

KLIC 講習会テキスト

KLIC システム編

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 KLIC システム利用法の入門テキストである。

KLIC システムの基本的な使い方, トレーサを使つてのデバッグ法と, KLIC システムのインストール方法について解説した。

講習会テキストとしては, 別に「KL1 言語編」がある。こちらは KLIC 言語の仕様とプログラミング・テクニックの基本を解説したものである。本編と合わせて御利用願いたい。

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

第 1 章	KLIC の使い方	1
1.1	KLIC システムの作り	1
1.2	まず, 走らせよう!	1
1.3	KL1 プログラムの構成	2
1.4	もうちょっと複雑な使い方	2
1.5	エラーメッセージ	8
1.6	実行時オプション	9
第 2 章	KLIC の実行機構とトレーサの使い方	10
2.1	KL1 のゴールリダクション方式	10
2.2	サスペンションメカニズム	12
2.3	トレーサの使用法	14
2.4	実行時エラーの表示とトレース	20
第 3 章	KLIC のインストール法	22
3.1	KLIC のコンパイル方式	22
3.2	必要とする環境	22
3.3	インストール方法	23
3.4	トラブルが起きたら	24
付録 A	簡易並列実装版の使い方	26
A.1	分散 KLIC 向けプログラムのコンパイル	26
A.2	分散 KLIC のプログラム実行	26
A.3	ランタイムモニタを使った実行	27
A.4	分散 KLIC の既知のバグ	28

第 1 章

KLIC の使い方

この章では, KLIC システムの基本的な使い方について説明する.

1.1 KLIC システムの作り

KLIC は, 基本的に,

KL1 で書かれたプログラムを C にコンパイルすることにより実行可能コードを作る

システムである. 但し, 手で, KL1⇒C コンパイラを立ち上げ, 次に C コンパイラを立ち上げ... とするのは, 面倒なので, `klic` というコマンドを起動するのみで KL1 のコンパイル, C のコンパイル/ リンクを行なうことができる.

1.2 まず, 走らせよう!

なにはともあれ, KLIC システムを走らせ, 実際にプログラムを動かしてみよう.

KL1 プログラムをコンパイルするためには,

```
% klic 'KL1 プログラムのファイル名'
```

とすればよい (KL1 プログラムのファイル名には, 拡張子 '`.kl1`' をつけること). 例えば, 図 1.1 に示す `nrev0.kl1` というプログラムをコンパイルするためには,

```
% klic nrev0.kl1
```

とする. この `klic` コマンドを実行すると, 実行形式のファイルが作成され, オプションを何も指定しないと (指定の仕方は後述) このファイルは `a.out` という名前になる. また, `nrev0.kl1` をコンパイルした結果できた C プログラムも同じディレクトリに `nrev0.c` というファイル名で作成されている.

この実行形式ファイルを実行すれば, KL1 プログラムを実行することができる.

```
% a.out  
[6,5,4,3,2,1]
```

1.3 KL1 プログラムの構成

では, KL1 プログラムの構成を見てみよう (図 1.1 参照).

```
% nrev0.kl1
% リストを逆転する.
:- module main.

main :- true |
    nrev([1,2,3,4,5,6],X),
    io:ostream([print(X),nl]).

nrev([], Y) :- true | Y=[].
nrev([A|X], Y) :- true |
    nrev(X, RevX),
    append(RevX, [A], Y).

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

図 1.1 nrev0.kl1

まず, 最初に`:- module main.`と書かれているが, これはモジュール宣言と呼ばれるものである (詳細はのちほど説明する). まず, KL1 プログラムの最初には必ず`:- module main.`と書いておけば良い, と理解しておけば良い.

次に, `main :- ...`と書かれているが, これは, 述語 `main` を定義していることを表す. `klic` コマンドでコンパイルしてできたコードは, まず, この述語 `main` を実行する決まりになっている. したがって, KLIC システムで利用するための KL1 プログラムには必ずこの述語 `main` が存在する必要がある (必ずしもファイルの先頭で定義されている必要はない). 「本当に行ないたいこと (今回であれば, `nrev([1,2,3,4,5,6],X)` という述語)」はこの述語 `main` から呼び出されるようにしておけば良い.

また, `io:ostream([print(X),nl])` とあるのは, 項 `X` を標準出力に印字するための述語呼出しであり, あらかじめシステムで用意されているものである. 述語 `io:ostream(Stream)` は, `Stream` に流れて来るメッセージに従い, 標準出力に印字を行う. 今回は, 以下の 2 種類のメッセージを利用している.

- `print/1 ...` 引数の項全体が具体化するまで待ち, 印字する.
- `nl/0 ...` 標準出力に改行コードを送る.

これで, 簡単な KL1 プログラムを書いて, `klic` コマンドでコンパイルし, 実行できるようになった.

1.4 もうちょっと複雑な使い方

先に述べた方法では, 1 つのファイルをコンパイルすることしかできず, 少し大きなプログラムを書こうと思うと, 少々面倒である. また, 必ず, `a.out` という実行形式のファイルができてしまう. ここでは, もうちょっと複雑な `klic` コマンドの使い方を解説する.

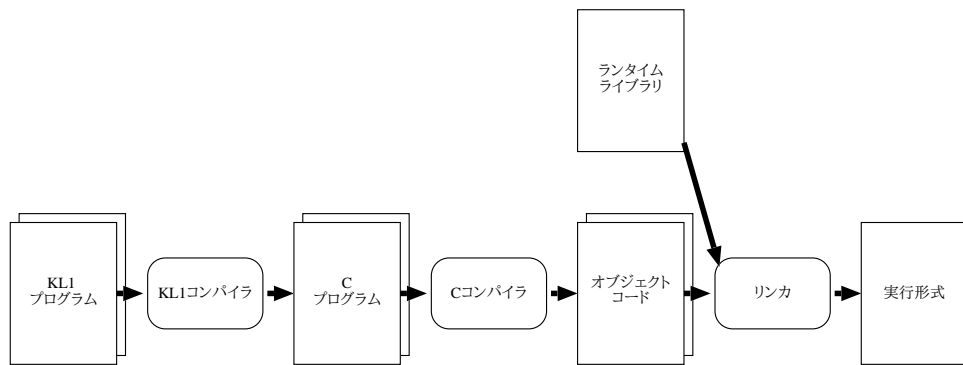


図 1.2 klic コマンドの仕組み

1.4.1 klic コマンドの仕組みを少し

先程も説明したように、klic コマンドによって、KL1 プログラムは C にコンパイルされ、実行形式になる。これをもう少し詳しく見ると、図 1.2 のようになる。

klic コマンドは、以下のようなプログラムを呼び出す。

KL1 ⇒ C コンパイラ: KL1 を C に変換するコンパイラ。このコンパイラは KL1 言語で記述されている。KLIC システムの心臓部分である。

C コンパイラ: 上の KL1 コンパイラが生成した C のコードをオブジェクトコードにする。

リンカ: C コンパイラが生成したオブジェクトコード^{*1} と KL1 プログラムが必要とするランタイム・ライブラリも併せてリンクする。もちろん、ここで複数のオブジェクトコードをリンクしても構わない。

1.4.2 2つのファイルの同時コンパイル/リンク

では、klic コマンドを使って 2 つの KL1 プログラムを同時にコンパイルし、1 つの実行形式にリンクしてみよう。

先ほどの nrev0.kl1 と同じ内容のプログラムを 2 つのファイルに分けたものが nrev1.kl1 と append.kl1 である (図 1.3 参照)。この場合、以下のようなコマンドにより 1 つの実行形式ファイルを得ることができる。

```
% klic -o nrev nrev1.kl1 append.kl1
```

-o nrev という指定は、「実行形式のファイル名は nrev とせよ」ということを表す。なにも指定しないと、(先ほどのように) a.out という名前のファイルが作成されてしまうが、この -o オプションを利用することによって実行形式のファイル名を自由に指定することができる。

^{*1} もちろん、KL1 コンパイラが生成した C をコンパイルしたオブジェクトコードでなくとも、さらには C コンパイラではなく、ほかのコンパイラで生成されたオブジェクトコードであってもリンクすることは原理的には可能である。

```

:- module main.

main :- true |
    nrev([1,2,3,4,5,6], X),
    io:ostream([print(X),nl]).

nrev([], Y) :- true | Y = [].
nrev([A|X], Y) :- true |
    nrev(X, RevX),
    app:append(RevX, [A], Y).

```

ファイル nrev1.kl1

```

:- module app.

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

```

ファイル append.kl1

図 1.3 nrev1.kl1, append.kl1

1.4.3 モジュール

では、2つのファイルを見てみよう (図 1.3 参照)。

まず、nrev1.kl1 の先頭には `:- module main.` と書かれているのは先程のプログラムと同様である。また、述語 `main` も同様に定義されている。最後に、`app:append(...)` と書かれているのに注意すること。

次に `append.kl1` を見ると最初には `:- module app.` と書かれており、先程の記述とは異なっている。

KL1 では複数のファイルに定義された述語を互いに利用することができる。しかしながら、いくつかのファイルを別々に記述すると、気がつかないうちに複数のファイルで同じ述語名が使われる可能性がある。これは、どの述語を呼び出すべきか分からなくなるので都合が悪い。

KLIC システムで利用できる KL1 プログラムはモジュールという単位で複数の述語を 1 まとめることができる。通常は 1 つのファイルが 1 つのモジュールとなる。モジュールにはあとで区別できるように名前をつけておく必要がある (この名前を「モジュール名」という)。実は、先頭にある `:- module main` という宣言は、「このファイルのモジュール名は `main` だよ」ということを宣言していたのであった。

ほかのモジュールの述語を呼び出そうとする場合には、その呼び出し先の述語にモジュール名も添える必要がある。

モジュール名:述語

たとえば、モジュール `app` の述語 `append(...)` を他のモジュールから呼び出すためには、`app:append(...)` とすれば良い。これで、どのモジュールに定義された述語を呼び出そうとしているのかを明確にすることができる。したがって、同じ名前 (で違う動作をするような) 述語を複数のモジュールで定義してしまっても、区別がすることができる。

この「モジュール」の取扱は `klic` システムで最初に呼び出される述語についても同様である。「`klic` で最初に呼び出される述語は述語 `main`」であることを既に説明したが、これは若干不正確で、実はモジュール `main` にある述語 `main` を呼び出すことになっている。

モジュールについて整理すると、以下のようになる。

- KL1 のプログラムはファイル毎に「モジュール」と呼ばれる述語の集合よりなる。
- 「モジュール」には名前をつける必要があり、それを「モジュール名」という。モジュールの先頭で、`:- module` 宣言をすることで、モジュール名を指定する。

- 他のモジュールで定義されている述語を呼び出すには、“呼び出す述語が定義されているモジュール名: 述語名”と書く。
- 最初に呼び出される述語は“main:main”である。

1.4.4 分割コンパイルの方法

先の使い方では「複数のファイルに対して同時に」コンパイル/リンクを行なっていた。しかしながら、実際にある程度大きなプログラムを作成し、デバッグ/修正を繰り返すようになると、「複数のファイルのうち1つだけ」を直してコンパイルし、それ以外のファイルのコンパイル結果 (オブジェクトコード) とリンクしたくなるような場面に遭遇することが少なくない。klic コマンドではそのような使い方をすることも可能である。以下のようにすれば、別々にコンパイルし、作成されたオブジェクトコードをリンクすることができる。

```
% klic -c nrev1.kl1
% klic -c append.kl1
% klic -o nrev nrev1.o append.o
```

ここで、-c というオプションは、実行形式は作成せず、オブジェクトコードまでしか作成しないことを指定している。通常、klic コマンドはなにも指定しなければ、実行形式まで作成する (リンクまで起動する) ようになっているが、ここでリンクを行なっても、所詮、プログラムの全てが与えられていないので、実行形式を作成することができない。そこで、オブジェクトコード (.o) の作成で止める必要がある。このオプションはそのため用いる。つまり、-c と指定した時には、オブジェクトのコードを作成したところで処理を止め、リンクを抑制する。

しかしながら、実際には、klic コマンドはファイルの作成された日付を見て、コンパイルを行なう必要があるかどうか^{*2}を自動的に判断する機能 (UNIX の make コマンドと同じ機能) を備えているので、以下のコマンドにより、自動的に最小限の処理が行なわれる。

```
% klic foo.kl1 bar.kl1
```

ここで、klic コマンドが行なっていることを調べてみよう。-v というオプションを使うと、途中行なっている処理の様子を見ることができる。

```
%klic -o nrev -v nrev1.kl1 append.kl1
KLIC compiler driver version 1.513 (Thu Dec 8 11:55:54 JST 1994)
/usr/local/lib/klic/kl1cmp nrev1.kl1 </dev/null
/usr/local/lib/klic/kl1cmp append.kl1 </dev/null
/usr/local/lib/klic/klicdb -X /usr/local/lib nrev1.ext append.ext
gcc -c -o nrev1.o -I/usr/local/include -I. nrev1.c
gcc -c -o append.o -I/usr/local/include -I. append.c
gcc -c -o atom.o -I/usr/local/include -I. atom.c
gcc -c -o funct.o -I/usr/local/include -I. funct.c
gcc -c -o predicates.o -I/usr/local/include -I. predicates.c
gcc -o nrev atom.o funct.o predicates.o nrev1.o append.o -L/usr/local/lib -lklic \
-L/usr/lib -L/lib -lm
%
```

^{*2} オブジェクトコードよりも、ソースコードの方が新しければ、最後に行なったコンパイル以降にソースコードが変更されていることになるので、コンパイルが行なわれる。さもなくば、コンパイルを行なう必要はない。


```
p(foo(X)) :- true | X=bar.
```

図 1.4 アトムとファンクタ

1.4.5 KLIC システムについてさらにもう少し

ここで、KLIC システムが作成するファイルについて説明を行なう。

klic を動かしたあとのディレクトリを見ると、klic.db や、atom.h とかいった、作った覚えのないファイルができていたのを発見するであろう。これらのファイルの働きについて解説する。

「記号」の扱い

KL1 のプログラムでは「アトム」や「ファンクタ」と呼ばれる「記号」が数多く使われる。例えば、図 1.4 に示すプログラムでは、foo, bar といったアトム、foo/1 というファンクタが用いられている。

これらは目で見た所、明らかに「文字列」である。しかしながら、これらの記号を内部的にも文字列として扱うことは、利用するメモリの量や、速度（たとえば、文字列 “bar” と、“bal” が違うことを見るためには、3 文字を 1 つずつ比較しなければならぬ）の面で不利になる。そこで、KLIC システムでは、これらの「記号」は内部的には全て「番号」で持っている。例えば、“foo” であれば 104 番、“bar” であれば 253 番... と内部では変換されているのである。この変換は KLIC の出力コードを効率化するため、コンパイル時に全て行なわれる。

さらにトレースやエラー表示のためには述語の印刷イメージを生成するための表も必要とする。

KLIC システムで作成されたプログラムは KL1 の項を読んだり書いたりすることができるが、その時にもこの記号 ⇔ 番号の変換が行なわれる。つまり、項が読み込まれる時には内部的な番号に置き換えられ、項を出力する時には、番号を記号に置き換え、人間に分かりやすく出力することができる。

複数ファイルでの「記号」の処理

さて、ここで 1 つ問題がある。

先にもあげたように、KLIC システムでは、複数の KL1 モジュールを 1 つずつ別々にコンパイルすることができる。これらのモジュールの間では「記号」がやりとりされることは当然考えられる。しかしながら、前章で説明したように、処理の効率を良くするため、「記号」は、内部では番号として扱われており、しかもコンパイルを行なう際に変換してしまう。モジュールの間で「記号」は受け渡されるので、異なったモジュールであっても同じ「記号」は同じ番号として変換される必要が生じる。「foo」という記号はこのモジュールでは 104 番だけど、あのモジュールでは 253 番で、それはそのモジュールの “bar” という記号の番号と同じになった」などということが起きてもらっては困るのである。

この問題の解決のために用いられているのが、klic.db というファイルである。このファイルの中には、「どの記号はどの番号に変換されたか」が記録されている。KL1 コンパイラは記号を見付けるたびに、このファイルにすでに登録されているかどうか調べ、すでに登録されていれば、その番号に変換するようにしている。登録されていなければ（他に、まだ、コンパイルを行なっていない、同じ記号を使っている KL1 プログラムがあるかも知れないので）そのファイルに「この記号は何番に変換したよ」ということを残しておく。つまり、このファイルを介して、複数の KL1 コンパイラが記号 ⇔ 番号の変換規則をやりとりしているわけである。

ファイル klic.db は、「KLIC データベース管理システム」(klicdb) というプログラムで管理されている。このプログラムにより atom.h, funct.h なるファイルも生成され KLIC コンパイラが生成した C プログラム

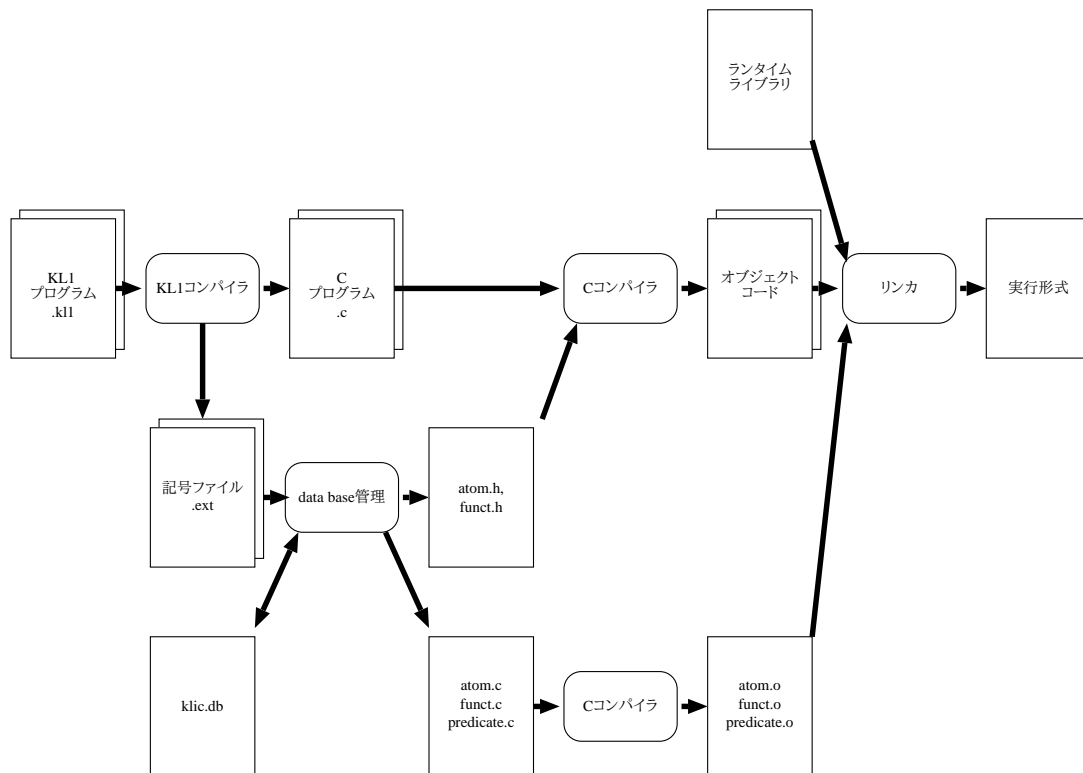


図 1.5 klic システム全体の流れ

内に取り込まれることにより記号 ⇒ 番号の変換が C コンパイル時に行なわれる。

この klic.db に書かれている情報は、KLIC システムで作られたプログラムが実行される時にも必要であるため、C プログラムに変換され、コンパイル/リンクされる。これが、`atom.c`, `funct.c`, `predicates.c` といったファイルである。

従って、KLIC システムを使って、複数のプログラムを作成してコンパイル/プログラム修正を繰り返している時に、ここで挙げたファイルが無くなってしまうと、それ以降の処理はおかしくなることがある。以下であげるファイルは作った覚えがないからといって、消してしまってはならない。^{*3}

- *.ext (.kl1 と同じ basename で拡張子が ext)
- klic.db
- atom.c atom.h
- funct.c funct.h
- predicates.c

もし、消してしまった場合は、`-R` という強制再コンパイルオプションで `.kl1` プログラムをコンパイルし直すようにすれば、これらのファイルはまた生成される。

この記号のメンテナンス関連の処理も含めた klic システム全体のファイル/処理の流れを図 1.5 に示す。

^{*3} 完成した実行形式のファイルはこれらのファイルを参照することはないので、その実行形式ファイルだけをどこかにコピーしてしまっても構わない。

表 1.1 主な klic コマンドのオプション

オプション	意味
-C	コンパイルのみ (C ソースが出力)
-c	コンパイルのみ (オブジェクトが出力)
-d	何を実行するかを表示 (実際のコンパイルはしない)
-g	C コンパイル時にデバッグオプション (-g) を付加
-Idirectory	C コンパイル時のインクルードパス指定
-Ldirectory	リンク時のライブラリパス指定
-lxxx	リンク時のライブラリ指定 (libxxx.a をリンク)
-o executable	実行ファイルのファイル名指定
-Odigit	C コンパイラに対する最適化レベル指定
-R	ファイルの依存性を無視して強制再コンパイル
-S	機械語アセンブラソース出力
-v	verbose mode
-?	ヘルプ
xxx.kl1	KL1 ファイル名
xxx.c	C ファイル名
xxx.o	object ファイル名
xxx.ext	記号表ファイル名

1.4.6 コンパイラオプション

ここまでで -c, -o, -v というオプションを使ったが、これら以外にもさまざまなオプションが用意されている。表 1.1 にまとめる。

1.5 エラーメッセージ

klic コマンドは、コンパイル対象の KL1 プログラムにエラーを発見すると、エラーを出力する。但し、このエラーは、C コンパイラ、リンカ等で出力されているケースもある。

klic コマンドの出力するエラーメッセージは大別すると以下のようになる。

- KL1 ⇒ C コンパイラの出力するもの ... これは比較的分かりやすいエラーメッセージとなっている筈なので、読めばその意味はわかるであろう。
- C コンパイラの出力するもの ... C コンパイラが出力するものとして、今の所、以下のものがあげられる。
 - － 未定義述語の使用 ... 外部モジュールではない (“モジュール名:” が付加されていない) 述語を呼び出しているが、その述語が当該モジュール内に定義されていない場合、C コンパイラが以下のようなメッセージを出力する。*4

*4 実際には、KLIC で利用する C コンパイラにより出力されるメッセージは異なる。ここであげたものは gcc version 2.6.2 のものである。

```
demo.c: In function 'module_demo':
demo.c:29: label 'bar_0_ext_interrupt' used but not defined
demo.c:28: label 'bar_0_0' used but not defined
C compilation failed for file demo.c
```

これは、「bar/0 という述語がない」ことを意味している。

- リンカが出力するもの ... リンカが出力するものとして、今の所、以下のものがあげられる。
 - － 未定義モジュールの利用
 - － 外部未定義述語の利用

ともに、以下のようなメッセージを出力する。

```
ld: Undefined symbol
      _predicate_bar_xbar_0
collect2: ld returned 2 exit status
Linkage failed
```

これは、「bar というモジュールがない。または、bar というモジュールに bar/0 という述語がない」ことを意味する。

1.6 実行時オプション

この章の最後に、klic コマンドで生成されたプログラムを実行する時に指定可能な実行時オプションを表 1.2 に挙げる。

表 1.2 主な実行時オプション

オプション	意味
-t	トレーサ付き実行
-h size	初期ヒープサイズをワード単位で指定
-H size	上限ヒープサイズをワード単位で指定
-g	GC 時間の計測指示
-s	サスペンド回数の計測指示
-v	実行時ルーチンのバージョン表示
-	ユーザ指定の実行時オプション列の開始

もちろんオプション指定の後に実行時プログラムへの引数が指定できる。引数の取り出しは unix モジュールの argc/1 述語および argv/1 述語を利用する。詳しくはマニュアルを参照されたい。

第 2 章

KLIC の実行機構とトレーサの使い方

この章では, KLIC の実行機構とトレーサの使用方法について述べる.

2.1 KL1 のゴールリダクション方式

2.1.1 ゴール書換えモデル

KL1 プログラムの実行は, ゴールを KL1 プログラムで定義された規則に従って書換えることにより行なわれる. KLIC 処理系では, ゴールはゴールプールと呼ばれる領域におかれ, リダクションの実行により生成されるサブゴール群は再びゴールプールに入れられる. ゴール書換えモデルは, 基本的に図 2.1 に示すように, 以下の処理の繰り返しにより実現される.

1. ゴールプールからゴールを 1 つ取ってくる (ゴール呼び出し操作).
2. ゴール引数の値を, ガード部の規則にしたがってチェックする.
3. チェックに適合した節を 1 つ選んでそのボディ部を実行し, 生成されたサブゴール群をゴールプールに入れる (リダクション操作).

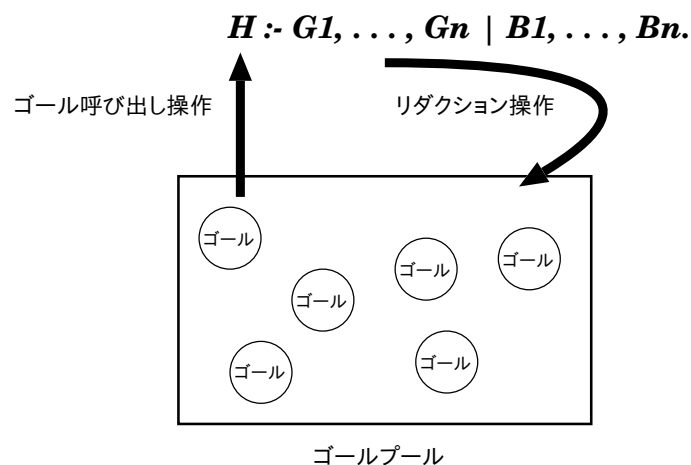


図 2.1 ゴールプールとリダクション操作

2.1.2 トレーサを用いたリダクション状況の観測

KLIC のトレーサは、上記のゴール呼びだし操作やリダクション操作などの、ゴール書換えモデルの各操作をモニターするものである。トレーサでは、各操作をモニターする部分を「ポート」と呼ぶ。

例えば、ゴール呼びだし操作をモニターする部分は「CALL ポート」と呼ばれ、リダクション操作をモニターする部分は「REDU ポート」と呼ばれる。以下に、実際のプログラムをトレースする例を示す。

プログラムをトレーサ付きで実行する方法

図 2.2 に例題として、数を要素とするあるリストが他のリストの部分集合となっているか調べるプログラムを示す。

```
:- module main.

main :- true | subset(yes, [3,1], [1,3,5], Ans),
        io:outstream([print(Ans),nl]).

subset(no, X, Y,Ans) :- true | Ans = no.
subset(yes, [], Y,Ans) :- true | Ans = yes.
subset(yes, [A|B],Y,Ans) :- true | member(A,Y,Ans1), subset(Ans1,B,Y,Ans).

member(A, B, Ans) :- B = [] | Ans = no.
member(A, B, Ans) :- B = [X|Y], A == X | Ans = yes.
member(A, B, Ans) :- B = [X|Y], A \== X | member(A, Y, Ans).
```

図 2.2 部分集合をチェックするプログラム

生成されたオブジェクトプログラムを `-t` オプションを付けて起動し、`<CR>` を入力し続けると以下に示すように、実行をトレースできる。

```
% klic -o subset subset.kl1 1
% subset -t
1 CALL:main:main?
1 REDU:main:main :-
2 0:+subset(yes,[3,1],[1,3,5],_3) 5
3 1:+io:outstream([print(_3),nl])?
2 CALL:main:subset(yes,[3,1],[1,3,5],_3)?
2 REDU:main:subset(yes,[3,1],[1,3,5],_3) :-
4 0:+member(3,[1,3,5],_11)
5 1:+subset(_11,[1],[1,3,5],_3)? 10
4 CALL:main:member(3,[1,3,5],_11)?
4 REDU:main:member(3,[1,3,5],_11) :-
6 0:+member(3,[3,5],_11)?
6 CALL:main:member(3,[3,5],_11)?
6 REDU:main:member(3,[3,5],yes)? 15
5 CALL:main:subset(yes,[1],[1,3,5],_3)?
5 REDU:main:subset(yes,[1],[1,3,5],_3) :-
7 0:+member(1,[1,3,5],_21)
8 1:+subset(_21,[],[1,3,5],_3)?
```

7 CALL:main:member(1,[1,3,5],_21)?	20
7 REDU:main:member(1,[1,3,5],yes)?	
8 CALL:main:subset(yes,[],[1,3,5],_3)?	
8 REDU:main:subset(yes,[],[1,3,5],yes)?	
3 CALL:io:ostream([print(yes),nl])? s	
yes	25
%	

トレース結果の読み方

- プログラム中のゴールには通し番号が付けられる。トレース結果の各行の最初の数字は、ゴールの番号である。
- 1 CALL:main:main? (3 行目) は、「main モジュールの main 述語が呼び出された。」ことを意味する。つまり、ゴールプールから main:main という述語が取り出されてきたことを意味している。
- 1 CALL:main:main? が表示された所で、トレーサは入力待ちになる。ここで<CR>を入力すると、プログラムを次に進めることができる。(?が出ている所ではさまざまなトレーサのコマンドが入力できるが、詳細は後で述べる)
- 4 行目から 6 行目の


```

1 REDU:main:main :-
2     0:+subset(yes,[3,1],[1,3,5],_3)
3     1:+io:ostream([print(_3),nl])?

```

は、main:main が REDU ポート を通過し、subset(yes, [3, 1], [1, 3, 5], _3) と io:ostream([print(_3), nl]) の 2 つのゴールをゴールプールに入れたことを意味する。KLIC のトレーサでは、具体値はその値がそのまま表示され、変数は全て _ で 始まる文字に変換されて表示される。

- 7 行目の 2 CALL:main:subset(yes,[3,1],[1,3,5],_3)? は、前のリダクションでゴールプールに入れられた、ゴール番号 2 番のゴールが呼び出されたことを意味する。以下、同様のリダクション操作が続く。
- 23 行目の 3 CALL:io:ostream([print(yes),nl])? s では、6 行目でゴールプールに入れられた、ゴール番号 3 番のゴールが呼び出されている。トレースからも分かるように、KLIC のゴールプールは LIFO のキューとなっている。

また、この行では s というコマンドを入力している。これは、io:ostream というゴールのトレースを行なわないことを意味する。io:ostream は、KL1 で書かれている KLIC システムのユーティリティであり、この内部まで詳細にトレースする必要はないからである。

2.2 サスペンションメカニズム

ゴール書換えモデルにおいて、ゴールプールからゴールを 1 つ取り出しゴール引数の値を KL1 節のガード部の規則にしたがってチェックする際に、そのチェックがサスペンドする場合は、そのゴールのリダクションは中断されゴールはサスペンドゴールプールに入れられる。(サスペンド操作)

サスペンドゴールプールに入れられたゴールは、中断原因となった変数が具体化されるとゴールプールに移される。(リジューム操作)

図 2.3 にサスペンドゴールプールを用いたサスペンションメカニズムの概略図を示す。

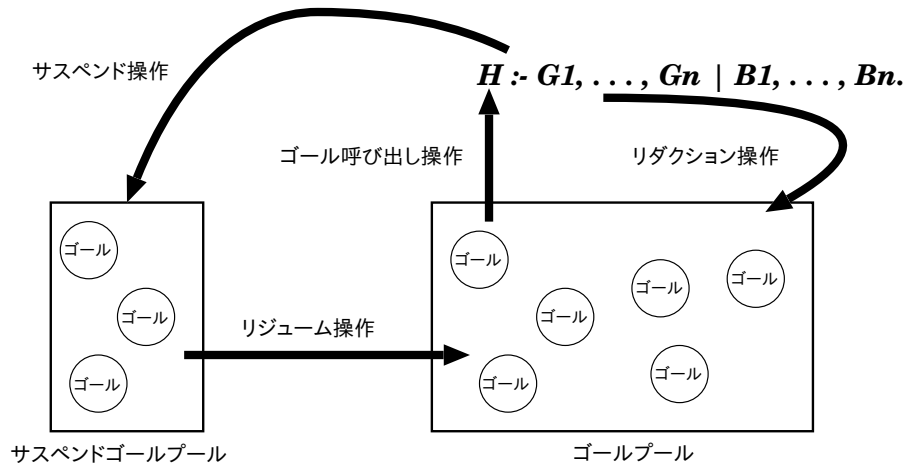


図 2.3 サスペンションメカニズム

サスペンドするプログラムのトレース

図 2.2 のプログラムで, subset の第 3 節を

```
subset(yes,[A|B],Y,Ans) :- true | subset(Ans1,B,Y,Ans), member(A,Y,Ans1).
```

とするとサスペンションが起きる.

このプログラムのトレース結果を以下に示す.

```
% klic -o susp susp.kl1
% susp -t
1 CALL:main:main?
1 REDU:main:main :-
2 0:+subset(yes,[3,1],[1,3,5],_3)
3 1:+io:outstream([print(_3),nl])?
2 CALL:main:subset(yes,[3,1],[1,3,5],_3)?
2 REDU:main:subset(yes,[3,1],[1,3,5],_3) :-
4 0:+subset(_13,[1],[1,3,5],_3)
5 1:+member(3,[1,3,5],_13)?
4 CALL:main:subset(_13,[1],[1,3,5],_3)?
4 SUSP:main:subset(_13,[1],[1,3,5],_3)?
5 CALL:main:member(3,[1,3,5],_63D7)?
5 REDU:main:member(3,[1,3,5],_63D7) :-
6 0:+member(3,[3,5],_63D7)?
6 CALL:main:member(3,[3,5],_63D7)?
6 REDU:main:member(3,[3,5],yes) :-
4 0!+subset(yes,[1],[1,3,5],_3)?
4 CALL:main:subset(yes,[1],[1,3,5],_3)?
4 REDU:main:subset(yes,[1],[1,3,5],_3) :-
7 0:+subset(_29,[],[1,3,5],_3)
8 1:+member(1,[1,3,5],_29)?
7 CALL:main:subset(_29,[],[1,3,5],_3)?
7 SUSP:main:subset(_29,[],[1,3,5],_3)?
```


8 CALL:main:member(1,[1,3,5],_63B8)?	25
8 REDU:main:member(1,[1,3,5],yes) :-	
7 0!+subset(yes,[],[1,3,5],_3)?	
7 CALL:main:subset(yes,[],[1,3,5],_3)?	
7 REDU:main:subset(yes,[],[1,3,5],yes)?	
3 CALL:io:ostream([print(yes),nl])? s	30
yes	
%	

- ゴールがサスペンドする操作をモニターするポートを SUSP ポートと呼ぶ。トレースの 12 行目の、4
SUSP: main:subset(_13, [1], [1, 3, 5], _3) は、ゴール番号 4 番の main:subset(_13, [1],
[1, 3, 5], _3) がサスペンドしサスペンドゴールプールに入れられたことを意味する。
- 18 行目の、
4 0!+subset(yes, [1], [1, 3, 5], _3)
は、ゴール番号 4 番の main:subset(_13, [1], [1, 3, 5], _3) がリジュームし、ゴールプールに
移されたことを示す。

2.3 トレーサの使用法

前章でも述べたように KLIC では、以下の操作の繰り返しによりリダクションを進める。

1. ゴールプールからゴールを 1 つ取ってくる。(ゴール呼びだし操作)
2. ゴール引数の値を、ガード部の規則にしたがってチェックし、チェックに適合した節を 1 つ選んでその
ボディ部を実行し、生成されたサブゴールをゴールプールに入れる。(リダクション操作)
3. 2 で呼び出されたゴールの引数のチェックがサスペンドする時、ゴールをサスペンドゴールプールに入
れる。(サスペンション操作)

KLIC のトレーサで提供されるポートは、前述の CALL ポート、REDU ポート、SUSP ポートと、後で述べ
るリダクションの失敗を扱う FAIL ポートの 4 つである。

トレースモードでコンパイルされたプログラムは、ゴールが上記の 4 つのポートを通過する時に、実行を停
止してユーザからの入力待ちとなったり、そのゴールの表示を行なったりする。

2.3.1 トレーサの出力形式

図 2.4 に各ポートでの出力形式を示す。

CALL ポート、SUSP ポート、FAIL ポートでは、ゴール番号、ポート名、ゴールレコードの内容が表示され
る。右端の?は、ユーザの入力待ちのプロンプトを意味する。

REDU ポートでは、先ずリダクションされるゴールが CALL ポートと同じ形式で表示され、続いて、ゴール
プールにエンキューされるゴール群が表示される。

ゴールプールにエンキューされるゴールは、ゴール番号、その節で一意に決まるサブゴール番号、エンキュー
されるゴールの種別、そのゴールのトレースフラグ、サブゴールの内容、の順に表示される。

エンキューされるゴールの種別は、以下の 4 つの記号で示される。

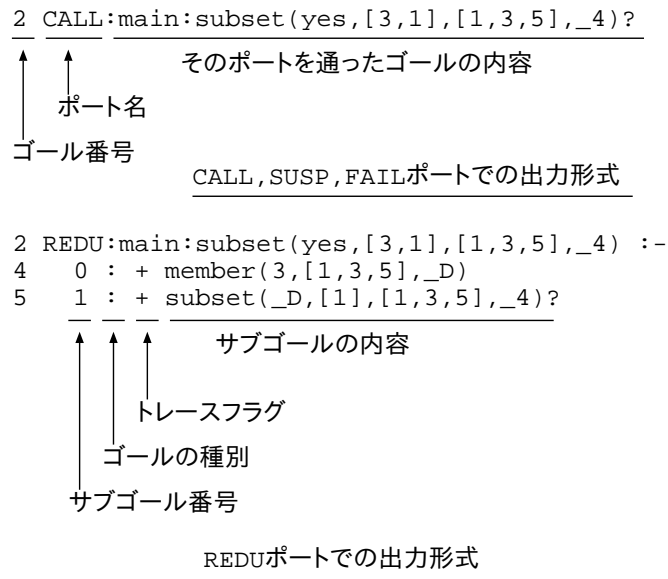


図 2.4 各ポートでの出力形式

ゴール種別	意味
:	通常ゴールのエンキュー
*	プライオリティつきゴールのエンキュー
!	通常ゴールのリジューム
#	プライオリティつきゴールのリジューム

トレースフラグは各サブゴールに付けられるフラグで、トレーサはトレースフラグが **on** であるゴールのみをトレースする。トレーサはトレースフラグが **on** ならば + を、トレースフラグが **off** ならば - を表示する。デフォルトでは、全てのゴールが トレース **on** である。

2.3.2 トレーサの各コマンドの詳細

?が出力された所で、トレーサはユーザのコマンド入力待ちになる。表 2.1 にコマンドの一覧を示し、以下に各コマンドの詳細を述べる。

実行制御コマンド

ここでは、トレースを続けたり中止したりするコマンドについて説明する。

- c, <cr> そのままトレースを続行し、次のポートで止まり、再び入力待ちとなる。
- a トレースを中止する。プログラムはその時点から最後までフリーに実行される。
- s その時点でコマンド入力待ちのポートにあるゴールのトレースフラグを **off** にする。新たにトレース **on** となっているゴールがポートを通過するまで、プログラムはフリーに実行される。
- l スパイされたゴールがポートを通るまで、トレースの表示・入力を抑制する。スパイされたゴールがポートを通ると、トレースモードに戻る。

また以下のコマンドは、サブゴールのトレースフラグの **on/off** を制御するもので、REDU ポートでのみ入

コマンド名	コマンド入力形式	入力可能ポート	コマンドの意味
Continue	<cr>,c	全てのポート	対象となるゴールのトレースを続ける
Abort	a	全てのポート	対象となるプログラムのトレースを止める
Skip	s	全てのポート	対象となるゴールのトレースを止める
Leap	l	全てのポート	spy されたサブゴールが現れるまで, トレースの表示・入力を抑制する
Enable port	E 㐀 ポート名	全てのポート	そのポートを観測する
Disable port	D 㐀 ポート名	全てのポート	そのポートは観測しない
Leash port	L 㐀 ポート名	全てのポート	そのポートでユーザからの入力待ちにする
Unleash port	U 㐀 ポート名	全てのポート	そのポートでゴールの表示だけ行なう
Trace	+ サブゴール番号	REDU ポート	そのサブゴールをトレース対象とする
Notrace	-サブゴール番号	REDU ポート	そのサブゴールをトレース対象外とする
Toggle trace	サブゴール番号	REDU ポート	そのサブゴールをトレースフラグを反転する
Notrace default	n 㐀 述語	全てのポート	その述語をトレース対象外とする
Trace default	t 㐀 述語	全てのポート	その述語をトレース対象とする
Spy	S 㐀 述語	全てのポート	その述語にスパイをかける
Nospy	N 㐀 述語	全てのポート	その述語のスパイを外す
Set Print Depth	pd 㐀 引数	全てのポート	表示する構造の深さの変更
Set Print Length	pl 㐀 引数	全てのポート	表示する構造の長さの変更
Toggle Verbose print	pv	全てのポート	verbose モードへの切替え
Status query	=	全てのポート	現在のパラメータの状況を表示
List Modules	lm	全てのポート	全モジュールの表示
List Predicates	lp	全てのポート	全述語の表示
Help	h,?	全てのポート	全モジュールの表示
Queue	Q	全てのポート	ゴールプール内のゴールの表示

表 2.1 トレーサのコマンド一覧

力可能である.

+ サブゴール番号: そのサブゴールのトレースフラグを on にする.

- サブゴール番号: そのサブゴールのトレースフラグを off にする.

サブゴール番号: そのサブゴールのトレースフラグを反転する.

図 2.2 の部分集合をチェックするプログラムを用いた, ‘c’, ‘<cr>’, ‘a’, ‘s’ コマンドの使用例を以下に示す.
‘l’ コマンドはリープコマンドと呼ばれる. 後で述べるスパイ機能と併せて使用例を示す.

```

% subset -t                                     1
1 CALL:main:main?
1 REDU:main:main :-
2   0:+subset(yes,[3,1],[1,3,5],_3)
3   1:+io:ostream([print(_3),nl])? c           5
2 CALL:main:subset(yes,[3,1],[1,3,5],_3)?
2 REDU:main:subset(yes,[3,1],[1,3,5],_3) :-
4   0:+member(3,[1,3,5],_11)
5   1:+subset(_11,[1],[1,3,5],_3)?
4 CALL:main:member(3,[1,3,5],_11)? s           10
5 CALL:main:subset(yes,[1],[1,3,5],_3)?
5 REDU:main:subset(yes,[1],[1,3,5],_3) :-
6   0:+member(1,[1,3,5],_1C)

```

7	1:+subset(_1C, [], [1,3,5], _3)? -0	
5	REDU:main:subset(yes, [1], [1,3,5], _3) :-	15
6	0:-member(1, [1,3,5], _1C)	
7	1:+subset(_1C, [], [1,3,5], _3)?	
7	CALL:main:subset(yes, [], [1,3,5], _3)?	
7	REDU:main:subset(yes, [], [1,3,5], yes)?	
3	CALL:io:ostream([print(yes),nl])? a	20
	%	

10 行目の ‘s’ コマンドにより、ゴール番号 4 番の member ゴールはトレースされなくなった。14 行目の ‘-0’ コマンドにより、ゴール番号 6 番の member 述語はトレースされなくなった。12 行目の ‘a’ コマンドにより、トレースが中止されプログラムが最後まで実行された。

デフォルトトレースフラグ設定機能

ある特定の述語を指定して、その述語を実行する全てのゴールのトレースフラグをデフォルトで設定することができる。

n 述語 その述語を実行するゴールがデフォルトでトレース off となる。

t 述語 その述語を実行するゴールがデフォルトでトレース on となる。

述語の指定方法には以下に示す 5 通りがある。

指定方法	意味
モジュール名:述語名/引数個数	述語の全情報の明示的な指定
モジュール名:述語名	指定したモジュールの指定した述語名を持つ全ての述語
モジュール名:	指定したモジュールの全ての述語
述語名/引数個数	実行中モジュール内の指定した述語名と引数個数を持つ述語
述語名	実行中モジュール内で指定した述語名を持つ全ての述語

デフォルトトレースフラグ設定の例を以下に示す。

% subset -t	1
1 CALL:main:main?	
1 REDU:main:main :-	
2 0:+subset(yes, [3,1], [1,3,5], _3)	
3 1:+io:ostream([print(_3),nl])? n member	5
Default trace reset on predicate main:member/3	
1 REDU:main:main :-	
2 0:+subset(yes, [3,1], [1,3,5], _3)	
3 1:+io:ostream([print(_3),nl])?	
2 CALL:main:subset(yes, [3,1], [1,3,5], _3)?	10
2 REDU:main:subset(yes, [3,1], [1,3,5], _3) :-	
4 0:-member(3, [1,3,5], _11)	
5 1:+subset(_11, [1], [1,3,5], _3)?	
5 CALL:main:subset(yes, [1], [1,3,5], _3)?	
5 REDU:main:subset(yes, [1], [1,3,5], _3) :-	15
6 0:-member(1, [1,3,5], _1C)	
7 1:+subset(_1C, [], [1,3,5], _3)?	

7 CALL:main:subset(yes, [], [1,3,5], _3)?	
7 REDU:main:subset(yes, [], [1,3,5], yes)?	
3 CALL:io:ostream([print(yes), nl])? s	20
yes	
%	

5 行目で, 述語 `main:member/3` のトレースフラグを `off` にしている. このことによりそれ以降は, 述語 `main:member/3` のトレースは行なわれない.

スパイ制御機能

スパイ制御機能コマンドを以下に示す.

S 述語 指定した述語にスパイを掛ける. 述語名を指定しないと, 現在トレースしている述語にスパイが掛かる.

N 述語 指定した述語のスパイを外す. 述語名を指定しないと, 現在トレースしている述語のスパイが外れる.

実行制御コマンドで説明した '`1`' コマンドは, スパイされたゴールがポートを通るまで, トレースの表示・入力を抑制する.

このため, スパイ機能と '`1`' コマンドを組合わせて使うと, ある特定の述語だけにトレースを掛けてデバッグを進めることができる.

以下にスパイと '`1`' コマンドの使用例を示す.

% subset -t	1
1 CALL:main:main? S member	
Spy point set on predicate main:member/3	
1 CALL:main:main? 1	
4 CALL:main:member(3, [1,3,5], _11)?	5
4 REDU:main:member(3, [1,3,5], _11) :-	
6 0:+member(3, [3,5], _11)?	
6 CALL:main:member(3, [3,5], _11)?	
6 REDU:main:member(3, [3,5], yes)?	
5 CALL:main:subset(yes, [1], [1,3,5], _3)?	10
5 REDU:main:subset(yes, [1], [1,3,5], _3) :-	
7 0:+member(1, [1,3,5], _21)	
8 1:+subset(_21, [], [1,3,5], _3)?	
7 CALL:main:member(1, [1,3,5], _21)?	
7 REDU:main:member(1, [1,3,5], yes)? N member	15
Spy point reset on predicate main:member/3	
7 REDU:main:member(1, [1,3,5], yes)?	
8 CALL:main:subset(yes, [], [1,3,5], _3)?	
8 REDU:main:subset(yes, [], [1,3,5], yes)?	
3 CALL:io:ostream([print(yes), nl])? s	20
yes	
%	

2 行目で, 述語 `main:member/3` にスパイを掛けている. 3 行目のゴール `main:main` において, '`1`' コマンドを実行することにより, `main:member/3` を実行したところからトレースが始まる.

ポートの制御

ここでは、ポートでの入出力を制御するコマンドについて述べる。

ポート制御コマンドは、‘コマンド_ポート名’の形式で入力する。ポート名の指定方法は以下に示す通りである。

ポート名	指定方法 (大文字, 小文字の区別なし)
CALL ポート	c, call
REDU ポート	r, redu, reduce
SUSP ポート	s, susp, suspend
FAIL ポート	f, fail
全ポート	a, all

またポート制御コマンドには、以下の4つがある。

- D ポート名 対象となるポートでのトレースを行なわない (ポートを Disable 状態にする.)
- E ポート名 対象となるポートでの表示・入力を再開する (ポートを Enable 状態にする.)
- U ポート名 対象となるポートでは表示のみを行ない、入力待ちにならない (ポートを Unleash 状態にする.)
- L ポート名 対象となるポートでコマンド入力待ちになる (ポートを Leash 状態にする.)

デフォルトでは、全てのポートが Enable 状態, Leash 状態である。各コマンドの使用例を以下に示す。

```
% subset -t 1
1 CALL:main:main? U redu
1 CALL:main:main?
1 REDU:main:main :-
2 0:+subset(yes,[3,1],[1,3,5],_3) 5
3 1:+io:ostream([print(_3),nl])
2 CALL:main:subset(yes,[3,1],[1,3,5],_3)?
2 REDU:main:subset(yes,[3,1],[1,3,5],_3) :-
4 0:+member(3,[1,3,5],_11)
5 1:+subset(_11,[1],[1,3,5],_3) 10
4 CALL:main:member(3,[1,3,5],_11)? L redu
4 CALL:main:member(3,[1,3,5],_11)?
4 REDU:main:member(3,[1,3,5],_11) :-
6 0:+member(3,[3,5],_11)?
6 CALL:main:member(3,[3,5],_11)? D call 15
6 CALL:main:member(3,[3,5],_11)?
6 REDU:main:member(3,[3,5],yes)?
5 REDU:main:subset(yes,[1],[1,3,5],_3) :-
7 0:+member(1,[1,3,5],_21)
8 1:+subset(_21,[],[1,3,5],_3)? a 20
yes
%
```

2行目で REDU ポートを unleash しているので、8-10行目の REDU ポートでは、ユーザの入力待ちになら

ない。その後、11 行目で REDU ポートを leash したので、13 行目以降は REDU ポートで入力待ちになった。15 行目で CALL ポートを disable しているので、以下のトレースでは CALL ポートのトレースは表示されない。

その他補助機能

その他、以下のトレース環境の設定・表示機能がある。

pd dep 表示する構造の深さを '*dep*' に設定する。
pl len 表示する構造の長さを '*len*' に設定する。
pv verbose モードフラグのトグルを行なう。
 verbose モード on 状態では、ゴールをサスペンドさせている変数が表示される際に、そのゴールも一緒に表示される。
= 現在のパラメータの状況を表示する。
lm プログラムを構成している全モジュールを表示する。
lp プログラムを構成している全述語を表示する。
Q ゴールプール内のゴールを全て表示する。
h ヘルプ画面を表示する。
? h コマンドと同じ。

2.4 実行時エラーの表示とトレース

2.4.1 節の失敗 (failure)

呼び出されたゴールを、ガード部の規則にしたがってチェックしたが、どの節の条件にもマッチしなかった時は、そのゴールは失敗する。以下に失敗するプログラムの例とその実行トレースを示す。

```
:-module main.

main:- true | gen(X), con(X).

gen(X):- true | X = a.

con(b):- true | true.
con(c):- true | true.

% g_fail -t
1 CALL:main:main?
1 REDU:main:main :-
2 0:+gen(_4)
3 1:+con(_4)?
2 CALL:main:gen(_4)?
2 REDU:main:gen(a)?
3 CALL:main:con(a)?
5
!!! Reduction Failure !!!
4 FAIL:main:con(a)? a
10
%
```

節が失敗したため 10 行目に失敗した節 4 `FAIL:main:con(a)?`が表示された。この時は図のように、`'a'` コマンドでトレースを中止することが推奨される。

2.4.2 永久中断

あるゴールがサスペンドしたが、その変数が具体化されないままゴールプールが空になると、永久中断状態となる。

以下に永久中断するプログラムの例と実行トレースを示す。

```
:-module main.

main:- true | gen(X),con(X).

gen(X):- true | true.

con([A|B]):- true | true.

% psus -t
1 CALL:main:main?
1 REDU:main:main :-
2 0:+gen(_4)
3 1:+con(_4)?
2 CALL:main:gen(_4)?
2 REDU:main:gen(_4)?
3 CALL:main:con(_4)?
3 SUSP:main:con(_4)?
1 perpetually suspending goals found
!!! Perpetual Suspension Detected !!!
3 PSUS: main:con(_B)? a
%
```

ゴール番号 3 番の `main:con(_4)` がサスペンドするが、リジュームされないので永久中断した。このため、10 行目に `!!! 1 suspending goal(s) !!!`と表示された。

第 3 章

KLIC のインストール法

この章では、KLIC システムのインストール方法について説明する。

3.1 KLIC のコンパイル方式

まず、インストール法の理解のため、KLIC システムが KL1 プログラムをどのようにコンパイルして実行するのかを説明する。

システムは以下のみつつのモジュールからなる。

- KLIC コンパイラ
- KLIC データベース管理
- KLIC 実行時システム

KL1 のプログラムは KLIC コンパイラで C プログラムに翻訳される。その時に“ファイル名.ext”という名の、プログラム中に出てくるアトムやファンクタの情報を持つファイルも出力する。後にデータベース管理プログラムが、一緒にリンクするプログラムの“.ext”ファイルの情報を統合して、“atom.h”、“funct.h”、“atom.c”、“funct.c”、“predicates.c”の諸ファイルを作成する。

次に、コンパイル結果の C プログラムを C コンパイラで、KLIC システムの提供するヘッダや“atom.h”、“funct.h”と共にコンパイルする。“atom.c”、“funct.c”、“predicates.c”のファイルもコンパイルし、実行時システムとリンクして実行可能ファイルを作成する。

コンパイル、データベース管理、リンクの一連の作業は“klic”という名のドライバプログラムが制御する。この“klic”は“cc”と“make”を合わせたような役目を果たす。“Cc”は C のプリプロセッサ、コンパイラ本体、リンクを制御するが、“klic”は KL1 から C へのコンパイラ、C コンパイラ、データベース管理プログラム、そしてリンクを制御する。“Make”はファイルの日付を調べて必要なコンパイルだけを行なうが、“klic”も同様の機能を持つ。

3.2 必要とする環境

KLIC はその部分毎に各々以下のようなソフトウェア環境を必要とする。

- KLIC コンパイラ ... これは KL1 自身で書かれている。KLIC の配布キットにはこれを既に C にコンパイルしたコードが含まれている。
- KLIC 実行時システム ... これは C で書かれている。ANSI C コンパイラ、さらに GNU C コンパイラであればより効率のよいコードが出力されることを念頭に書かれているが、K&R コンパイラであっても問題なく利用できる。

- KLIC データベース管理 ... これも KLIC 実行時システムと同様 C で書かれている。

また、オペレーティングシステムとしては通常の UNIX (Berkeley 版, ATT 版を問わず) であれば、動作することを念頭に書かれている。

3.3 インストール方法

KLIC システムのインストールは以下のように行なう。

3.3.1 システムの入手

まず、KLIC システムのソースコードを入手する。最も手軽な方法は、ICOT の IFS (ICOT Free Software) アーカイブ ([ftp.icot.or.jp](ftp://ftp.icot.or.jp)) より匿名 ftp する方法である。この際、ftp を binary (image) mode にすることを忘れずに。

3.3.2 展開

入手したシステムをディレクトリに展開する必要がある。展開のためには、展開を行なう先のディレクトリ (このディレクトリを以下では KLICDIR と呼ぶ) で、`zcat`, `uncompress`, `tar` などを使って行なう。

```
% zcat ‘‘ftp してきたファイル’’ | tar xf -
```

この結果、全てのファイルはディレクトリ KLICDIR 以下に展開される。

3.3.3 コンフィギュレーション

KLIC をインストールするシステムに合わせて、システムのパラメタなどを設定する。このためにはシェルスクリプト `KLICDIR/Configure` を走らせて、会話的に質問に答えていけば良い。種々のライブラリ関数などが利用できるかどうか自動的に判定するようになっている。

コンフィギュレーション・スクリプトへの応答は、大部分の項目については通常デフォルト値で良いが、特に注意すべき項目として、以下のものがある。

- 並列処理系のインストールを行なうかどうか。並列処理系のインストールには並列処理ライブラリなどの環境設定が必要である。最初は並列処理系はインストールしないことを奨める。この文書では並列処理系のインストールについてまでは触れない。
- 非同期入出力のためのレディ・シグナルが使えるかどうか。これはシステムに依存するが、ソケットの非同期入出力を利用しないのなら『使えない』と答えておいてもよい。
- インストール先のルートとなるディレクトリ。デフォルトは “`/usr/local`” としているが、システム管理者以外の場合は適当な自分のディレクトリを指定する。多くの場合ホームディレクトリを指定すれば良い。

コンフィギュレーションの結果、以下のファイルが生成される。

```
KLICDIR/Makefile
KLICDIR/include/klic/config.h
```

3.3.4 コンパイル

KLICDIR で, `make` コマンドによりコンパイルする.

```
[frame=single]
% make
```

これによってコンパイラのドライバ, KL1 から C へのコンパイラ, データベース管理プログラム, 実行時システムのコンパイルが行なわれる.

ここでエラーがおきた場合には設定に問題がある可能性があるので, もう一度コンフィギュレーションが正しく行なったか確認すること.

3.3.5 確認

KLIC システムが正しくコンパイルされたかどうか, テストプログラムを実行することで確認できるようになっている. KLICDIR で “`make tests`” とすることで, 基本的なテストプログラムがコンパイル, 実行され, 結果が確認される.

```
% make tests
```

3.3.6 インストール

ここまで問題なければ, KLIC システムは正しく作成されているので, トップディレクトリで “`make install`” とすることにより, Makefile で指定したディレクトリに必要なファイルをコピーする. これでインストール終了である.

```
% make install
```

3.3.7 掃除

インストール後にインストール用に作った作業ファイルなどを消去するには以下を行なう.

```
% make distclean
```

3.4 トラブルが起きたら

トラブルがおきた際には, 以下にメールで問い合わせを.

`klic-bugs@icot.or.jp`

問い合わせの際には以下の情報を添付していただければ, 問題の迅速な解析に役立つだろう.

- 使用システムの機種, OS と版名.

- Make のログ.
- コンフィギュレーションの結果. すなわち以下のファイルの内容.

KLICDIR/Makefile

KLICDIR/include/klic/config.h

付録 A

簡易並列実装版の使い方

本章は、現在 (1995 年 9 月現在) リリースされている簡易並列版の使い方について説明する。なお 内容は非常に暫定的なものが多いので、実際に使用される際はマニュアルを読み直す必要がある。

KLIC の 2 種類の並列実装が、配布される KLIC に含まれている。このうち現在の配布に含まれる分散メモリ実装と呼ばれる方式には PVM3.3 を利用した版が存在し、これは簡易並列実装版とも呼ばれ、同種のマシンがネットワーク結合された分散環境でも利用することができる。この版の簡単な利用法について説明したいと思う。

この版は PVM に基づいているが、異種構成をサポートしてない。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ないことに注意されたい。

A.1 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ある。

-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。

なお -dp の代わりに -shm オプションを付けると共有メモリ並列実装版向けのコードが生成される。

A.2 分散 KLIC のプログラム実行

A.2.1 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る。

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る。

- 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) の数を指定する.
- e バッチ転送モードに切り替える. 通常, KLIC は要求時にプロセッサ間でデータ構造を転送する. ネストしたデータ構造は通常, 1 度に 1 レベルずつ転送する. バッチ転送では, ネストしたデータ構造を 1 度に転送する. これをモードでは, ある種のプログラムはより効率的に実行する. しかし, 別のプログラムでは性能を劣化させることもある.
- n 実行時の統計情報を表示する.

A.3 ランタイムモニタを使った実行

並列プログラムの開発では, デバッグングの他チューニングも必要となる. プログラムのチューニングのためにビジュアルにノードの稼働状況を見るツールは非常に威力を発揮する. そのために KLIC ではランタイムモニタと呼ばれるツールを用意する. このツールの利用方法について簡単に述べる.

A.3.1 ランタイムモニタの起動

ここでは PVM を利用したランタイムモニタについて説明する. 起動は次の通り.

`klicmon` 最大ノード数

すると図 A.1 のウィンドウが表示される (図の場合は, ノード数 12 とした). この時 rmonitor の PVM におけるタスク ID を覚えておくこと.



図 A.1 ランタイムモニタの初期状態

A.3.2 ランタイムモニタの利用

ランタイムモニタを利用する. 例えば `a.out` をノード数 12 で実行したとする. またランタイムモニタの PVM におけるタスク ID が 40004 とする. `a.out` を以下の通り実行することで各ノードの稼働状況が図 A.2 のように見ることができる.

```
a.out -p 12 -rmon 40004
```

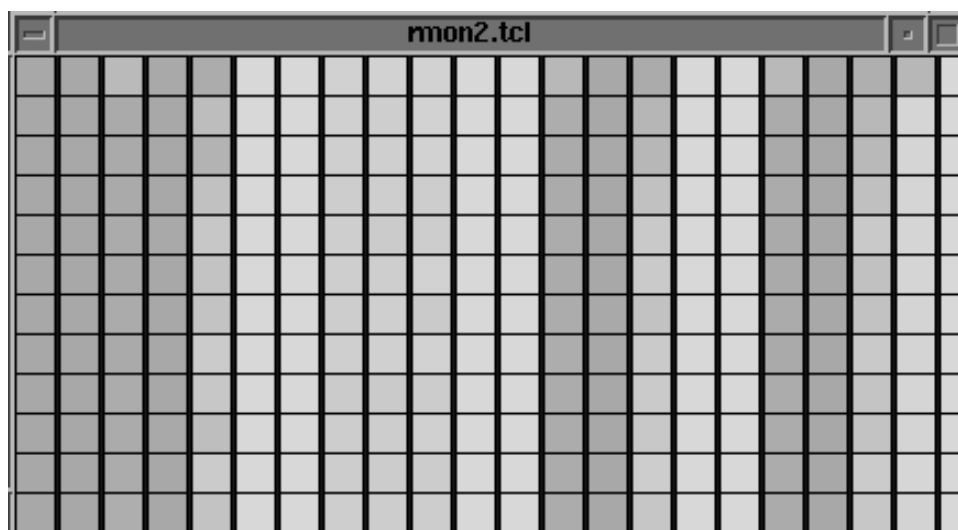


図 A.2 稼働状況の表示

この図の横軸は右方向への時間軸となる。画面が一杯になった場合は、横スクロールが起こる。縦軸は各ノードを示す。上から下に昇順に並ぶ。

A.4 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る。
- スパイの指定は、指定した計算ノードの内部だけに効果がある。
- 永久中断ゴールの検出は行なっていない。
- PVM は版によってはハングアップすることがある。念のため PVM のコンソールから稼働しているかどうかを時々確認して欲しい。