれのオブジェクト・クラスに依存します。

2.6.2 ジェネリック・データ・オブジェクトのガード・メソッド

ガード・メソッドを呼ぶことで、ジェネリック・データ・オブジェクトに依存して節が選択できます。ガード・メソッドは、以下の形式を持っています。

`generic:`*Method*`(`*Object*`,` *Input*`,` `...):`*Output*`:...`

Input`,...` では、入力引数を指定します。もし、入力引数のどれかが未定義のままである場合には、この呼出しは中断されます。*Output*`:...` では、出力引数を指定し、このメソッドからの返却値がここに入れられます。もし、 *Output* が既に何らかの具体値を持っていた場合には、その値と返却値との間のガード・ユニフィケーションが行われます。ガード・メソッドが返却値を持たない場合には、コロン (:) とそれに続く *Output* は省略されます。

2.6.3 ジェネリック・データ・オブジェクトのボディ・メソッド

ジェネリック・データ・オブジェクトに対する操作は、次に示すボディ・メソッドの呼出し形式で行います。

`generic:`*Method*`(`*Object*`,` *Args*`,` `...)`

ガード・メソッドとは異なり、入力引数と出力引数は構文的には区別されません。また、実行時に動的に呼び出されるメソッドを決めることもできます。その場合は、次の呼出し形式を用います。

`generic:generic(`*Object*`,` *Functor*`)`

この呼出し形式では、 *Functor* は (コンパイル時、または実行時に) *Method*`(`*Args*`,...)` というファンクタ構造でなければなりません。*Functor* が未定義の場合、具体化されるまで呼出は中断します。

2.7 優先順位の指定

ゴールには、実行優先度 (Execution Priority) が付いています。実行優先度には、正の整数を指定します。大きな優先度を持つゴールは、(通常) 小さな優先度を持つゴールよりも先に実行されます。

しかし、優先度の指定は単なる示唆にすぎず、実装方式によっては、優先度が厳密に守られないこともあります。

ボディ・ゴールの実行優先度は、次の形式で指定されます。

Goal`@priority(`*AbsPrio*`)`
Goal`@lower_priority`
Goal`@lower_priority(`*RelPrio*`)`

上記の *AbsPrio* と *RelPrio* は、負以外の整数定数、または実行時に負以外の整数になるような変数でなければなりません。現在の実装方式では、優先度に負が指定された場合には 0 が指定されたものとして扱われます。

priority による絶対優先度指定によって、ゴールには *AbsPrio* で指定された優先度が与えられます。@lower_priority(RelPrio) による相対優先度指定では、ゴールは親のゴールの優先度より RelPrio だけ小さい優先度を持ちます。*Goal*`@lower_priority` という指定は、*Goal*`@lower_priority(1)` と同じ効果を持ちます。優先度指定のないゴールは、親のゴールと同じ優先度を持ちます。

優先度の最大値は整数の最大値であり、ホスト・システムに依存します (see Section 3.2.2 [整数アトム], page 20)。初期ゴール `main:main` は、ホストシステムにおいて最大の優先度を持ちます (see Section 2.5 [初期ゴール], page 8)。

2.8 節の優先関係

KL1 の述語では、 2 つ以上の節が適用可能なために非決定性を持つことがあります。その場合、節同士の優先関係を `alternatively` 指示によって指定できます。

`alternatively` というキーワードを 2 つの節の集合の間に書いた場合、 `alternatively` より前の節が、後ろの節よりも優先的に選ばれます。もし、 `alternatively` 指示より前の節が情報不足 (変数値の具体化が足りず) のため、選択不可能な場合、この指示よりも後の節が選択の対象となります。この機能は、実行状況に依存した投機的実行 (speculative execution) の制御に利用できます。

alternatively と `otherwise` を混同しないでください (see Section 2.2 [述語], page 7)。例えば、次の 2 つの述語を考えてみます。

```
p(1, Y, R) :- R = a.
alternatively.
p(X, 2, R) :- R = b.

q(1, Y, R) :- R = a.
otherwise.
q(X, 2, R) :- R = b.
```

第 1 引数が未定義で、第 2 引数が 2 の場合、述語 `p` は 2 番目の節の実行によって第 3 引数に `b` を返しますが、述語 `q` は第 1 引数が束縛されるまで待ちます。つまり、第 1 引数が最終的に `1` になるならば、述語 `q` では `a` が返されることが保証されますが、述語 `p` では `a` か `b` のどちらかが返されることになります。

2.9 引数対の簡略表記

KL1 のプログラムでは、述語に対して 1 つを入力、 1 つを出力というように 2 つの引数を対として与えることがよくあります。KLIC では、このような場合のために簡略表現を用意しています。

2.9.1 引数対と引数対の展開

述語のヘッドや、ガード部やボディ部にあるゴールに対して、 1 つの変数名をマイナス符号で続けることで、引数対を与えることができます。この疑似変数名は、マクロ展開のために用いられる変数名であり、通常の KL1 の変数と区別するために引数対名と呼びます。次に例を示します。

```
p(X,Y)-Pair :- q(X)-Pair, s(Z)-Pair, r(Pair,Y), t(Z)-Pair.
```

この場合、疑似変数 Pair が引数対名です。この節は、次の節と同じであると解釈されます。

```
p(X,Y,P0,P) :- q(X,P0,P1), s(Z,P1,P2), r(P2,Y), t(Z,P2,P).
```

ヘッドやゴールに付いているマイナス符号に続く引数対名は、引数列の最後に追加された 2 つの異なった変数の対と解釈されます。これ以後、この 2 つの変数を、引数対から生成された展開対と呼びます。

あるゴールの中の展開対の 2 番目の変数は、次の同じ引数対名を持つゴールの展開対の 1 番目の変数と同じです。上の例では、 P1 はゴール q/3 の第 3 引数に現れ、また s/3 の第 2 引数にも現れています。これらは、元々同じ引数対名 Pair だったものです。

ヘッドにある展開対の 1 番目の変数は、節内で同じ引数対名を持つ最初のゴールの展開対の 1 番目の変数と同じです。またヘッドにある展開対の 2 番目の変数は、節内で同じ引数対名を持つ最後のゴールの展開対の 2 番目の変数と同じです。

上の例では、ヘッドにある展開対の 1 番目の変数 P0 が、最初のゴール q/3 の第 2 引数にも現れており、ヘッドにある展開対の 2 番目の変数 P は、最後のゴール t/3 の第 3 引数にも現れています。

引数対名がヘッドにしか現れていない場合には、展開対の 2 つの変数がボディにおいて単一化されるコードが生成されます。例えば、

```
p(X)-Y :- q(X).
```

は、次のように展開されます。

```
p(X,Y0,Y) :- Y0=Y, q(X).
```

先ほどの例では、ゴール r/2 の 1 番目の引数に見られるように、引数対名は、ヘッドやボディにマイナス符号を付けて指定されるだけでなく、通常の引数位置に置かれることもあります。この場合、引数対名は、単一の変数に展開されます。この変数は、直前の展開対の 2 番目の変数と同じで、直後の展開対の 1 番目の変数とも同じです。したがって、この例では、 r/2 の第 1 引数 Pair は P2 に展開され、これは s/3 の第 3 引数および t/3 の第 2 引数と同じになります。

ゴールやヘッドには、任意の数の引数対名が書けます。例えば、次の例で、

```
p-X-Y :- q-X, r-Y, s-Y-X.
```

は、次のように解釈されます。

```
p(X0,X,Y0,Y) :- q(X0,X1), r(Y0,Y1), s(Y1,Y,X1,X).
```

時には、通常の引数を引数対名の後に指定したくなる場合もあるでしょう。その場合には、プラス符号 + の後ろに通常の引数を付けることで指定できます。例えば、

```
p-X+Y :- q-X+35, r(Y), s+Y-X.
```

は、次のように解釈されます。

```
p(X0,X,Y) :- q(X0,X1,35), r(Y), s(Y,X1,X).
```

以上、述べた引数対の展開規則は、ゴールの位置に依存することに注意してください。しかし、これによってボディ・ゴールの実行順序が何らかの制約を受けることを意味するものでは全くありません。

引数対の記法は、節のマクロ展開にすぎないということも覚えておいてください。1 つの述語のいくつかの節を引数対の記法を用いて書き、他の節は通常の記法で書くことも可能です。

2.9.2 引数対のためのマクロ

KLIC では、引数対をより多くの場面で利用するため、次のようなマクロを用意しています。

S `<=` *M*　*S0* `=` `[`*M*`|`*S1*`]` に展開されます。*S0* と *S1* は引数対名 *S* を展開した展開対です。

M `=>` *S*　`[`*M*`|`*S0*`]` `=` *S1* に展開されます。*S0* と *S1* は引数対名 *S* を展開した展開対です。

S `+=` *E*

S `-=` *E*

S `*=` *E*

S `/=` *E*　*S1* `:=` *S0* `+` *E0* 等に展開されます。*S0* と *S1* は引数対名 *S* を展開した展開対です。

S `<==` *X*　*S1* `=` *X* に展開されます。*S0* と *S1* は引数対名 *S* を展開した展開対です。*S0* は展開型には現れません。つまり、元々の引数対 *S* の値は失われます。その代わり、次から現れる *S* は *X* を意味することになります。このマクロは、通常、引数対が対にならずに出現する場合に用いられます。例えば、

```
...
        , p-S, q(S), S <== X, r-S, ...
```

は、次のことを意味します。

```
...
        , p(S0,S1), q(S1), S2 = X, r(S2,S3), ...
```

2.9.3 引数対の使い方

ここでは、引数対の標準的な使用例をいくつか示します。

次のプログラムは、整数リストの各要素を加え合わせるプログラムです。

```
sum(List,Sum) :- sum(List)+0+Sum.

sum([])-Acc.
sum([H|T])-Acc :- Acc += H, sum(T)-Acc.
```

ここでは、引数対 `Acc` はアキュムレータの役割を果たします。

次のプログラムは、整数リストの各要素の符号を反転します。

```
inv(List,Inv) :- inv(List)+Inv-[].

inv([])-Inv.
inv([H|T])-Inv :- MH := -H, Inv <= MH, inv(T)-Inv.
```

2.10 C 言語コードのインライン挿入

インライン C コード機能によって、 KL1 プログラム中にオブジェクト・コードの中に埋め込む C プログラムが記述できます。この機能は、 C における `asm` 文に多少似ています。

挿入された C コードが適切なものであるか否かは、 KLIC システムの内部実装方法に完全に依存します。そして、これは将来変わるかもしれません。したがって、一般ユーザがこの機能を使うことは勧められません。

2.10.1 ファイル先頭でのインライン挿入

ソース・ファイルの先頭では、以下の方法によって C プログラムに挿入される文字列が指定できます。

```
:- inline:"挿入されるべき C プログラム・テキスト".
```

オブジェクトの C プログラム中では、ここで指定したテキストは、標準的な宣言の後で、かつユーザが定義したモジュールの前に挿入されます。

インライン指定は、いくつも書くことができます。標準的な例を次に示します。

```
:- inline:"#include <stdio.h>"
```

挿入される C プログラムは、 KLIC の文字列定数として書く必要があるので、C 中の二重引用符文字は、バックスラッシュ (\) によってエスケープする必要があります。典型的な例は次のとおりです。

```
:- inline:"#include \"myheader.h\""
```

この部分で、節のガード部に書くインライン・コードで使用するマクロや関数を定義しておくのは、良い考えかもしれません。

2.10.2 ガード部でのインライン挿入

次に示す形式のどちらかを使って、ガード・ゴールでもインライン挿入が指定できます。

```
inline:"C プログラム・テキスト"
inline:"C プログラム・テキスト":[ArgSpec, ...]
```

どちらの形式でも、プログラム・テキスト文字列中のパーセント記号 (%) が特別なフォーマットを指定することの他は、 C プログラムのテキストは指定したとおり、ガード部に対応するオブジェクト・コード中に挿入されます。次の表に、パーセント記号の後ろに指定される特殊なフォーマット文字とその意味を列挙します。

〈数字〉　〈数字〉番目 (ゼロから始まる) の *ArgSpec* に対応する C 変数の名前。10 個までの引数しか許されないので注意してください。

`f`　この節が失敗や中断した場合に、分岐する `goto` の分岐先ラベルの名前

`%`　パーセント文字自身

ArgSpec は、次のいずれかの形式です

変数+型　挿入されたプログラム・テキストが、変数の値を使用することを指示します。オブジェクト・コードには、コンパイラによって、同期と型チェックのために必要なコードが生成されます。

変数-型　挿入されたプログラム・テキストによって変数に値が与えられることを意味します。この変数は、ここで最初に出現するものでなければなりません。コンパイラは挿入されたコードを実行した後は、変数が型の値を持つと、仮定されます。

型フィールドは、以下のいずれかでなければなりません。

`any`　未束縛の変数を含む任意のもの

`bound`　任意の束縛されているもの

`atomic`　アトム値 (アトムまたは整数)

`int` 整数

`atom` 記号アトム

`list` リスト構造

`functor` ファンクタ構造 (ジェネリック・オブジェクトを含む)

`object` ジェネリック・データ・オブジェクト

`object(Class)`
クラスが *Class* であるようなジェネリック・データ・オブジェクト

any 以外の型では、間接表現は取りません。入力モード (+) 引数では、コンパイラが出力するコードによって、挿入されたプログラム・テキストが実行される時点では、引数は間接参照ではなく、指定した型の値そのものであることが保証されています。逆に、出力モード (-) 引数では、挿入されたプログラム・テキストの実行後は、その変数の値は間接参照ではなく、指定した型の値そのものであるとコンパイラは仮定しており、この情報を最適化のために利用します。もし、出力変数に関するこの仮定に確信が持てない場合には、効率は悪いが安全な型として any を指定してください。

2.10.3 KL1 項の C レベル表現

KL1 変数に対応する C の変数の型は、 KL1 のデータ型から推測される C の型とは違うことに注意してください。

KL1 変数に対応する C の変数はすべて、 1 ワードを占めるという以外の意味を持たない q という型を持ちます。また、 KL1 の値は (タグが付加される等によって) エンコードされています。例えば、 KL1 における整数 3 は、 C の整数 3 とは異なったビットパターンを持ちます。

このドキュメントは、 KLIC におけるデータ表現形式の詳細を表すことを目的としたものではありませんし、そのような記述や、それに依存するプログラムは、将来の KLIC の版では、使えなくなるでしょう。しかしながら、インライン機能を用いて書かれるほとんどの C のプログラムにおいて、整数値の操作は最も簡単で役に立つものなので、整数のためのデータ変換用マクロを以下に説明します。これらは将来のバージョンにおいても変更されることはないでしょう。

C の変数から KL1 の整数値に対応する整数値を得るには、`intval(`*X*`)` というマクロを用います。C の整数を KL1 表現にするには、`makeint(`*N*`)` を用います。

2.10.4 例

例 **1: 2** つの整数の加算

2 つの整数は、次の節によって加算できます。

```
p(X,Y,Z) :- W := X+Y | Z = W.
```

同じ機能は、次のようにインライン機能を用いて実現できます。

```
p(X,Y,Z) :-
  inline:"%2 = makeint(intval(%0)+intval(%1));":
    [X+int, Y+int, W-int] | Z=W.
```

挿入されるテキストは、次のようになります。

```
x0 = makeint(intval(a0)+intval(a1));
```

変数 a0 と a1 は KL1 プログラムの X と Y に対応し、 x0 は W に対応します。

例 **2: 2** つの整数の比較

2 つの整数は、次の節によって比較できます。

```
p(X,Y) :- X > Y | ...
```

同じことは、インライン機能を用いて次のように実現できます。

```
p(X,Y) :-
  inline:"if (intval(%0) <= intval(%1)) goto %f;":
    [X+int, Y+int] | ...
```

挿入されるテキストは、次のようになります。

```
if (intval(a0) <= intval(a1)) goto p_2_interrupt;
```

変数 a0 と a1 は、 KL1 プログラムの X と Y に対応し、 p_2_interrupt はコンパイラによって自動的に生成されたラベルです。

2.10.5 インライン C コード機能を使う上でのヒント

- できる限りインライン機能を使うのは避けてください。KL1 の版によっては、コードが使えなくなるかもしれません。
- 複数の行を連続して挿入する場合には、 1 回のインライン記述にそれらすべてを書いてください。そうでないと、コードの間にガードのための他のコードが割り込むかもしれません。挿入するプログラム・テキストには改行を入れることが許されています。
- 二重引用符やバックスラッシュの前にはバックスラッシュを付けることを忘れないでください。Hello, world と出力したいならば、次のように書く必要があります。

  ```
  hello :-
      inline:"printf(\"Hello, world\\n\");" |
      ...
  ```

  二重引用符の前のバックスラッシュと n の前の 2 つのバックスラッシュに注意してください。もし n の前にバックスラッシュを 1 つしか付けないと、インライン展開の結果、これは改行コードとなってしまいます。この改行コードは展開された C プログラムの中で文字定数内に入り、途中で改行された文字列となります。幸運にもこの場合は、バックスラッシュを 2 つ入れた場合と同じように動きますが、 C コンパイラによっては警告メッセージが出力されるかもしれません。
- もし、インライン・コードを含むプログラムが期待どおりに動かない場合には、生成された C コードを調べることが、問題を見つける最良の方法かもしれません。

3 組込み述語とライブラリ機能

この章では、 KLIC の組込み述語とライブラリ機能について説明します。

3.1 共通操作

述語には、すべてのデータ型に共通するものや依存しないものなどがあります。

3.1.1 単一化

= *?X ?Y* ガード述語 on builtin

X と Y が単一化可能か否かをチェックします。この述語の結果は、他の節で説明する変数には影響しません。

= *?X ?Y* ボディ述語 on builtin

X と Y を単一化します。X が未定義で、かつ Y が具体値である場合、X を Y と同じ値にします。Y が未定義で、かつ X が具体値である場合、Y を X と同じ値にします。両方が未定義の場合、2 つの変数は、同じ変数を意味するようになります。両方が具体値の場合、マッチング処理を行います。両方が同じ種類のデータ構造の場合、この単一化操作を、2 つの構造の対応する要素に対して再帰的に行います。

3.1.2 同期

wait *+X* ガード述語 on builtin

X が具体化されるまで待ちます。

3.1.3 比較とハッシュ

compare *+X +Y -R* ガード述語 on builtin

X と Y を比較し、R へ結果を返します。$X < Y$ の場合、$R < 0$ の整数値にします。$X = Y$ の場合、$R = 0$ にします。$X > Y$ の場合、$R > 0$ の整数値にします。

比較は、以下に示した標準順序に従って行います。この述語では、色々な型のデータが比較できます。X と Y の両方が同じ数値型の場合、通常の数値比較を行います。ただし、整数と浮動小数点数は、同じ型ではないので注意してください。このような比較を行なっても意味がありません。文字列の場合は、(いわゆる) 辞書式順序で比較します。

型の異なる任意の 2 つのデータの順序は、システムによって定義されます。ただし、この順序付けは、単独の実行可能プログラム内でしか保証しません。この述語が提供する順序付けを使用して、データ列をファイルに保存しても、プログラムを、再コンパイルしたり、他のプログラムとリンクしたりすると、そのプログラムには、データ列を順序どおりに認識するという保証がなくなってしまいます。プログラムが異なると当然、順序付けも異なってきます。

比較を行うためには、X と Y がともに、十分具体化されていなければなりません。例えば、‘`f(V) @< f(W)`’ は、‘`V`’ と ‘`W`’ の両方が具体化されない限り、比較の処理を中断します。これに対し、‘`f(1,V) @< f(2,W)`’ は、‘`V`’ や ‘`W`’ の値を調べるまでもなく、順序が決定できるので成功します。

データ型によっては、結果 R の絶対値が意味を持つ場合があります。2 つの文字列の比較の場合、結果 R の絶対値は、最初に異なる要素位置を示す (C の `strcmp`の) インデックスよりも 1 つ大きな値になります。

ジェネリック・オブジェクトは、比較メソッドを実装しない場合があります。このような場合、その比較は、重大エラーを引き起こすことになります。

`@<` +X +Y ガード述語 on builtin
`@=<` +X +Y ガード述語 on builtin
`@>=` +X +Y ガード述語 on builtin
`@>` +X +Y ガード述語 on builtin

標準順序で X と Y を比較します。条件を満たさない場合、述語の呼出しは失敗します。

`\=` +X +Y ガード述語 on builtin

X と Y を比較し、主ファンクタが異なる場合だけ成功します。アトム値の場合、それらが異なるアトムであることを意味します。ファンクタ構造の場合、それらが異なるファンクタ名を持つか、または異なる引数個数を持つか、のどちらかであることを意味します。ジェネリック・オブジェクトの場合、2 つのオブジェクトが異なるクラスのときに成功します。

‘`f(a) \= f(b)`’ は、2 つの項が同じ主ファンクタを持つ場合、失敗するので注意してください。浮動小数点数はジェネリック・オブジェクトなので、任意の 2 つの浮動小数点数が同じクラスのオブジェクトの場合、‘`X \= Y`’ は失敗するのでこの場合にも注意してください。

hash +X -H ガード述語 on builtin

X のハッシュ値を計算し、その結果を H へ返します。ハッシュ値は、負以外の整数値です。

ハッシュ機能は、構造型データの要素を再帰的に調べます。ハッシュ値を計算するには、X は、十分に具体化されていなければなりません。ジェネリック・オブジェクトは、ハッシュ・メソッドを実装しない場合があります。このような場合、そのハッシュ値は定数値となります。

3.1.4 実行ステータス

current_priority -P ガード述語 on builtin

リダクションしたゴールの優先順位の値を P へ返します。優先順位の仕組みについての詳細は、See Section 2.7 [優先順位の指定], page 9。

current_node -*Node* -*NumNodes* ボディ述語 on builtin

並列実装では、述語を実行するノード番号を *Node* へ、使用可能な実行ノード総数を *NumNodes* へ返します。初期ノード番号は、0 です。このため、*Node* へ返す最大値は、*NumNodes* へ返す値より小さくなります。逐次実装では、*Node* へ 0 を、*NumNodes* へ 1 を返します。

3.1.5 デバッグ

unbound *?X -Result* ボディ述語 on builtin

X が、未定義変数か否かをチェックし、その結果を *Result* へ返します。

X が未定義変数でない場合、*Result* には、形式 `{X}` の単一要素のベクタを単一化します。*X* が構造体の場合、その要素が未定義変数でない、という保証はありません。

X が未定義変数の場合、*Result* には、形式 `{`*Addr1*, *Addr2*, *X*`}` の 3 つの要素ベクタを単一化します。*Addr1* と *Addr2* は、変数 X の現在のアドレスを示す整数となります。変数のアドレスは、ガーベージ・コレクション、並列実装における自動データ移動、または低レベル実装などによって、変化するので注意してください。これらは、デバッグする上では、ヒント以上の情報にはなり得ません。

通常のアプリケーション・プログラムでは、この述語を使用してはなりません。逐次 Prolog の `var/1` 機能とは異なり、並列実装では、未定義であると判断された直後に、未定義でなくなる可能性があります。したがって、この述語の使用は、システム実装の低レベルを詳細に調べる、デバッグツールのようなプログラムに限定されます。

3.2 アトム・データ

KLIC は、数値と記号の 2 種類のアトム・データ型を提供します。

KLIC は、数値データに対し、整数と浮動小数点数のデータ型、およびそれらを操作する演算を提供します。浮動小数点数は、ジェネリック・オブジェクトとして実装されているため、実際にはアトムではありません。

整数データと浮動小数点データ間では、暗黙の型変換は行わないので注意してください。整数と浮動小数点数は、完全に別個のものとして扱います。

データがアトムか否かは、次のガード述語でテストします。

atomic *+X* ガード述語 on builtin

X がアトムか否かをテストします。この述語では、浮動小数点数は、アトムとして判定しません。

3.2.1 記号アトム

記号アトム(symbolic atoms) は、概念に名前を与えるアトム・データです。同じ名前を持つ記号アトムは同じものであり、異なる名前を持つ記号アトムは異なるものとなります。

3.2.1.1 記号アトムの表記

記号アトムの表記は、次に示すように Edinburgh Prolog に似ています。

- 先頭がアルファベットの小文字で、そのあとに任意個 (ゼロ個を含む) の文字、数字 、または下線の並びが続きます。

  例:

  ```
  icot   kl1   a_symbolic_atom_with_a_long_name
  ```

- 特殊文字の並び (`~`,`+`,`-`,`*`,`/`, `\`,`^`,`<`,`>`,`=`,`` ` ``(backquote), `:`,`.`,`?`,`@`,`#`,`$`,`&`).

  例 :

  ```
  +   >=   :-   =:=
  ```

- 単一引用符で囲まれた任意の文字の並び単一引用符が文字の並びに含まれる場合は、 2 個続けるか、またはバックスラッシュ (\) を前に付けてエスケープします。

  例：

  ```
  'Hello world'    'an atom with \'singlequotes\' in it'
  ```

- 1 文字の特殊アトムには !、|、および ; の 3 種類があります。なお、| はリスト表記では、特別な意味を持っています (see Section 3.3.2.1 [リストの表記], page 26)。
- 特殊アトムの [] は通常、リストの終端を表わすために使用します (see Section 3.3.2 [リスト], page 25)。[と] の間には、空白を許しています。

Edinburgh Prolog の構文と異なる点のうち、重要なものを以下に示します。

- 縦棒 (|) は、 1 文字アトムを表します。演算子として使用する場合でも、セミコロン (;) と同じ扱いはせず、異なるアトムとして扱います。
- 1 対の中括弧 ({}) は、記号アトムではありません。要素のないベクタを表します (see Section 3.3.3.1 [ベクタの表記], page 28)。

3.2.1.2 記号アトムの操作

データが記号アトムか否かは、次のガード述語でテストします。

atom +*X* ガード述語 on builtin

X が記号アトムか否かをテストします。

記号アトムの一意性を保持するために、処理系では、各記号アトムに一意な番号を付けて、記号アトムの名前文字列と記号アトム番号との関係を保持しています。記号アトムと名前の関係は、atom_table モジュールで定義した次の述語によって調べることができます。

make_atom +*String* -*Atom* 述語 on atom_table

String が与えられたとき、その名前を持つ *Atom* を返します。そのような記号アトムが存在しない場合、新しい記号アトムを登録します。

atom_number +*Atom* -*Number* 述語 on atom_table

Atom の内部表現で利用している通し番号を、整数で *Number* へ返します。

get_atom_string +*Atom* -*String* 述語 on atom_table

Atom の名前文字列を *String* へ返します。

intern +*String* -*Result* 述語 on atom_table

返却値が normal(*Atom*) 形式のファンクタ構造である他は、atom_table:make_atom と同じです。

get_atom_name +*Atom* -*Result* 述語 on atom_table

返却値が normal(*String*) 形式のファンクタ構造である他は、atom_table:get_atom_string と同じです。

記号アトムは名前文字列を持っていますが、それらの名前文字列を、文字列操作のために使用すべきではありません。文字列データは、より豊富な機能性と、より良い性能を提供しています (see Section 3.3.4 [文字列], page 29)。

3.2.2 整数アトム

KLIC は、基本的な標準機能として、通常、28 ビットか 60 ビットの整数データを提供します。そのビット長は、使用する C コンパイラに依存します。整数データは、 `long int` 型のビット長に比べ、4 ビット短くなります。

integer +*X* ガード述語 on builtin

X が整数アトムか否かをテストします。

3.2.2.1 整数の表記

KLIC は、整数定数を表す方法を提供します。

- 通常の 10 進表記: 任意指定のマイナス符号のあとに、10 進数字の並びを続けます。例:‘`123`’,‘`-35`’。
- 基数表記: 任意指定のマイナス符号のあとに、基数を規定する 10 進数字の並び (1〜36) 、アポストロフィ、そして、数字とアルファベット (大文字、小文字の区別なし) からなる基数の数字列を続けます。例 : ‘`2'1010`’、‘`16'0D0a`’ 。基数 1 の整数の値は、数字列中の 1 の数です。例えば、‘`1'10110`’ は 3 を表します。
- 文字コード表記: 任意指定のマイナス符号のあとに、数字の 0、アポストロフィと文字を続けます。例:‘`0'a`’ は、アルファベットの小文字の `a` の文字コードを表します。

上記の定数表記は、KL1 プログラムと、Prolog 風の入出力インタフェースで読み込まれる KL1 データの双方で使用できます (see Section 3.6.2 [Prolog 風のインタフェースを用いた入出力], page 40)。

以下に示す内容も PIM マシン上の PIMOS システムとの互換性のために、 KL1 プログラムは許しています。

- 基数表記: 任意指定のマイナス符号のあとに、基数を規定する 10 進数字の並び (1〜36) 、シャープ記号 (#) 、そして、数字とアルファベット (大文字、小文字の区別なし) からなる基数の数字列を二重引用符で囲んで続けます。例:‘`2#"1010"`’、‘`16#" 0D0a"`’ 。
- 文字コード表記: 任意指定のマイナス符号のあとに、シャープ記号 (#) と、二重引用符で囲んだ 1 個の文字を続けます。例:‘`#"a"`’ は、アルファベットの小文字の `a` の文字コードを表します。

3.2.2.2 整数演算

:= -*Var* +*Expr* ガード述語 on builtin
:= -*Var* +*Expr* ボディ述語 on builtin

整数式 *Expr* の値を計算し、*Var* と単一化します。次の演算子が利用可能です。

X `+` *Y*	加算
`+` *X*	演算せず *X* が結果となります
X `-` *Y*	減算
`-` *X*	符号の反転
X `*` *Y*	乗算
X `/` *Y*	整数除算

X `mod` *Y* モジュロ

`\(`*X*`)` ビットごとの補数

X `/\` *Y* ビットごとの論理積

X `\/` *Y* ビットごとの論理和

X `xor` *Y* ビットごとの排他的論理和

X `<<` *Y* 左論理シフト

X `>>` *Y* 右論理シフト

`int(`*X*`)` 浮動小数点から整数への変換　*X* は浮動小数点式で、その値を整数値に丸めます (see Section 3.2.3.3 [浮動小数点演算], page 22)。

算術オーバフローは無視します。つまり、すべての演算は、使用する C コンパイラに依存して、モジュロ 2^28 か モジュロ 2^60 を行います。32 ビットの `long int` を持つ C コンパイラは、28 ビットの KLIC 整数を提供し、64 ビットの `long int` を持つ C コンパイラは、60 ビットの KLIC 整数を提供します。

この述語は、節のガードとボディの両方で利用可能です。

式中に、具体化されていないオペランドがある場合、すべてのオペランドが具体化されるまで計算は中断します。

式中のいくつかのオペランドに対して、再帰的に式を指定できます。ただし、プログラム中に変数として書くオペランドは、‘3+5’ のような合成項に具体化しては なりません。具体化するのは整数だけにしてください。整数以外に具体化すると、型不一致のエラーが発生します。

3.2.2.3 整数比較

整数データの比較は、ここで説明する述語を用いて行うことができます。より一般的な比較述語も提供しています (see Section 3.1.3 [比較とハッシュ], page 16)。ただし、ここで説明する述語とメソッドは、オペランドが整数であることが明らかな場合に使用すると、より効率的です。

> +*X* +*Y* ガード述語 on builtin

>= +*X* +*Y* ガード述語 on builtin

=:= +*X* +*Y* ガード術語 on builtin

=\= +*X* +*Y* ガード述語 on builtin

=< +*X* +*Y* ガード述語 on builtin

< +*X* +*Y* ガード述語 on builtin

これらは 2 つの整数引数の比較演算を行います。値の同値関係のチェックには `=:=` と `=\=` を使用してください。比較の両側には算術式が指定できます。`:=` で用いたものと同じ演算子群が使えます。

3.2.3 浮動小数点数

64 ビット精度の浮動小数点数を、ジェネリック・オブジェクトとして提供します。次のメソッドと述語は、与えられたデータが浮動小数点数か否かを判定します。

float +*X* ガード・メソッド on float

float +*X* ガード述語 on builtin

X が浮動小数点数か否かをテストします。

3.2.3.1 浮動小数点数の表記

浮動小数点数は、次のような定数表記の構文を持っています。

符号 整数 . 小数 e 符号 指数

ここで、整数、小数、および指数は、10 進数の並びです。符号は、 + 、 - または 指定なし (この場合、 + を仮定する) となります。指数部は、文字の e、符号、および指数で構成し、すべてを省略することも可能です。

浮動小数点定数の例を以下に示します。

```
3.14159 -6.02e23  1234.5678e-25
```

3.2.3.2 新しい浮動小数点数の生成

新しい浮動小数点数は、次のように生成できます。Section 3.2.3.3 [浮動小数点演算], page 22 で説明する浮動小数点の演算用述語も、算術演算の結果として浮動小数点数を生成します。

new *-Float +Init* float on オブジェクト生成

新しい浮動小数点数を生成し、 *Float* で単一化します。引数の *Init* には、浮動小数点数の値として整数を指定する必要があります。例えば、 ‘`generic:new(float, F, 3)`’ では `F` を `3.0` で単一化します。

3.2.3.3 浮動小数点演算

$:= *-Var +Expr* ボディ述語 on builtin

浮動小数点式の値 *Expr* を計算して *Var* で単一化します。次の演算子が利用可能です。

X `+` *Y* 加算

X `-` *Y* 減算

X `*` *Y* 乗算

X `/` *Y* 除算

`pow(`*X*`, `*Y*`)`
X の *Y* 乗

`sin(`*X*`), cos(`*X*`), tan(`*X*`)`
X の三角関数

`asin(`*X*`), acos(`*X*`), atan(`*X*`)`
X の逆三角関数

`sinh(`*X*`), cosh(`*X*`), tanh(`*X*`)`
X の双曲線関数

`exp(`*X*`)` 指数関数

`log(`*X*`)` 自然対数

`sqrt(`*X*`)` 平方根

`ceil(`*X*`)` 切り上げ関数 (正の無限方向への切り上げ)

`floor(`X`)` 切り捨て関数 (負の無限方向への切り捨て)

`float(`X`)` 整数の浮動小数点数への変換 X は整数式で、結果は浮動小数点数に変換します (see Section 3.2.2.2 [整数演算], page 20)。

`+`X X が結果となります。

`-`X 符号の反転

この述語は、節のガードとボディの両方で利用可能です。

式中に、具体化されていないオペランドがある場合、すべてのオペランドが具体化されるまで計算は中断します。

式中のいくつかのオペランドに対して、再帰的な式を指定できます。ただし、プログラム中で変数として書くオペランドは、'`3.0 + 5.0`' のような合成項に具体化しては なりません。具体化するのは浮動小数点数だけにしてください。浮動小数点数以外に具体化すると、型不一致のエラーが発生します。

上記の演算は、浮動小数点数のジェネリック・メソッドとしても提供しています。

add `+`X `+`Y `-`R ボディ・メソッド on float
subtract `+`X `+`Y `-`R ボディ・メソッド on float
multiply `+`X `+`Y `-`R ボディ・メソッド on float
divide `+`X `+`Y `-`R ボディ・メソッド on float
pow `+`X `+`Y `-`R ボディ・メソッド on float
sin `+`X `-`R ボディ・メソッド on float
cos `+`X `-`R ボディ・メソッド on float
tan `+`X `-`R ボディ・メソッド on float
asin `+`X `-`R ボディ・メソッド on float
acos `+`X `-`R ボディ・メソッド on float
atan `+`X `-`R ボディ・メソッド on float
sinh `+`X `-`R ボディ・メソッド on float
cosn `+`X `-`R ボディ・メソッド on float
tanh `+`X `-`R ボディ・メソッド on float
exp `+`X `-`R ボディ・メソッド on float
log `+`X `-`R ボディ・メソッド on float
sqrt `+`X `-`R ボディ・メソッド on float
ceil `+`X `-`R ボディ・メソッド on float
floor `+`X `-`R ボディ・メソッド on float

これらのメソッドは、メソッド名で規定した算術演算を実行します。オペランドを指定すると、その結果を R へ返します。

3.2.3.4 浮動小数点比較

浮動小数点データの比較は、ここで説明する述語を用いて行えます。より一般的な比較述語も提供しています (see Section 3.1.3 [比較とハッシュ], page 16)。ただし、ここで説明する述語とメソッドは、オペランドが浮動小数点数であることが明らかな場合に使用すると、より効率的です。

$> *+X +Y* ガード述語 on builtin
$>= *+X +Y* ガード述語 on builtin
$=:= *+X +Y* ガード述語 on builtin
$=\= *+X +Y* ガード述語 on builtin
$=< *+X +Y* ガード述語 on builtin
$< *+X +Y* ガード述語 on builtin

これらの述語は、2 つの浮動小数点引数の比較演算を行います。値の同値関係のチェックには (浮動小数点数の場合は、たいして意味のあることではありませんが)、 `=:=` と `=\=` を使用してください。比較の両側には浮動小数点式を指定できます。`$:=` で用いたものと同じ演算子群が使えます。

バグに注意 現在の版 (2.001) では、これらの述語に対して演算子付き式の使用は避けてください。単純な変数と定数だけを使用してください。

浮動小数点数の比較は、次のメソッドでも行えます。

less_than *+X +Y* ガード・メソッド on float
not_greater_than *+X +Y* ガード・メソッド on float
not_less_than *+X +Y* ガード・メソッド on float
greater_than *+X +Y* ガード・メソッド on float
equal *+X +Y* ガード・メソッド on float
not_equal *+X +Y* ガード・メソッド on float

これらの方法では、X が Y より小さいか否か等をテストします。

3.3 構造型データ

構造型データ・オブジェクトは、0 個以上の要素で構成します。

3.3.1 ファンクタ構造

ファンクタ構造は、与えられた名前と 1 個以上の任意の型の要素を持つ構造です。ファンクタは、あらかじめ大きさが分かっているデータ構造を表すのに向いています。ファンクタは、C 風言語のレコード構造に対応します。

3.3.1.1 ファンクタの表記

ファンクタ定数は、主ファンクタ名、左括弧、コンマで区切られた要素、および最後の右括弧で書くことができます。ファンクタ名は記号アトムと同じ構文です。主ファンクタ名とそれに続く左括弧を、空白文字や区切り記号等で分けてはなりません。要素には、変数やファンクタ自身を含む任意の型が指定できます。

例：

```
f(a, 3)   'a recursive functor structure'(X, 'child functor'(Y))
```

3.3.1.2 ファンクタの操作

ファンクタ構造を操作する述語は、以下に示すように組込み述語として提供したり、`functor_table` モジュール中に提供したりしています。

現在の実装では、次に示すボディ組込み述語のすべてが、`functor_table` モジュールの述語に展開したマクロ形式で実装しています。この実装方法は、将来のリリースで変更する可能性があります。

functor *+X −Functor −Arity* ガード述語 on builtin
functor *+X −Functor −Arity* ボディ述語 on builtin

X は、主ファンクタ名が *Functor* で、引数個数が *Arity* のファンクタです。上記の述語は、主ファンクタ名とその引数個数、またはそれらのどちらか一方を得るために使用します。ガード述語として呼び出された場合には、*X* が名前 *Functor* と引数個数 *Arity* を持つこと、またはそれらのどちらか一方を持つこと、のテストにも使用できます。アトム・データ、文字列、ベクタなどの、ファンクタ構造でない具体化データは、引数個数が 0 となり、自分自身が主ファンクタ名になります。リスト構造は、ファンクタ `./2` で構成しているので注意してください。

この述語では、新しいファンクタは生成できません。

arg *+Pos +Term −Arg* ガード述語 on builtin
arg *+Pos +Term −Arg* ボディ述語 on builtin

Term の *Pos* 番目の引数は *Arg* です。引数は 1 から番号付けしています。ガード述語として呼び出された場合、 *Pos* が範囲外なら単に失敗するだけです。この述語をファンクタ構造以外のデータ構造に使用すると、それらのデータ構造は引数を持たないので、常に失敗します。

new_functor *−Functor +Atom +Arity* ボディ述語 on builtin

主ファンクタ名が *Atom* で引数個数が *Arity* のファンクタ構造を、 *Functor* へ返します。生成したファンクタの引数は、整数 `0` で初期化しています。

setarg *+Pos +Fnct ?NewE −NewFnct* ボディ述語 on builtin
setarg *+Pos +Fnct ?OldE ?NewE −NewFnct* ボディ述語 on builtin

Pos 番目の引数だけが *Fnct* と異なる新しいファンクタ構造を生成して、 *NewFnct* へ返します。*NewFnct* の *Pos* 番目の要素は *NewE* になります。5 個の引数を持つ述語の場合、*Fnct* の *Pos* 番目の引数を、 *OldE* へ返します。

=.. *−NewFnct +List* functor_table on 述語

新しいファンクタを生成して、*NewFnct* へ返します。主ファンクタ名は、*List* の最初の要素に記号アトムで指定しなければなりません。引数は、*List* の残りの部分に指定します。*List* の要素が 1 個だけの場合、その要素を *NewFnct* へ返します。

この述語は、ファンクタ構造をリストに分解するためには使えません。

3.3.2 リスト

リストは、任意の長さのデータ・オブジェクトの並びです。KL1 では、リスト構造は、名前 `.` と 2 個の引数を持つファンクタ構造、 `./2` で構成します。リスト構造は、これらコンス・セルとも呼ばれるファンクタ構造で構成します。

コンス・セルの 1 番目の要素は、時にはセルの *car* とも呼ばれますが、リストの 1 番目の要素を表します。セルの *cdr* である 2 番目の要素は、リストの残りの部分を表します。リストの終端は、その cdr が記号アトム `[]` であることで示します。

与えられた引数がリストか否かは、次のガード述語でテストできます。

list *+X* ガード述語 on builtin

X がコンス・セルか否かをテストします。この述語は、X が空リスト `[]` であると、X の名前とは関係なく、失敗するので注意してください。

徐々に具体化されるリスト構造は、メッセージ・ストリームとして使用すると便利です。

3.3.2.1 リストの表記

KL1 のリストは、Lisp のようにコンスのデータ構造を用いて構築します。コンスのデータ構造は、実際にはファンクタ構造 `./2` です。

リストの基本的な表記は、[*Car*|*Cdr*] です。すなわち、リストの最初の要素が *Car* で、末尾が *Cdr* で構成します。これは、.(*Car*, *Cdr*) とまったく同じ意味になります。空リストは、アトム `[]` で表します。

Cdr が空の場合、つまり、リストが 1 個の要素 *Car* だけで構成する場合、リストは .(*Car*, []) 、または [*Car*|[]] と表記します。後者の場合、 リストの末尾の `|[]` は省略できるので、 [*Car*] と書くこともできます。つまり、car が *Car* で、cdr が [*Cadr*, ...] であるリストは、[*Car*|[*Cadr*, ...]] と書けます。これは、[*Car*, *Cadr*, ...] のように省略形でも書けます。例えば、4 個の要素、`first`、`second`、`third`、`fourth` からなるリストは、`[first, second, third, fourth]` と書けます。

4 個以上の要素からなるリストで、最初の 4 個の要素が `first`、`second`、`third`、`fourth` の場合、`[first, second, third, fourth | Rest]` と書けます。ここで、変数 `Rest` は、5 番目の要素で始まるリストに対応します。リスト全体が 4 個の要素だけの場合には、`Rest` は空リストに対応します。

Edinburgh Prolog とは異なり、文字の並びの`,..`は `|`の代わりには使用できないので注意してください。

3.3.2.2 メッセージ・ストリームの操作

ストリーム・マージャは、メッセージのリストとして表現される複数のメッセージ・ストリームを入力として受け取り、それらすべての入力ストリームのメッセージを、リストとして表現する単一の出力ストリームに渡すプロセスです。

出力は、入力中のすべてのメッセージを複製したものから成り立ちます。2 つのメッセージに対して、どれか 1 つの入力ストリームで順序付けを行うと、それらのメッセージの順序は出力中でも変わることはありません。出力中のメッセージが複数の入力ストリームから渡される場合、メッセージの出力順序は予測できません。その出力順序は、同じプログラムでも実行ごとに異なる可能性があります。このように、マージャは非決定的な動作をします。

例えば、2 つの入力ストリーム、`[1, 2, 3]` と `[a, b, c]` が存在する場合、出力は、`[1, 2, a, b, 3, c]` や `[1, a, 2, b, c, 3]` にはなりますが、決して `[1, a, 3, b, c, 2]` になることはありません。

KL1 では、2 入力のストリーム・マージャを、次のように定義できます。

```
merge([M|In1], In2, Out) :- Out=[M|OutT],  merge(In1, In2, OutT).
merge(In1, [M|In2], Out) :- Out=[M|OutT],  merge(In1, In2, OutT).
merge([], In2, Out) :- Out=In2.
merge(In1, [], Out) :- Out=In1.
```

- 上記のマージャの定義において、1 番目の節（clause）は、最初の入力ストリームから、1 つのメッセージを出力ストリームに転送します。最初の入力ストリームは最初の引数であり、出力ストリームは述語の 3 番目の引数です。続いて、繰返し実行のために再帰的に述語 `merge/3` を呼び出します。
- 2 番目の節は、2 番目の入力ストリームに対して、同じ処理を行います。

- 3 番目の節は、最初の入力ストリーム中にメッセージが存在しない場合に使用します。この場合、2 番目の入力ストリームを直接、出力に接続します。最初の入力ストリームから転送するメッセージはないので、マージの結果は 2 番目の入力ストリームと同じになります。
- 4 番目の節は、2 番目の入力ストリーム中にメッセージが存在しない場合に使用します。

メッセージを同時に両方の入力ストリームから受け取る場合、1 番目か 2 番目のどちらか一方の節が任意に選ばれます。このことがマージャの非決定性の原因です。

2 入力のマージャは、KL1 では容易に定義できますが、任意の多入力ストリームのマージャを定義することは、容易ではありません。新しい入力ストリームを動的に追加することも望まれますが、さらに難しくなります。また、マージャは KL1 プログラムで非常によく使用されるので、効率的である必要があります。したがって、KLIC システムでは、マージャを標準的な機能として提供します。

新しいマージャは、以下に示す疑似述語によって生成します。

new *?Input ?Output* オブジェクト生成 on merge
単一の入力ストリームを持つ新しいマージャを生成します。入力ストリームは *Input* で、出力ストリームは *Output* になります。

上記の疑似述語で生成したマージャ・プロセスは、実際には、生成直後はマージを開始しません。単に *Input* からのメッセージを *Output* に、順序を変えずに転送するだけです。

新しい入力ストリームをマージャに追加するには、入力をベクタで単一化してください。そのベクタの要素が新たな入力ストリームになります。例えば、バイナリのマージャが必要な場合、次のようにしてください。

`generic:new(merge,` *Input*`,` *Output*`),`
Input `= {`*In1*`,` *In2*`}`

これは、次の処理と同じ意味になります。

`generic:new(merge, {`*In1*`,` *In2*`},` *Output*`)`

上記処理後、マージャは 2 個の入力ストリーム *In1* と *In2* からのメッセージを、出力ストリーム *Output* へマージします。

マージャへの入力ストリームの追加は、生成直後だけでなく、要求に応じて任意に行えます。例えば、2 個以上の入力ストリームを追加する例を、次に示します。

In2 `= {`*In2A*`,` *In2B*`,` *In2C*`}`

この結果、マージャは 4 個の入力ストリーム *In1*、*In2A*、*In2B*、*In2C* を持つことになります。

入力ストリームの 1 つが必要なくなった場合、その入力ストリームをアトム `[]` で単一化するだけで、簡単に閉じることができます。

入力ストリームに単一化されるベクタのサイズは、任意に変更できます。ベクタが 1 個の要素しか持っていない場合、入力ストリームの数は変更されません。ベクタが要素を持っていない場合、ベクタを単一化すると、ストリームを閉じることになります。

すべての入力ストリームを閉じたとき、出力リストの末尾を `[]`と単一化するので、出力ストリームも閉じます。

マージャを使用する際のヒントを、次に示します。

- マージするメッセージには、未束縛変数を含むデータ構造を許します。そのようなメッセージは、不完全メッセージ とも呼びます。不完全メッセージは、クライアント/サーバのプロセス構造を構築する場合に向いています。メッセージ中の変数に値を与えると、サーバからクライアントへの返信に使用できます。

- 逐次実装におけるマージは、決定的に見えるかも知れません。しかし、並列実装におけるマージは、非決定的になりますので、決して頼らないでください。

3.3.3 ベクタ

ベクタは、固定長の 1 次元配列の KL1 データです。ベクタの長さは、生成の際に決定します。要素には、任意の KL1 データを許し、データ構造を生成した時点で、未定義の状態でいることも許します。

要素は 0 から始まる整数によって、インデックス付けします。例えば、3 個の要素を持つベクタは、0、1、2 と番号が付いた要素になります。

3.3.3.1 ベクタの表記

ベクタは、1 対の中括弧 (`{}`) の中で、要素の並びをコンマで区切ることで表します。

```
{ 1, a, f(b), X }
```

空ベクタ (要素のないベクタ) は 1 対の中括弧だけで表します。

```
{}
```

中括弧は、Edinburgh Prolog とは完全に異なった意味で使用するので、注意してください。Edinburgh Prolog では、`{}`はアトムを意味し、`{...}` はファンクタ構造 `{}((...))`を意味します。

3.3.3.2 ベクタの生成

前項で示した表記に加えて、ベクタをプログラムの実行中に動的に生成できます。次に示す述語は、新しいベクタを生成するために使えます。

new -*Vector* +*Init* オブジェクト生成 on vector
new_vector -*Vector* +*Init* ボディ述語 on builtin

新しいベクタを生成して、*Vector* へ返します。

引数 *Init* が整数の場合、要素数を指定したことになります。この場合、要素を整数 `0` で初期化します。例えば、'`generic: new(vector, V, 2)`' はベクタ '`{0, 0}`' を生成して、`V` へ単一化します。

引数 *Init* がリストの場合、新しく生成したベクタを、リストの要素によって初期化します。当然、ベクタの要素数はリストの長さと同じになります。例えば、'`generic:new(vector, V, [a, b, c])`' は ベクタ `{a, b, c }`を生成して、`V` へ単一化します。

3.3.3.3 ベクタの述語

vector +*Vector* -*Length* ガード・メソッド on vector
size +*Vector* -*Length* ボディ・メソッド on vector
vector +*Vector* -*Length* ガード述語 on builtin

(ガード述語で呼び出された場合、)*Vector* がベクタ・オブジェクトか否かをテストします。要素数を *Length* へ返します。

element +*Vector* +*Index* −*Element* ガード・メソッド on vector
element +*Vector* +*Index* −*Element* ボディ・メソッド on vector
vector_element +*Vector* +*Index* −*Element* ガード述語 on builtin
vector_element +*Vector* +*Index* −*Element* ボディ述語 on builtin
ベクタ *Vector* の *Index* 番目の要素を、*Element* と単一化します。インデックスは、ゼロから始まります。

set_element +*Original* +*Index* ?*NewElement* −*New* ボディ・メソッド on vector
set_vector_element +*Original* +*Index* ?*NewElement* −*New* ボディ述語 on builtin
新しいベクタを *New* と単一化します。*Index* 番目の要素を更新して、*NewElement* にする他は、 *Original* と同じ要素になります。オリジナルのベクタには影響しません。インデックスは、ゼロから始まります。

set_element +*Original* +*Index* ?*Element* ?*NewElement* −*New* ボディ・メソッド on vector
set_vector_element +*Original* +*Index* ?*Element* ?*NewElement* −*New* ボディ述語 on builtin
新しいベクタを *New* と単一化します。*Index* 番目の要素を更新して、 *NewElement* にする他は、 *Original* と同じ要素になります。オリジナルのベクタには影響しません。インデックスは、ゼロから始まります。*Index* 番目のオリジナルの要素を *Element* へ返します。

split +*Original* +*At* −*Lower* −*Upper* ボディ・メソッド on vector
ベクタ *Original* を *At* 番目で分割し、2 つのベクタを *Lower* と *Upper* に単一化します。*At* は、オリジナルのベクタのサイズ以下で、かつゼロ以上の整数です。*Lower* は、0 番目以上 *At*−1番目以下の要素で構成します。*Upper* は、*At* 番目以上の要素で構成します。

join +*Lower* +*Upper* −*Joined* ボディ・メソッド on vector
Lower と *Upper* の 2 つのベクタを結合して、新しいベクタ *Joined* にします。

KLIC では、既存のベクタと 1 要素だけが異なる新たなベクタの生成は、ベクタのサイズとは関係なく、時間と領域がともに一定のオーバヘッドで済むマルチバージョン配列表現を用いて実装しています。

3.3.4 文字列

文字列は、限定された範囲の整数の 1 次元配列です。現在の版では、0 〜 255 個の要素を持つ 8 ビット要素の文字列だけを提供しています。それらは、文字の列を表す場合に向いています。異なるサイズの要素からなる文字列を、将来計画しています。

Edinburgh Prolog とは異なり、文字列は、文字コードのリスト用の表記規約ではありません。文字列は、それ自身のデータ型を持ちます。

3.3.4.1 文字列の表記

文字列定数は次のように、 1 対の二重引用符 (") で囲まれた文字の並びで表します。

```
"A string of the characters written here"
```

次のエスケープ・シーケンスは、(ANSI C のように) 二重引用符、バックスラッシュ、制御コードなどを文字列の要素として指定するために使います。

`\a` ビープ音

`\b` バックスペース

`\t` タブ

`\n` 改行

`\v` 垂直タブ

`\f` 改ページ

`\r` 復帰

`\'` 単一引用符

`\"` 二重引用符

`\?` 疑問符

`\\` バックスラッシュ　2 個の連続するバックスラッシュで文字列中の 1 個のバックスラッシュを指定する

`\`*ooo* 8 進数 *ooo* で指定するコード最大 3 桁の 8 進数が指定できる

`\x`*hh* 16 進数 *hh* で指定するコード任意桁の 16 進数が指定できる

`\`⟨NEWLINE⟩
直後に改行コードの付いたバックスラッシュを無視する文字列中では、この並びには文字が無くなる

例:

```
"The character \'\"\' (doublequote)"
```

上記の例では、次の文字を含む文字列として解釈します。

```
The character '"' (doublequote)
```

文字列には、改行や二重引用符を直接含んではなりません。文字列内に改行を含む標準的な方法は、行を '`\n\`' で終了させることです。この方法では、改行コードを、'`\n`' で挿入し、2 番目の '`\`' に続くソース・コード中の実際の改行を、無視します。

Edinburgh Prolog とは異なり、文字列は文字コードのリストではありません。

3.3.4.2 文字列の生成

上記の文字列定数に加えて、文字列を実行中に動的に生成できます。次の述語が新しい文字列を生成するために使えます。

new *-String +Init +ElemSize* オブジェクト生成 on string
new_string *-String +Init +ElemSize* ボディ述語 on builtin
新しい文字列を生成して、*String* と単一化します。最後の引数 *ElemSize* には要素のビット幅を指定します。現在の版では、8 ビット文字列しか利用できないので、これは 8 になります。

引数 *Init* が整数の場合、要素数が指定されたことになります。この場合、要素を整数 0(ヌル・コード) で初期化します。例えば、‘`generic:new(string, S, 3, 8)`’ は ‘`"\0\0\0"`’ を生成します。

引数 *Init* が整数のリストの場合、新しく生成された文字列を、リストの要素によって初期化します。当然、文字列の要素数はリストの長さと同じになります。この場合、リストの要素は、与えられたビット幅に適合する値、つまり 8 ビット文字列で構成する場合、0 〜 255 の値になります。例えば、‘`generic:new(string, S, [0'a, 0'b, 0'c], 8)`’ は ‘`"abc"`’ を生成します。

3.3.4.3 文字列の述語

string +*String* −*Length* −*ElemSize* ガード・メソッド on string
string +*String* −*Length* −*ElemSize* ボディ・メソッド on string
string +*String* −*Length* −*ElemSize* ガード述語 on builtin
(ガード述語で呼び出された場合)、*String* が文字列オブジェクトか否かをテストします。*String* の要素数を *Length* へ、要素サイズ (現在の版では常に 8) を *ElemSize* へ返します。

size +*String* −*Length* ボディ・メソッド on string
String の要素数を *Length* へ返します。

element_size +*String* −*ElemSize* ボディ・メソッド on string
String の要素サイズを *ElemSize* へ返します。

element +*String* +*Index* −*Element* ガード・メソッド on string
element +*String* +*Index* −*Element* ボディ・メソッド on string
string_element +*String* +*Index* −*Element* ガード述語 on builtin
string_element +*String* +*Index* −*Element* ボディ述語 on builtin
文字列 *String* の *Index* 番目の要素を *Element* と単一化します。インデックスは、ゼロから始まります。

less_than +*String1* +*String2* ガード・メソッド on string
string_less_than +*String1* +*String2* ガード述語 on builtin
辞書式順序で、*String1* が *String2* より小さい場合だけ成功します。

not_less_than +*String1* +*String2* ガード・メソッド on string
string_not_less_than +*String1* +*String2* ガード述語 on builtin
辞書式順序で、*String1* が *String2* 以上の場合だけ成功します。

string +*String* −*Length* −*ElemSize* ガード・メソッド on string
string +*String* −*Length* −*ElemSize* ガード述語 on builtin
String の要素数を *Length* へ返し、要素サイズ (常に 8) を *ElemSize* へ返します。

set_element +*Original* +*Index* ?*Element* +*New* ボディ・メソッド on string
set_string_element +*Original* +*Index* ?*Element* +*New* ボディ述語 on builtin
新しい文字列を *New* で単一化します。新しい文字列の、*Index* 番目の要素を更新して、*Element* にする他は、*Original* と同じ要素になります。オリジナルの文字列には影響しません。インデックスは、ゼロから始まります。

split *+Original +At -Lower -Upper* ボディ・メソッド on string
文字列 *Original* を、 *At* 番目で分割し、 2 個の文字列を *Lower* と *Upper* に単一化します。*At* は、オリジナルの文字列のサイズ以下で、かつゼロ以上の整数です。*Lower* は、0 番目以上 *At*-1 番目以下の要素で構成します。*Upper* は、*At* 番目以上の要素で構成します。

join *+Lower +Upper -Joined* ボディ・メソッド on string
Lower と *Upper* の 2 つの文字列を結合して、新しい文字列 *Joined* にします。

search_character *+String +Start +End +Char -Where* ボディ・メソッド on string
search_character *+String +Start +End +Char -Where* ボディ述語 on builtin
文字 *Char* を *String* 中で探索します。探索は、位置 *Start* から開始し、*End* の前で終わります。探索する文字が見つかった場合、そのインデックスを、*Where* と単一化します。見つからなかった場合、*Where* を `-1` と単一化します。インデックスは、ゼロから始まります。

KLIC では、既存の文字列と 1 要素だけが異なる新たな文字列の生成は、文字列のサイズとは関係なく、時間と領域がともに一定のオーバヘッドで済むマルチバージョン配列表現を用いて実装しています。

3.4 プログラム・コードのデータとしての扱い

KLIC では、データ・オブジェクトとして、実行コードの高次な操作が可能です。プログラム・モジュールは、モジュール・データ・オブジェクトとして扱い、個々の述語は、述語データ・オブジェクトとして扱います。

3.4.1 モジュール

プログラム・モジュールは、`module` 型のジェネリック・データ・オブジェクトによってデータとして扱います。

new *-Module +ModuleName* オブジェクト生成 on module
記号アトムの *ModuleName* で指定されたプログラム・モジュールに対応する、新しいオブジェクト *Module* を生成します。指定されたモジュールが定義されていない場合、記号アトム自身を *Module* へ返します。オブジェクト生成ゴールのフォーマットについては、See Section 2.6.1 [ジェネリック・オブジェクトの生成], page 9。

module *+Module* ガード・メソッド on module
Module がモジュール・オブジェクトか否かをテストします。

name *+Module -ModuleName* ボディ・メソッド on module
Module のモジュール名を、記号アトムとして *ModuleName* へ返します。

3.4.2 述語

プログラムの述語は、述語 (`predicate`) 型のジェネリック・データ・オブジェクトによってデータとして扱います。

述語型データは、定数として表わすか、実行時に動的に生成するかのどちらかです。ホスト・システムの機能の制約のため、ホスト・システムによっては、動的な生成をサポートしない場合があります。

述語定数の構文は次のとおりです。

`predicate#(`*module*`:`*predicate*`/`*arity*`)`

ここで、*module* と *predicate* がモジュールと述語の名前アトムであり、*arity* が整数 (述語の引数個数) になります。例えば、

```
predicate#(main:main/0)  predicate#(quicksort:partition/4)
```

これらは、プログラムで有効な述語定数です。

述語定数は、KLIC のパーサでなく、KLIC のコンパイラが認識するので注意してください (see Section 3.6.2 [Prolog 風のインタフェースを用いた入出力], page 40)。したがって、Prolog 風の I/O ストリームを用いて単に読み込むだけの場合、上記の表記は、通常のデータ構造を意味します。

new -*Predicate* +*Module* +*PredName* +*Arity* オブジェクト生成 on predicate
Module(モジュール・オブジェクト)、*PredName*(記号アトム)、*Arity*(整数) で指定する述語に対応する新しいオブジェクト *Predicate* を生成します。オブジェクト生成ゴールのフォーマットについては、See Section 2.6.1 [ジェネリック・オブジェクトの生成], page 9。

predicate +*Predicate* ガード・メソッド on predicate
Predicate が述語オブジェクトか否かをテストします。

arity +*Predicate* -*Arity* ガード・メソッド on predicate
arity +*Predicate* -*Arity* ボディ・メソッド on predicate
述語 *Predicate* の引数個数を *Arity* へ返します。

apply +*Predicate* +*ArgVec* ボディ・メソッド on predicate
述語オブジェクトの *Predicate* で指定した述語を *ArgVec* で指定した引数で呼び出します。*ArgVec* は、*Predicate* に渡す引数のベクタです。したがって、ベクタのサイズは、述語の引数個数と一致させる必要があります。

call +*Predicate* +*Args...* ボディ・メソッド on predicate
述語オブジェクトの *Predicate* で指定した述語を *Args...* で指定した引数で呼び出します。引数の数は、述語の引数個数と一致させる必要があります。

module +*Predicate* -*Module* ボディ・メソッド on predicate
Predicate が属するプログラム・モジュールを、モジュール・データ・オブジェクトとして *Module* へ返します。

name +*Predicate* -*Name* ボディ・メソッド on predicate
述語 *Predicate* の名前を、記号アトムとして *Name* へ返します。

3.5 Unix インタフェース

`unix` という名前のモジュールによって、ホスト・オペレーティング・システム (代表例として `Unix`) の機能を KL1 プログラムから利用できるようになります。

機能のほとんどは、`unix` モジュールが提供する述語 `unix/1` で獲得したストリームに対するメッセージとして利用可能です。いくつかの機能は、述語として提供しています。

3.5.1 Unix インタフェース・ストリームの獲得

ユーザは、unix モジュールをメッセージ・ストリームを介して利用します。そのストリームは、次の述語を呼び出すことによって得られます。

unix *?Stream* predicate on unix
Unix インタフェースに対応するメッセージ・ストリームを *Stream* へ返します。

述語の呼出し順序は、保証できないため、Unix インタフェース機能のほとんどは、述語として提供しません。仮に、Unix インタフェースが述語として提供されるとしたら、次に示す例の `ls` の結果は、実行順序に依存することになります。

```
unix:cd("a", 0),
unix:cd("b", 0),
unix:system("ls", 0)
```

つまり、実行順序によっては、ディレクトリ `a` を表示したり、 `b` を表示したり、または、2 つの `cd` を実行する前にどこかのディレクトリを表示したりするかもしれません。一方、次に示す例の場合は、実行順序によってではなく、リスト中に並んだ要素の順序に従って、2 つの `cd` と、`ls` を実行します。

```
unix:unix([cd("a", 0),
           cd("b", 0),
           system("ls", 0)])
```

並列実装では、KLIC は複数プロセスを構成します。すべてのメッセージは、unix ストリームを獲得するプロセスで処理します。例えば、`cd(Path)` メッセージは、単一プロセスのワーク・ディレクトリと、他に何も存在しないワーク・ディレクトリを変更します。

複数のメッセージ・ストリームを獲得した場合、異なるストリームへ送られたメッセージ間の同期はとりません。

3.5.2 入出力用ストリームのオープン

Unix ストリームに、次のメッセージを送って、Unix I/O ストリームをオープンします。入出力を実行するために (Unix ストリーム自身ではなく) Unix I/O ストリームに送るメッセージについては、別の場所で説明します (See Section 3.6.1 [C 風のインタフェースを用いた入出力], page 38, Section 3.6.2 [Prolog 風のインタフェースを用いた入出力], page 40)。

次に示す例は、「hello world」と出力する KLIC プログラムです。

```
main :- unix:unix([stdout(R)]), check_and_write(R).

check_and_write(normal(R)) :- R = [fwrite("hello world\n")].
```

stdin *-Result* メッセージ on unix ストリーム
stdout *-Result* メッセージ on unix ストリーム
stderr *-Result* メッセージ on unix ストリーム
これらのメッセージによって、プロセスの標準入力、標準出力、および標準エラー・ファイルと関連する個々のストリームをオープンし、*Result* へ `normal(`*String*`)`を返します。

read_open +*Path* -*Result* メッセージ on unix ストリーム
write_open +*Path* -*Result* メッセージ on unix ストリーム
append_open +*Path* -*Result* メッセージ on unix ストリーム
update_open +*Path* -*Result* メッセージ on unix ストリーム

これらのメッセージによって、文字列の *Path* に指定したファイルをオープンし、*Result* へ `normal(`*String*`)`を返します。オープン・モードは、入力、出力、追加、および入力/出力の各モードです。ファイルのオープンに失敗した場合、 `abnormal` を返します。

3.5.3 ソケットの使い方

Unix ストリームに以下のメッセージを送って、Unix とインターネットのプロトコルのソケットを獲得します。SOCK_STREAM 型のソケットだけを提供しています。

connect +*Spec* -*Result* メッセージ on unix ストリーム

ソケットを生成し、Spec で指定されたソケット・アドレスと接続し、*Result* へ `normal(`*String*`)` を返します。*Spec* は、以下に示す形式のいずれかです。

`unix(`*Path*`)`
: パス名 *Path* を持つ Unix ドメインのソケットをオープンします。

`inet(`*HostName*`,` *Port*`)`
: ホスト名 *HostName* とポート番号 *Port* で指定したホストのインターネット・ドメインのソケットをオープンします。

`inet({`*B1*`,` *B2*`,` *B3*`,` *B4*`},` *Port*`)`
: インターネット・ドメインのソケットをオープンし、インターネット・アドレスの *B1* から *B4* で指定したホストをオープンします。

獲得したストリームで、入力と出力の両方のメッセージを処理します。

bind +*Spec* -*Result* メッセージ on unix ストリーム

ソケットを生成し、*Spec* で指定したソケット・アドレスへバインドします。*Spec* の形式は、インターネット・ドメインのソケットに対する *HostName* を省略している他は、`connect` の場合と同じです。`normal(`*Stream*`)` を *Result* へ返します。この *Stream* はバインドされたソケット・ストリームであるため、直接 I/O メッセージを処理しません。I/O メッセージ・ストリームを得るためには、さらに `accept` メッセージを使用します。ソケットが `unix` ドメインの場合、バインドされたソケット・ストリームを閉じ、*Spec* 中の *Path* で指定された名前付きソケットをアンリンクします。

accept -*Result* メッセージ on バウンド・ソケット

ソケットへの接続を受け入れ、*Result* へ `normal(`*Stream*`)` を返します。ここで、*Stream* は入力と出力の両メッセージ用の I/O メッセージ・ストリームです。

ソケットは、非同期 I/O を提供します。すなわち、KLIC システムでは、接続待ちや接続の受け入れ時に他のプロセスを中断することはありません。ソケットに対して、空のバッファから読み込んだり、満杯のバッファに書き出したりしても全体の演算実行は中断しません。そのような入出力操作は、操作が可能になるまで延期します。

制限事項: ソケットへの操作を延期した場合、ソケットに対する残りの操作のすべても、延期した操作が完了するまで延期します。このことは、入力と出力の双方をポーリングする必要がある場合に問題です。この問題は、将来のリリースで解決する予定です。

Linux に関する制限事項: 現在の版では、非同期 I/O は (少なくとも Slackware 1.2.0 の)Linux に対しては機能しません。

3.5.4 ファイルとディレクトリ

unix ストリームに次のメッセージを送り、ファイルとディレクトリを処理します。

cd +*Path* -*Result* メッセージ on unix ストリーム

ワーク・ディレクトリを *Path* に変更します。変更が成功した場合、*Result* へ `0` を返し、それ以外の場合、`-1` を返します。これは、システム・コールの `chdir` に相当します。

unlink +*Path* -*Result* メッセージ on unix ストリーム

Path で指定されたディレクトリの登録を削除します。削除が成功した場合、*Result* へ `0` を返し、それ以外の場合、`-1` を返します。これは、システム・コールの `unlink` に相当します。

mktemp +*Template* -*Filename* メッセージ on unix ストリーム

指定された *Template* からユニークなファイル名を生成し、*Filename* へ返します。これは、C ライブラリ・ルーチンの `mktemp` に相当します。ただし、ライブラリ・ルーチンとは異なり、テンプレートは 6 個の終了文字 X を持つ必要がありません。何らかの理由で、ユニークなファイル名が作成できない場合、*Filename* へ空文字列を返します。

access +*Path* +*Mode* -*Result* メッセージ on unix ストリーム

パス名 *Path* のファイルに、モード *Mode* でアクセス可能か否かをチェックします。その結果を *Result* へ返します。これは、C ライブラリ・ルーチンの `access` に相当します。ファイルがアクセス可能な場合、`0` を返し、それ以外の場合、`-1` を返します。

Mode は、各ビットが以下に示す意味を持つ整数です。*Mode* に 0 を指定した時は、ファイルが存在するか否かだけを調べます。

`4`	読み込み可
`2`	書き出し可
`1`	実行可

chmod +*Path* +*Mode* -*Result* メッセージ on unix ストリーム

パス名 *Path* を持つファイルの許可モードを *Mode* に変更します。これは、システム・コールの `chmod` に相当します。モードの変更が成功した場合、*Result* へ `0` を返し、それ以外の場合、`-1` を返します。*Mode* は、標準的な Unix の許可ビットを持つ整数です。

umask -*OldMask* メッセージ on unix ストリーム
umask -*OldMask* +*NewMask* メッセージ on unix ストリーム

現在のファイルの生成マスクを *OldMask* へ返します。引数が 2 個の場合、ファイル作成マスクに *NewMask* を設定します。これは、システム・コールの `umask` に相当します。

3.5.5 シグナル割込みの処理

Unix シグナルは、 unix ストリームに対して次のようなメッセージを使用して、整数のリストに変更します。

signal_stream *+Signal -Result* メッセージ on unix ストリーム

Signal(整数値) で指定された unix シグナルを捕捉して報告します。引数 *Result* は `normal(`*Stream*`)`になり、指定された種類のシグナルが到着したとき、そのシグナル番号を Stream に送ります。例えば、シグナル 2(BSD と SVR4 での SIGINT) が見つかると、Stream は `[2|`*Rest*`]` になります。その後、到着したシグナルは、*Rest* に報告されます。

制限事項: Unix の制限によって、シグナルを捕捉する前に同じシグナルの発生が 2 回を超えると、シグナルは無視することがあります。

3.5.6 Unix ストリームへの種々のメッセージ

Unix ストリームに次のメッセージを送り、Unix の種々の機能を利用します。

system *+Command -Result* メッセージ on unix ストリーム

新たに生成したサブシェルで、 *Command*(文字列) を実行し、終了コードを *Result* へ返します。これは、システム・コールの `system` に相当します。

getenv *+Name -Value* メッセージ on unix ストリーム

名前 *Name* の環境変数の値を *Value* へ返します。これは、ライブラリ・ルーチンの `getenv` に相当します。そのような環境変数が存在しない場合、整数 `0` を *Value* へ返します。

putenv *+String -Result* メッセージ on unix ストリーム

最初の引数 *String* は、*Name* `=` *Value* の形式にしてください。環境変数 *Name* を *Value* で追加または更新します。これは、ライブラリ・ルーチンの `putenv` に相当します。追加または更新が成功した場合、*Result* へ `0` を返し、それ以外の場合、0 以外の整数値を返します。

kill *+Pid +Sig -Result* メッセージ on unix ストリーム

Pid で指定した 1 つのプロセスまたはプロセス・グループへシグナル *Sig* を送り、送信が成功した場合、*Result* へ `0` を返し、失敗した場合、`-1` を返します。

fork *-Pid* メッセージ on unix ストリーム

現行プロセスのコピーである新しいプロセスを、フォークします。これは、システム・コールの `fork` に相当します。子プロセスの生成が成功した場合、親プロセスの *Pid* へ子プロセスのプロセス ID を返し、子プロセスへ `0` を返します。

fork_with_pipes *-Result* メッセージ on unix ストリーム

パイプを生成して、新しいプロセスをフォークします。新しいプロセスは、現行プロセスのコピーです。親プロセスでは、*Result* を `parent(`*Pid*`,`*In*`,`*Out*`)` で単一化します。ここで、*Pid* は新たに作成されたプロセスのプロセス ID です。新たに生成した子プロセスでは、*Result* を `child(`*In*`,`*Out*`)` で単一化します。

In と *Out* は パイプへの Unix I/O ストリームです。ここで、親の *Out* は出力ストリームであり、子の入力ストリーム *In* に接続し、子の *Out* は親の *In* に接続しています。

3.5.7 述語インタフェース

いくつかの unix インタフェースを、 `unix` モジュール中で定義した述語として提供しています。

argc *-Argc* unix on predicate
KLIC で使用しないコマンド行引数の数を *Argc* へ返します。そのような引数は、`-`で始まらない最初の引数や、コマンド行の`--`の後から始まる引数です。

argv *-ArgList* predicate on unix
KLIC で使用しないコマンド行引数を、文字列のリストとして *ArgList* へ返します。

exit *+ExitCode* predicate on unix
直ちに、終了コードの *ExitCode* でプロセスを終了します。

times *-Utime -Stime -CUtime -CStime* predicate on unix
ミリ秒単位でプロセス・タイムを返します。*Utime* はユーザ・タイムであり、*Stime* はシステム・タイムです。*CUtime* と *CStime* は、それぞれ子プロセス用のものです。
`HZ`(秒当たりのクロック数) が標準的な場所で定義されていない場合、システムは 60 を仮定します。

3.6 入出力

KLIC は 2 種類の入出力操作群を提供しています。1 つは C 風のインタフェース、もう 1 つは Prolog 風のインタフェースです。

C 風の機能は低レベルであり、速度とコード・サイズの両面で高性能です。しかし、プロトタイプとデバッグのフェーズでは、Prolog 風の高レベルなインタフェースの方が、データ構造の入出力が直接可能なので有利になります。

3.6.1 C 風のインタフェースを用いた入出力

この節では、C 風のインタフェースを用いた入出力操作を説明します。

このインタフェースは、ファイル、ソケット、パイプなどへのストリームに対するメッセージとして提供しています。これらのストリームは、Unix ストリームへメッセージを送って獲得します (see Section 3.5.2 [入出力用ストリームのオープン], page 34)。

3.6.1.1 C 風のインタフェースを用いた共通メッセージ

次に示すメッセージは、C 風の入出力用の入力ストリームと出力ストリームの両方に利用できます。

feof *-Result* メッセージ on C 風の I/O
ストリームがファイルの終端の場合、*Result* へ `1` を返し、それ以外の場合、`0`を返します。これは、ライブラリ・ルーチンの `feof` に相当します。

fseek *+Offset +Ptrname -Result* メッセージ on C 風の I/O
オフセットとポインタ名を、それぞれ *Offset* と *Ptrname* に指定してストリームの位置を変更します。*Ptrname* が 0 のときファイルの先頭からの、1 のとき現在位置からの、2 のときファイルの末尾からのオフセットを、符合付き整数で指定します。変更が成功した場合、*Result* へ `0` を返し、それ以外の場合、`-1` を返します。

あまり大きなファイル (long int が 32 ビットのシステムでは 128MB より大) を対象とした場合、整数型の値の範囲による制限のため、このメッセージでは任意位置への移動はできない場合があることに注意して下さい。

ftell *-Result* メッセージ on C 風の I/O
現在の位置 (バイト単位) のオフセットを、 *Result* へ返します。
あまり大きなファイル (long int が 32 ビットのシステムでは 128MB より大) を対象とした場合、整数型の値の範囲による制限のため、得られた位置が誤っている場合があることに注意して下さい。

fclose *-Result* メッセージ on C 風の I/O
ストリームをクローズします。クローズが成功した場合、*Result* へ `0` を返し、それ以外の場合、`-1` を返します。クローズの後、`sync/1` の他はストリームにメッセージを送ってはなりません。

sync *-Result* メッセージ on C 風の I/O
Result へ `0` を返します。先行するすべてのメッセージが、既に処理済みであることを確認するのに有効です。

3.6.1.2 C 風のインタフェースを用いた入力メッセージ

次のメッセージが C 風の入出力用の入力ストリームに利用できます。

getc *-C* メッセージ on C 風の I/O
ストリームから 1 バイトを読み込み、*C* へ返します。ファイルの終端の場合、`-1` を返します。

ungetc *+C* メッセージ on C 風の I/O
1 バイトの *C* を、ストリームにプッシュ・バックします。

fread *+Max -String* メッセージ on C 風の I/O
ストリームから最大 *Max* バイトを読み込み、そのデータを、バイト文字列として *String* へ返します。現在の実装では、4,096 バイトまでしか処理できません。読み込んだ文字列の長さが、指定した最大値 *Max* より小さい場合がありますので、注意してください。この現象は、標準ファイルではファイルの終端で発生し、パイプやソケットではいつでも発生する可能性があります。

linecount *-Count* メッセージ on C 風の I/O
これまでに検出した改行文字の数を、*Count* へ返します。ファイルの第 1 行では、まだ改行文字を検出していないので `0` を返します。行数のカウントが 1 で始まる場合 (通常) は、改行文字の数に 1 を加えることで行数が計算できます。
ただし、`fseek/2` メッセージを使用する場合、この行数は正しく計算されません。

3.6.1.3 C 風のインタフェースを用いた出力メッセージ

次のメッセージが C 風の入出力用の出力ストリームに利用できます。

putc *+C* メッセージ on C 風の I/O
1 バイトの *C* をストリームに書き出します。

Number メッセージ on C 風の I/O

1 バイトの *Number* をストリームに書き出します。これは、`putc(`*Number*`)` に相当します。

fwrite +*String* -*Result* メッセージ on C 風の I/O

バイト文字列 *String* の内容をストリームに書き出します。実際に書き出したバイト数を *Result* へ返します。実際に書き出したバイト数は、*String* の長さより短くなる場合があります。

fwrite +*String* メッセージ on C 風の I/O

バイト文字列の *String* の内容をストリームに書き出します。引数 *Result* を持つ `fwrite` のメッセージと違って、*String* 中のすべてのバイトが出力されるまで待ちます。これは、インターネットのソケットやパイプのような、出力に予測できない時間を必要とするストリーム用には望ましくありません。

fflush -*Result* メッセージ on C 風の I/O

ストリームに残っている出力をフラッシュします。フラッシュが成功した場合、*Result* へ `0` を返し、それ以外の場合、`-1` を返します。

3.6.2 Prolog 風のインタフェースを用いた入出力

演算子順位文法に基づく Prolog 風の項の処理機能を持つ Unix インタフェースのストリームは、以下に示すモジュールの `klicio` 述語によって獲得できます。

KLIC の項の構文は、Edinburgh Prolog によく似ていますが、細かい点で異なっています。詳細については、Section 3.2.1.1 [記号アトムの表記], page 18、Section 3.2.2.1 [整数の表記], page 20、Section 3.2.3.1 [浮動小数点数の表記], page 22、Section 3.3.1.1 [ファンクタの表記], page 24、Section 3.3.2.1 [リストの表記], page 26、Section 3.3.3.1 [ベクタの表記], page 28、Section 3.3.4.1 [文字列の表記], page 29 などを参照してください。

3.6.2.1 Prolog 風の I/O ストリームのオープン

klicio *?Stream* klicio on predicate

Prolog 風の項のインタフェースに対応するメッセージ・ストリームを *Stream* へ返します。獲得したストリームは、 Unix のインタフェース・ストリームのように機能します。つまり、このストリームは、実際に入出力を行うメッセージ・ストリームを獲得するために使用します。このストリームを介して獲得した I/O ストリームは、通常の C 風の I/O メッセージに加えて、この節で説明する Prolog 風の項の 入出力に対するメッセージも受け取ります。

パーシングやアンパーシング用のモジュールは、無視できない大きさなので、Prolog 風の項の入出力を必要としないプログラムに対して、そのモジュールなしで実行できるように、別モジュールとして提供します。

stdin *-Result* メッセージ on klicio ストリーム
stdout *-Result* メッセージ on klicio ストリーム
stderr *-Result* メッセージ on klicio ストリーム
read_open *+Path -Result* メッセージ on klicio ストリーム
write_open *+Path -Result* メッセージ on klicio ストリーム
append_open *+Path -Result* メッセージ on klicio ストリーム
update_open *+Path -Result* メッセージ on klicio ストリーム

これらのメッセージは、Prolog 風の I/O ストリームをオープンします。実際に入出力を実行するために、(`klicio` ストリーム自身ではない) Prolog 風の I/O ストリームに送るメッセージについては、次に述べます。

これらのメッセージは、返された I/O ストリームが C 風の I/O ストリーム用メッセージに加えて、Prolog 風の項の入出力用メッセージも解釈する他は、 `Unix` ストリームに対するメッセージと全く同じように機能します。

Prolog 風の I/O ストリームは、演算子定義を持ちます。さらに、各ストリームは異なる演算子定義を持つことができます。したがって、演算子をあるストリームに追加したり、あるストリームから取り除いたりしても、他のストリームで使用する演算子には、影響しません。各ストリームは、生成後すぐにデフォルトの演算子の集合を持ちます。

3.6.2.2 Prolog 風のインタフェースを用いた共通メッセージ

以下に示す、C 風の I/O ストリーム用メッセージは、Prolog 風の I/O ストリームに使用できます。

feof *-Result* メッセージ on Prolog 風の I/O
fseek *+Offset +Ptrname -Result* メッセージ on Prolog 風の I/O
ftell *-Result* メッセージ on Prolog 風の I/O
fclose *-Result* メッセージ on Prolog 風の I/O
sync *-Result* メッセージ on Prolog 風の I/O

詳細については、See Section 3.6.1.1 [C 風のインタフェースを用いた共通メッセージ], page 39。

addop *+Op +Type +Prec* メッセージ on Prolog 風の I/O

優先順位 *Prec* を持つ型 *Type* の演算子 *Op* を追加します。

rmop *+Op +Type* メッセージ on Prolog 風の I/O

型 *Type* の演算子 *Op* を取り除きます。

3.6.2.3 Prolog 風のインタフェースを用いた入力メッセージ

gett *-Term* メッセージ on Prolog 風の I/O

関連する入力ストリームから KLIC の構文項の形で *Term* へ読み込みます。構文解析に失敗した場合、メッセージを `stderr` に出力した後、別の項を読み込みます。ファイルの終了時には、アトム `end_of_file` を返します。

getwt *-Result* メッセージ on Prolog 風の I/O

関連する入力ストリームから KLIC の構文項の形で *Term* へ読み込み、結果を *Result* へ返します。構文解析が成功した場合、*Result* は、`normal(`*WrappedTerm*`)` の形式に

なります。ここで、*WrappedTerm* とは、読み込んだ項の基底項表現のことです。変数は、変数名の情報を持つ基底項として表現します。構文解析が失敗した場合、メッセージを `stderr` に出力した後、別の項を読み込みます。ファイルの終了時には、`normal(end_of_file)` を返します。

ラップした項の操作については、See Section 3.6.2.5 [ラップした項], page 43。

以下に示す、C 風の I/O ストリーム用メッセージは、Prolog 風の I/O ストリームに使用できます。

getc *-C* メッセージ on Prolog 風の I/O
ungetc *+C* メッセージ on Prolog 風の I/O
fread *+Max -String* メッセージ on Prolog 風の I/O
linecount *-Count* メッセージ on Prolog 風の I/O

詳細については、See Section 3.6.1.2 [C 風のインタフェースを用いた入力メッセージ], page 39。

3.6.2.4 Prolog 風のインタフェースを用いた出力メッセージ

putt *+Term* メッセージ on Prolog 風の I/O
puttq *+Term* メッセージ on Prolog 風の I/O
putwt *+WrappedTerm* メッセージ on Prolog 風の I/O
putwtq *+WrappedTerm* メッセージ on Prolog 風の I/O

項 *Term* やラップした項 *WrappedTerm* を、関連する出力ストリームへ書き出します。

文字 `q` の付いていないメッセージは、再び、読み込みをする必要がある場合でも、記号アトムを 2 つの引用符で囲みません。しかし、現在のところでは `q` の付いたメッセージと全く同じように機能します。

現在の版の出力フォーマットは、マシンが読めるだけで、人間が読める形式ではありません。つまり、演算子を全く使用しないですべてのアトムを括弧で囲んでいます。

ラップした項の操作については、See Section 3.6.2.5 [ラップした項], page 43。

以下に示す、C 風の I/O ストリーム用メッセージは、Prolog 風の I/O ストリームに使用できます。

putc *+C* メッセージ on Prolog 風の I/O
Number メッセージ on Prolog 風の I/O
fwrite *+String -Result* メッセージ on Prolog 風の I/O
fwrite *+String* メッセージ on Prolog 風の I/O
fflush *-Result* メッセージ on Prolog 風の I/O

詳細については、See Section 3.6.1.3 [C 風のインタフェースを用いた出力メッセージ], page 40。

項を終了させるピリオドは、これらのメッセージでは書き出せないので注意してください。ピリオドと空白や改行文字は通常、再び、読み込みをするために書き出す必要があります。以下に示すゴール列は、`/tmp/foo.bar`という名前のファイルをオープンして、変数 *X* の完全な具体化を待ち、後にピリオドと改行が続く Prolog 風の形式で出力します。

```
klicio:klicio([write_open("/tmp/foo.bar", normal(S))]),
S = [putt(X), putc(0'.), nl].
```

nl メッセージ on Prolog 風の I/O
改行コードを出力します。メッセージの `putc(10)` を同じストリームに送ることと、同じ意味になります。

Prolog 風の I/O ストリームも、`putc/1` や `getc/1` といった C 風の入出力が受け取るメッセージのすべてを受け取るので注意してください (see Section 3.6.1 [C 風のインタフェースを用いた入出力], page 38)。

3.6.2.5 ラップした項

変数を含む項のメタレベルの操作を可能にするため、KLIC では ラップした項 *(wrapped term)* と呼ばれるデータ表現を提供しています。ラップした項は、その中に変数を含まない基底項です。ラップした項は、以下に示す形式を持っています。

`variable(`*VarName*`)`
: *VarName* という名前文字列を持つ変数

`atom(`*Atom*`)`
: 記号アトム *Atom*

`integer(`*Int*`)`
: 整数 *Int*

`floating_point(`*Float*`)`
: 浮動小数点数 *Float*

`list([`*Car*`|`*Cdr*`])`
: *Car* と *Cdr* で構成される コンス・セル；*Car* と *Cdr* は自己再帰的なラップした項

`functor(`*Functor*`(`*Arg*`, ...)`
: ファンクタの構造；その引数の (*Arg*, ...) は自己再帰的なラップした項

`vector({`*Elem*`, ...})`
: ベクタ；その要素の (*Elem*, ...) は自己再帰的なラップした項

`string(`*Str*`)`
: 文字列 *Str*

`unknown(`*Term*`)`
: 不明データ；この場合ラッピングが不正確になることがあります

例えば、以下の項

```
f(a, X, 3, ["abc"|X], 3.14)
```

を
ラップした表現は、以下のようになります。

```
functor(f(atom(a),
        variable("X"),
        vector(integer(3), list([string("abc")|variable("X")])),
        floating_point(3.14))).
```

次の述語は、ラップした項を通常の項に変換します。

unwrap *-Wrapped ?Term* predicate on 変換
ラップした項 *Wrapped* を通常の項 *Term* に変換します。

ラップした項は、通常、入力操作の結果として獲得します (see Section 3.6.2.3 [Prolog 風のインタフェースを用いた入力メッセージ], page 42)。

ラップした項も、通常の KL1 の項と変わりがないので、通常のユーザ・プログラムで作成できます。通常の項を、ラップした項に変換する次の述語も、ある場合には有効です。

wrap *?Term -Wrapped* predicate on variable

通常の項 *Term* を、ラップした項 *Wrapped* に変換します。

現在の版では、すべての変数に同じ名前の _ を与えています。したがって、ある項をラップし、その結果をアンラップすると、元の項の中のすべての変数が、同じ名前で参照できるようになってしまいます。これはバグですので、将来のバージョンでは修復する予定です。

Term が同じ変数を複数個含んでいるとき、並行して演算を続けていると、変数を具体化することがあります。このような場合、同じ変数を 2 つ含む項は、1 つは変数のまま、もう 1 つは変数でない項を含む形でラップした項に変換されるかもしれません。このことは、この述語の仕様が内包する特有の問題ですので、おそらく修復は無理でしょう。したがって、この述語を非基底項 (non-ground terms) に適用する場合、デバッギング・ユーティリティといった、メタレベルのプログラムに限定すべきです。

3.7 システム動作の制御

以下に示す述語は、`system_control` モジュール中に提供しています。

postmortem *+Module +Goal -Result* predicate on system_control

メイン・プログラムの正常終了や異常終了の後に実行する後処理を、登録します。*Goal* は、ゴールの後処理の述語と引数を指定するファンクタ構造にする必要があります。*Module* は、後処理を行う述語のモジュールを指定する記号アトムにする必要があります。1 つのゴールだけを指定できます。この場合、コンマで区切られたゴールの並びは許していません。

登録が終了したら *Result* を `[]` で単一化します。登録の完了まで待ち状態になるので、他の処理は実行しません。

この述語が何度も呼び出された場合、最後の登録が有効となります。

gc *-Before -After* predicate on system_control

ガーベージ・コレクションを要求し、ガーベージ・コレクションの前後のヒープ・サイズを、それぞれワード単位で、*Before* と *After* へ返します。ワードのサイズは、インストールで使用した C 言語システムの `long` 型のサイズと同じです。

並列実装では、局所的な記憶域に対するガーベージ・コレクションだけが要求されます。大域的なガーベージ・コレクションの要求はできません。

3.8 タイマ

KLIC では実時間タイマを提供しています。Unix では 1 プロセスにつき 1 タイマしか提供されていませんが、KLIC ではこの機構を仮想化して必要な個数のタイマを使用できます。

実時間タイマが利用できないホストシステム上の実装では、この機能は提供していません。

時間の値は (時刻、時間間隔ともに) `time(`*Day*`,` *Sec*`,` *Usec*`)` の形式の項で表現します。ここで、*Day*, *Sec* and *Usec* はそれぞれ日、秒、マイクロ秒を表す負でない整数です。また、*Sec* は 86,000 (1 日) 未満、 *Usec* は 1,000,000 (1 秒) 未満でなければなりません。

次の述語は、モジュール `timer` 中に提供しています。

get_time_of_day *-Time* 述語 on timer

1970 年 1 月 1 日の真夜中を起点とし、秒とマイクロ秒で表現した現在の時刻を、 *Time* に返します。

得られる時刻は、この述語が実際に実行されたときの時刻です。ゴールの実行順序は KLIC のシステムによることに注意して下さい。報告される時間は、このゴールの親ゴールがリダクションされてから、*Time* の値が検査されるまでの間であることのみ、保証されています。

また、報告される時間は、この述語を実行するタスクが動作している OS が、返すものであることに注意して下さい。分散システムにおいては、各システム構成要素の時刻は完全に一致しているとは限りません。

add *Time1 Time2 -Time* 述語 on timer
sub *Time1 Time2 -Time* 述語 on timer

それぞれ、二つの時間値の加算および減算を行います。

compare *Time1 Time2 -Result* 述語 on timer

二つの時間値 *Time1* および *Time2* を比較し、結果を *Result* に返します。結果は、*Time1* が *Time2* より小さい (早い) 場合は `<`、等しい場合は `=`、*Time1* が *Time2* より大きい (遅い) 場合は `>` になります。

instantiate_at *Time -Var* 述語 on timer
instantiate_after *Interval -Var* 述語 on timer

指定された時間に、*Var* を記号アトム `[]` と単一化します。前者の述語は指定された時刻にこの動作を行いますが、後者は指定された時間間隔後にこの動作を行います。指定時間をすでに過ぎていた場合は、変数はただちに単一化される場合があります。

単一化操作には、任意時間の遅れが生じる可能性があることに注意して下さい。合理的な実装では、この遅れは短いはずです。

instantiate_every *Interval Stop -Var* 述語 on timer

Var を記号アトム `[]` のリストで順に単一化していきます。リストの最初の要素は指定した時間間隔後に単一化され、第二要素はさらに同じ時間間隔を経て単一化されます。この操作は引数 *Stop* が単一化されるまで繰り返され、リストの終端を生成して終了します。

単一化操作には、任意時間の遅れが生じる可能性があることに注意して下さい。合理的な実装では、この遅れは短いはずです。

3.9 乱数生成器

オブジェクトクラス `random_numbers` により、疑似乱数を生成することができます。この乱数生成器は `nrand48` によるものであり、ホストシステムに `nrand48` がない場合はこの機能は利用できません。

new -*Randoms Range* オブジェクト生成 on random_numbers
new -*Randoms Range Seed* オブジェクト生成 on random_numbers

0 から *Range* - 1 を含んだ範囲をとる整数の疑似乱数からなる、無限長のリストを *Randoms* に返します。*Range* は正の整数です。

任意指定の引数 *Seed* には乱数生成の種を指定します。同じ種を与えた場合、リストの要素は同じになることが保証されています。

リストは仮想的に無限ですが、リスト要素はプログラムでその値を検査することによって、遅延的に計算されることに注意して下さい。

4 KLIC の使い方

この章では、KLIC システムの使い方について説明します。

4.1 KLIC におけるプログラムのコンパイル

インストールが完了すると、klic コマンドで KL1 プログラムを C 言語プログラムへコンパイルし、実行可能コードを生成することが可能になります。klic というプログラムは、様々なオプションが可能なコンパイラ・ドライバです。

4.1.1 コンパイル用コマンド

引数に、拡張子 .kl1 が付いた KL1 ソース・プログラム・ファイル名を指定して、klic コマンドを実行すると、 KL1 プログラムは C 言語にコンパイルされ、実行可能コードが生成されます。

例えば、 XXX.kl1 をコンパイルしてコードを生成する場合、次のように入力します:

```
% klic XXX.kl1
```

実行結果は a.out となります。実行結果の名前を *YYY* としたい場合、次のように入力します。

```
% klic -o YYY XXX.kl1
```

例えば、プログラムが XXX.kl1 、YYY.kl1 、ZZZ.kl1 のように複数個のソース・ファイルに分割されている場合、次のように入力すると、一括してコンパイルとリンクができます。

```
% klic XXX.kl1 YYY.kl1 ZZZ.kl1
```

複数個の KL1 ソース・ファイルを別個にコンパイルして、後でリンクすることもできます。リンケージ・エラーを避けるために、次のように -c オプションを指定して、リンク前にコンパイル・コマンドの実行を終わらせる必要があります。

```
% klic -c XXX.kl1
% klic -c YYY.kl1
% klic -c ZZZ.kl1
```

最後に、次のように入力して、複数のコンパイル結果をリンクすることができます。

```
% klic XXX.o YYY.o ZZZ.o
```

コンパイラ・オプションの詳細については、See Section 4.1.2 [コンパイラ・オプション], page 47。

直接 C でかかれたプログラム (例えば *CCC*.c と *DDD*.c) を KL1 で書かれたプログラム (例えば *XXX*.kl1 と *YYY*.kl1) のリンクは、次のように入力するだけで実行できます。

```
% klic CCC.c DDD.c XXX.kl1 YYY.kl1
```

ファイル指定の順番は影響しません。C の関数は KL1 のインライン展開コード (see Section 2.10 [C 言語コードのインライン挿入], page 12) から呼び出すことが出来ます。

4.1.2 コンパイラ・オプション

コンパイル・コマンド klic のオプションの利用方法を、以下に示します。

`-c` 再配置可能オブジェクトを生成し、klic コマンドを終了します。リンクは行ないません。

`-C` C 言語への変換を終了し、klic コマンドを終了します。

`-d` 実際に実行するコマンド列を表示します。表示するだけで実行はしません。 -v を仮定します。

`-D` *database_manager*
: 指定されたデータベース管理プログラムを使用します。

`-g`
: dbx 、 gdb などのデバッガ用の情報を含んだ実行可能コードを生成します。

`-I` *directory*
: 指定されたインクルード・パスを C コンパイル時に使用します。

`-K` *klic_compiler*
: 指定された変換プログラムを KL1 から C への変換時に使用します。

`-l` *library*
: 指定されたライブラリをリンク時に使用します。

`-L` *directory*
: 指定されたライブラリ・パスをリンク時に使用します。

`-o` *file*
: 指定された実行可能コードファイルの名前を生成します。

`-O`
`-O`*level*
: 指定された最適化レベルで、コンパイルします。ゼロ以外の最適化レベルが指定されたとき、いくつかの付加的な最適化フラグも C コンパイラに渡します。そのような付加的な最適化フラグは、システムに依存するため、 KLIC システムのインストール手続きの際に決定されます。

  このオプションでは、 `-O` と *level* の間に空白があってはなりません。

`-P` *parallel*
: サブタスク (C コンパイラなど) を並列で実行します。最大時には、並列のサブタスクを同時にフォークします。

`-R`
: ファイルの日付とは無関係に、再コンパイルをします。

`-S`
: アセンブリ・コードを生成し、klic コマンドを終了します。

`-n`
: デバッグなし実行時ライブラリをリンクします。このオプションが省略された場合、デバッグあり実行時ライブラリをリンクします。

`-v`
: 実際に実行したすべてのコマンドを、標準エラー出力に出力します。

`-x`*directory*
: 指定されたディレクトリ中のデータベース・ファイル `klic.db` を使用します。また、`atom.c` 、 `funct.c` 、 `predicates.c` や対応するオブジェクトも指定されたディレクトリに置きます。このオプションは、リンクするプログラムが複数のディレクトリに分散している場合に便利です。

`-X`*directory*
: データベース・ファイル `klic.db` が存在しない場合、指定されたディレクトリ下のデータベース初期化ファイル `klicdb.init` から `klic.db` を作成します。このオプションが省略された場合、デフォルトのライブラリ・ディレクトリを仮定します。

次に示す環境変数によって、コンパイラのデフォルトの動作を変えることができます。コンパイル時に指定されたオプションの方は、環境変数の値よりも優先します。

`KLIC_LIBRARY`
: 実行時ライブラリ用のディレクトリです。`-X` オプションの方を優先します。

KLIC_DBINIT
: 初期データベース用のディレクトリです。デフォルトは実行時ライブラリ用のディレクトリです。

KLIC_COMPILER
: KL1 から C への変換プログラムです。-K オプションの方を優先します。

KLIC_DBMAKER
: データベース管理プログラムです。-D オプションの方を優先します。

KLIC_INCLUDE
: コンパイル時のインクルード・パスです。-I オプションの方を優先します。

KLIC_CC
: 使用する C コンパイラです。

KLIC_CC_OPTIONS
: コンパイラ・オプションです。

KLIC_LD
: 使用するリンカです。

KLIC_LD_OPTIONS
: リンカ・オプションです。

4.1.3 KLIC コンパイラの動作内容

KL1 プログラムのコンパイル過程や実行可能コードの生成過程を知ることによって、 KLIC の使い方の理解を深めることができます。

KLIC システムは、以下のように 3 個のモジュールで構成されます。

- KLIC コンパイラ
- KLIC データベース管理プログラム
- KLIC 実行時ライブラリ

KLIC コンパイラは、KL1 プログラムを C 言語プログラムにコンパイルします。KLIC コンパイラは同時に、.ext ファイル も生成します。このファイルは、プログラムで使用しているアトムとファンクタの情報などを持ちます 。リンクする個々のプログラムの .ext ファイル中にある情報は、後でデータベース管理プログラムによってマージされ、 atom.h 、 funct.h 、 atom.c 、 funct.c 、 predicates.c になります。

C コンパイラは、KLIC コンパイラが生成した C プログラムをコンパイルします。その際、ヘッダ・ファイル atom.h 、 funct.h を使用します。atom.c 、 funct.c 、predicates.c もコンパイルして、実行時ライブラリとリンクします (predicates.c は、デバッグあり実行時ライブラリを使用するときだけリンクします)。

コンパイル、データベース管理、およびリンクは、コンパイラ・ドライバ・プログラム klic が制御します。この klic プログラムは、 cc と make を組み合わせた機能に類似した機能を持っています。cc は、 C プリプロセッサ、 C コンパイラ・カーネル、およびリンカを制御します。同様に、klic は、 KL1 から C へのコンパイラ、 C コンパイラ、 KL1 プログラムのデータベース管理プログラム、およびリンカを制御します。make は、ファイルの日付をチェックし、コンパイルが必要なファイルだけをコンパイルします。klic も make と同様の処理を行います。

4.2 KLIC におけるプログラムの実行

生成された実行可能コードは、簡単に動かせます。KL1 プログラムをコンパイルして、実行可能コードを `a.out` に出力した場合、ファイル名 `./a.out` を使用中のシェルに渡すだけで実行できます。

`main` モジュール中の引数なし述語 `main` が、最初に実行されるゴールになります (see Section 2.5 [初期ゴール], page 8)。

4.2.1 実行時オプション

実行可能コードを動かす際、以下に示すオプションが利用できます。

`-h size` ワード単位のヒープの初期サイズです。コピー方式のガーベージ・コレクションを採用しているので、実際に使用するヒープ・サイズは、この 2 倍になります。ヒープ・サイズは、(例えば、`2097152` のように) 数値を直接指定したり、`2^10` 単位や `2^20` 単位で `k` や `m` を数字の後ろに付けて (`2048k` や `2m` のように) 指定したりできます。このオプションが省略された場合、初期サイズはマクロの `HEAPSIZE` によって決定します。配布版では、 HEAPSIZE は 24k になっています。1 ワードの長さは、C の `long int`型と同じです。実際の長さは、使用するハードウェアや C コンパイラに依存します。ヒープ・サイズは、 `-H` や `-a` オプションに従って自動的に増加します。

`-H` *size* ワード単位の最大のヒープ・サイズです。ヒープは、指定されたサイズまでは自動的に拡張します。このオプションが省略された場合、無限大を仮定します。

`-a` *ratio* ヒープ領域中でアクティブ・セル (ゴミでないセル) が占める割合の上限です。浮動小数点数で指定してください。アクティブ・セルの占有する割合が、指定された上限値を超えると、 `-H` オプションで指定された最大サイズを超えない限り、ヒープ・サイズを、次回のガーベージ・コレクションで 2 倍にします。このオプションが省略された場合、0.5 を仮定します。

`-g` ガーベージ・コレクションに要した時間を計測します。ヒープが小さいとガーベージ・コレクションは非常に短い時間で行うので、計測オーバヘッドの方が大きくなってしまう可能性が高くなります。このオプションが省略された場合、ガーベージ・コレクションの計測は行いません。

`-s` 中断情報を計測します。プログラムの実行が終了すると、中断した述語とその中断回数を報告します。このオプションは、デバッグあり実行時ライブラリをリンクした場合にだけ有効です。このオプションを省略しても、デバッグあり実行時ライブラリがリンクされるので、中断情報を計測します (see Section 4.1.2 [コンパイラ・オプション], page 47)。

`-t` トレース状態で、プログラムの実行を開始します (see Section 4.3 [プログラム実行のトレース], page 51)。このオプションは、デバッグあり実行時ライブラリをリンクした場合にだけ有効です。このオプションを省略しても、デバッグあり実行時ライブラリがリンクされるので、トレース状態になります。デバッグなし実行時ライブラリをリンクさせる場合には、コンパイル時のオプションで指定します (see Section 4.1.2 [コンパイラ・オプション], page 47)。

すべての実行可能ゴールが実行を終了すると、プログラムは停止します。デバッグあり実行時ライブラリがリンクされている場合、(実行可能ゴールは存在しないが)、入力データを待って中断しているゴールが存在した場合、中断の原因となっているゴールが探索されて報告されます。デバッグなし実行時ライブラリがリンクされている場合は、中断しているゴールの数だけが報告されます。

4.3 プログラム実行のトレース

KLIC では、スパイ(ブレーク・ポイント) 機能を持つデバッグ・トレーサを提供します。

4.3.1 トレース実行の準備

トレース機能を使うには、プログラムをデバッグありの実行時ライブラリでリンクする必要があります。特に指定がなければ、デバッグありの実行時ライブラリがリンクされますが、コンパイル・コマンドの `klic` に `-n` オプションを指定すれば、トレースは行われません。

トレース対象のプログラムを、既に、`-n` オプションでコンパイル・リンクしている場合でも、プログラムを、初めから再コンパイルする必要はありません。このような場合、 `-n` オプションなしで再度 `klic` コマンドを実行すると、オブジェクトをデバッグありの実行時ライブラリとリンクするだけなので、短時間の操作で済みます。

プログラムの実行をトレースするには、`-t` オプションを付けてプログラムを実行するだけです (see Section 4.2.1 [実行時オプション], page 50)。

4.3.2 トレース・ポート

KL1 プログラムの実行の様子を以下に示します。

1. 初期ゴールの `main`：`main` は、実行するゴールを貯めておくためのゴール・プールに入れられます。
2. 1 つのゴールがゴール・プールから取り出されます (`CALL` ポート)。
3. ゴールは、プログラム節とマッチングされます。
4. ゴールと一致する節があれば、ゴールはサブゴールにリダクションし、それらのサブゴールはゴール・プールに戻されます (`REDUCE` ポート)。
5. ゴールと一致する節がない場合は、計算全体が放棄されます (`FAIL` ポート)。
6. ゴール引数の値やそれらの構造が確定していないため、ゴールと一致する節があるか否かを決定できない場合、ゴールは必要な値が揃うまで、ゴールを貯めておくための別のゴール・プールに入れられます (`SUSPEND` ポート)。
7. ゴール・プールにゴールが残っていれば、ステップ 2 にループします。

ゴールの実行は、上記した 4 つのポイント、すなわち 2 、 4 、 5 、 6 番がトレースされます。このような対象ポイントは トレース・ポートと呼ばれ、それぞれ `CALL` 、 `REDUCE` 、 `FAIL` 、`SUSPEND` ポートと呼びます。

Prolog の 4 ポート・トレース・モデルに慣れ親しんでいれば、 Prolog の他の 2 つのポートの `EXIT` と `REDO` がないことを疑問に思うかもしれません。KL1 プログラムは、バックトラックをしないので `REDO` ポートは存在しません。`EXIT` ポートは、次の 2 つの理由によってトレースされません。1 つは、ゴール - サブゴール階層のすべてについて、実行履歴を維持することは、 Prolog のような逐次処理言語に比べて、 KL1 のような並列処理言語の方が負担が重くなるからです。これは、階層における多くの異なるサブツリーが、データフローの同期機能のために、互いにインタリーブで動くためです。他の 1 つは、 KL1 プログラムがしばしば、同じ述語を末尾再帰呼出しの形式で呼び出すゴールとして各々が定義された、通信プロセスの集合として書かれているためです。そのようなプロセス (永久プロセス: perpetual processes と呼ばれる場合がある) は、ほとんど終了することがないので、終了を探索することは Prolog の場合に比べ、意味がないためです。

4.3.3 トレース表示のフォーマット

以下に、サンプル・プログラムを示します。

```
:- module main.

main :- nrev([1,2],X), builtin:print(X).

nrev([], R) :- R = [].
nrev([W|X], R) :- nrev(X, XR), append(XR, [W], R).

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

以下に、サンプル・プログラムの実行をすべてトレースした出力結果を示します。

```
1 CALL:main:main?
1 REDU:main:main :-
2   0:+nrev([1,2],_4)
3   1:+builtin:print(_4)?
2 CALL:main:nrev([1,2],_4)?
2 REDU:main:nrev([1,2],_4) :-
4   0:+nrev([2],_D)
5   1:+append(_D,[1],_4)?
4 CALL:main:nrev([2],_D)?
4 REDU:main:nrev([2],_D) :-
6    0:+nrev([],_18)
7    1:+append(_18,[2],_D)?
6 CALL:main:nrev([],_18)?
6 REDU:main:nrev([],[])?
7 CALL:main:append([],[],_D)?
7 REDU:main:append([],[2],[2])?
5 CALL:main:append([2],[1],_4)?
5 REDU:main:append([2],[1],[2|_1F]) :-
8   0:+append([],[1],_1F)?
8 CALL:main:append([],[1],_1F)?
8 REDU:main:append([],[1],[1])?
3 CALL:builtin:print([2,1])?
[2,1]
3 REDU:builtin:print([2,1])?
```

このプログラムでは、中断や失敗がないので、すべてのトレース出力は、 CALL ポートか REDUCE ポート (トレース例では、REDU と出力されている) のどちらかになります。上記の 1 行目は、初期ゴール main:main の CALL ポートのトレースです。

```
1 CALL:main:main?
```

トレースされたすべてのゴールには、ゴール間で区別するために、一意な識別子（整数値）が与えられています。1 カラム目の番号 1 は、初期ゴールの識別子です。

初期ゴールは、プログラムで定義した最初の節と一致するので、プログラム節中で定義しているように、サブゴールにリダクションします。このリダクションは、以下のようにトレースされます。

```
1 REDU:main:main :-
2   0:+nrev([1,2],_4)
```

```
3   1:+builtin:print(_4)?
```

これは、識別子 1 を持つオリジナル・ゴール `main:main` が、2 つの新しいゴールにリダクションしていることを示しています。この 2 つの新しいゴールは、`main:nrev([1,2],_4)` と `builtin:print(_4)` で、それぞれ識別子 2 と 3 を持ちます。

新しいゴールの識別子 2 と 3 に続く、番号の 0 と 1 は、リダクションによって生成されたサブゴールのシーケンス番号です。これらのシーケンス番号は、コマンドを適用するサブゴールを識別するために、トレーサ・コマンドによって使用されます。グローバルな意味を持つ一意なゴール識別子と違って、これらのサブゴール番号は、この特定のポートでしか意味を持ちません。

次に続く : は、サブゴールが、親ゴールの通常のサブゴールであることを意味しています。: 以外にも、次に示すものがトレース表示されます。* の場合は、 * に続くゴールもサブゴールですが、親とは異なる優先順位が与えられていることを意味します。優先順位は疑似プラグマ形式で表示されます。! の場合は、 ! に続くゴールは、実際に親からリダクションしたサブゴールではなく、このリダクションによって、具体的な値が変数に与えられたことで、実行可能となったゴールであることを意味します。

次に + か - のどちらかが続きます。ただ実行を継続するだけの場合、 + は、サブゴールをトレースし、 - は、トレースしないことを意味します。これは、あとの項で述べるトレース・コマンドを指定することで変更できます。上記の例では、すべてのサブゴールに + が付いているので、すべてのサブゴールがトレースされます。

続いて、モジュール名、コロン（:）、そしてサブゴールの述語名が表示されます。サブゴールの述語のモジュール名は、その述語が親ゴールの述語と同じモジュールに定義されている場合は、(:) とともに省略されます。上記の例では、サブゴール `nrev` (モジュール名 `main`) は、親ゴールの `main:main` と同じモジュール名なので、モジュール名は表示されません。

最後には、括弧の中にコンマ（,）で区切られた引数リストが続きます。nrev の第 2 引数と print の引数は `_4` です。これは、ソース・プログラム中の `X` と対応する変数に相当します。変数は、述語節の適用ごとに新しく割り当てられ、また、2 つ以上の変数は単一化できるので、ソース・プログラム中のオリジナルな名前を表示することは意味がありません。このような理由で、`_4` のような一意な名前が与えられています。

実際、この番号 4 は、変数の物理メモリ・アドレスと関連します。そのため、この番号は、ガーベージ・コレクションによって変更されます。しかし、ガーベージ・コレクションは、それほど頻繁には行われないので、アドレス情報は、デバッグでは有効です。

トレースは、すべてのサブゴールと疑問符（?）を表示したあと、停止します。ここで、次に示すトレース・コマンドを入力できます。

4.3.4 トレース制御コマンド

トレースは、 *leashed* された各ポートで制御します (see Section 4.3.6 [トレース・ポートの制御], page 55)。トレースは、トレース対象のゴール全体や、リダクション・ポートにおいて新しく生成された各サブゴールを制御します。各述語ごとに、ゴールをトレースするか否かのデフォルト値を設定することもできます。

4.3.4.1 トレース対象ゴールのトレース制御

以下に示すコマンドは、プログラムの実行を制御するために利用します。

Continue : `c` または単に〈リターン〉
ステップ実行します。- が付いたサブゴールは、ステップ・モードでもトレースしません。

Leap：`l` スパイ・ポイントに達するまで、トレースしないで実行を継続します。詳細については、See Section 4.3.5 [スパイ], page 55。

Skip：`s` トレース対象のゴールとすべてのサブゴールを、トレースしないで実行します。スパイ・ポイントがあっても無視します。

Abort：`a` プログラム全体の実行を放棄します。

これらのコマンドには、引数はありません。

4.3.4.2 新しく生成されたサブゴールのトレース制御

各サブゴール (+ や - で表示されている) のトレースは、以下に示すコマンドで変更できます。

Trace：+ *subgoal_number* ...
指定したサブゴールのトレースのスイッチをオンにします。複数のサブゴール番号を指定する場合、空白で区切ります。サブゴール番号を指定しない場合、すべてのサブゴールがトレースされます。

No Trace：- *subgoal_number* ...
指定したサブゴールのトレースのスイッチをオフにします。複数のサブゴール番号を指定する場合、空白で区切ります。サブゴール番号を指定しない場合、すべてのサブゴールはトレースされません。

Toggle Trace：*subgoal_number* ...
指定したサブゴールのトレース・スイッチを切り替えます。トレースのスイッチがオンならオフに、オフならオンになります。複数のサブゴール番号を指定する場合、空白で区切ります。

4.3.4.3 述語ごとのトレース・スイッチのデフォルト値の変更

デフォルトでは、ゴールのすべてのサブゴールに対して、トレース・スイッチは、最初の reduce ポートではオン (+) の状態です。このデフォルトの設定は、この項で説明するコマンドを使えば述語ごとに変更できます。変更の結果、トレース・スイッチがオフ (-) となった述語は、デフォルトではトレースされません。

コマンドの引数 <述語> は、以下の形式のいずれかになります。

Module:*Predicate*/*Arity*
明示的に、1 つの述語を指定します。例えば、 `main:nrev/2` のように指定します。

Module:*Predicate*
引数個数に関係なく、モジュール 内のすべての述語を指定します。

Module: モジュールで定義されたすべての述語を指定します。述語名と区別するため、モジュール名のあとにはコロン（:）が必要です。

Predicate/*Arity*
与えられた名前と引数個数を持つ現在トレース対象であるゴールの述語と同じモジュールで定義された述語を指定します。

Predicate 与えられた名前を持つ現在トレース対象であるゴールの述語と同じモジュールで定義された述語を指定します。

与えられた述語のトレース・スイッチのデフォルトを変更するコマンドを、以下に示します。

No Trace Default: `n Predicate ...`
述語のトレース・スイッチのデフォルトをオフに設定します。引数に述語を指定しない場合、トレース対象ゴールの述語が指定されたものと仮定されます。

Trace Default: `t Predicate ...`
述語のトレース・スイッチのデフォルトをオンに設定します。引数に述語を指定しない場合、トレース対象ゴールの述語が指定されたものと仮定されます。

4.3.5 スパイ

特定の述語だけをデバッグ対象にすることはよくあることです。そのような場合、その述語のポートを スパイ・ポイント として指定できます。leap (`l`) コマンドを使って、任意のスパイ・ポイントに達するまで、プログラムをトレースせずに実行させることができます。詳細については、See Section 4.3.4.1 [トレース対象ゴールのトレース制御], page 53。

この項で説明するコマンドは、スパイ・ポイントの設定や解除を行います。

Spy: `S` *Predicate* ...
述語をスパイします。引数に述語が指定されない場合、トレース対象ゴールの述語をスパイします。

No Spy: `N` *Predicate* ...
述語に設定されているスパイ・ポイントを解除します。引数に述語が指定されない場合、トレース対象ゴールの述語のスパイ・ポイントを解除します。

4.3.6 トレース・ポートの制御

4 つのトレース・ポートを使用可能状態 (enabled) にしたり、使用禁止状態 (disabled) にしたりできます。使用禁止状態になったポートはトレースされません。

さらに各ポートに対して、停止状態になってコマンド入力を待つか否かを指定できます。実行を停止して、コマンドを待つポートのことを *leashed* と呼びます。使用可能状態で leashed されていないポートの場合、トレース結果の出力表示後すぐに continue コマンド (キャリッジ・リターン) が入力されたように、実行が継続されます。スパイ対象の述語の場合、 unleashed なポートも leashed になります。

この項で説明するコマンドは、そのようなポートの属性を制御するために使用されます。これらのコマンドでは、以下に示す方法で引数にポート名を指定します。

Call: `c`, `call`

Reduce: `r`, `redu`, `reduce`

Suspend: `s`, `susp`, `suspend`

Fail: `f`, `fail`

All ports: `a`, `all`

ポートを制御するコマンドを以下に示します。

Enable Port: `E` *port* ...
指定されたポート (群) を使用可能状態にします。

Disable Port: `D` *port* ...
: 指定されたポート (群) を使用禁止状態にします。

Leash Port: `L` *port* ...
: 指定されたポート (群) を leashed にします。

Unleash Port: `U` *port* ...
: 指定されたポート (群) を unleashed にします。

4.3.7 表示を制御するコマンド

トレース対象ゴールの完全情報が常に必要であるとは限りません。逆にそういった過剰な完全情報がプログラム動作の理解を妨げる場合さえあります。そういった場合のために、トレース・ポートで表示される情報量を制御するコマンドが、提供されています。

表示量は、以下に示すオプションの組合せで制御されます。

表示の深さ制限：深さ制限以下の引数構造は、次に示す省略形で表示します。

```
f(a,b,c,d,e)    ↦    f(..)
[a,b,c,d,e]     ↦    [..]
```

表示の長さ制限：長さ制限を超えた引数リストの構造や文字列は、次に示す省略形で表示します。

```
f(a,b,c,d,e)    ↦    f(a,b,c,..)
[a,b,c,d,e]     ↦    [a,b,c,..]
"abcde"         ↦    "abc.."
```

サブタームの表示指定：指定されたトレース対象ゴールの特定の部分だけを表示します。

以下に示すコマンドは、オプション制御のために使用できます。

Set Print Depth: `pd` *depth*
: データ構造の表示の深さ制限に *depth* を設定します。引数なしの場合、現在の深さ制限値で表示します。

Set Print Length: `pl` *length*
: データ構造の表示の長さ制限に *length* を設定します。引数なしの場合、現在の長さ制限値で表示します。

Toggle Verbose Print: `pv`
: 詳細表示モードのスイッチを切り替えます。詳細表示モードでは、ゴールの実行を中断している変数を表示する際に、そのゴールも一緒に表示します。

Set Subterm: `^` *N*

Reset Subterm:`^`
: トレース対象ゴールの N 番目のサブタームを表示します。N に 0 が指定された場合、サブタームは 1 レベル上がります。N が省略された場合、サブタームの表示を解除します。リスト構造の場合、 1 は car 、 2 は cdr を意味します。

  このコマンド実行中は、トレース対象ゴールのサブタームだけが表示されます。表示されているサブタームの前には、タームのどの部分が表示されているのかを示す位置情報が表示されます。以下に例を示します。

```
10 CALL: foo:bar(f(a,g(..),[..]))? ^1
10 CALL: ^1 f(a,g(b,c),[d,e])? ^2
10 CALL: ^1^2 g(b,c)? ^0
```

```
10 CALL: ^1 f(a,g(b,c),[d,e])? ^3
10 CALL: ^1^3 [d,e]? ^2
10 CALL: ^1^3^2 [e]? ^
10 CALL: foo:bar(f(a,g(..),[..]))?
```

REDUCE ポートでは、サブタームを指定していると、リダクションで生成されたサブゴールは、表示されません。指定した親ゴールのサブタームだけが表示されます。現在の版では、ベクタ要素はサブタームとして指定できません。

深さ制限と長さ制限の初期設定値は、それぞれ 3 と 7 です。詳細表示モードのスイッチは、最初はオフに設定されています。

4.3.8 ゴールのダンプ

最後の手段として、システム中のすべてのゴールをダンプすることが望ましい場合があります。ダンプする場合のコマンドを以下に示します。

レディ・キューのダンプ：`Q`
: レディ・キュー (ゴール・プール) 中のすべてのゴールを、優先順位とともに表示します。

中断 (待ち状態) ゴールのダンプ：`W`
: システム中のすべての中断ゴールを、優先順位とともに表示します。

4.3.9 その他のコマンド

ステータスの問合せ：`=`
: トレーサのステータス情報を、以下のように表示します。

```
   port: Call Susp Redu Fail
enabled:  +    +    +    +
leashed:  +    +    +    +
print terse; depth = 3; length = 7
```

モジュール表示：`lm`
: 現在実行しているプログラムの、すべてのモジュールを表示します。

述語表示：`lp`
: 現在実行しているプログラムの、すべての述語とデフォルトのトレース状態を表示します。

キュー表示：`Q`
: レディ・キュー (ゴール・プール) の内容を表示します。

ヘルプ：`?`または `h`
: 現在のポートで利用できる、すべてのコマンドとその簡略説明を表示します。

4.3.10 永久中断の探索

あるゴールが、他のどのゴールからも具体化されない変数の具体化を待ち続ける場合、そのゴールは永久に先に進めなくなります。このような状態を、永久中断(perpetual suspention) と呼びます。永久中断は、 KLIC のガーベージ・コレクタが見つけ出すので、プログラム実行中に、ガーベージ・コレクションが発生すると、永久中断が見つけ出されることがあります。

システムは中断ゴールの数を管理しています。実行できるゴールが全くない状態で中断ゴールが残っている場合、システムは永久中断を見つけ出すためにガーベージ・コレクションを試みます。

永久中断は、以下のように報告されます。

```
!!! Perpetual Suspention Detected !!!
   3 PSUS: Module:Predicate(Args...)?
```

ここでは、 FAIL ポートと同じコマンド群が利用できます。

4.4 インストール

KLIC のインストールは、比較的簡単に行えます。

ホスト依存とユーザの選択に基づくカスタマイズは、配布された構成スクリプトを実行させて行います。その後、 `make all` でシステム全体をコンパイルしてください。次に、 `make tests` の実行によって、システムが問題なくコンパイルされたことが確認できます。続いて、 `make install` でシステムのインストールができます。

4.4.1 コンフィギュレーション

KLIC のインストールで最初に行うことは、ホスト・コンピュータ・システムと、ユーザの選択に依存した KLIC システムを構成することです。

配布物のルート・ディレクトリ (以下、 *ROOT* と呼びます) に進んでください。その後、'`./Configure`' コマンドで構成スクリプトを実行してください。スクリプトが、利用できるソフトウェア・ツールを、システムから探索して、選択の問合せをしてきます。

一部の BSD 4.2 をベースとする Unix システムでは、この構成スクリプトの構文の一部が、デフォルトで用意されたシェルでは実行できないことがあります。その場合は、新しいシェル (GNU `bash` など) を入手して、次のようにスクリプトを実行して下さい。

```
% bash Configure
```

KLIC システムを前に構成したことがあり、同じディレクトリで再構成を行う場合は、前回指定した値をデフォルト値にするかどうか、問い合わせてきます。

次の (初めてシステムを構成する場合は最初の) 問い合わせは、KLIC の並列版を構成するか否かというものです。逐次システムだけをインストールしたい場合は、 `no` と答えてください。並列版システムの構成の詳細については、See Section 4.5 [分散 KLIC], page 59 , Section 4.6 [共有メモリ KLIC], page 61。

構成スクリプトは、以下の 3 個のファイルを作成します。

'*ROOT*`/Makefile`'
'*ROOT*`/include/klic/config.h`'
'*ROOT*`/config.sh`'

最後のファイルは、システム再構成に備えて今回指定したオプションを保存します。

構成スクリプトは、インストール手続きで使用される並列機能 (parallelism) について問い合わせてきます。このとき、システムを負荷の軽いマルチプロセッサ・システムにインストール中であれば、ここでゼロ以外の並列機能を指定できます。`make` の並列実行機能を使用してはなりません。

4.4.2 KLIC システムのコンパイル

システムの構成後は、 `make all` と入力して、 KL1 から C へのコンパイラと、実行時ライブラリを含む KLIC システム全体をコンパイルしてください。

4.4.3 コンパイル結果のテスト

システム全体のコンパイルが終了した後は、コンパイルが問題なく終了したか否かをテストすることをお勧めします。テストする場合には、提供されたルート・ディレクトリ内 (そのサブディレクトリ `test` ではありません) で `make test` と入力してください。何本かの KL1 テスト・プログラムがコンパイル・実行されて、出力結果と予想結果が比較されます。

4.4.4 オブジェクトのインストール

コンパイルが終了したあと、 `make install` と入力すると、コンパイラ、ヘッダ・ファイル、および実行時ライブラリが、コンフィギュレーションで指定したディレクトリにインストールされます (see Section 4.4.1 [コンフィギュレーション], page 58)。

4.4.5 インストール・ディレクトリのクリーン・アップ

インストールが終了したあと、 `make distclean` と入力すると、配布物中に含まれて いない すべてのファイルが削除されます。

一般ユーザは、`make realclean` を試しては なりません。このコマンドは、 KL1 から生成された C プログラムのソース・ファイルを削除します。削除した C プログラムのソース・ファイルを再生成するには、既に動いている KL1 から C へのコンパイラが必要です。

4.4.6 うまく行かない場合

構成ミスのために、インストール手続きがうまく行かない場合は、コンフィギュレーションからやり直した方がよいでしょう (see Section 4.4.1 [コンフィギュレーション], page 58)。構成スクリプトは、コンフィギュレーションのやり直しのために、システムをクリーン・アップするか否かを問い合わせてきます。そのときには yes と答えてください。

Makefile に書かれる依存ルールは、`make` のいくつかのバージョンで提供される make の並列機能を使用する場合には適切ではありません。アトムとファンクタのデータベースが単調に増大しているという事実に依存しているので、それらのデータベースに依存するようには書けません。代わりに、コンパイラ・ドライバ `klic` の並列実行機能を使用してください。使用する並列機能は、コンフィギュレーションの段階で指定します (see Section 4.4.1 [コンフィギュレーション], page 58)。

配布されたコードに問題があると思われる場合は、以下に示すアドレスまでご報告ください。

```
klic-bugs@icot.or.jp
```

その際、ホスト・システム (ハードウェアとオペレーティング・システム) と構成 (トップレベルの ‘`Makefile`’ と ‘`include/klic/config.h`’ ファイル) に関する情報があると、問題分析に役立ちます。

4.5 分散 KLIC

KLIC の分散並列実装版も、配布される KLIC に含まれています。この分散実装は PVM3.3 に基づいています。MPI 等の並列処理ライブラリや、システム固有のプロセス間通信ライブラリに基づく実装も行われていますが、この配布版にはまだ統合されていません。

PVM に基づいていますが、現在の版では異種構成をサポートしていません。複数のアーキテクチャを持つプロセッサで構成されるシステムや、異なるオペレーティング・システムを動かすシステムでは機能しません。現時点では、異種システムをサポートする計画は全くありません。

4.5.1 分散 KLIC のインストール

分散 KLIC の PVM 版をインストールする場合、まず並列版を構成するか否かを構成スクリプトが最初に問い合わせてくるので、yes と答えてください。次に分散 KLIC を構成するか否かを、問い合わせてくるので、これにも yes と答えてください。そのあと、 PVM が利用できる場合は、 PVM システムのインストール先のディレクトリや、使用する PVM ライブラリなどについて、問い合わせてきます。

問合せの内容を、以下に示します。

PVM システムのルート・ディレクトリ

システムのアーキテクチャのキーワード (例えば、 `SUNMP` など)

PVM ライブラリの名前 (例えば、 `pvm3` など)

現在の版では、デーモンプロセスを使用しない PVM の実装には、問題があります。例えば、Solaris2 が動いている共有メモリ・マルチプロセッサの Sparc システムの場合、ライブラリの `pvm3` が機能しません。プロセス間通信には、共有メモリの代わりにソケットを使う `pvm3s` を使用してください。

インストール手続きの残りの部分は、分散 KLIC をインストールしない場合の手続きと同じです。

分散処理のオプション (`-dp`) が、コンパイル時に指定されていない場合、分散 KLIC システムは、逐次版と全く同じように動作します。

4.5.2 分散 KLIC 向けプログラムのコンパイル

コンパイル手続きは、以下に示すオプションが利用できる他は、逐次版とほぼ同じです。

`-dp` 分散 KLIC システムを用いたコンパイルを指定します。このオプションの指定がない場合、コンパイルされるオブジェクト・コードは、逐次処理でしか動きません。

4.5.3 分散 KLIC のプログラム実行

4.5.3.1 PVM のセット・アップ

分散実行用にコンパイルされたプログラムを実行する前に、 PVM システムがシステム上で稼働している必要があります。以下に示すセット・アップが必要です。

次の環境変数を設定してください。

`PVM_ROOT` システムにインストールした PVM システムのルート・ディレクトリ

`PVM_ARCH` システムのアーキテクチャを指定するキーワード

これらは、分散 KLIC システムをインストールした時に指定したものと同じものを指定してください。

PVM のデーモンを開始してください。デーモンは、 PVM コンソールを起動することで開始できます。PVM コンソールは、 `$PVM_ROOT/lib/$PVM_ARCH/pvm` に存在します。このコンソール用のウィンドウがあると便利です。

他のパラメタの設定と PVM コンソールの操作の詳細については、専用のマニュアルをご覧ください。

4.5.3.2 分散 KLIC の実行時オプション

分散 KLIC システムでプログラムを動かす場合、逐次版で利用できるオプションに加えて、以下のオプションが利用できます。

`-p` *N* プログラムを動かすための疑似プロセッサ (Unix プロセス) の数を指定します。klic 2.0 版ではパラメータが -dp から -p に変更になっています.

`-e` バッチ転送モードに切り替えます。通常、 KLIC は要求時にプロセッサ間でデータ構造を転送します。ネストしたデータ構造は通常、1 度に 1 レベルずつ転送します。バッチ転送では、ネストしたデータ構造を 1 度に転送します。これは、いくつかのプログラムに対して、より効率的に実行します。しかし、他のプログラムの性能を劣化させることもあります。

`-E` *Level* ネストしたデータ構造について、何レベルまで一度に送信するかを指定します。

`-I` *MicroSec*
プロセッサ間通信のポーリング間隔を指定します。このポーリングが必要かどうか、またどのような値が適切であるかは、ホストシステムと物理通信層の実装に依存します。多くの場合は、デフォルト値である 10000 が適切です。

`-n` 実行時の統計情報を表示します。

`-notimer` タイマ駆動の通信ポーリングを行いません。このポーリングが必須かどうかは、物理通信層の実装に依存します。

`-relsp` 疑似プロセッサプロセスを生成する時の実行ファイルを、相対パスで探します。

`-S` 通信パケットの受信側プロセスに対し、シグナル送信による通知を行いません。実装によっては、この指定によりシグナル送信のオーバヘッドが減るため、実行速度が向上する場合があります。

4.5.3.3 分散 KLIC の既知のバグ

新しく登録されたアトムとファンクタは、プログラムの実行中に、正常に処理されないことがあります。

スパイの指定 (see Section 4.3.5 [スパイ], page 55) は、指定した計算ノードの内部だけに効果があります。

4.6 共有メモリ KLIC

KLIC の共有メモリ並列実装版も、配布される KLIC に含まれています。この実装は、ハードウェア、 OS 、 C コンパイラの各々に依存する部分から成ります。このバージョンでは、SunOS 5.3 が動作している Sparc ベースのシステム、および DEC OSF/1 が動作している Alpha ベースのシステムに対応しています。コンパイルには、 Gnu CC が必要です。

4.6.1 共有メモリ KLIC のインストール

共有メモリ版の KLIC をインストールする場合、まず共有メモリ KLIC を構成するか否かを問い合わせてくるので、yes と答えてください

インストール手続きの残りの部分は、共有メモリ KLIC をインストールしない場合の手続きと同じです。

共有メモリ並列処理のオプション (`-shm`) が、コンパイル時に指定されていない場合、共有メモリ KLIC システムは、逐次版と全く同じように動作します。

4.6.2 共有メモリ KLIC 向けプログラムのコンパイル

コンパイル手続きは、以下に示すオプションが利用できる他は、逐次版とほぼ同じです。

`-shm` 共有メモリ KLIC システムを用いたコンパイルを指定します。このオプションの指定がない場合、コンパイルされるオブジェクト・コードは、逐次処理でしか動きません。

4.6.3 共有メモリ KLIC のプログラム実行

4.6.3.1 共有メモリ KLIC の実行時オプション

共有メモリ KLIC システムでプログラムを動かす場合、逐次版で利用できるオプションに加えて、以下のオプションが利用できます。

`-p` *N* プログラムを動かすための疑似プロセッサ (Unix プロセス) の数を指定します。

`-D` 子である疑似プロセッサのプロセス番号を報告します。低レベルのデバッグで便利かもしれません。

`-S` *Size* 共有ヒープ領域の大きさを指定します。現在の実装では、共有ヒープ領域は初期化の際に確保され、以後拡張されることはありません。

4.6.3.2 共有メモリ KLIC の既知のバグ

トレーサが正しく動かないことがあります。

データ型索引

(Index is nonexistent)

述語、メソッド、メッセージ名索引

(Index is nonexistent)

モジュール名索引

(Index is nonexistent)

索引

A

accumulator 12
alternatively 10

B

body 6

C

C 12, 38
car 25
cdr 25
chdir 36
clause 6
clause preference 10

D

difference list 12

E

execution 6

G

GHC 6
guard 6

I

I/O 38
ICOT 無償公開ソフトウェア 1

K

KL1 6

L

Linux 36

O

otherwize 7

P

preference of clauses 10
priority 9
PVM 58, 59

U

Unix インタフェース 34

あ

アトム 18
アトム・データ 18
新たなリリース 5

い

インストール 58
インターバルタイマ 45
インクリメント 12
インライン 12

え

演算子順位文法 40

お

オープン 35, 41
オブジェクト生成 9
オペレーティング・システム 34

か

改訂 12
書き出し 40
環境変数 37

が

ガード・メソッド 9
ガーベージ・コレクション 44

き

記号 18
記号アトム 18
共有メモリ KLIC 61
切り上げ 22
切り捨て 23

く

クラス 4

こ

高次 32
構造 24
コピーライト 1
コマンド行引数 38
コンス・セル 25
コンパイル 47
コンフィギュレーション 60, 61

コンフィギュレーション 58

ご
ゴール 8
ゴール・プール 57

さ
サブターム 56
三角関数 22

し
シーク 39
シェル・コマンド 37
シグナル 37
シグナルを送る 37
指数 22
終了コード 38
出力 38
出力のフラッシュ 39
出力引数 5
主ファンクタ 17, 24
詳細表示 56
初期ゴール 8

じ
ジェネリック・オブジェクト 8
ジェネリック・オブジェクトの生成 9
ジェネリック・メソッド 4, 8
時間 45
辞書式順序 31
実行 50
実数 21
述語 4, 7

す
ストリーム 26
スパイ 55

せ
正弦 22
整数 20
整数演算 20
整数からの浮動小数点変換 22
整数の演算 20
整数の比較 21
整数の浮動小数点への変換 22, 22
整数比較 21
正接 22
節 7

そ
双曲線関数 22

た
対数 22
タイマ 45
単一化 16

ダンプ 57
中止 38
中断ゴール 57
展開対 11
ディレクトリ 36
デクリメント 12
デバッグ 51
トレース 51
トレース表示 56
トレース表示の長さ制限 56
トレース表示の深さ制限 56
同期 39
入力 38, 39
入力引数 5
配布 1
配列 28
ハッシング 16
バグ報告 5
比較 16
引数対 10, 11
引数モード 5
否定 7
非同期 I/O 36
標準出力 35, 41
標準順序 16
標準入出力 35
標準入力 35, 41
ファイル 36
ファイルのクローズ 39
ファイルの終端 39
ファンクタ 24
ファンクタの操作 24
ファンクタ表記 24
フォークプロセス 37
浮動小数点演算 22
浮動小数点数 21
浮動小数点数の生成 22
浮動小数点の演算 22
浮動小数点の比較 23
浮動小数点比較 23
浮動小数点表記 22

ブレーク・ポイント 55
分散 KLIC 59
プログラム 32
プロセス間通信 5
プロセスのフォーク 37
並行論理プログラム言語 6
平方根 22
並列処理 59, 61
ヘッダ・ファイル 13
ベクタ 28
ベクタの生成 28
ボディ・メソッド 9
ポート 51, 55
ポストモーテム処理 44
マージング 26
丸め 22
未束縛 17
メイル・リスト 5
メイン 8
メソッド 4, 8
メッセージ 5
メッセージ送り 12
メッセージ・ストリーム 26
文字コード 20, 20
文字出力 40
文字入力 39
モジュール 7
文字列 29
文字列出力 40
文字列入力 39
文字列の比較 31
優先度 17
余弦 22
読み込み 39
ラップした項 43
乱数 46
リスト 25
リストの表記 26
リンケージ 47
レコード構造 24
レディー・キュー 57
割り込み 37

ア
アンリンク 36

Table of Contents

ICOT 無償公開ソフトウェアの利用条件 **1**

JIPDEC 無償公開ソフトウェアの利用条件 **3**

1 はじめに **4**
1.1 述語とメソッドの説明について 4
1.1.1 述語とメソッド 4
1.1.2 メッセージ 5
1.1.3 引数モード 5
1.2 バグ報告とコメントの送付 5

2 **KL1** とは **6**
2.1 基本的な実行の仕組み 6
2.2 述語 7
2.3 モジュール 7
2.4 ゴール 8
2.5 初期ゴール 8
2.6 ジェネリック・オブジェクト 8
2.6.1 ジェネリック・オブジェクトの生成 9
2.6.2 ジェネリック・データ・オブジェクトのガード・メソッド 9
2.6.3 ジェネリック・データ・オブジェクトのボディ・メソッド 9
2.7 優先順位の指定 9
2.8 節の優先関係 10
2.9 引数対の簡略表記 10
2.9.1 引数対と引数対の展開 11
2.9.2 引数対のためのマクロ 12
2.9.3 引数対の使い方 12
2.10 C 言語コードのインライン挿入 12
2.10.1 ファイル先頭でのインライン挿入 13
2.10.2 ガード部でのインライン挿入 13
2.10.3 KL1 項の C レベル表現 14
2.10.4 例 14
2.10.5 インライン C コード機能を使う上でのヒント 15

3 組込み述語とライブラリ機能 **16**
3.1 共通操作 16
3.1.1 単一化 16
3.1.2 同期 16
3.1.3 比較とハッシュ 16

3.1.4 実行ステータス 17
3.1.5 デバッグ 17
3.2 アトム・データ 18
3.2.1 記号アトム 18
3.2.1.1 記号アトムの表記 18
3.2.1.2 記号アトムの操作 19
3.2.2 整数アトム 20
3.2.2.1 整数の表記 20
3.2.2.2 整数演算 20
3.2.2.3 整数比較 21
3.2.3 浮動小数点数 21
3.2.3.1 浮動小数点数の表記 22
3.2.3.2 新しい浮動小数点数の生成 22
3.2.3.3 浮動小数点演算 22
3.2.3.4 浮動小数点比較 23
3.3 構造型データ 24
3.3.1 ファンクタ構造 24
3.3.1.1 ファンクタの表記 24
3.3.1.2 ファンクタの操作 24
3.3.2 リスト 25
3.3.2.1 リストの表記 26
3.3.2.2 メッセージ・ストリームの操作 26
3.3.3 ベクタ 28
3.3.3.1 ベクタの表記 28
3.3.3.2 ベクタの生成 28
3.3.3.3 ベクタの述語 28
3.3.4 文字列 29
3.3.4.1 文字列の表記 29
3.3.4.2 文字列の生成 30
3.3.4.3 文字列の述語 31
3.4 プログラム・コードのデータとしての扱い 32
3.4.1 モジュール 32
3.4.2 述語 32
3.5 Unix インタフェース 33
3.5.1 Unix インタフェース・ストリームの獲得 34
3.5.2 入出力用ストリームのオープン 34
3.5.3 ソケットの使い方 35
3.5.4 ファイルとディレクトリ 36
3.5.5 シグナル割込みの処理 37
3.5.6 Unix ストリームへの種々のメッセージ 37
3.5.7 述語インタフェース 38
3.6 入出力 38
3.6.1 C 風のインタフェースを用いた入出力 38
3.6.1.1 C 風のインタフェースを用いた共通メッセージ 38
3.6.1.2 C 風のインタフェースを用いた入力メッセージ 39

3.6.1.3 C 風のインタフェースを用いた出力メッセージ 39
3.6.2 Prolog 風のインタフェースを用いた入出力 40
3.6.2.1 Prolog 風の I/O ストリームのオープン 40
3.6.2.2 Prolog 風のインタフェースを用いた共通メッセージ 41
3.6.2.3 Prolog 風のインタフェースを用いた入力メッセージ 41
3.6.2.4 Prolog 風のインタフェースを用いた出力メッセージ 42
3.6.2.5 ラップした項 43
3.7 システム動作の制御 44
3.8 タイマ 44
3.9 乱数生成器 45

4 KLIC の使い方 47
4.1 KLIC におけるプログラムのコンパイル 47
4.1.1 コンパイル用コマンド 47
4.1.2 コンパイラ・オプション 47
4.1.3 KLIC コンパイラの動作内容 49
4.2 KLIC におけるプログラムの実行 50
4.2.1 実行時オプション 50
4.3 プログラム実行のトレース 51
4.3.1 トレース実行の準備 51
4.3.2 トレース・ポート 51
4.3.3 トレース表示のフォーマット 52
4.3.4 トレース制御コマンド 53
4.3.4.1 トレース対象ゴールのトレース制御 53
4.3.4.2 新しく生成されたサブゴールのトレース制御 54
4.3.4.3 述語ごとのトレース・スイッチのデフォルト値の変更 54
4.3.5 スパイ 55
4.3.6 トレース・ポートの制御 55
4.3.7 表示を制御するコマンド 56
4.3.8 ゴールのダンプ 57
4.3.9 その他のコマンド 57
4.3.10 永久中断の探索 57
4.4 インストール 58
4.4.1 コンフィギュレーション 58
4.4.2 KLIC システムのコンパイル 58
4.4.3 コンパイル結果のテスト 59
4.4.4 オブジェクトのインストール 59
4.4.5 インストール・ディレクトリのクリーン・アップ 59
4.4.6 うまく行かない場合 59
4.5 分散 KLIC 59
4.5.1 分散 KLIC のインストール 60
4.5.2 分散 KLIC 向けプログラムのコンパイル 60

4.5.3 分散 KLIC のプログラム実行 60
4.5.3.1 PVM のセット・アップ 60
4.5.3.2 分散 KLIC の実行時オプション 61
4.5.3.3 分散 KLIC の既知のバグ 61
4.6 共有メモリ KLIC 61
4.6.1 共有メモリ KLIC のインストール 61
4.6.2 共有メモリ KLIC 向けプログラムのコンパイル 62
4.6.3 共有メモリ KLIC のプログラム実行 62
4.6.3.1 共有メモリ KLIC の実行時オプション 62
4.6.3.2 共有メモリ KLIC の既知のバグ 62

データ型索引 63

述語、メソッド、メッセージ名索引 64

モジュール名索引 65

索引 66