第1章

MiLogとは

MiLog は，私（福田）が，学部生時代のときから温めていたアイデアを，修士１年の終わりごろから少しずつ形にしていき，その後も，周囲の叱咤激励を受けながら，いろいろな人から吸収したアイデアや新しい機能を追加し，今日まで成長させてきたものである．

MiLog は，マルチエージェント技術・モバイルエージェント技術を駆使して，新しいアイデアに基づくフレームワークやアプリケーションを構築するための，メタフレームワークである．MiLog では，マルチエージェント技術を効果的に利用できるように，エージェント１つ１つに Prolog ベースの言語処理系を備え，アクティブオブジェクトモデルに基づいてエージェント間の通信プログラムを比較的容易に作成できるようにしている．また，モバイルエージェントの移送（マイグレーション）機能としてはもっとも強力な，強いモビリティを比較的小さな通信オーバヘッドで利用できるようにしており，モビリティのアプリケーション開発への利用を行いやすくしている．MiLog は，特に Web に関連したアプリケーションの構築支援にフォーカスしており，Web に関連した強力なライブラリ群を備えている．MiLog は，Prolog に習熟した研究者のための，研究アイデアを具体的なアプリケーションにしていくためのプラットフォームとして，また，Prolog やエージェントに興味を持った学習者が，楽しみながらアプリケーションを実際に作成しながら学んでいくのに役立つものと，期待している．

MiLog は，次のような開発方針に沿って作られている．

- アプリケーション第一主義：MiLog の開発に当たって，アプリケーションの試作・構築に実際に使えるものであることを最も重視している．MiLog への機能の実装・改良では，アプリケーションの実現に必要となる機能をもっとも優先的に行っている．逆に，アプリケーションの構築にとって重要にならない限り，言語処理系としての機能追加（特に，既存の処理系にある機能と同等のものの追加）は，極力行わないようにしている．これは，機能の後追いをしてもあまりよいことはない，という経験則による．

- マルチプラットフォーム対応：MiLog は，できるだけ多くの環境 (OS) で利用できるように，Java 言語上に実装されており，環境への依存性を最大限排除しながら，可能な範囲で環境固有の問題に対応を試みている．例えば，自宅で Macintosh を使いながら研究室で Linux を利用しているような大学院生には，いつでもどこでも研究用のプログラムが作成できるという点で，有益であろう．逆に，特定の機種に対象を絞り込めば，もう少し使いやすくしたり，機能を豊富にすることが可能であっても，マルチプラットフォームで使う場合に問題になるような点については，あえて機能追加や改良を行っていない部分もある．

- 継続性のあるメンテナンス：私自身が，MiLog をできるだけ末永く使っていきたいと考えているせいもあって，MiLog は今まで，ゆっくりとしたペースではあるものの，継続的にメンテナンスされてきている．幸いなことに，私自身は学部１年のころからアルバイトで商用ソフトウェアのプログラムを書いていた経験があり，ソフトウェアのメンテナンスの泥臭い部分なども含めて，多くのノウハウを学んでこれた．この経験を最大限に生かしている．MiLog 内部の設計は，メンテナンス性を最優先した設計になっており，その後，必要に応じて，アプリケーション構築の要求にあうように，性能面などでの改善を行ってきている．メンテナンスにかかる時間的負担を最小限にするために，現在では MiLog 内部に MiLog(Prolog) で作られたコードが多く含まれており，これ自身が MiLog の信頼性を底上げする１つの要因になっている．また，この過程を通じて，Prolog という言語の持つすばらしい特長の１つ，高い記述力からくるプログラムの記述長の短さと，それに起因するプログラムのメンテナンスのしやすさを肌身で実感している．

- 自前主義：これがいいことなのか悪いことなのかは異論のあるところだが，MiLog では可能な限りクラスライブラリには頼らず，作者がその技術を根本から丁寧に理解しながら，自前でライブラリを構築している．また，開発当初はライブラリを用いた部分であっても，その後に順次必要に応じて自前のものに置き換えている．実際に，HTTP 通信制御や Prolog 言語インタプリタなどは，完全にフルスクラッチから仕上げたもの（つまり，ゼロから自分で作ったもの）であり，環境 (Java VM やクラスライブラリの実装の違い) の影響を受けにくいようになっている．こうすると，機能の実装自体（あるいは，そのことを論文にするまでにかかる時間といったほうがピンと来るかもしれない）には時間がかかるが,「既存のライブラリではできないこと」ができるようになる可能性は，逆に高くなると考えている．また，これがマルチプラットフォーム性の実現やメンテナンス性の向上に実際に非常に役立ってもいる．何よりも，作っ

ている本人にとっては，特定のクラスライブラリの API や設定方法を覚えたとしても後々に役立つ「知恵」にはつながらないことが多く，自前で理解しながら機能を作っていくことが,（自己満足ではあるが）非常によい勉強になっている．

MiLog のこのような開発方針から，単純に言語処理系として見た場合には，不完全な部分や未完成な部分が見え隠れしているのも事実である．今までに見つかってきている，いろいろな課題・挑戦事項については，時間はかかるものの，アプリケーション構築にとって重要なものから順次改善していく予定でいるので，それまで温かく見守っていただければ幸いである．また，温かく見守るだけでは生ぬるいとお考えの諸氏は，必要に応じて，作者らを叱咤していただきたい．

2005.8.4 福田 直樹

第2章

MiLog 入門（概略）

本章では，MiLog を初めて使うプログラマにとって必要な情報が整理されている．最初に，MiLog の動作環境について説明する．次に，MiLog のインストール方法について述べる．その後，MiLog を使ったプログラムの開発を体験するためのガイドを示す．

なお，本書を読み進めるにあたり，最低限必要な知識として，Prolog の基礎知識，Java アプリケーションの起動しかた，および PATH や ClassPath の概念が必要となる．また，できれば並行プログラミングの基礎知識と，(もし Web を扱うプログラムを作るなら）HTML や HTTP に関する知識があるとよい．これらの点については本書では解説しないので，良書を読者自身で探して読んでほしい．

2.1 動作環境

MiLog を最低限動作させるためには，JDK1.1 以上が動作する環境（計算機プラットフォーム）が必要である．また，できるだけ高速な JIT コンパイラが動作することが望ましい．ただし，内蔵の SSL サーバ機能などを使う場合には，JDK1.4 以降が必要となる．

現状では，以下の環境において動作が確認されている．これら以外でも，条件を満たしていればたいていの環境でなら動くはずである．

- Mac OS X(10.0 以降，執筆時点で Tiger まで動作確認済[1])
  JDK1.4.1/1.4.2/5.0
- Windows 98/2000/XP
  Sun JDK1.3.1/1.4.2/1.5.0_03 以降 (64bit 版にも対応)
  BEA JRockit(R) JDK1.4.2/5.0
  IBM JDK1.4/1.5
- Linux(Debian3.1/SuSE9.2/Gentoo)
  Sun JDK1.3.1/1.4.2/1.5/BEA JRocket(R) JDK1.4/5.0
- Solaris x86
  Sun JDK1.4.2/1.5.0

なお，環境によっては，使用可能なスタック／ヒープメモリ量の制限などからモバイルエージェント機能が正常に動作しない場合があるが，そのときは JDK のバージョンを変えて再確認すると良い．

なお，現時点で，一部のバージョンの Windows 用 JDK[2]では，JDK そのもののバグが原因で，文字入力や操作上の不具合などが確認されているケースがあるので注意されたい．

2.2 インストール方法

ここでは MiLog のインストール方法について説明する．

2.2.1 実行に必要なファイル

MiLog の起動に必要なファイルは，Java の実行環境と，以下の 2 つのファイルである．

- weblog2.jar
- agentMonitor.jar

[1]Mac OS Classis と Mac OS X 10.2 以前で使う場合には，専用にビルドされたバージョンの MiLog を使うと動作する．
[2]1.4 系だと，1.4.1 以前．これらは GUI から _ (アンダースコア) 記号が入力できない不具合がある．1.5 系だと 1.5.0_02 以前．これらには，write/1 等による出力結果が自動でスクロールされない不具合がある．

ただし，エージェントモニタ機能を使用しない場合には，agentMonitor.jar は必要ない．動作環境によっては，専用のパッケージが用意してある場合もある．

（注意）起動時のパスワード入力について
July 8 2005 以降のバージョンでは，起動時にデフォルトでパスワードを聞くダイアログが開かれる．（このダイアログがウィンドウの裏側に隠れてしまって，MiLog がうまく起動しないように見える時もあるので注意！）ここで入力するパスワードは，MiLog プラットフォーム間通信につかう共通鍵で，同一の鍵を入力した MiLog プラットフォーム間でのみ，エージェント間通信やエージェントの移動などが行えるようになっている．（なお，この鍵の管理はではあまり安全ではないので，他で使っている大切なパスワードなどを間違って入力しないように注意すること．）

2.2.2 Macintosh 上で利用する場合

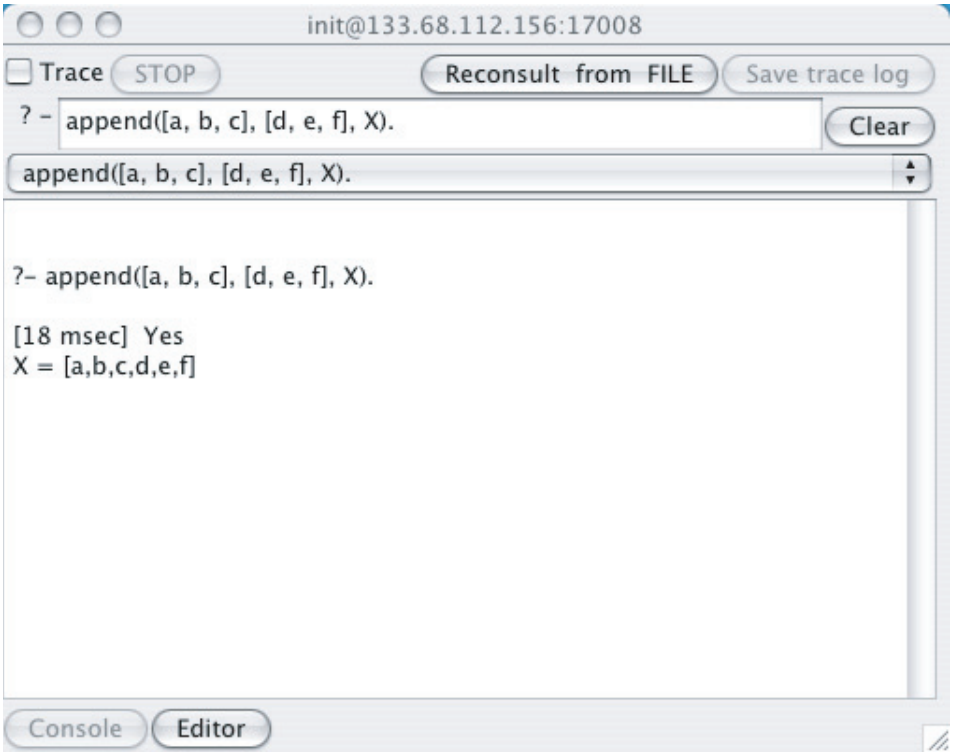
1. MiLog のホームページから Macintosh 用の MiLog パッケージをダウンロードして，適当なフォルダに展開する．

2. 展開されたフォルダにある，MiLog アイコンをダブルクリックして起動する．[3]

3. MiLog が起動するまで少し待つ．すると下のようなウィンドウが出てくるので，例として append プログラムを実行してみる．

   ?- の横にあるテキストフィールドに append([a,b,c],[d,e,f],X). と入力してリターンキーを押してみる．すると，下のテキストエリアに結果が表示される．

[3]Mac OS X の場合，Unix 上での利用方法にならって，java コマンドを使って起動することももちろんできる．



Unix/Linux 上で利用する場合

1. MiLog のホームページから最新の JAR ファイル (weblog2.jar) をダウンロードする．**(このときに，weblog2.jar というファイル名を変更しない．)**

2. java をコンソールから起動 (ダウンロードした jar ファイルをクラスパスに追加して，' weblog.Boot' クラスを起動．なお，ここの例では，古いシステムで使われていた jre という実行専用の java コマンドを使用しているが，最近の Java であれば，java コマンドを普通に使えばいい．なお，-cp はクラスパスに jar ファイルなどを追加するためのオプションである．)

3. MiLog が起動するまで少し待つ．すると下のようなウィンドウが出てくるので，例として append プログラムを実行してみましょう．

   ?- の横にあるテキストフィールドに append([a,b,c],[d,e,f],X). と入力して Enter キーを押してみる．すると，下のテキストエリアに結果が表示される．

Windows 上で利用する場合

1. MiLog のホームページから最新の JAR ファイル (weblog2.jar) をダウンロード．**(このときに，weblog2.jar というファイル名を変更しない．)**

2. java をコンソールから起動

   昔の Windows 上には標準で Micorsoft 製の Java 実行環境が入っているので，コマンドプロンプトから

   ```
   %C:\> set CLASSPATH=.\weblog2.jar
   %C:\> jview weblog.Boot
   ```

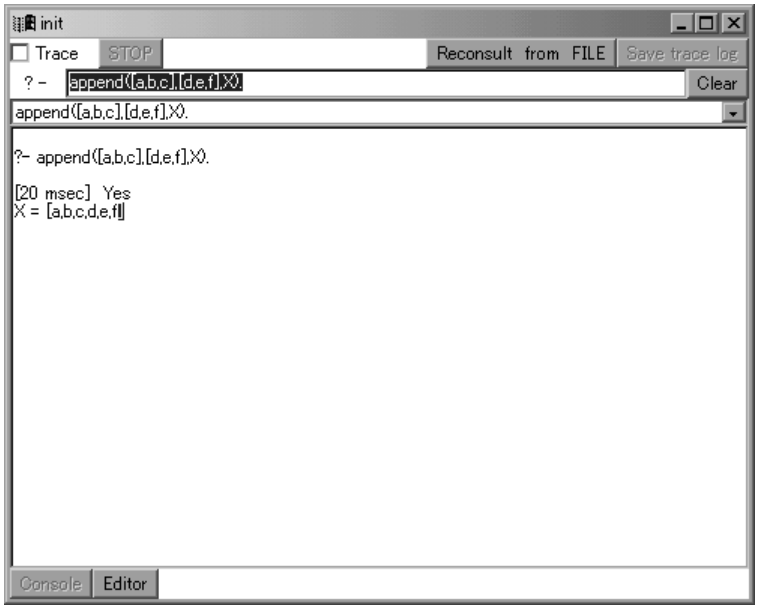
   とすると，MiLog を起動できたが，最近は入っていないので，別途，Sun の JDK などをダウンロードしてインストールしておく．java コマンドに PATH が通っていれば，

   ```
   %C:\> java -cp .\weblog2.jar;.\agentMonitor.jar weblog.Boot -Xmonitor
   ```

   のようにして起動できる．(パスの区切り文字が ;(セミコロン) な点に注意．これは，たとえ Cygwin を使っている場合でも変わらない．)

3. MiLog が起動するまで少し待つ．すると下のようなウィンドウが出てくるので，例として append プログラムを実行してみる．

   ?- の横にあるテキストフィールドに append([a,b,c],[d,e,f],X). と入力して Enter キーを押してみる．すると，下のテキストエリアに結果が表示される．



2.2.3 起動時のオプション

MiLog の起動時に，以下のコマンドラインオプションが利用できる．

- `-server <PORT>`
  MiLog 内蔵の WWW Server 機能で使用するポート番号を指定する．このオプションを指定しない場合には，自動的に 17008 番が使用される．off を指定すると，内蔵 WWW Server 機能が Off になる．（なお，この内蔵 WWW Server 機能は MiLog プラットフォーム間通信にも利用されているので，この機能を Off にすると，エージェントの移動やエージェント間通信機能に支障が出る場合があるので注意.）

  例 1：ポート番号を 17007 番に指定する
  ```
  java weblog.Boot -server 17007
  ```

  例 2：内蔵 WWW Server 機能を off にする
  ```
  java weblog.Boot -server off
  ```

- `-autoport`
  このオプションを指定すると，サーバのポート番号として，デフォルトのポート (17008) か，それより上で，未使用のポート番号をスキャンする．デフォルトのポート番号は，-server オプションで別の値に指定することができる．これで，単純に MiLog を 2 回起動すれば，2 つの異なるポートに別々に MiLog が立ち上がるようになる．デバッグ等に利用するのがよい．なお，意図しないポート番号で MiLog が動作してしまうことを防ぐために，デフォルトではこの機能は Off になっている．

- `-agents <AGENT> [<AGENT>...]`
  起動時に自動的に生成するエージェントを指定する．起動オプションは -agents エージェント名 1 エージェント名 2 ... のように指定する．

  ```
  java weblog.Boot -agents agent1 agent2 agent3
  ```

  この例では，MiLog の起動後に自動的に agent1 agent2 agent3 という 3 つのエージェントが生成され，それぞれに agent1.mlg agent2.mlg agent3.mlg というプログラムが（もしカレントディレクトリに置い

てあれば）自動的に reconsult される．-agents オプションを指定した場合，init エージェントは生成されない．-agents オプションは，かならず末尾で指定する．例えば，-agents agent -Xmonitor と指定すると，-Xmonitor という名前のエージェントができてしまうので，注意する．

- -autorun <AGENT>
起動時に，(-agents でエージェントの生成が終了したあと）指定したエージェントに対して ?-run. という問い合わせを行う．MiLog 起動時に自動的にプログラムを実行したい場合に便利である．例えば，

```
java weblog.Boot -autorun agent1 -agents agent1 agent2 agent3
```

のように使う．

- -Xmonitor
エージェントモニタを使用する場合にこのオプションを指定する．エージェントモニタを使用する場合には，このオプションと合わせて，起動時に agentMonitor.jar をクラスパスに追加しておく．例えば，

```
java -cp ./weblog2.jar:./agentMonitor.jar weblog.Boot -Xmonitor
```

のように使う．

- -XmonitorOFF
起動時にエージェントモニターの表示を Off にする場合にこのオプションを指定する．起動時にはエージェントモニターが表示されないが，debug_macro_setMonitorVisible をメタエージェントから実行することにより，エージェントモニタを表示する．

- -proxy <PROXY>
MiLog が外部の WWW Server へアクセスするときに使用する Proxy Server のアドレスを指定する．例えば，学外の WWW Server へアクセスする場合に proxy.XXXX.XXXXX.ac.jp の 8080 ポートを Proxy Server に指定する必要がある．これには，次のようにオプションを指定する．

```
java weblog.Boot -proxy proxy.XXXX.XXXXX.ac.jp:8080
```

Technology Preview Version May.19 2000 以降では，デフォルトで Proxy Server が使用されない設定になっている．

- -silent
MiLog をサーバーモードとして起動する．このモードで起動すると，エージェントウィンドウが表示されず，そのかわり標準出力にエージェントの移動記録などが出力される．なお，-Xmonitor オプションとの併用はできない．また，かならず-passwd オプションを併用してパスワードファイルを指定する必要がある．

- -password <PASSWORD>
MiLog プラットフォーム間の認証に用いるパスワードファイルを設定する．MiLog では，セキュリティのためにパスワードファイルが同じプラットフォーム間のみで通信やエージェントの移動を行うことができる．パスワードファイルは，MiLog のメタエージェントから述語 generatePassword/0, 1 を使用することによって作成できる．(注：-password 指定を省略した場合，Technology Preview Version July. 8 2005 以降のバージョンでは，使用するパスワードを起動時にダイアログで聞いてくるようになっている.）ユーザのパスワードファイル password.key を MiLog で利用するには，次のようにオプションを指定する．

```
java weblog.Boot -password password.key
```

- `-proxypassword <PASSWORD>`
  MiLog の proxyAgent 機能を利用する際に，Proxy 利用時のパスワードファイルを指定する．proxyAgent 機能を用いることにより，MiLog を Web Proxy として利用できるようになるが，利用時に Web ブラウザから Proxy 認証のパスワードを聞かれる．その際に答えるパスワードを指定する．パスワードファイルは-password オプションと同じ方法で生成する．このオプションを省略した場合，デフォルトのユーザ名とパスワードとして，ユーザ名：milog，パスワード：milog が使用される．ユーザのパスワードファイル password.key を MiLog の proxyAgent 機能で利用するには，次のようにオプションを指定する．

  ```
  java weblog.Boot -proxypassword password.key
  ```

- `-Xposition <WxH+X+Y>`
  エージェントモニタの表示位置を指定する．例えば，400x300 の大きさで，(200,100) にエージェントモニタの左上端がくるようにするには次のよう指定する．

  ```
  java weblog.Boot -Xmonitor -Xposition 400x300+200+100
  ```

  なお，まだ '-'(X window でいうところの，画面の右下端からの座標指定) は使えないので注意する．

- `-autocgi`
  MiLog で HTTP リクエストをエージェントにリダイレクトする際に，そのエージェントがまだ存在しない場合，エージェントを自動生成する．例えば，

  ```
  http://localhost:17008/agent1?message=hello
  ```

  というリクエストを受け取った場合，agent1 というエージェントが生成され，htdocs/ フォルダ内にあらかじめ置いておいた agent1.mlg というプログラムがロード[4]されたあと，そのエージェントにリクエストがリダイレクトされるようになる．**注意:エージェント名にフォルダ名がつく場合（例えば /myfolder/myagent）の動作はまだ調整中であるため，かならずドキュメントルート（例えば /myagent）でエージェント名を指定する．**

- `-use_sslserver`
  このオプションを起動時につけると，SSL モードで起動する．(なお，SSL の利用には，JDK1.4 以降が必要)．SSL モードでは，エージェントの移動プロトコルや内蔵 WWW Server 機能に HTTPS を利用するようになる．[5]

- `-Xyield`
  このオプションを起動時につけると，個々のエージェントが処理に使う CPU 時間を控え目にして，GUI の操作等に CPU パワーを使えるようする．主に，システムに大きな負荷をかけるような処理を行う場合に，操作感が向上する．(いくつかのエージェント上で loop,fail. のような問い合わせを実行してみると，効果がわかる．ただし，単体のエージェントでの処理能力はかなり低下する.)

- `-relay <ADDRESS>`
  ADDRESS で指定した MiLogServer を中継サーバとして利用する．詳細は章 4.4 を参照すること．

- `-uniquename <UNIQUENAME>`
  MiLogServer を UNIQUENAME で指定した名前で，中継サーバを用いた通信で利用する．詳細は章 4.4 を参照すること．

[4]もしも，このとき agent1.mlg でなくて agent1.mip という (.mip で終わる) ファイルだった場合，MiPage プログラムとして認識され，MiLog コードに自動コンパイルされたものが自動で reconsult されるようになっている．

[5]このとき，Web ブラウザから内蔵 WWW Server 機能を介して Web ページにアクセスするための URL には (たとえポート番号が 80 であっても) ポート番号の指定を省略してはいけないので注意．HTTPS では，デフォルトのポート番号は 80 ではない．ちなみに，このとき内蔵 WWW Server で提供されるすべてのページが HTTPS モードになってしまうのは仕様であり，今のところ変更の予定はない．もし一部のページを http モードでアクセスさせる必要がある場合には，MiLog を異なるポート番号で同時に複数起動して，それぞれを HTTP モードと HTTPS に使い分けてほしい．

- `-debug`
  このオプションを指定すると，MiLog インタプリタの最適化レベルを 1 つ下げ，デバッグ時にプログラムのトレースが多少見やすくする．ただし，プログラムの実行効率は下がる．

- `-noAnimation`
  このオプションをつけて MiLog を起動すると，エージェント移動時のアニメーション処理を省略する．

2.3 開発環境を使ってみる

ここでは MiLog の開発環境の使い方を例を用いて説明する．Macintosh 上の画面を使って説明する．**（この章は，現在改訂中... 差し換え予定.）**

2.3.1 MiLog のプログラムを書くには?

MiLog にはプログラムエディタが内蔵されている．Editor ボタンを押すと，エディットモードに切り替わる．テキストエリアにプログラムを記述する．Save ボタンや Load ボタンを使うことで，今つくったプログラムをファイルにセーブしたり，セーブしたプログラムをエディタに読み込むことができる．Reconsult ボタンを押すと，プログラムが実行可能になる．

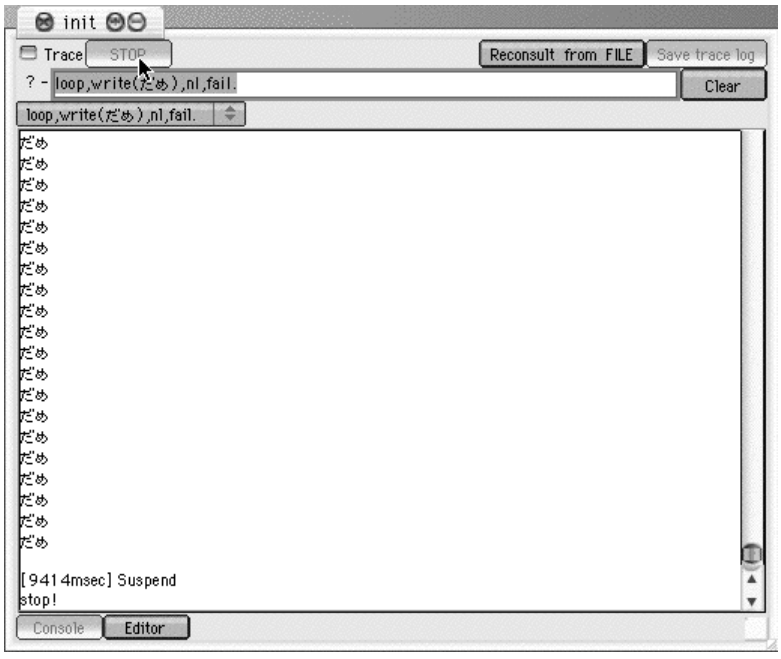
さっそく今つくったプログラムを実行してみる．
?- の横のテキストフィールドに **てすと.** と入力してリターンキーを押す.

プログラムが暴走したら?

次に，暴走したプログラムを停止させてみる．まず，無限ループするプログラムを書く．

?- の横のテキストフィールドに **loop,write(だめ),nl,fail.** と入力してリターンキーを押す．

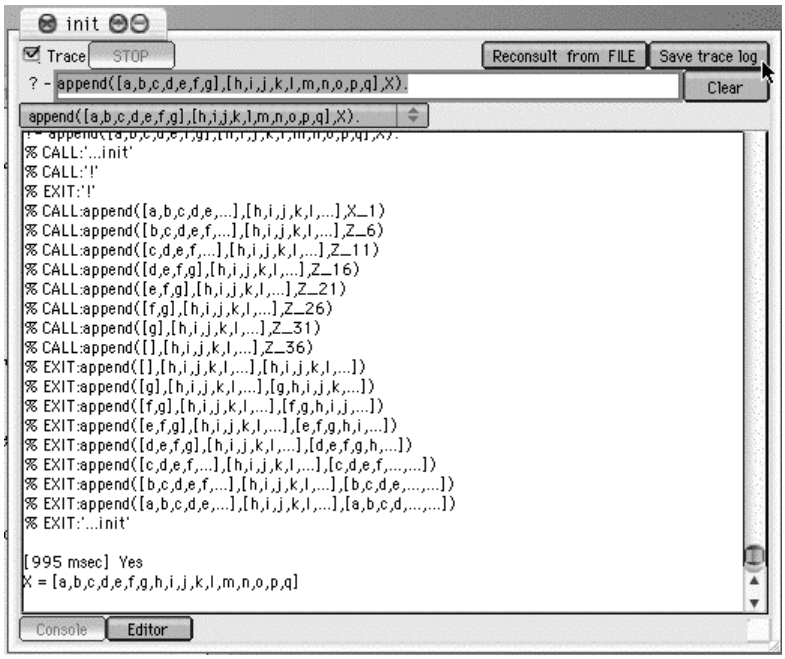
このプログラムは，無限に「だめ」を出力し続ける．こんな不快なプログラムは，さっさと停止させてしまう．**STOP** ボタンを押すと，プログラムの実行が停止する．



2.4 デバッガを行う

プログラムのトレースをとるには?

?- の上にある **trace** チェックボックスを押して，トレースを On にする.（これは，プログラムを実行中でも On/Off できるようになっている.）ただ，トレースはあまりにも高速で目がついていかない．おまけに，一定以上の長さになると昔のものが消えてしまう．こんなときは，**Save trace log** ボタンを押すと，これまでのトレース情報をまとめて全部ファイルに保存してくれる.



終了するには?

MiLog を終了するには，**halt.** を実行する．?- の横のテキストフィールドに **halt.** と入力してリターンキーを押す.

2.5 MiLog の文法（概略）

MiLog の文法は，**Prolog 言語からオペレータと';' を使った or の記述を取り除いたもの**である．ただし，述語 `is` の右側には式を書くことができる．たとえば,

```
calc(N,M) :-
  N is M + 2.
```

というプログラムは正しいが,

```
comp(N,M) :-
  N < M.
```

```
findPage(P) :-
  checkThePage(P);write(P).
```

というプログラムは次のように書き直す必要がある.

```
comp(N,M) :-
  '<'(N,M).

findPage(P) :-
  checkThePage(P).
findPage(P) :- write(P).
```

'!'(カットオペレータ) や，'' で囲ったアトム，""で囲った文字列など，それ以外の表現は Prolog のものをそのまま使うことができる．

```
getImageFine(URL) :-
  'StartsWithCapsedChar'(URL),!,
  append("Find URL",URL,X),write(X).
```

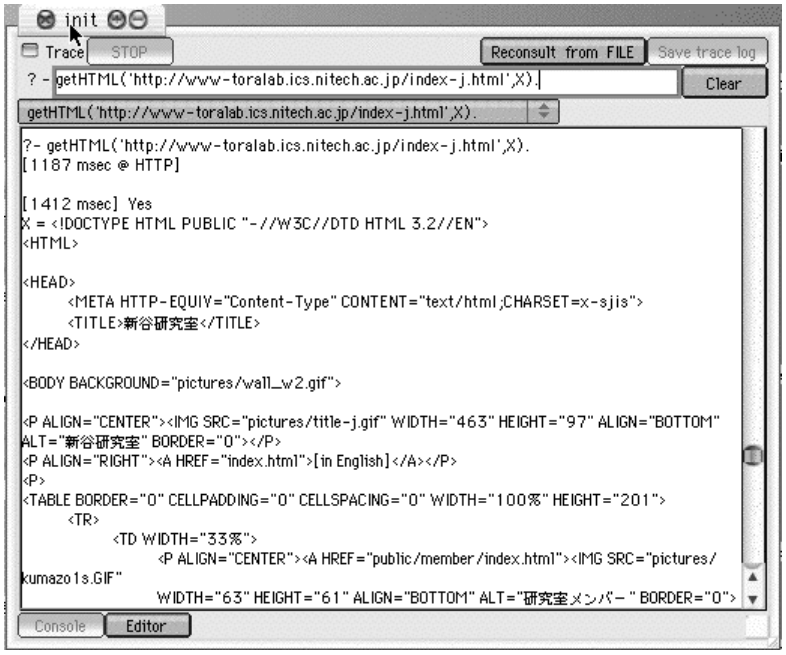
コメント文は'%' を使い，その行の'%' 以降の部分がコメントになる．

```
% これはコメント文
this :-
  is,a,test. % 行の後ろの部分にコメントをつけることもできる．
```

2.6 より高度な機能を使ってみる

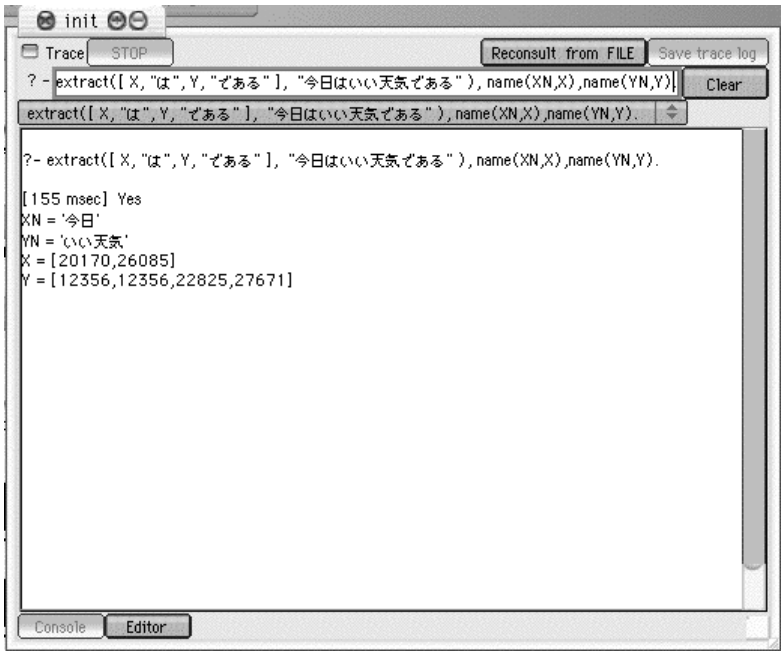
2.6.1 WWW へのアクセス

WWW へのアクセスは述語 1 つで簡単に行うことができる．Web ページの内容（HTML ドキュメント）を取得するには，組込述語 **getHTML/2** を使う．(getHTML/2 の getHTML は述語の名前を， /2 は引数が 2 つということを意味している．つまり，getHTML(HOGE1, HOGE2). の形式（2 引数）の述語である．MiLog では，取得した Web ページを解析するために，いろいろな述語が用意されている．たとえば，ハイパーリンク先をリストとして抽出するには，**getREF2/2** を使うことができる．それ以外にも，いろいろな述語がある．



2.6.2 パターンマッチング

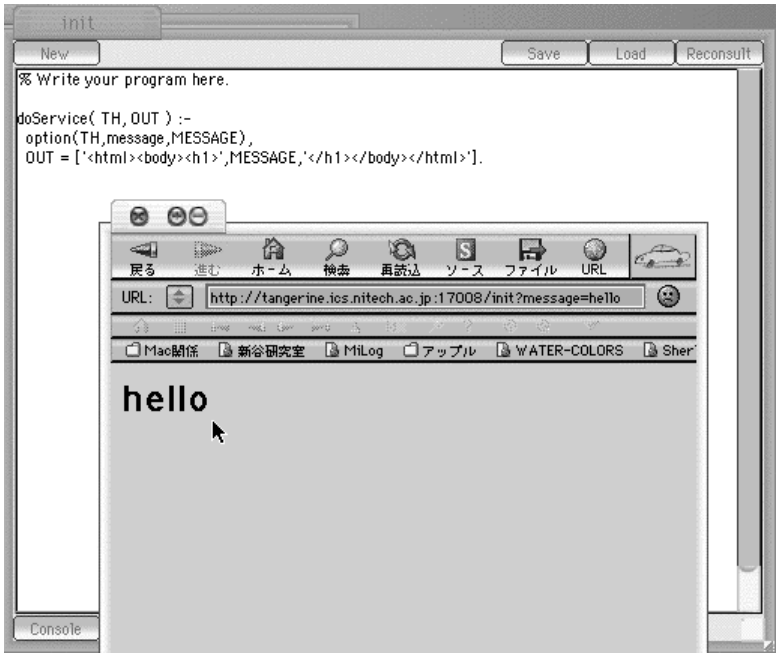
バックトラック機能をうまく利用すると，簡単にパターンマッチングを記述できる．たとえば,「X は Y である」というパターンを記述して,「今日はいい天気である」から「今日」と「いい天気」を取り出す場合を考える．これには，述語 **extract/2** を使う．この extract/2 は，論理型言語のリスト処理とバックトラックをうまく利用した MiLog プログラムとして実装されている．[6]



2.6.3 WWW Server and built-in CGI

MiLog には，WWW Server 機能が内蔵されている．この機能を使うと，簡単な MiLog プログラムで知的な Web ページが生成することができる．たとえば，次のようなプログラムを書いて，`http://`(自分の計算機の IP アドレス)`:17008/init?message=hello` というアドレスで Web ブラウザからアクセスすると，message パラメータの値を反映したページを表示させることができる．

[6] アトムへのパターンマッチングを簡単に行うために，extractA/2 という述語も用意されている．



2.6.4 マルチエージェントと並行動作

MiLog では，1 つの Java 実行環境上で，複数の MiLog 言語インタプリタを同時に (並行して) 動かすことができる．ここで，1 つのインタプリタを**エージェント**と呼ぶ．エージェントは，単に独立して複数同時に動くだけでなく，互いに情報を交換したり，仕事を分担したりして協調的に動作することができる．MiLog では，**複数のエージェントの間で情報を共有**したり，**情報を交換**したり，**仕事を分担**したりするための仕組みを用意している．

新しいエージェントを作るには，述語 new/1 を使う．例えば，

```
?- new(agent1).
```

とすると，agent1 という名前のエージェントが生成される．新しいエージェントがつくられると，そのエージェントのウィンドウが出てくる．それぞれのエージェントは独立して並行に動作することができる．たとえば，agent1 に論文の検索をさせている間に，agent2 にニュース内容のチェックをさせるという使い方ができる．

エージェント同士は，お互いを名前で識別する．他のエージェントへ質問 (Query) をする (＝プログラムを起動する) には，述語 query/2 を用いる．例えば，agent1 に今の気分 (最高だよ)．というプログラムを定義しておけば，他のエージェントから下のような質問を実行することができる．

```
?- query(agent1, 今の気分 (X)).
X = 最高だよ
```

MiLog では，オリジナルエージェントのコピーとして，クローンが利用できる．クローンを利用することにより，節データベース（MiLog 言語で書いたプログラムや assert された節が格納されているデータベース）を複数のエージェント（クローン）で共有することができる．節データベースを共有したクローンをつくるには，述語 clone2/1 を用いる．

```
?- clone2( cloneAgent1 ).
```

クローンを使うと，同じような作業を複数のエージェントに並行して行わせたり，1 つのツリー探索を複数のエージェントで並行に行ったりするプログラムが簡単に記述できる．

2.6.5 モバイルエージェントの記述

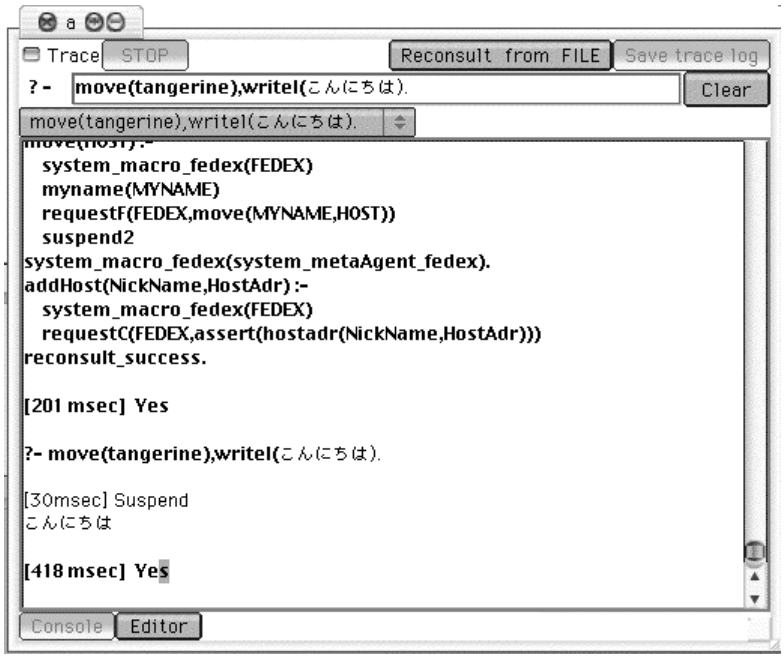
MiLog のもっとも重要な特長の 1 つが，モバイルエージェントを簡単に記述できることである．モバイルエージェントというのは，複数の計算機上を移動しながら計算を継続的に行うプログラムのことである．既存のモバイルエージェント記述言語を用いた場合，サーバの設定や起動などの面倒なセッティングが必要である．さらに，プログラミングの際に，エージェントを移動させるためのオブジェクトのシリアライズ方法などの知識が必要になる．MiLog では，オブジェクトのシリアライズ方法などをいっさい考慮することなく，ごく簡潔な記述でエージェントの移動を可能にする．MiLog で記述したエージェントは，特別なプログラムを付加することなくモバイルエージェントとして機能する．次に簡単な例を示す．

今回，エージェントをリモートホスト “192.168.0.2” 上で稼働中の MiLog に移動させる例を示す．やり方はとても簡単で，移動させたいエージェントに以下のような質問を評価させるだけである．[7]ただし，移動先のリモートホスト “192.168.0.2” 上で MiLog が起動している必要がある．[8]

```
?- move('192.168.0.2').
```

上記問い合わせを実行したエージェントは，そのエージェントウィンドウとともにリモートホスト “192.168.0.2” 上の MiLog に移動する．

次の実行例では，Windows2000 上で稼働している MiLog から iMac 上で稼働している MiLog にエージェントを移動させて，iMac 上で「こんにちは」と挨拶をしている例を示している．



2.6.6 補足

ユーザインタフェースからの別解探索

MiLog では，いわゆる’;’ の入力による別解探索機能は使用頻度が少ないという理由から省略している．

この機能がついた状態で問い合わせを行いたい場合には，述語 interactive/1 を使うとよい．[9]

[7]MiLog は Strong Mobility と呼ばれるもっとも強力なエージェントの移動方式を実現しているので，move/1 はプログラム中の任意の位置に入れることができ，このときエージェントが単に移動するという副作用が起きるだけででそれ以外の副作用はなく，バックトラック等のプログラムの実行制御には何も影響をおよぼさないようになっている．

[8]現在のホストのアドレスは，エージェントモニターを使っている場合なら，画面左上にいるメタエージェントの吹き出しのところに表示されている．

[9]interactive/1 中で write(hello),nl. のように複数の述語の AND を問い合わせしたいときには，my_call/1 を併用するとよい．このあたりは，もしかすると将来改善するかもしれない．

また，複数解の探索には，findall/3 を使うと便利である．findall/3 の使い方は，Prolog の教科書および付録のリファレンスを参照のこと．

第3章

MiLog 基礎

3.1 MiLog 言語の仕様

3.1.1 Prolog との文法的な相違

MiLog の文法は，基本的には一般的な Prolog の文法に近いが，いくつか標準的な Prolog とは異なる点がある．以下に一般的な Prolog との違いを列挙する．なお，Prolog 言語に関する一般的な知識に関しては本書では与えないので，必要に応じて，Prolog に関する参考書を別途参照されたい．

- オペレータに関する機能が省略されている
  MiLog では，実行速度の高速化および内部構造の単純化のために，オペレータに関する処理を省略してある．したがって，オペレータを用いたプログラムや，オペレータを内部的に使う機能 'read' や オペレータに対する制御を行う機能 'op' といったものは利用できない．ただし，述語 'is' などでは，例外的に，四則演算オペレータを用いた記述ができるようになっている．

- 数値計算に関する記述方法が異なる
  MiLog では，数値計算をする述語 'is' では，整数演算のみが可能である．
  浮動小数点演算を行う場合には 'isD' という述語が別途用意されている．また，数はアトムとは区別されない．したがって，浮動小数点数から整数への変換は，アトム名に対する文字列処理として行う（例えば，0.5 を足して '.' 以前を extractA などで取り出す）必要がある．また，数として扱える範囲は，整数値なら Java 言語での long 型で扱える範囲のもの，浮動小数点数値なら Java 言語での double 型で扱える範囲のものであり，それぞれ，アトム名としての表現も Java 言語上での数値の文字表現と対応している．

- 選言節を表す ';' オペレータを用いない
  プログラム中で選言を記述する場合には，複数のヘッド部分に分けて記述する．
  例：(通常の Prolog の場合)
  head1 :- body1 ; body 2.
  (MiLog での書き換え例)
  head1 :- body1.
  head1 :- body2.

- 単項，2 項オペレータを使わない
  "X > Y" は，大小記号 '>' をクオートで囲ってからファンクタ名とし，" '>'(X,Y) " のように書く
  "X =.. Y" は，組込述語 univ/2 を用いて "univ(X,Y)" と書く

- 'is' オペレータの右辺で使用できる演算子が限定されている
  基本的な四則演算（'+'，'-'，'*'，'/'）だけが利用可能
  単項オペレータ '-' は利用不能
  -1 を表したいときは '-1' のように -1 をクォートで囲む必要がある

- Unification を行う 'X = Y' の記述は，equal(X,Y) とも記述できる．call/1 で使うときは，こちらの記述を使う．

- オペレータを利用できないので，call/1 の引数に ',' などで複数の節の and で構成されたものを与えることはできない．たとえば，通常の Prolog では ?- call((Q1,Q2)) といった書き方が可能であるが，これは MiLog では受理されない．そのようなことをしたい場合，簡易メタインタプリタ my_call/1 を使い，?- my_call([Q1,Q2]). とするとよい．ただし，my_call/1 は簡易な実装であるため，カットオペレータの利用はできない．

- 一般的な Prolog の組み込み述語がすべて MiLog で用意されているというわけではない．用意されていないなものは，自力で作成する必要がある．

- アトムを記述する際に，英数字と'_'(アンダースコア) 以外を含む名前のアトムを作成する場合には ”(クオート) で囲う必要がある．アトム名自身にクオート記号を含めたい場合には，エスケープ文字として\ (バックスラッシュ) を使う．また，改行記号 (LF,CR) やタブ文字を表すために，それぞれ \n, \r, \t を使うことができる．これ以外のバックスラッシュによる記号の表現は，基本的にサポートしていない．

なお，system_ から始まる節の定義はシステムによって予約されている．

3.1.2 用語の意味

Prolog は，もともと日本で開発されたものではないため，解説書も英語などで書かれたものの邦訳である場合が多い．それらの解説書では，用語の日本語への対訳方法が必ずしも一意でないため，本書で用いられている言葉との相違があり，混乱することがあるかもしれない．本節では，本書で用いる用語とそれに対応する意味について，以下で補足しておく．といっても，用語の意味の定義をきちんと文章として書いたものは難解で読みづらくなるため，ここでは，用語の意味の説明をかねて，その用語を実際に使いながら MiLog のプログラムに関する基礎を説明していく，という形式をとることにする．他の用語が用いられている Prolog 解説書を併読する際には，参考にされたい．[1]

MiLog の**プログラム (program)** は，**ヘッド (head)** と**ボディ (body)** を持つ**ホーン節 (Horn clause)** の集合として構成される．プログラムは，各エージェントの持つ**節データベース (clause database)** に格納されることで呼び出し可能となる．節データベースへプログラムを**ロード (load) する**ことを，そのときに使う述語名にならって，**reconsult する**，あるいは **consult する**と呼ぶこともある．

ロードされたプログラムを呼び出すための記述は，**問い合わせ (query)** と呼ばれる．節データベースに対して問い合わせを行うことで，その問い合わせと一致するヘッド部を持つホーン節が順に呼び出されていく．問い合わせは通常，先頭に ?- を付けて記述され，例えば，?- member(X,[a,b]). という問い合わせにより，述語 member/2 が**実行**される，あるいは**呼び出される (call)**，という表現をする．(ただし，MiLog のユーザインタフェースを使って問い合わせを行うときには，先頭の ?-の部分は入力しない.) 問い合わせの結果は，問い合わせ中に含まれる変数への変数**束縛 (binding)** として得られる．

ホーン節のヘッド部には，**アトム (atom)** か**述語 (predicate)** を使う．アトムとはプログラムの実質上の最小の単位で，**アトム名 (atom name** を用いて，識別子を表現する．述語というのは，**述語名 (predicate name)** に続いて括弧中にカンマ (comma) で区切った**引数 (argument)** を書くことができるもので，その引数の数のことを**アリティ (arity)** と呼んだりする．また，述語名は，それを得るために使う述語名に由来して**ファンクタ (functor) 名**あるいは**関数子**と呼ぶことがある．述語の引数には，アトムや**変数 (variable,var)** のほかに，**リスト (list)** や述語と同じ形をした**構造**を入れることができる．また，ホーン節をいくつか使って，特定の述語名に対応するヘッド部を持つプログラムを作ることを，**述語を定義する**という言い方をすることがある．また，そのときに，ヘッド部の述語名とアリティを ‘/’(スラッシュ記号) で区切って表現することがある．たとえば，２つ引数を持つ述語 member のことを，member/2 と書いたりする．

あらかじめそのボディ部で実行する内容が作られているような述語のことを，**組み込み述語 (built-in predicate, built-in)** と呼ぶ．

ヘッド部とボディ部を結ぶ :-記号のことを，その形が似ていることから**メダカ**と呼んだりする．

ボディ部のないホーン節を**基底節 (ground clause)**，ヘッド部がないホーン節を**コマンド (command)** あるいは **directive clause** と呼ぶ．コマンドは，プログラムの reconsult 時に即座に問い合わせとして実行される．たとえば，:- reconsult(mylib). とプログラムの先頭に書いておくと，mylib というファイルに書いておいたライブラリプログラムを自動でロードすることができる．ただし，MiLog では，コマンドからはエージェントの移動を伴う問い合わせを実行できないので注意が必要である．

節データベースに対するプログラムの reconsult 以外に，特定の組み込み述語を使うことで,(主に基底節を) 追加したり削除したりすることができる．これを，そのときに使う述語名から，追加することを**アサート (assert)** するといい，削除することを**リトラクト (retract)** する，という．一部の Prolog 処理系では，アサートやリト

[1] あくまでも便宜上書いたもので，これらの用語の使い方が学術的に正しいことを主張するものではない．

ラクトする述語を事前にプログラム中で宣言しておかなければならないことがある．これを，そのときに使う述語名から，**ダイナミック (dynamic)** 宣言と呼んだりする．なお，MiLog では今のところこのダイナミック宣言を行う必要はなく，普通にアサートやリトラクトができる．

述語の呼び出しが**失敗 (fail)** したときには，**バックトラック (backtrack)** が起きる．このバックトラックを制御するための仕組みに，!記号で書かれる**カットオペレータ (cut operator)** がある．カットオペレータは，その述語呼び出しに関して，ボディ部のそのカットオペレータ以前の部分へのバックトラックと，その述語の他のヘッド部へのバックトラックを禁止する．

Prolog では，文字列は，その文字コードを 1 次元のリストにしたものとして表現することが多い．ただし，プログラム中では，文字コードをカンマで区切ったリストを書くのは面倒なため，””(ダブルクオート) で囲った**文字列**を，自動で文字コードのリストに置き換えてくれる機能が付いているのが普通である．このような理由から，たとえば，`?- write("日本語").` とすると，日本語とは表示されずに，‘日’ と ‘本’ と ‘語’ を示す文字コードのリストが表示されるが，これは決してバグではない．‘日本語’ と出力したいときは，アトムに直して表示すればよい．リストによる文字列表現とアトムとの相互変換には，name/2 という組み込み述語を使うとよい．

あと，これは内部の実装に近い話になってしまうが，このバックトラックを行う可能性のあるような，実行上の分岐点のことを，**choice point** と呼んだりする．また，バックトラック時には必要に応じて choice point 以後に行われた変数の束縛を解除するが，このためのデータ格納場所のことを**トレイルスタック (trail stack)** と呼んだりする．

通常の Prolog 処理系では，アトム名として使える文字列の長さや個数が制限されていたりすることもあるが，MiLog では，メモリーが不足しない範囲でならそういった制限は一切ないため，例えばアトムを文字列を表現するためのデータ構造として使うこともできる．また，述語のアリティにも制限がないため，1000 くらいのアリティの述語を平気で作ったりもできる．

MiLog の現時点での実装では，中間言語へのコンパイラを持っておらず，インタプリタとしての実装ではあるものの，**末尾呼び出し最適化 (Last Call Optimization)** と呼ばれる非常に重要な最適化については，ちゃんと独自のやり方で実装してあるので，安心してほしい．この最適化があるかないかが，使える処理系か否かを分けるといっても過言ではない．

3.1.3 日本語の扱い

MiLog では，データとして扱うだけでなく，プログラムの記述そのものにも，日本語を使うことができるようになっている．たとえば，

```
こんにちはと叫ぶ :-
  叫ぶ(こんにちは), 改行.
叫ぶ( _叫ぶこと) :-
  write( _叫ぶこと ).
改行 :-
  nl.
```

というプログラムを作成して，

```
?- こんにちはと叫ぶ.
```

という問い合わせを与えても，ちゃんと動くようになっている．日本語で変数名を書きたい場合には，_ (アンダースコア) か英大文字を前につければよい．ただし，’:-’ や ’,’ や ’.’ は，半角英文字で書く必要がある点には注意されたい．また，アトムの名前として半角英数字と日本語の文字を混ぜて使う場合には，”(クオート) で囲う必要がある．

MiLog には，テキストファイルを読み出してそれを 1 つのアトムに変換する述語 loadText/2 や，アトムをテキストファイルとして書き出す述語 saveText/2 が用意されている．これら述語を用いてテキストファイル

第4章

エージェント間通信とモビリティ

本章では MiLog におけるエージェントの動作モデルと，そのモデルに沿ったエージェントの実装方法について紹介する．また，エージェント間通信とマイグレーションの方法も例を示しながら紹介する．

4.1 MiLog の動作モデル:Multiple Interpreter Model

MiLog では，複数のエージェントが並行にに動作するモデルとして，Multiple Interpreter Model を採用している．Multiple Interpreter Model では，エージェント１つ１つに独立した MiLog インタプリタ (とその操作のためのインタフェース) が割り当てられ，それぞれのエージェントが全く別個のプログラムを持ち，独立してプログラムを動作させられるようになっている．

4.1.1 新しいエージェントを作る

新しいエージェントを作るには，述語 new/1 を使う．たとえば，

```
?- new(a).
```

とすると，'a' という名前の新しいエージェントが１つできる．ここで，もう一度

```
?- new(a).
```

としてみても，No とでて，新しいエージェントはできない．エージェントには，唯一の名前を付けることになっている．すでに 'a' という名前のエージェントはいるので，同じ名前のエージェントは作れないようになっているからである．

次に，

```
?- new(b).
```

とすると，今度は 'b' という名前の新しいエージェントが１つできる．

4.1.2 エージェントの並行動作

a というエージェントと b というエージェントでは，異なるプログラムを独立して動かせるようになっている．

a というエージェントにプログラムをロードさせて，実行してみる．たとえば，

```
helloMiLog :-
  writel('Hello MiLog World!').
```

というプログラムを reconsult してから

```
?- helloMiLog.
```

とすると，`Hello MiLog World!` と出てくる．このときに，エージェント b のほうは，ぴくりとも動かない．

次に，今度は，エージェント b のほうに

```
helloMiLog :-
  writel('MiLog World He Youkoso!').
```

というプログラムを reconsult してから

```
?- helloMiLog.
```

とすると，`MiLog World He Youkoso!` となる．ここで，あらためて，エージェント a のほうで

```
?- helloMiLog.
```

とすると，エージェント a のほうは，やっぱり Hello MiLog World! と出てくる．
こんどは，エージェント a のプログラムを少し書き換えて

```
helloMiLog :-
  writel('Hello MiLog World!'),
  helloMiLog.
```

としてから reconsult してみる．これでエージェント a のプログラムが，無限ループのプログラムに書き換わっているはずである．ここで，エージェント a から，

```
?- helloMiLog.
```

とすると，

```
Hello MiLog World!
Hello MiLog World!
Hello MiLog World!
...
```

と表示を続けるようになる．この状態で，エージェント b に対して

```
?- helloMiLog.
```

とすると，エージェント a が実行中であることに関係なしに MiLog World He youkoso! と表示がでて，実行ができることがわかる．次に，エージェント b のプログラムを

```
helloMiLog :-
  writel('MiLog World He Youkoso!'),
  helloMiLog.
```

として reconsult してから，エージェント b に

```
?- helloMiLog.
```

とすると，

```
MiLog World He Youkoso!
MiLog World He Youkoso!
MiLog World He Youkoso!
...
```

と表示を続けるようになる．このときに，エージェント a とエージェント b は独立して，かつ同時に（学術用語では，これを「並行に」，という）動作していることに注意する．すなわち，エージェント a とエージェント b は完全に独立しており，それぞれ異なった内容のプログラムを持つことができ，それぞれ独立して並行にプログラムの実行をすることができるようになっている．
ここで，エージェント a について（先ほどの実行を止めずに）

```
?- write('Interrupt Me!').
```

としても，これは実行されない．そこで，STOP ボタンを押してエージェント a のプログラムの実行を止めてからあらためて，

```
?- writel('Interrupt Me!').
```

とすると，今度はうまく実行される．つまり，各エージェントは，原則として，同時に 1 つだけ，プログラムを実行するようになっている．

4.1.3 エージェントの削除

使い終わって不要になったエージェントは，述語 delete/0 を使って，削除することができる．たとえば，エージェント a に対して

```
?- delete.
```

とすると，エージェント a のウィンドウが消え，エージェントモニター上のアイコンも消える．これは，エージェント a が消滅したことを意味する.（エージェントモニターが起動しているときなら，エージェントのアイコンをリサイクルマークのアイコンへドラッグ&ドロップすることで，そのエージェントを削除することができる.）一度消したら，そのエージェントの名前は空き状態になるので，もう一度同じ名前のエージェントを作ることができる．

4.2 MiLogの実現するモビリティ: Interpreter-level(Strong) Mobility

4.2.1 エージェントが「移動する」とは？

MiLog では，エージェントが「移動する」ことができる．では，エージェントが移動することができるとはどういうことか？例えば，あるコンピュータ X 上で MiLog が起動していて，別のコンピュータ Y 上でも MiLog が起動しているときを考える．コンピュータ X 上にあるエージェント a が，コンピュータ Y 上に「移動する」とは，コンピュータ X 上にいたエージェント a が消えてなくなり，そのかわりに，消えたエージェント a と全く同じ内部状態を持ったものが，コンピュータ Y 上に生成される，ということである．

ここで重要なのは，エージェントの「内部状態」とは何か，である．MiLog では，移動されるエージェントの内部状態とは，エージェントインタプリタの持つ情報すべてである．その情報には，そのエージェントの名前，ロードされていたプログラム（節データベース）から，その時点で実行していた問い合わせの実行の実行状況までを全部含むものである．つまり，移動後のエージェントは，移動前と同じ名前で，同じプログラム（節データベース）を持っており，かつ，移動前に問い合わせを実行中であったならその問い合わせを引き続きその続きから実行するようになっている．つまり，エージェントが移動したときに変化するのはそのエージェントを実行するコンピュータが何であるかであって，それ以外は何も変化しない．このようなエージェントの移動機能のことを，Strong Mobility と呼ぶことがある．[1]

エージェントのモビリティに関しては，アプリケーションとして見たときに，データ通信遅延の低減，データ通信量の最適化，アプリケーションのインストールと更新の自動化／簡易化などの利点をもちうることが，これまでに報告されている．また，プログラミングの中にモビリティを取り入れることで，クライアント/サーバプログラムの統合とプロトコル設計の省略が可能で，コーディングも簡単になるといわれている．ただし，実際にモビリティをうまく役立てたアプリケーションは少数であり，またモビリティを持ったプログラムをコーディングする際の，効果的な記述方法についても，よくわかっていないのが現状である．そこで，MiLog の開発にあたっては，エージェントのプログラム中でエージェントの移動をできる限り簡潔に記述でき，かつ，効果的なエージェント移動の記述方法を，MiLog を使うプログラマ自身の手で開拓していけるように，Strong Mobility を，可能な限りピュアな形で実現するよう努めた．

本節では，MiLog のモバイルエージェント機能を説明する．MiLog のモバイルエージェント機能は，より厳密に言うと，ある ホストコンピュータ上で動いている MiLog の実行環境（MiLogServer）上のエージェントが，どこか別のホストコンピュータ上で動作する別の MiLogServer 上に移動するための機能である．[2]

[1] 他の多くのモバイルエージェント処理系では，このエージェントの内部状態に，問い合わせの実行状況（スレッドを実行するためのスタック）を含んでいない．つまり，エージェントは，移動前にプログラムをどこまで実行していたのか，どんな述語（メソッド）を呼び出したが値をまだ戻していなかったのか，といった情報を，移動後には思い出せない．仕方がないので，移動前にメモ（データ領域）に書いておいた内容を元に，移動後に改めてプログラムをあらかじめ決められたメソッド呼び出しの部分から実行し直すようになっている．これを，Weak Mobility と呼んだりする．

[2] さらに厳密に言うと，複数の MiLogServer を同じホストコンピュータ上で起動することもでき，その場合にも MiLogServer 間をエージェントが移動できるようになっている．

4.2.2 エージェント移動のモデル

エージェントの移動は，move/1 によって行われる．たとえば，’192.168.0.2’ という名前で識別可能な MiLogServer へエージェントを移動させるには，

```
?- move('192.168.0.2').
```

とする．こうすると，エージェントは，エージェントウィンドウごと ’192.168.0.2’ に移動する．

move/1 は，問い合わせとして実行するだけでなく，プログラム中の任意の位置で使うことができる．たとえば，’192.168.0.1’,’192.168.0.2’,’192.168.0.3’,’192.168.0.4’,’192.168.0.5’ の 5 つの MiLogServer があり，エージェント a が ’192.168.0.1’ にいるとき，エージェント a に

```
test_program(X) :- writel([goto,X]),move(host).
test_program_loop([]) :- writel(done).
test_program_loop([X|R]) :-
  test_program(X),
  test_program_loop(R).
```

というプログラムをロードして

```
?- test_program_loop(['192.168.0.2','192.168.0.3',
                      '192.168.0.4','192.168.0.5']).
```

という問い合わせを実行すると，エージェントは，’192.168.0.2’ から順に，’192.168.0.3’,’192.168.0.4’,’192.168.0.5’ の MiLogServer へと移動していく．このように，move/1 でのエージェントの移動では，エージェントの問い合わせ実行が途中で止まってしまうというようなことは，起きず，移動先で問い合わせ実行が継続される．

エージェント移動後の，問い合わせ処理中のバックトラックは，移動先で実行される．たとえば，エージェント a が ’192.168.0.1’ にいて，

```
test_program2 :- move('192.168.0.2'),fail.
test_program2 :- writel(ok).
```

というプログラムがあるときに

```
?- test_program2.
```

という問い合わせを実行させると，エージェントは ’192.168.0.2’ に移動したあと，その直後の fail により失敗がおき，test_program2 のヘッド部分のマッチングのところへバックトラックする．ここで，その test_program2 のヘッド部分へ最初にマッチングを試みたときにはエージェントは ’192.168.0.1’ にいたが，そこへ再び移動するのではなくて，’192.168.0.2’ 上で，バックトラック動作が起き，2 番目のヘッド `test_program2` が見つかって，そのボディ部にある `writel(ok)` が実行される．この `writel(ok)` が実行されるのも ’192.168.0.2’ でのことで，この問い合わせの終了後もエージェントは ’192.168.0.2’ に留まり続ける．

バックトラックとエージェントの移動を組み合わせると，例えば特定のエージェントを探しまわって，そのエージェントにエージェント間問い合わせをする，というようなプログラムを作ることができる．例えば，たとえば，’192.168.0.1’,’192.168.0.2’,’192.168.0.3’,’192.168.0.4’,’192.168.0.5’ の 5 つの MiLogServer があり，エージェント a が ’192.168.0.1’ にいるとき，エージェント a に

```
test_program3(X) :- member(H,X),move(H).
test_program4 :-
  test_program3(['192.168.0.2','192.168.0.3',
                 '192.168.0.4','192.168.0.5']),
  interruptAndQuery(init,delete).
```

というプログラムをロードして

```
?- test_program4.
```

という問い合わせを実行すると，エージェント a は，エージェント init が見つかるまで '192.168.0.2','192.168.0.3','192.168.0.4','192
と順に移動していき，エージェント init を見つけると，そのエージェントを，エージェント間問い合わせ機能を使って，削除する．

ここで重要な点は，MiLog ではこの例のようなエージェントの移動を伴うプログラムの断片を，エージェントの移動を隠蔽した形でプログラムに組み込むことができる点である．例えば,「エージェントを実際に移動させながら，エージェントの活動にふさわしい場所を探して，そこに最終的に移動させる」といったような，高度なエージェントの移動をライブラリ化することができる．また，このライブラリ化をうまく使えば，`move/1` を拡張して,「特定の問い合わせの部分が必ず特定の MiLogServer 上で実行されるようにする」というような述語や,「特定の移動先に移動したら，特定のプログラムをロードする」というような述語を，MiLog ユーザ自身の手で作っていくことができる．既存の多くのモバイルエージェントシステムでは Weak Mobility しか実現していないため，このライブラリ化や自前でのエージェント移動の記述方法の拡張ができないことが多い．これができることが，MiLog の 1 つのアドバンテージになっている．

参考までに，以下に，特定の問い合わせ `Q` をバックトラックも含めて特定の MiLogServer `Host` 上で行われるようにする述語 `call_at/2` を， `move/1` を使ってプログラミングした例をあげておく．

```
call_at( Q, Host) :-
  thisHostForMobile(HERE),
  go_or_back(Host,HERE),
  call(Q),
  go_or_back(HERE,Host).

go_or_back(GO,BACK) :-
  move(GO).
go_or_back(GO,BACK) :-
  move(BACK),fail.
```

これは，たとえば,

```
?- call_at( member(X,[a,b,c]), '192.168.0.2' ), member(X,[c,d,e]).
```

のように使う．こうすると，最初の `member(X,[a,b,c])` の実行はバックトラックも含めて必ず '192.168.0.2' 上で行われるが，2 つ目の `member(X,[c,d,e])` を実行するときには，エージェントは移動先から戻ってくる．この結果，上の問い合わせではエージェントが'192.168.0.2' との間を何度か行ったり来たりする．[3]

プログラミング上のテクニックとしては，go_or_back/2 の動作が鍵になる．この述語は最初，第 1 引数で指定された MiLogServer に移動するが，その後，バックトラックが生じた際には，第 2 引数で指定されたホストに戻り，さらにその直後に fail することでそれ以前の場所へとバックトラックさせている．この作り方だと，厳密に言うなら，バックトラック時になんらかの理由で移動先に `move` で移動できないようなときには，移動せずにその場でバックトラックをさせてしまう，という挙動をする．こういった，エラー時の挙動まで含めて自分でエージェントの移動の作法をプログラミングしていけるのが，MiLog の持ち味である．

4.2.3 MiLogServer の指定方法

MiLog エージェントが複数の MiLogServer 間を移動するためには，移動先の MiLogServer を指定する必要がある．MiLogServer の指定方法として，URL，IP アドレス，および NAT 越えのための特殊な URI の 3 つが挙げられる．これらに関して以下で詳細に説明する．

[3] この例だと，`member/2` の実行結果はどの MiLogServer 上でもそもそも変わらないので，あまり有り難みがわからないが...

1. URL
   MiLogServer の指定に URL を用いる方法である．MiLogServer が存在するホストの URL をアトムとして表現する．例えば URL が，“http://192.168.0.3/” のホスト上の MiLogServer は，アトム ’http://192.168.0.3/’ として表現される．ここで URL だけでなくポート番号も指定したいときは，’URL:ポート番号’ のように記述する．例えば URL が，“http://192.168.0.3” のホストの，ポート番号 12345 に接続している MiLogServer は，アトム ’http://192.168.0.3:12345/’ と表現される．ただし，-use_sslserver をオプションとして使用している MiLogServer へ通信する場合，’http://’ の部分を，’https://’ とする．これは，SSL モードでは HTTP プロトコルではなく HTTPS プロトコルを使用するためである．なお，いずれのばあいでも，URL の指定では，記述の一部を省略することが可能になっている．デフォルトのポート番号 (17008) を使っているホストの場合，ポート番号の記述（たとえば，:12345）を省略することができる．また，プロトコルが HTTP の場合には，verb+’http://’+の部分を省略できる．また，末尾の’/’ も省略できる．これらを省略したものが，実際には，次に紹介する IP アドレスによる指定ということなる．

2. IP アドレス
   プロトコルとして HTTPS を使っている場合を除き，MiLogServer の指定には， IP アドレスを用いることができる．MiLogServer が存在するホストの IP アドレスをアトムとして表現する．例えば IP アドレスが，“192.168.0.1” のホスト上の MiLogServer は，アトム ’192.168.0.1’ として表現される．ここで IP アドレスだけでなくポート番号も指定したいときは，’IP アドレス:ポート番号’ のように記述する．例えば IP アドレスが，“192.168.0.1” のホストの，ポート番号 12345 に接続している MiLogServer は，アトム ’192.168.0.1:12345’ と表現される．なお，IP アドレスだけでなく，DNS に登録されたホスト名も MiLogServer の指定に一応利用可能ではあるが，完全な動作の保証はしていない．また，DynamicDNS などの仕組みとの整合性などが完全に確認できていないので，このような用途で使う場合は，下記の特殊 URI を使った方が確実である．

3. 特殊 URI
   MiLogServer の指定に 特殊な URI を用いる方法である．(NAT 越えに関しての詳細は後の章で述べる.) MiLogServer が存在するホストのユニークネームと通信を経由させる relay server の URL を指定する形式であり，NAT やファイアウォールで囲まれた環境間でエージェントを移動させる際に利用する．例えば URL が，“http://relay.com” の MiLogServer を中継し，通信先の MiLogServer のユニークネームが milog の場合，アトム ’relay:milog@http://relay.com/’ と表現される．

4.2.4 エージェント移動機能の詳細

MiLog エージェントが MiLogServer 間を移動するための組込述語として，move/1 がある．MiLog において実装されたマイグレーションは強いマイグレーションなので，move/1 はプログラム中の任意の箇所で使用でき，述語の呼び出し関係やバックトラック情報なども移動の前後で副作用をうけない．move/1 によって移動したエージェントは，移動後に移動前と同じ状態で実行を再開する．

move(MiLogServerAddr) を評価したエージェントは，MiLogServerAddr に移動する．MiLogServerAddr は，4.2.3 で説明した形式で記述する．例えば，MiLogServer “http://192.168.0.3/” に移動したい場合は，以下のようにすればよい．

例: MiLogServer ’’http://192.168.0.3/’’ に移動

```
?- move('http://192.168.0.3/').
Yes
```

上記を評価したエージェントは，MiLogServer “http://192.168.0.3/” 上に移動する．

また組込述語 goHome/0 を用いることによっても，エージェントの移動が達成される．goHome/0 を評価したエージェントは，そのホームに移動する．通常，あるエージェントのホームは，そのエージェントが作成され

た MiLogServer である．エージェントのホームは，エージェントの節データベース上に myhome/1 の形式で登録されている．myHomeIsHere/0 を用いると，エージェントのホームは，現在そのエージェントが稼働している MiLogServer に変更される．例えば，“http://weblog.com” から “http://milog.com” に移動したエージェント ‘a’ が存在したとする．‘a’ のホームが “http://milog.com” であるとき，‘a’ の節データベースには “myhome('http://weblog.com').” 登録されている．このとき ‘a’ が移動後に myHomeIsHere/0 を評価すると，‘a’ の節データベース中の “myhome('http://weblog.com').” が削除され，代わりに “myhome('http://milog.com').” が追加される．すなわち，‘a’ のホームが，“http://weblog.com” から “http://milog.com” に変更される．

エージェントウィンドウの内容を保存したまま移動させる

エージェント移動のための述語 `move/1` および `goHome/0` では，移動のオーバヘッドを小さくするために，移動前のエージェントウィンドウの内容が，移動後に復元されずクリアされるようになっている．プログラムのデバッグのために，トレース（`trace/0`）を行っていたり，あるいは `write/1` 述語を使ってデバッグ情報を表示していた場合には，これらの情報がエージェントの移動に伴って消えてしまう．主にプログラムのデバッグを楽にする目的で，これらのエージェントウィンドウの情報を保存した状態でエージェントを移動させる述語 `moveL/1` および `goHomeL/0` を用意している．使い方は，述語 `move/1` および `goHome/0` と同じである．

4.3 MiLog のエージェント間通信のモデル

MiLog では，エージェントはそれぞれ独立したインタプリタを持ち，異なったエージェントプログラムを並行に動作させることができることを，前節までで見てきた．ここで重要な点は，例えばエージェント a とエージェント b があったとして，エージェント b のプログラムからは，エージェント a のプログラムやデータは直接参照できない，という点である．つまり，ここで出てくる疑問として，

- a というエージェントから b というエージェントに仕事を依頼するには？(問い合わせの起動)
- a というエージェントが b というエージェントが今何を考えているかを知るには？(値の参照)

というものがある．このため機能を統合して扱いやすくしたものが，MiLog の持つ，エージェント間通信機能である．

エージェント間の通信モデルには，非同期メッセージ通信など，いろいろな通信モデルが考えられる．MiLog では，エージェント間の通信モデルとして，Active Object Model を採用している．Active Object Model とは，メッセージの受け渡しとそのメッセージの処理のためのスレッドの起動・破棄が連動している並行処理モデルで，複雑な並行処理をミス無く平易に扱うことができるモデルとして提案されたものである．

次に，実際に動かしながら，このモデルに沿った通信の動きを見てみる．

4.3.1 単純なエージェント間問い合わせの起動： query

先ほどの例と同じように，まずエージェント a とエージェント b を用意する．エージェント a には，プログラム

```
helloMiLog :- writel('Hello MiLog World!').
```

を，エージェント b には，プログラム

```
helloMiLog :- writel('MiLog World He Youkoso!').
```

をそれぞれ reconsult しておく．ここで，エージェント a から

```
?- query( b, helloMiLog ).
```

とすると，エージェント b 側で MiLog World He Youkoso! と表示されるはずである．これは，エージェント a からエージェント b へ「 ?- helloMiLog. を実行しなさい」という通信メッセージが送信され，それがエージェント b に届くと，エージェント b のインタプリタが自動的に動作を開始して，?- helloMiLog. という問い合わせをエージェント b のプログラムで実行され，その結果として，MiLog World He Youkoso! という出力がエージェント b 側で表示されたわけである．さらに，その helloMiLog の実行がエージェント b 上でがうまくいったことがエージェント a に伝達され，その結果として エージェント a 側で Yes と出てきている．

こんどは逆に，エージェント b の側から，

```
?- query( a, helloMiLog ) .
```

とすると，エージェント a 側で Hello MiLog World! と表示され，エージェント b 側は Yes と出てくる．

つまり，エージェントは双方向に問い合わせを行うことができるようになっている．

4.3.2 実行中のエージェントに対する問い合わせをしたときの挙動

今度は，エージェント b の 側のプログラムを

```
helloMiLog :- writel('MiLog World He Youkoso!').
helloMiLogLoop :- writel('MiLog World he Youkoso!'),helloMiLogLoop.
```

と変えて，さらにエージェント b の側で

```
?- helloMiLogLoop.
```

として，エージェント b を無限ループでずっと問い合わせ実行中の状態にする．この状態で，エージェント a の側から

```
?- query( b, helloMiLog).
```

としても，結果は No. となる．query/2 では，なにか問い合わせを実行中のエージェントに対しては問い合わせを行うことはできず，単純に失敗するようになっている．そこで，エージェント b のプログラムの実行を STOP ボタンで止めてからエージェント a の側から

```
?- query( b, helloMiLog).
```

とすると，今度は期待通り，エージェント b で MiLog World He Youkoso! と表示され，エージェント a の側は Yes となる．

4.3.3 問い合わせの結果の受け取り

先ほどの例では，エージェント a からエージェント b のプログラムを起動してみたが，エージェント b で文字が出力されただけで，エージェント a には単にそれが成功したという情報しか戻されていなかった．次の例では，エージェント b で実行した問い合わせの結果（変数束縛）をエージェント a で受け取るような例を作ってみる．先ほどと同様にエージェント a とエージェント b を用意する．こんどは，エージェント b には，プログラムとして

```
helloMiLog(X) :-
  helloMessage(X).
helloMessage('MiLog World He Youkoso!').
```

というものを用意しておく．この状態でエージェント b で

```
?- helloMiLog(X).
```

とすると

```
X = 'MiLog World He Youkoso!'
```

と当然出てくるはずである．今度は，エージェント a から

```
?- query( b, helloMiLog(X)).
```

とすると，X = 'MiLog World He Youkoso!' と出てくる．つまり，問い合わせ結果は，エージェントに対する通常の問い合わせ実行同じようなイメージで，変数束縛として受け取ることができる．

今度は，問い合わせ結果が失敗するような例を見てみる．エージェント a から

```
?- query( b, helloMiLog(hoge)).
```

とすると No. とでてくる．これは，hoge と 'MiLog World He Youkoso!' が単一化できないからで，問い合わせは失敗した，という結果が返ってきている．その結果として，No. と出る．

次に，バックトラックが起きたときにどうなるかを見てみる．先ほどのエージェント b のプログラムを書き換えて，

```
helloMiLog(X) :-
  helloMessage(X).
helloMessage('MiLog World He Youkoso!').
helloMessage('Hello MiLog World!').
```

としておく．ここで，まず手始めに，エージェント b で

```
?- helloMiLog(X),writel(X),fail.
```

とすると，次のようになるはずである．

```
MiLog World He Youkoso!
Hello MiLog World!


No
```

これは，一度 helloMiLog/1 が成功して writel/1 で出力された後，バックトラックが起きて，helloMiLog/1 の別解が検索されさらにそれが出力された後，もう一回バックトラックが起き，今度は別解がないので，問い合わせが失敗し No と出てくる．何の変哲もない，通常のバックトラックの動作である．似たようなこととして，エージェント a から

```
?- query( b, helloMiLog(X)),writel(X),fail.
```

とすると，

```
MiLog World He Youkoso!


No
```

と，先ほどと異なった結果になる．これは，MiLog におけるエージェント間問い合わせでは，一度問い合わせが成功すると，別解を求めないようになっているからである．ただし，これは成功した後の話であり，そのエージェント間問い合わせの実行そのものは，通常通り，バックトラックのある解の探索が行われている．(つまり，問い合わせが成功した段階で，あたかもカットオペレータがそこに挿入されたかのように，別解探索が破棄されるようになっている）たとえば，

```
?- query( b, helloMiLog('Hello MiLog World!') ).
```

とした場合，きちんと，２つめの解である 'Hello MiLog World!' が見つかり Yes と出てくる．少しわかりにくいかもしれないが，一度だけ解を探索して，解が見つかったらバックトラックをしない述語 once/1 を使って，エージェント b から

```
?- once( helloMiLog(X) ),writel(X),fail.
```

と実行したときと，同じような挙動を示す．

なぜ，一度成功したエージェント間問い合わせではバックトラックが行われないようになっているかというと，性能，プログラミング時の扱いやすさ，実際の用途などを考えた上でのトレードオフとして，開発時点でそれがもっとも都合がよかったからである．(というわけで，もしかすると，将来的にはバックトラックが選択式になったりあるいはデフォルトの動作が変わったりするかもしれない．)

4.3.4 しつこく待って必ず問い合わせを行う：queryF

query/2 では，なにか問い合わせを実行中のエージェントに対しては問い合わせを行うことはできず，単純に失敗するようになっていた．このような場合に，何か問い合わせを実行中のエージェントに対して，その問い合わせの完了を待ってから，改めて問い合わせを行う機能として，queryF/2 を用意している．

エージェント a と エージェント b を用意し，エージェント b の 側のプログラムを

```
helloMiLog :- writel('Hello MiLog World!').
helloMiLogLoop :- writel('MiLog World he Youkoso!'),helloMiLogLoop.
```

として reconsult しておき，さらにエージェント b の側で

```
?- helloMiLogLoop.
```

として，エージェント b を無限ループでずっと問い合わせ実行中の状態にする．この状態で，エージェント a の側から

```
?- queryF( b, helloMiLog).
```

とすると，この述語の実行は終了せず，待ち状態になる．ここで，エージェント b 側の無限ループプログラムを STOP ボタンで止めてやると，その直後に，エージェント b で問い合わせが実行され Hello MiLog World! とエージェント b に出力されたはずである．このように，queryF/2 では，エージェントの問い合わせ実行終了までしつこく待って問い合わせを行うようになっている．

(ヒント：通信相手のエージェントが暇そうにしているときには，queryF/2 と query/2, のどちらを使うといいか？という疑問が出てくる．このときには，なるべく，queryF/2 の方を使いたい．query/2 は，明示的な問い合わせの処理中以外に，MiLog 内部での処理の最中にも，その実行を拒絶されることがあるからである．たとえば，同一にエージェントに対して，２度連続して query/2 で問い合わせを実行させた場合，その２つがある特定のタイミングで実行されると，そのときだけ，なぜか理由もなく２度目の問い合わせが失敗してしまうことがある．その原因は，１つめの query/2 の結果が返された直後には，まだその問い合わせ先のエージェントは先ほどの後始末を MiLog 内部処理として行っている最中で，そのときに query/2 を続けて行っても，その後始末が処理中と解釈されて，拒絶されてしまうからである．実際には MiLog の内部処理にかかる時間は非常にわずかなのだが，CPU が１つしかない場合，OS のスレッドの管理方法の都合によっては，待っていたスレッド (query を送っているエージェントのスレッド) が起きた瞬間にそのスレッドに非常に大きな比重で CPU 時間を割くようになっていたりして，ごくわずかな時間で済むはずの問い合わせ後の内部処理の実行が，ずっと後回しにされてしまうことがある．これは並行プログラミングに固有な問題の１つだが，知らないと解決に苦しむ問題であるので，できれば避けて通りたいところである．)

4.3.5 queryF と query の典型的な使用方法

queryF/2 は，主な用途として，連続的な問い合わせの実行，共有データへのアクセスとそのときのセマフォの自動化への使用を想定している．

query/2 は，主な用途として，負荷分散（複数のエージェントの中でとりあえず手の空いたエージェントへのタスクの依頼）への使用を想定している．

queryF/2 のサンプル | 共有 DB

```
---以下は db_agent 用のプログラム---
  data(chika,0).
  data(fukuta,1).
  data(itota,2).
  modify_db(add,X) :-
    data(X,Data),
    NewData is Data + 1,
    retract(X,Data),
    assert(X,NewData).
  modify_db(del,X) :-
    data(X,Data),
    NewData is Data - 1,
    retract(X,Data),
    assert(X,NewData).
  modify_db(move,X,Y) :-
    modify_db(del,X),
    modify_db(add,Y).

  ---以下は client_agent 用のプログラム---
  name_data([chika,fukuta,itota]).
  cmd_data([add,del,move]).
  db_agent_name(db_agent).
  shuffle(List,XX) :-
    length(N,[a,b,c]),
    sys_random(X),
    Y isD X * N + 1,
    once(extractA(Y,[NN,'.',_])),
    nth(NN,List,XX).
  nth(0,[XX|_],XX) :- !.
  nth(NN,[_|RR],XX) :-
    N is NN-1,
    nth(N,RR,XX).
  random_cmd_loop :-
    cmd_data(CMDLIST),
    shuffle(CMDLIST,CMD),
    random_cmd(CMD),
    random_cmd_loop.
  random_cmd(move) :-
    !,name_data(NAMELIST),
    shuffle(NAMELIST,NAME1),
```

```
    shuffle(NAMELIST,NAME2),
    db_agent_name(DBAGENT)
    queryF(DBAGENT,modify_db(move,NAME1,NAME2)).
  random_cmd(CMD) :-
    name_data(NAMELIST),
    shuffle(NAMELIST,NAME),
    db_agent_name(DBAGENT),
    queryF(DBAGENT,modify_db(CMD,NAME)).
```

このプログラムでは，複数のクライアントプログラムからランダムに与えられるコマンド (add/del/move の３種類) を逐次処理する共有 DB 制御プログラムを作っている．あらかじめ共有 DB 管理エージェントとして db_agent という名前のエージェントを作って，db_agent 用のプログラムの部分を reconsult しておく．次に，適当な名前のエージェントをいくつか作って，そのそれぞれに，client_agent 用のプログラムを reconsult して，クライアントエージェントとする．各クライアントエージェントから，

```
?- random_cmd_loop.
```

と実行すると，クライアントにランダムにコマンドを送り始める．複数のクライアントから同時に db_agent に問い合わせが届くが，ある１つの問い合わせ中には別の問い合わせの実行はブロックされており，各コマンドの処理中（retract して，まだ assert しなおす前）に間違って別のコマンドが実行されてしまうことが無いようになっている．このように，非常に簡単に共有 DB の構築ができる（注：ただし，このやり方では，あくまでも簡単なサンプルを示すという都合から，いわゆる待ち行列という構造にはなっておらず，アクセスの公平性は必ずしも保証されない.）

```
query のサンプル | 仕事の押しこみ
---- 以下は prof_agent 用のプログラム----
person(fukuta).
person(chika).
person(itota).
task_push_loop :-
  person(X),
  query(X,push_task(heavytask)),
  task_push_loop.

---- 以下は エージェント fukuta,chika,itota用のプログラム ----
push_task(_) :-
  writel('moudamedesuu...').

---- サンプルを動かす場合には，それぞれのエージェントに
     それぞれ対応した部分のプログラムをコピーして reconsult する --
```

このプログラム例では，１つのクライアントエージェント (prof_agent) が，とりあえずその場で手の空いているサーバエージェント (fukuta,chika,itota) のうちの１つに，仕事を投げるようなプログラムである．

基本的には，クライアントエージェントは特定のサーバエージェントに対してひたすら仕事を投げ続けるが，ときどき，いそがしそうにしていると，他のサーバエージェントにも仕事を投げる．たとえば，普段タスクを投げられ続けるサーバエージェント上で `?- sleep(10000).` として 10 秒ほど眠らせると，他のサーバエージェントにタスクを投げるようになる．(なお，このサーバエージェントは，あまりにもたくさん仕事を投げられすぎていて，sleep という問い合わせを受け付けない (眠る間もない?) かもしれないので，そのようなときには，いったんクライアントエージェントの動作を停止してから sleep を実行させて眠らせ，その後にクライアントエージェントの動作を再開させるとよい．)

4.3.6 非同期通信：request/requestF/wait/know

述語 query/2 および queryF/2 は，他のエージェントに問い合わせを依頼したら，その問い合わせの結果が得られるまでじっと待つようにできている．一方で，問い合わせを依頼したらその結果が返ってくるまでの間に別の仕事をしたいような場合もある（これを非同期通信といい，また，query/2 のような通信を同期通信という.）.

このために，MiLog には，非同期エージェント間問い合わせ述語として，request/2 および requestF/2 が用意してある．request/2 と requestF/2 の違いは，queryf/2 と queryF/2 の違いと同じで，依頼先のエージェントが依頼時に仕事中の場合，仕事の依頼をあきらめるか，あるいはしぶとくその仕事が終わるのを待って依頼をするか，の違いである．

request/2 あるいは requestF/2 による問い合わせの依頼では，依頼先でその問い合わせが開始された直後に（その問い合わせの実行終了を待たずに），この述語自身の実行から抜け，続けて他の処理を行うことができるようになっている．つまり，他のエージェントに仕事を依頼しつつも，自分も他の仕事をこなすようなことができる．request/2 あるいは requestF/2 による問い合わせは依頼を行うだけですぐに終了してしまうため，その問い合わせの結果がそのままでは得られない．request/2 や requestF/2 での問い合わせの結果を得るには，wait/2 あるいは know/3 を使う．wait/2 は，問い合わせの結果が届くまで待ってその結果を受け取り，問い合わせが万が一失敗であった場合には，wait/2 も失敗するようになっている．

たとえば，次のようなプログラムの場合，

```
test_program :-
   request( student, do_heavy_task(Result) ),
   writel('Eat an apple.'),
   writel('Eat anther apple.'),
   wait(student,query(do_heavy_task(Result))).
```

このプログラムを持ったエージェントが

```
?- test_program.
```

を実行した場合,（student という名前のエージェントがいると仮定すると）最初に，エージェント student に問い合わせ ?- do_heaby_task(Result). の実行を依頼し続けてリンゴを２つ食べた（writel/1 の２行）後，のんびり問い合わせの結果を待つ．ここで注意すべき点は，wait/2 の第２引数には，問い合わせを query(...) で囲って記述するになっている点である．エージェント student 上での問い合わせ実行が成功すれば，その結果は変数 Result に束縛される．もちろん，エージェント student が ?-do_heavy_task(Result). の問い合わせ実行に失敗した場合には wait/2 の実行も失敗し．この傲慢なテストプログラムの野望も fail することになる．

一方で，know/3 を使った場合には，その場で問い合わせ結果が届いているかどうかを確認し，届いていればその結果を得ることができ，届いていなければ即座に失敗する．たとえば，次のようなプログラムの場合，

```
test_program2 :-
  request( student, do_heavy_task(Result) ),
  writel('Eat an apple.'),
  writel('Eat another apple.'),
  loop,
  writel('mada?'),
  know( student, query( do_heavy_task(Result) ), TorF ),
  !,
  TorF = true.
```

このプログラムを持ったエージェントが

```
?- test_program2.
```

を実行した場合,(先ほどと同様に，エージェント student が存在したと仮定すると）最初に，エージェント student に問い合わせ ?- do_heaby_task(Result). の実行を依頼し続けてリンゴを２つ食べた (writel/1 の２行) 後，問い合わせの結果が届くまで，何度もしつこく 'mada?' と言い続ける．ここで注意すべき点は，wait/2 の第２引数と同様に，know/3 の第２引数も，問い合わせ部分を query(...) で囲って記述するになっている点である．問い合わせが成功したかどうかは，know/3 の第３引数である TorF に束縛され，成功した場合は，true，失敗した場合は fail にそれぞれ束縛される．問い合わせが成功すれば，その結果は変数 Result に束縛される．

なお，know/3 も，wait/2 も，１つの request に対する結果を受け取れるのは，１度だけである．たとえば，

```
test_program3 :-
  request( student, do_heaby_task(Result) ),
  know( student, query( do_heavy_task(Result) ), TorF),
  know( student, query( do_heavy_task(Result) ), TorF).
```

というようなプログラムを作って，

?- test_program3. とした場合，

(以前 request した結果を偶然受け取り忘れている場合を除き）仮にエージェント student 上での問い合わせ ?-do_heavy_task(Result). の実行結果が戻ってきていたとしても，それを受け取れるのは１つ目の know/3 だけで，２つ目の know/3 は成功しない．

また，同様に

```
test_program4 :-
  request( student, do_heaby_task(Result) ),
  wait( student, query( do_heavy_task(Result) )),
  wait( student, query( do_heavy_task(Result) )).
```

というようなプログラムを作って，

```
?- test_program4.
```

とした場合,(以前 request した結果を偶然受け取り忘れている場合を除き）仮にエージェント student 上での問い合わせ?- do_heavy_task(Result が成功していたとしても，その答えを受け取れるのは １つ目の wait/2 だけで，２つ目の wait/2 は無限に答えを待ち続け，プログラムは終了しない．

4.3.7 request/requestF の典型的な使い方

requestF/2 は，複数エージェントへのタスクのアサインによる並行処理の実現に使用されることを想定している．

request/2 は，たとえば契約ネットプロトコルのようなエージェント間強調プロトコルを簡易に実装するような場合に使用されることを想定している．

```
---- 以下は prof_agent 用のプログラム ----
person(fukuta).
person(chika).
person(itota).
task_push_loop :-
  findall(_,my_call([person(X),
```

```
                requestF(X,push_task(heavytask))]),_),
    trush_response,
    task_push_loop.
trush_response :-
    know(_,query(push_task(heavytask)),TorF),
    fail.
trush_response.
---- 以下はエージェント fukuta,chika,itota 用のプログラム----
push_task(_) :-
    writel('moudamedesuu...').
---- プログラムを実際に動かす場合には，対応するエージェント用の
     部分をコピーして reconsult しておく ------
```

このプログラムでは，

```
?- task_push_loop.
```

が エージェント prof_agent で実行されると， 3つのサーバエージェント (fukuta, chika, itota) に対して，同時並行して問い合わせを行い，その結果を何も見ずに破棄するのを繰り返す，という動作をしている．このプログラムでは，requestF/2 の特性から，エージェントがすでに問い合わせ実行中の場合は，その次の問い合わせ実行を依頼するのを待つようになっているため， 3つのプログラムのが，ほぼ同期して動く様子がわかる．

```
---- 以下は，contractor_agent 用のプログラム ----
person(fukuta).
person(chika).
person(itota).
contract_net :-
    findall(Agent,person(Agent),Agents),
    send_contract_request(Agents),
    sleep(1000),
    receive_contract_bids(Bids),
    sort_bids(Bids,[[Agent,Price]|_]),
    queryF(Agent,push_task(heavytask)).
send_contract_request([]).
send_contract_request([A|R]) :-
    request(A,bid_to_task(lighttask,Bid)),
    !,send_contract_request(R).
send_contract_request([_|R]) :-
    send_contract_request(R).
receive_contract_bids([[A,Bid]|R]) :-
    know(A,query(bid_to_task(_,Bid)),true),
    !,receive_contract_bids(R).
receive_contract_bids([]).

sort_bids(BidList,Ordered) :-
    sort(BidList,Ordered,Key,[_,Key]).
sort([],[],Key,Ptn).
sort([PtnX|R],RES,Key,Ptn) :-
    renameVars([Key,Ptn],[NKey,NPtn]),
```

```
    PtnX = NPtn,
    sort_divide(NKey,R,RR1,RR2,Key,Ptn),
    sort(RR1,RRS1,Key,Ptn),
    sort(RR2,RRS2,Key,Ptn),
    append(RRS1,[PtnX|RRS2],RES).
sort_divide(N,[],[],[],_,_).
sort_divide(N,[PtnX|R],[PtnX|RR1],RR2,Key,Ptn) :-
    renameVars([Key,Ptn],[NKey,NPtn]),
    PtnX = NPtn,
    '>'(N,NKey),!,
    sort_divide(N,R,RR1,RR2,Key,Ptn).
sort_divide(N,[X|R],RR1,[X|RR2],Key,Ptn) :-
    !,sort_divide(N,R,RR1,RR2,Key,Ptn).

---- 以下は，contractee agent（つまり契約を受けるエージェント
      fukuta chika itota の３つ) 側のプログラム．-----

bid_to_task(lighttask,X) :-
    sys_random(RND),
    X isD RND * 3.
push_task(heavytask) :-
    writel('moudekimashen...').

----- プログラムを実際に動かす場合には，それぞれのエージェント用
      の部分をコピーして reconsult しておく ------
```

このプログラムでは，交渉を開始するエージェント contractee_agent から

```
?- contract_net.
```

とすると交渉が開始され，まず最初に交渉相手となる３つのエージェント itota,chika, fukuta に対して，あるタスク lighttask に関するコストの見積もりを問い合わせする．その問い合わせは，答えを待つことなくエージェントに対して一斉に行われ，それらのエージェントの中で現在問い合わせを実行中であるときには，その問い合わせがスキップされる．その後，もっともコストの見積もりの低いエージェントを見つけ，そのエージェントに対して改めて queryF/2 を用いてタスクの割り当て (問い合わせ実行) を行い，結果を受け取るよういなっている．

sort の部分が少々複雑だが，実際にエージェント間問い合わせを行っている部分はごく単純なものである．

4.3.8 割り込み問い合わせ — interruptAndQuery/2 述語の動作の意味

割り込み問い合わせ述語を使うと，何か問い合わせを実行中のエージェントに対して，即座に割り込んで別の問い合わせを実行させることができる．そのエージェントが割り込み問い合わせを受ける前に実行していた問い合わせは，割り込み問い合わせを受け付けた段階で保留され，割り込み問い合わせが終了した段階で，再開される.（なお，即座に割り込みで問い合わせを行うため，request や query にあった 'F' のついた述語はなく，１種類のみである．）

たとえば，２つのエージェント a および b があって，エージェント b に

```
loop_and_count :-
```

```
    retract(count(N)),
    write(N),
    M is N + 1,
    assert(count(M)),
    loop_and_hello.
count(1).
```

というプログラムがあって,

```
?- loop_and_count.
```

とされているとき，この問い合わせにより，随時節データベースにある count/1 の値が 1 つづつ増やされながら，更新されている．このときに，エージェント a から,

```
?- interruptAndQuery( b, count(X) ).
```

とすると，エージェント b では，すでに行っている問い合わせ ?- loop_and_count．が中断され，割り込みで，?- count(X)．が実行される．その結果として，たとえばそのときに 256 までカウントが進んでいれば X = 256 が割り込み問い合わせの結果として得られる．

エージェント b では，割り込みの問い合わせが終了した後，中断されていた ?- loop_and_count．が再開され，割り込み問い合わせの前と同様に count/1 の値を 1 つづつ増やしていく．このように，割り込み問い合わせを使うと，問い合わせ実行中のエージェントが持つ内部データを，即座に参照することができる．

割り込み問い合わせは，副作用を伴うような処理に使うこともできる．たとえば，2 つのエージェント a および b があって，エージェント b に

```
loop_and_hello :-
  writel('Hello MiLog World'),
  loop_ok,
  loop_and_hello.
loop_stop :-
  writel('Stopping loop...'),
  retract(loop_ok).
```

というプログラムがあって，?- loop_and_hello．とされているとき,（つまり，無限ループしながら文字を表示し続けるようなとき）エージェント a から,

```
?- interruptAndQuery( b, loop_stop ).
```

とすると，エージェント b では，すでに行っている問い合わせ ?- loop_and_hello．が中断され，割り込みで，?- loop_stop．が実行される．この問い合わせの実行により，loop_ok が retract/1 によって節データベースから削除される．この状態で，もとの問い合わせ ?- loop_and_hello． が再開される．loop_ok が削除されているため，結果として無限ループが止まることになる．

節データベースの削除以外にも，エージェントを別の場所移動させるといった用途にも使える．たとえば，エージェント b に

```
loop_and_count :-
  retract(count(N)),
  write(N),
  M is N + 1,
  assert(count(M)),
  loop_and_hello.
count(1).
```

というプログラムがあって，?- loop_and_count．とされているとき，そのエージェントに対して，エージェント a で

```
?- interruptAndQuery( b, move('192.168.0.3' ).
```

とすると，エージェント b は，ホスト '192.168.0.3' に移動して，それまで通り count/1 の更新を続ける．

4.3.9 非同期割り込み問い合わせ：interruptAndRequest/2

query/2 に request/2 があるように，interruptAndQuery/2 にも，それに対応した非同期動作版の述語 interruptAndRequest/2 がある．interruptAndRequest/2 の動作は，interruptAndQuery とほぼ同様だが，割り込みで行った問い合わせの完了を待たず，その結果を know/3 や wait/2 で受け取る．

たとえば，エージェント a, b, c, d があって，エージェント a, b, および c が何らかの問い合わせを実行中の時，エージェント d に

```
agent_to_be_moved(a).
agent_to_be_moved(b).
agent_to_be_moved(c).
moveThemAll :-
  agent_to_be_moved(X),
  interruptAndRequest(X,move('192.168.0.3')),fail.
moveThemAll.
```

というプログラムがあって ?- moveThemAll．とした場合，3つのエージェント a, b, および c を，一斉に '192.168.0.3' に動かして，問い合わせを続けさせることができる．

4.3.10 異なる MiLog 実行環境間でのエージェント間問い合わせ

エージェント間問い合わせ述語 query, queryF, request, requestF, interruptAndQuery, および interruptAndRequest は，3番目の引数として MiLog 実行環境のアドレスを記述することで，異なる MiLog 実行環境上で動いているエージェントに遠隔で問い合わせを行うことができる．

たとえば，'192.168.0.1' にエージェント a がいて，'192.168.0.3' にエージェント b がいる場合，エージェント a から

```
?- queryF( b, do_task, '192.168.0.3').
```

とすると，エージェント b に対して問い合わせを行い，その結果を受け取ることができる．

エージェントが移動をした場合，エージェント間問い合わせが行えなかったり，あるいは問い合わせの結果を受け取れなかったりする場合がある．

すでに他の場所へ移動してしまったエージェントに対する問い合わせは，その移動先のアドレスを指定しない限り，失敗する．たとえば，エージェント b が '192.168.0.4' に移動してしまった場合，エージェント a から

```
?- queryF( b, do_task, '192.168.0.3').
```

としても，問い合わせは単純に失敗する．

エージェントに問い合わせを行っている最中に他の場所に移動すると，その問い合わせが完了するまでの間にもとの場所に戻ってこない限りその問い合わせの結果を受け取ることができない．たとえば，エージェント b が '192.168.0.4' にいるときエージェント a から

```
?- requestF( b, do_task, '192.168.0.4' ), move('192.168.0.3').
```

とすると，エージェント a は エージェント b への問い合わせの依頼の直後に別の場所に移動してしまうため，この問い合わせの結果を know/3 や wait/2 で受け取ることはできない．なお，これとは逆に，問い合わせを受ける側のエージェントが移動した場合には，その問い合わせが完了したときに通信路が切断されているようなな場合を除き，問い合わせの結果を受け取ることができる．たとえば，エージェント a から

```
?- queryF( b, move('192.168.0.5'), '192.168.0.4').
```

とすると，エージェント b が '192.168.0.5' に移動し，その移動がうまくいったかどうかの結果をエージェント a は受け取ることができる．

4.3.11 節データベースの共有 : clone2

clone2 を使うと，節データベース（＝エージェントのプログラム）を複数のエージェントで共有することができる．clone2 の基本的な使い方は，new/1 と同じである．

たとえば，エージェント a から，

```
?- clone2(x).
```

とすると，指定した名前'x' のエージェントが生成される．

ここで生成されたエージェントは，それを生成した元のエージェント a と節データベースを共有している．このとき，エージェント x を エージェント a のクローンエージェントと呼ぶことにする．

節データベースが共有されているので，エージェント a で行われた assert や retract による節データベースへの操作は，エージェント x へも反映される．たとえば，エージェント a で，

```
?- asserta( task(one) ).
```

とすると，エージェント a に task(one). という節が挿入されるが，それはエージェント x にも反映されている．たとえばこの後，エージェント x で

```
?- task(X).
```

とすると，

```
X = one
```

となる．この操作と逆に，クローンエージェントである エージェント x で行われた節データベースへの操作も，その元のエージェント a に反映される．たとえば，エージェント x で

```
?- retract(task(one)).
```

とすると，エージェント x で task(one). という節が削除されるが，それと同時に，エージェント a でも task(one). という節が削除される．

クローンエージェントを使った節データベースの共有は，現在のところ，エージェント間通信述語よりも高速に動作するため，並行処理の最中に頻繁なデータのやりとりが必要な場合に使うと，処理速度を高速にできる．

クローンエージェントとの節データベース共有は，元のエージェントかクローンエージェントのいずれかが別の場所へ移動するまでの間，有効である．つまり，節データベース共有は，同一の MiLog 実行環境上でのみ可能である．クローンかオリジナルかいずれかのエージェントが移動したときに，その移動したエージェントの節データベースは，他のエージェントとの共有が解除され，その時点での節データベースのコピーを持った，独立した１つのエージェントとして動作するようになる．

なお，クローンエージェントは複数作ることができる．

クローンエージェント機能は，１つの共有データに対して並行して探索・処理を行うようなプログラムの作成に利用できる．

たとえば，

```
task(one).
task(two).
task(three).
task(four).
task(five).
do_task :-
  retract(task(X)),
  next_task(X,Y),
  assert(task(Y)),
  !,do_task.
do_task :-
  sleep(1000),
  !,do_task.
next_task(X,Y) :-
  name(X,[I|XNAMELIST),
  writel([done,I]),
  name(Y,XNAMELIST).
```

というプログラムを持つエージェント a とそのクローン x, y がいて，それぞれに対して，問い合わせ

```
?- do_task.
```

を実行すると，one, two, three, four, five の５つのアトムに対して，先頭から１文字ずつ取り出して表示する処理をエージェントとクローンエージェントが並行して行う．

CPU コアが複数あるような状況なら，クローンを使うことで複数 CPU の並列処理による高速化が期待でき，仮に CPU コアが１つしかない状況でも，ネットワークを介した処理など，待ち時間の多い処理を複数行う必要がある場合には，その待ち時間に他の処理を行うことで効率的に処理を進めることができる．

作成したクローンは，`delete/0` で削除することができる．エージェントかそのクローンの最後の１つが削除されたときに，そこで共有されていた節データベースも消滅する．

4.3.12 一定時間おきに同じ問い合わせ実行を繰り返す：repeatRequest

エージェント間通信とは直接関係ないが，エージェントに一定時間ごとに繰り返し同じ問い合わせを実行させるための述語として `repeatRequest/3` が用意してある．これは，たとえば一定間隔ごとに特定の Web サイトの情報を収集して更新するようなエージェントの処理に向いている．

本来であれば，プログラム中にループを作って，一定時間 `sleep/1` 等で待たせた後に処理を実行するように書けばいいのだが，こうすると，その待っている期間に他のエージェントから（割り込みでない）エージェント間問い合わせを受け付けることができない．この制限を緩和するために，`repeatRequest/3` が用意してある．

たとえば，エージェント a に

```
helloMiLog :-
  writel('Hello MiLog World!').
```

というプログラムがあるとき，

```
?- repeatRequest( helloMiLog, 1000, X).
```

とすると，約１秒ごとに，`Hello MiLog World!` が表示される．（このとき，エージェントモニターを開いているとわかるが，繰り返し問い合わせを実行させるための監視エージェントが別途生成される．このエージェントは，間違ってもリサイクルマークのアイコンへドラッグアンドドロップして消してはならない．）このとき，変数 X に特定の ID が束縛される．この ID は，問い合わせ実行の繰り返しをやめるときに使う．

問い合わせを繰り返し行うのをやめたい場合には，deleteRepeatRequest/1 を使う．先ほどの問い合わせで，変数 X の値が repeatRequest0 だった場合，

```
?- deleteRepeatRequest(repeatRequest0).
```

とすると，繰り返し行われていた問い合わせが，最後の問い合わせの実行の実行が完了したあとは，行われなくなる.（このとき，エージェントモニターを開いていると，監視エージェントが削除されていなくなるのがわかる.）なお，deleteRepeatRequest/1 の引数に変数を使うこともでき，

```
?- deleteRepeatRequest(X).
```

とすれば，任意の最近の repeatRequest/3 による問い合わせの繰り返しを停止させることができる．また，この機能を使うことが前提で ID を取る必要がないときのために，３番目の引数を省略した repeatRequest/2 も用意している．

repeatRequest/2,3 による問い合わせの繰り返しは，エージェントが他の場所へ移動したときも継続されるようになっている．また，１つのエージェントに対して複数の repeatRequest/2,3 を与えることもできる．ただし，repeatRequest/2,3 の時間管理はかなりいい加減なので，正確な時刻ごとに問い合わせを行う必要がある場合には，repeatRequest/2,3 は，あまり向いていない．

4.3.13 query 系問い合わせ述語全般に関する注意

エージェントの停止を伴う動作を query するとハングアップしたように見えることがあるが，これは仕様である．なぜこのような現象が起きるのかを，以下で説明する．

具体的な状況としては，今，init というエージェントがいたとして，

```
?- query(init,moveToFile).
```

とか

```
?-query(init,suspend).
```

といったエージェント間問いあわせを適当なエージェントから実行すると，この問い合わせの実行直後に，moveToFile/0 や suspend の副作用として相手のエージェントのインタプリターが無くなったり止まったりするため，query/2 の結果が相手のエージェントのインタプリタ―から返ってこない．そのため，問い合わせを送った側のエージェントは，届くはずのない返事を無限に待ち続けることになり，query/2 は永遠に終わらなくなる．このような場合には reqest 系述語を使ってほしい．

なお，これはバグではなくて，そのような仕様してある．人間にとっては，moveToFile/0 や suspend/0 という述語の実行がどういう状態になったら完了なのかはすぐわかるように思えるのだが，それは人間には背景知識があって，状況をメタな視点から見ることができて，かつ，偶然その問い合わせ内容が簡単だったからである．実際に（人間が）行っている認知動作は，エージェントがいなくなるのを目で確認し，その結果として問い合わせが完了したことを認知している，つまり，プログラムに置き換えるなら，

```
?- request(init,moveToFile), loop, not(thereExists(init)).
```

というようなことをしている．これは，moveToFile というものの副作用としてそのエージェントが存在しなくなることを知っていなくては，できない．

つまり，こういうことを意図した動作をプログラムにさせたい場合には，背景知識を生かして request と loop などを組み合わせたプログラムを書いてやれば，意図通りに動く．人間は何も意識して考えなくても「目で見てわかる」のでこういう違いを意識しないのだが，プログラムとしてはきちんと意識して，区別して書いてやる必要がある．

（ちなみに，moveToFile/0 という述語は，ファイルに移動したらそれで終わりというわけではなくて，ファイルから戻ってきたときに自分自身の身なりを直す処理も（内部的な処理として）含まれている．だから，実際に

第6章

Java との連携

この章では，MiLog で作ったプログラムを Java で記述したプログラムから利用したり，MiLog に新しい組込述語を追加したり，さらに MiLog 自身を拡張するための方法について説明する．

6.1 Java プログラムからの呼び出し

MiLog には，Java プログラムから MiLog プログラムを利用するための機能が提供されている．ただし，Java プログラムからの MiLog プログラムの呼び出し機能に関しては，もともと，MiLog 自身の構築に使っていた内部 API を単に外部からも使えるようにした程度のものなので，まだまだ扱いにくい部分があったり，MiLog の言語仕様と用語の使われ方が違っていたりすることもある．これらの扱いにくさはあるものの，自分で作った Java プログラムの中に部分的に MiLog プログラムを組み込めるというのは，やはりメリットが大きい．

MiLog エージェントは，Java プログラムからは，MiLog クラスのインスタンスとしての Java オブジェクトとして扱うことができる．よって，Java 言語で作成したプログラム中から，MiLog エージェントを利用することができる．Java プログラムから MiLog エージェントを扱う手順は，(1) エージェントの生成，(2) エージェントでの query の実行，(3) query の結果の取り出し，(4) エージェントの削除，の 4 ステップである．Java プログラムから呼び出されたエージェントは，他の計算機へ移動する事ができないという制限がある．しかし，そのエージェントのクローンは，マイグレート可能である．クローンをマイグレートする方法を，(5) モバイルエージェントの生成と利用として簡単に説明する．最後に，簡単なサンプルプログラムを示す．(なお，文章中にクラス名やパッケージ名として ‘weblog’ という単語が多数出現する．これは，MiLog の初期の開発コードネームであり，その名残として残っている．)

6.1.1 エージェントの生成

MiLog エージェントは，weblog.Weblog クラスのインスタンスとして，任意の Java プログラム中で利用することができる．MiLog エージェントのインスタンス生成には，weblog.Weblog クラスのコンストラクタ weblog.Weblog(String name) を利用する．コンストラクタの第 1 引数には，そのエージェントの名前を指定する．このエージェントの名前は，MiLog 上での new/1 述語と同様に，ユニークである必要がある．例えば，次のようにする．

```
weblog.Weblog myAgent = new weblog.Weblog("taro");
```

結果として，weblog.Weblog 型の変数 myAgent に MiLog エージェント ‘taro’ が代入される．以降 myAgent を用いて MiLog エージェントを制御することが可能になる．

生成したばかりの MiLog エージェントは，標準ライブラリ述語だけを consult する．このため，ユーザのプログラムを MiLog エージェントに読み込ませるためにはに，明示的にエージェントに reconsult を実行させる必要がある．MiLog エージェントにプログラムを reconsult させるには，void reconsult(String program) メソッドを使えばよい．この reconsult メソッドの引数には，“write(ok).” のように，MiLog プログラムを表す String 型のオブジェクトを指定する．例えば，次のようする．

```
// 論理プログラム "test :- write(ok)." を reconsult
myAgent.reconsult("test :- write(ok).");
```

これにより myAgent の節データベースには，プログラム `test :- write(ok).` が追加される．

MiLog プログラムが記述されたのファイルを reconsult させたい場合は，reconsultFromFile(String filename) メソッドを使う．例えば，次のようする．

```
// ソースファイル "samplefile.txt" を reconsult
myAgent.reconsultFromFile("samplefile.txt");
```

これにより myAgent の節データベースには，ソースファイル ”samplefile.txt” に記述されたプログラムが追加される．

制限事項：このようにして作ったエージェントには，move/1 述語などを使ったエージェントの移動はサポートされていない．また，このようにして作ったエージェントに対する，他のエージェントからの interruptAndQuery/2 などによる割り込み問い合わせも，サポートされない．

6.1.2 エージェントへの query の受け渡し

MiLog エージェントに，質問を評価させるには，PrologObject syncQuery(String query) メソッドを使う．この query メソッドの引数には，そのエージェントに実行させたい質問を MiLog のフォーマットで記述する．例えば，“?- doYourWork(Result).” という質問を Java プログラム中から評価させるには， 次のように記述する．

```
PrologObject answer;
answer = myAgent.syncQuery("doYourWork(Result).");
```

例のように質問の末尾には，’.’（ピリオド）を付加する必要がある．MiLog エージェントは，シングルタスクなので，同時に 1 つの質問だけを評価する．さらに，一度評価し終わった質問の別解探索ができないという制限がある．これは，MiLog 上での制限と同様である．

上記の syncQuery メソッドは，MiLog 上の query/2 に相当する．request/2 に相当するメソッドは，void query(String query) メソッドである．このメソッドでは，query を起動するが，query の評価の開始と同時にメソッド呼び出しが終了する．これは，特定のエージェントの動作を開始（trigger）するのに便利である．例えば，無限ループを含むプログラムを実行させる場合などに利用するのが有効である．以下に，例を示す．

```
myAgent.query("startYourWork.");
```

この例は，myAgent に質問 ”startYourWork.” を評価させる例である．syncQuery メソッドの場合と異なり，query メソッドを呼び出しによりエージェントが質問の評価を開始した時点で，query メソッドの呼び出しは終了する．ここで，質問 ”startYourWork.” が無限ループをするプログラムであるならば，syncQuery メソッドで質問を評価させるとメソッド呼び出しが終了しないので不都合である．よってこの場合は，sync メソッドの利用が適している．

6.1.3 query の実行結果の受け取り

メソッド syncQuery は，返値として，クラス PrologObject 型のオブジェクトを返す．クラス PrologObject 型は，MiLog における基本的なデータ構造，すなわちアトムや関数子などを表現するためのクラスである．返値は，複数の PrologObject がリスト状に連なったものである．syncQuery メソッドが，返値として PrologObject のインスタンスを返す理由は，Java のプリミティブ（int や String など）だけでは，MiLog 上で扱われている全ての種類のデータを表現できないからである．ただし，あらかじめ答えの形式が（例えば，アトムのリストなどと）わかっている場合のために，PrologObject を（例えば，String 型の要素を含む Vector などの）特定の Java プリミティブへ変換するためのユーティリティメソッドが用意されているが，これについては，後述する．以下，PrologObject クラスについて，簡単に説明する．後述のユーティリティメソッドで不十分な場合に，以下の情報が役に立つ．

PrologObject 型のオブジェクトは，外見上は単一の Java クラスのインスタンスであるが，内部に独自の型情報を持っている．なぜ内部に型情報を持つかというと，Java の型キャストによるオーバヘッドを抑えることで，MiLog インタプリタの動作速度を向上させるためである．PrologObject 型が内部に持つ型情報は，type メンバーから得ることができる．例えば，アトム ’a’ は，単一の PrologObject から構成され，type メンバー

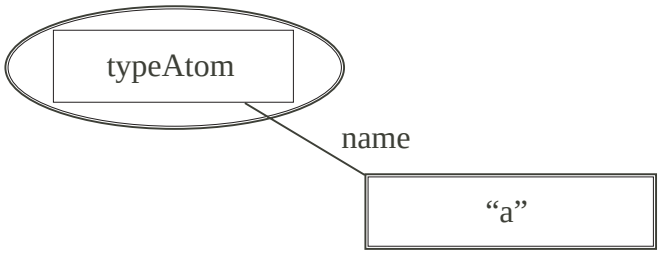


図 6.1: アトム

に PrologObject.typeAtom が int 型の定数として入っており，name メンバーにそのアトムの名前 ’a’ が String 型で入っている．

関数子 ’f(X)’ は，２つの PrologObject から構成される．先頭のオブジェクトの type メンバーに PrologObject.typeFunc が int 型の定数として入っており，name メンバーにその関数子の名前 ”f” が入っている．先頭のオブジェクトの cdr メンバーに，引数”X”を表現するためのオブジェクトの参照が入っている．引数オブジェクトはこの場合１つの引数のみを取るので，第１引数そのものとなる．この場合，引数（変数 ”X”）オブジェクトの name メンバーにその変数の変数名”X”が入っている．もし，その変数に何か値が束縛されているならば，束縛されている内容を表現するオブジェクトへの参照が（何も束縛されていないときには null が），cdr メンバーに入っている．

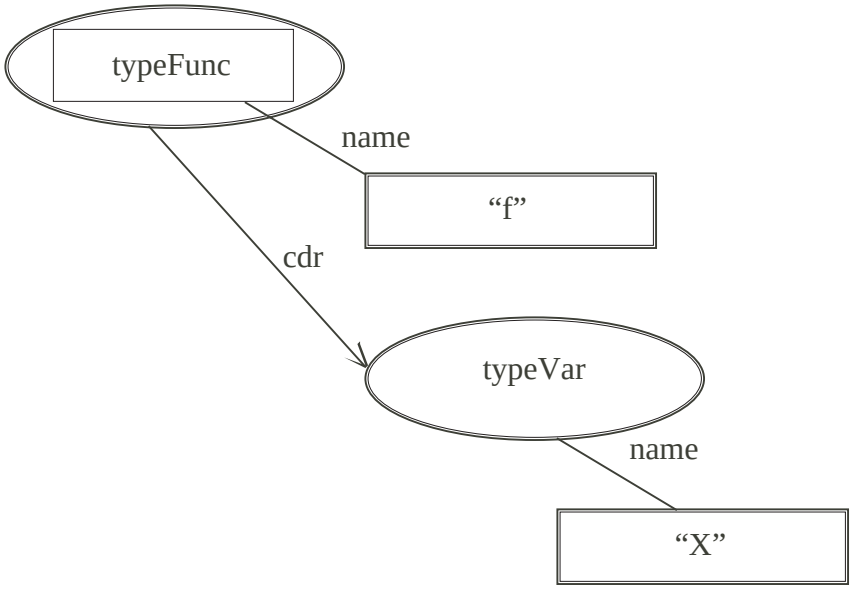


図 6.2: 1 引数関数子の図

引数が２つ以上ある関数子の場合，引数は，typeAnd 形式のリストとして表現される．この typeAnd 形式のリストは，オブジェクトの car 部分に先頭の要素を持ち，cdr 部分に残りの要素を含むリストへの参照を持つ．ただし，残りの要素がただ１つの時には，その要素そのものへの参照を持つ．例えば，関数子”g(h, i, j)”の引数は，２つの typeAnd 形式のオブジェクトに格納される．先頭のオブジェクトの car メンバーには，アトム ’h’ を表現する PrologObject への参照が入っている．先頭のオブジェクトの cdr メンバーには，残りの引数を格納している typeAnd 形式のオブジェクトへの参照が入っている．このオブジェクトの car メンバーには，アトム’i’ を表現する PrologObject への参照が入っており，cdr メンバーには，アトム’j’ を表現する PrologObject への参照が入っている．

”[a,b,c]”などのリスト構造は，typeList 形式のリストとして表現される．この typeList 形式は，typeAnd 形式と異なり，残りの要素がただ１つの時には，cdr メンバーに，その要素を含む typeList 形式のオブジェクト

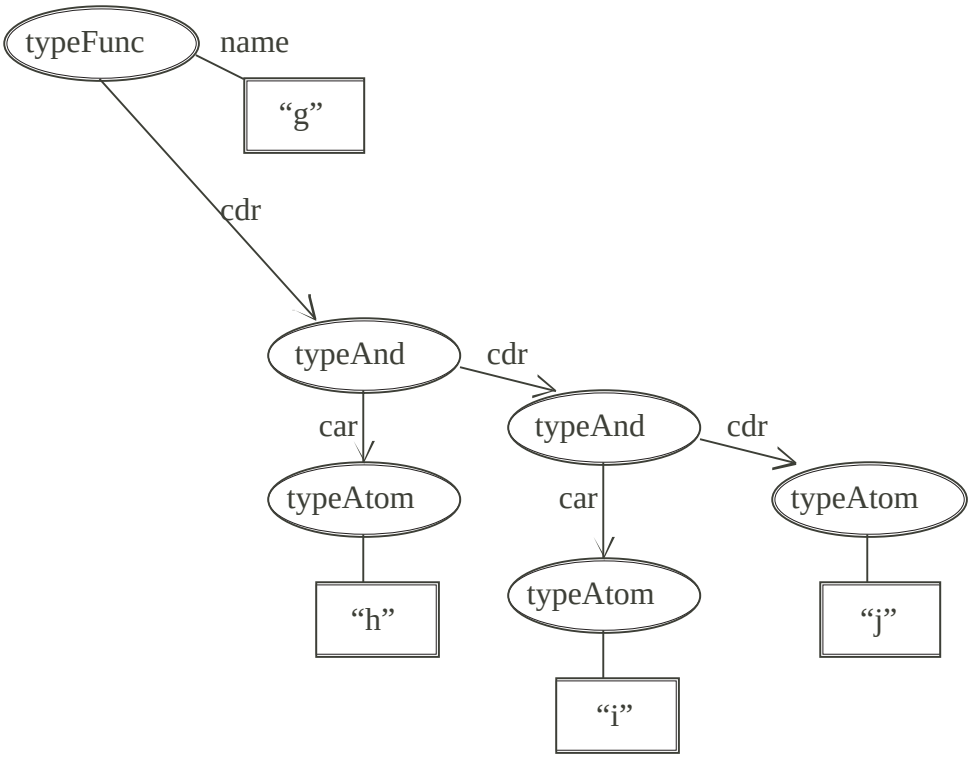


図 6.3: 3 引数関数子の図

への参照を持ち，その末尾の typeList 形式のオブジェクトの cdr メンバーは null となる．

PrologObject 型のデータを Java 言語上で扱いやすくするために，ここから String などのデータを取りだしてやる必要がある．MiLog エージェントから答えとして返される PrologObject 型のデータには，もし実行に失敗した場合には null が，それ以外の場合には，query に含まれていた変数（およびその束縛内容）のリストが入る．例えば，query(”append([o],[k],X).”); と query を実行した場合，PrologObject 形式での答えは，”[(X=[o,k])]” のようなイメージになる．実際には，先頭は typeList 形式のオブジェクトで，その car メンバーに変数”X”を表現する PrologObject への参照が入っている．そのオブジェクトの cdr メンバーに，束縛内容であるリスト”[o,k]” を表現するオブジェクトへの参照が入っている．ここで注意すべき点は，変数の束縛内容に変数が含まれる場合があるという点である．たとえば，(X = (Y = (Z = [o, (K = k)]))) というイメージで値が格納されている場合がある．変数から値を取り出すには，この点を考慮して，変数の束縛内容が変数でないことを仮定しないようにプログラムを書く必要がある．変数の束縛内容をたどるのに便利なように，PrologObject クラスに binding() メソッドが用意してある．この binding() メソッドでは，変数の束縛内容が変数でなくなるまで（あるいは，未束縛の変数へ行き着くまで）繰り返し cdr メンバーをたどって，その値を返す．例えば，上の例では，オブジェクト x が変数’X’ を表現する場合に，x.binding() の値は，変数’Z’ を表現するオブジェクトへの参照となる．ここでさらに注意が必要なのは，変数’K’ が，いまだに変数’Z’ の束縛内容として含まれている点である．

PrologObject 型で表現されたデータを，Java 上で扱いやすい構造に変換するために，ユーティリティメソッドを用意してあるので，適宜利用されたい．このメソッドは，Weblog クラスのクラスメソッドとして定義されている．

指定したの名前の変数の束縛内容を，Java オブジェクトに変換したものを得る．Object getAnswerAsObjects(String varName)

このメソッドでは，次の対応関係で Java オブジェクトへ変換される．

アトム/数値 → *String*

関数子 → 関数名をクラス名として，引数をコンストラクタの

パラメータとしてとる *Java* クラスのオブジェクト,
あるいは該当するクラスがない場合は *String*
リスト → *Vector*
変数 → *String*

例えば， MiLog 上での [hello,world] は，Vector 型のインスタンスに ‘hello’ と ‘world’ の２つの String 型データが格納されたものへ変換される．

6.1.4 エージェントの削除

MiLog エージェントは，独自のネームサーバ（名前管理機構）を持つので，局所変数に代入されたエージェントのインスタンスは，そのままではガベジコレクタによって消去されない．よって，エージェントを，Java プログラム中で明示的に消去する必要がある．エージェントを消去するためには，そのエージェントに ”?- delete.” という質問を評価させる．例えば，次のようにする．

```
myAgent.query("delete.");
```

これにより myAgent を消去することができる．

6.1.5 モバイルエージェントの使用方法

モバイルエージェント機能を有効にするためには，モバイルエージェントを制御するためのサーバを起動する必要がある．サーバの起動といっても，実際にはほんの数行の Java プログラムを記述するだけでよい．例えば，次のように書く．

```
String[] args = { "-server", "17008", "-agents" };  // no agent is generated.
weblog.Boot.main(args);
while( weblog.Weblog.getTarget("_boot") != null ) {
    Thread.yield();
}
```

args は，エージェントサーバに渡すオプションで，MiLog 開発環境 weblog.Boot クラスを起動するときに指定するのと同じオプションが指定できる．上の例では，サーバのポート番号を指定するためのオプション ”-server” と，”init” エージェントを自動的に生成させないようにするためのオプション ”-agents” を指定している．エージェントのクローンの生成するためは，そのエージェントに対して述語 clone1/1 あるいは clone2/1 を実行させればよい．また，new/1 および new2/2 を用いてエージェントを作成しても良い．生成したクローンは，述語 move/1 を使って，他の計算機上へ移動できる．（このときに，あらかじめ目的の計算機上で MiLog が起動している必要がある．これは，通常の MiLog の使用と同様である．）

```
myAgent.query("clone1(CLONE), request(CLONE, move('127.0.0.1:17007').");
```

6.1.6 MiLog エージェントを組み込んだ Java プログラムのコンパイルおよび実行方法

MiLog エージェントを組み込んだ Java プログラムのコンパイルおよび実行時には，クラスパスに weblog2.jar を含める必要がある．例えば，次のようにする．（この例では，bash を利用した場合を示す．$ はプロンプトである．）

```
$ export CLASSPATH=./weblog2.jar:$CLASSPATH
$ javac  sample/Sample.java
$ java  sample.Sample
```

6.1.7 サンプルプログラム

このサンプルプログラムでは，エージェントが localhost 上の別の MiLog 実行環境へ移動するので，あらかじめ，localhost に MiLog を（ポート 17007 で）起動しておくこと．

```
package sample;

import java.util.*;
import weblog.*;

public class Sample {

  weblog.Weblog milogAgent;

  public Sample() {
    milogAgent = new Weblog("sample");
    milogAgent.reconsult("test :- write('hello world'). ");
  }

  // no return value
  public void test() {
    milogAgent.syncQuery("test.");
  }

  public String simpleReturnValue(String URL) {
    String quotedURL;
    PrologObject atom = new PrologObject(PrologObject.typeAtom);
    atom.name = URL;
    quotedURL = atom.toString();
    PrologObject answer =
        milogAgent.syncQuery("getHTML("+ quotedURL + ",X).");
    return(answer.car.binding().name);
  }

  public String simpleReturnValue2() {
    // another method
    PrologObject answerObj;
    answerObj = milogAgent.syncQuery("concat([a,b],X).");
    if( answerObj != null ) {
      String answer = (String)Weblog.convertPrologObjectToJavaObject(
                        Weblog.getValueFromAnswer(answerObj,"X"));
      return(answer);
    }
    return(null);
  }

  public Vector complexReturnValue() {
    if( milogAgent.syncQuery("append([abc],[def,ghi],[X|Y]).") != null ) {
```

```
    String answer1 = (String) milogAgent.getAnswerAsObjects("X");
    Vector answer2 = (Vector) milogAgent.getAnswerAsObjects("Y");
    return(answer2);
  }
  return(null);
}

public void triggerOnly() {
  milogAgent.query("test.");
}

public void bootServer() {
  String[] args = { "-agents" };  // no agents are generated.
  Boot.main(args); // please call this only once at the execution.
  while( Weblog.getTarget("_boot") != null ) {
    Thread.yield();
  }
}

public void mobileSample() {
  String additionalProgram = "hitAndAway(URL) :- move(URL),query(init,write(hehe)),goHome. ";

  milogAgent.reconsult(additionalProgram);
  milogAgent.syncQuery("clone1(A),assert(c(A)),query(A,hitAndAway('127.0.0.1:17007')).");
  milogAgent.syncQuery("retract(c(A)),query(A,delete).");
}

public String communicationSample() {
  Weblog agent1 = new Weblog("agent1");
  Weblog agent2 = new Weblog("agent2");

  agent1.syncQuery("query(agent2,assert(manjyu(tabetai))).");
  PrologObject answer = agent2.syncQuery("manjyu(TABETAI).");
  agent1.syncQuery("delete.");
  agent2.syncQuery("delete.");
  return( (String) Weblog.convertPrologObjectToJavaObject(
            Weblog.getValueFromAnswer(answer,"TABETAI") ) );
}

public void debugSample() {
  Weblog bug = new Weblog("debug");
  bug.reconsult(" myapp([],X,X). myapp([X|L],Y,[X|Z]) :- myapp(L,Y,Z). ");

  weblogConsole agentWindow = new TabbedConsole("debug");  // arg 1 is the name of the agent.
  bug.setWeblogConsole(agentWindow);
  bug.syncQuery("trace.");
  bug.syncQuery("myapp([a,b,c],[d,e,f],X).");
```

```
    bug.syncQuery("sleep(10000).");
    bug.syncQuery("delete.");
  }

  public static void main(String[] args) {
    Sample sample = new Sample();
    String html = sample.simpleReturnValue( "http://weblog.ics.nitech.ac.jp/" );
    String ab = sample.simpleReturnValue2();
    Vector defghi = sample.complexReturnValue();
    String tabetai = sample.communicationSample();
    System.out.println(html + "\n" + ab + "\n" + defghi + "\n" + tabetai );
    sample.bootServer();
    sample.mobileSample();
    sample.debugSample();
    sample.milogAgent.syncQuery("delete.");
    System.exit(0);
  }
}
```

6.2 Java 言語を用いた MiLog エージェントインタプリタの拡張

Java 言語を用いて，独自の組み込み述語を持った MiLog エージェントを作成することができる．作成した独自の MiLog エージェントは，述語 new2/2 を用いることで，MiLog エージェントプラットフォーム上で使用することができる．また，ある条件を満たしていれば，その拡張したエージェントを他のエージェントプラットフォームへ移動させることも可能である．この文書では,（1）独自の MiLog エージェントの作成方法と,（2）そのエージェントの MiLog エージェントプラットフォーム上での使用方法について述べる．独自の MiLog エージェントにモビリティを持たせるための条件については，この文書の付録に示す.（この文書中において，'weblog' というキーワードがいくつか出現するが，これは MiLog の初期の開発コードネームを意味する.）

（1）独自の MiLog エージェントの作成方法独自の MiLog エージェントの作成を行うには，MiLog エージェントのコアクラスである weblog.Weblog クラスを継承して，新たな Java クラスを設計する．この Java クラス上において，"sys_" という名前で始まる public メソッドを定義することで，組み込み述語を追加することができる．weblog.Weblog クラスには，Java のリフレクション機能を利用して，自動的にメソッドの定義を検索し，MiLog エージェントインタプリタ内部からそのメソッドを呼び出すことができるようにする機能が備わっている．ここで述べる方法で拡張できる組み込み述語は，バックトラックを行なわない種類の組み込み述語である．組み込み述語の実装には，ネットワークアクセスや GUI などの，Java 言語上の機能を利用することができる．JNI(Java Native Interface) 等を用いれば，C 言語等，Java 以外の言語を組み込み述語の記述に利用することも可能である．JNI 等の使用方法については，関連書籍を参照されたい．MiLog エージェントを拡張する Java プログラムのひな形は，次のようになる．

```
import weblog.*;
public class myClass extends Weblog {
   public myClass(String name) {
     super(name);
     stringProperty = name + " is myClass";
   }
   public myClass(String name, int cm, boolean dv) {
     super(name,cm,dv);
```

```
    stringProperty = name + " is myClass";
  }
  String stringProperty;
  public PrologObject sys_sampleBuiltin( PrologObject  args ) {
    if( arity(args) == 1 && (
      args.binding().typeVar() || args.binding().typeAtom())  ) {
      PrologObject p = new PrologObject( PrologObject.typeAtom );
      if( match( p, args ) ) {
        return(success);
      }
  }
  return(null);
}
```

クラスのコンストラクタには，String 型の引数を１つとるものと，String 型に int 型と boolean 型をあわせて計３つの引数を取るものの２種類を用意しておく．いずれのコンストラクタ中でも，必ず super(name); のように引数の数に対応した親クラスのコンストラクタを必ず最初に呼ぶ必要がある．親クラスのコンストラクタにより，MiLog エージェントインタプリタ内部の初期化が行われるようになっている．

クラスは，任意のオブジェクトをメンバとして持つことができる．上の例では，String 型のオブジェクトを１つ，インスタンスのメンバとして持っている．組み込み述語を定義するメソッドは，”sys_” で始まる public メソッドで，引数に PrologObject 型を１つ持ち，返り値として PrologObject 型を返すようにする．このメソッドの名前の先頭から”sys_”を除いたのもが，MiLog エージェント上から参照される組み込み述語の名前となる．メソッドの返り値には，その組み込み述語の実行が成功したか否かを示すフラグを指定する．その組み込み述語の実行が成功した場合には，success を，失敗した場合には null をメソッドの返り値に指定する．ここで，オブジェクト success は，親クラスである weblog.Weblog クラスで定義されている定数である．組み込み述語を定義するメソッドの引数には，その述語の呼び出し時における引数が渡される．例えば，上の例では，MiLog エージェント上で?- sampleBuiltin(a) という形で組み込み述語が呼ばれた場合に，アトム’a’ を表現するオブジェクトが，sys_sampleBuiltin メソッドの引数として渡される．この引数は PrologObject クラスのオブジェクトで表現される．PrologObject クラスによる表現については，MiLog users guide for Java Programmers(1) を参照されたい．例えば，述語の引数が１つで，atom が指定されていた場合には，typeAtom の属性を持つ PrologObject インスタンスがメソッドの引数として渡される．述語がいくつ引数を持つかを判定するためには，int arity(PrologObject args) メソッドを使用する．このメソッドは，親クラスである weblog.Weblog クラスで定義されており，述語の引数の数を int 型の数値で返す．述語の引数が変数を含む場合には，メソッド boolean match(PrologObject arg1, PrologObject arg2); を用いて，その変数に値を束縛させることができる．この match メソッドは，親クラスである weblog.Weblog クラスで定義されている．この match メソッドでは，２つの引数の単一化を行い，単一化が成功すれば true を，失敗すれば false を返す．この match メソッドの引数には，組み込み述語を定義するメソッドに渡された PrologObject 型の引数（あるいはその表現の一部）と，ユーザの手で新たに作成した PrologObject クラスのインスタンスを指定することができる．ただし，ユーザの手で新たに作成した PrologObject クラスのインスタンスでは，変数を含まないようにする．例えば，述語の引数に指定された変数 ’X’ と，ユーザが作成したアトム ’a’ を単一化することはできるが，述語の引数に指定された関数子 ’f(a,X)’ と，ユーザが作成した関数子 ’f(A, A)’ を単一化することはできない．これは，ユーザが作成した関数子に含まれる変数が，MiLog エージェントインタプリタによって初期化されていないためである．この match メソッドは，組み込み述語を定義するメソッド内部で何度でも使用することができる．例えば，２つの引数を持つ組み込み述語において，その２つの引数の両方に値を束縛して返すような場合には，次のようにメソッドを定義する．

```
  public PrologObject sys_twoArgs( PrologObject args ) {
    if( arity(args) == 2 ) {
```

```
      PrologObject arg1 = args.car.binding();
      PrologObject arg2 = args.cdr.binding();
      PrologObject ret1 = new PrologObject(PrologObject.typeAtom);
      ret1.name = "this is ret1";
      PrologObject ret2 = new PrologObject(PrologObject.typeList);
      PrologObject ret2b = new PrologObject(PrologObject.typeAtom);
      ret2b.name = "this is ret2b";
      ret2.car = ret2b;

      if( match( arg1, ret1) && match( arg2,ret2 ) ) {
        return(success);
      }
    }
    return(null);
  }
```

match メソッドによって変数束縛を行ったあと，組み込み述語を定義するメソッドが null を返した場合，その変数束縛は自動的に解除される．例えば上の例で，もし最初の match メソッドが true を返し，その直後の match メソッドが false を返した場合でも，最初の match メソッドにおいて行われた変数束縛は MiLog エージェントインタプリタ内部で自動的に解除される．[1]

（2）MiLog エージェントプラットフォームにおける利用方法 Java 言語を用いて拡張した独自の MiLog エージェントは，MiLog エージェントプラットフォーム上で使用することができる．このためには，MiLog エージェントプラットフォームの起動時に，作成した MiLog エージェントの Java クラスを，クラスパスに追加しておく必要がある．例えば，作成した MiLog エージェントが myClass.class という名前でカレントディレクトリにおいてある場合，MiLog エージェントプラットフォームの起動時に次のようにクラスパスを指定する．(この例では，UNIX 上で bash を用いた場合を示している．行頭の'$' はコマンドプロンプトを意味する．)

```
$ export CLASSPATH=./:./weblog2.jar:$CLASSPATH
$ java weblog.Boot &
```

作成した MiLog エージェントは，組み込み述語 new2/2 を用いることで，MiLog エージェントプラットフォーム上にエージェントとして生成することができる．例えば，myClass クラスのエージェントを 'myagent' という名前で生成したい場合には，次の query を実行する．

```
?-  new2(  myagent,  myClass  ).
```

この query 実行後に，myagent という名前のコンソールウィンドウが現れる．このコンソールウィンドウを用いて，他の MiLog エージェントと同様に操作を行うことができる．

6.2.1 独自に拡張した MiLog エージェントにモビリティを持たせる方法

Java 言語を用いて独自に拡張した MiLog エージェントであっても，ある条件を満たせばモビリティを持たせることができる．その条件とは，次の 2 つである．(a) エージェントが Java のシリアライズ機能を用いてシリアライズ可能であること．また，そのときにエージェントのメンバ変数として，特定の計算機環境あるいは Java VM に依存したデータや資源を持たないこと．(b) 事前に移動先の MiLog エージェントプラットフォーム上に，拡張した MiLog エージェントのクラスファイルを置いて，クラスパスを通しておくこと．上の（a）に

[1] 値を返すような組み込み述語を設計する場合には必ず match メソッドを用いて変数束縛を行うようにし，決して引数の内容そのものを改変してはならない．このような改変操作は，MiLog エージェントインタプリタ内部でのバックトラック動作を破壊してしまい，プログラムの挙動が不正確になる．