Prolog

入門と演習

中村勝則，平塚聡

2020 年 4 月 14 日

目 次

1　序：Prologとは

Prologは人工知能（AI）の分野でしばしば利用されるプログラム言語である．C言語やFORTRAN言語といった手続き型言語が記述された処理の手順を実行するのに対して，Prologでは与えられた事実や推論規則を元にして質問に答える．

例えば,「太郎は人である」という事実と「Aさんが人であるならば，Aさんはいずれ死ぬ」という規則があると,「太郎はいずれ死にますか？」という質問に答えることができる．

事実と規則に基づいた推論は**数理論理学**の分野で扱われる演算であり，数理論理学上のアルゴリズムを実装することを**論理プログラミング**という．Prologは論理プログラミングのための言語処理系である．

現在利用できるProlog処理系の多くは，エディンバラ大学で開発されたいわゆる「DEC-10 Prolog」[1]に倣っており，ISOによる標準化（文献[7, 8]）もされている．本書で取り上げるPrologもDEC-10 Prologを前提とする．

1.1　述語論理に基づく推論

Prolog以外の言語でも，連言（and），選言（or），否定（not）といった論理演算（ブール演算）によって真理値の算出ができるが，ブール演算は**命題論理**に基づくものであり，主として真理値（ture/false）の演算を行うものである．

これに対してPrologで扱うのは**述語論理**であり,「AはBである」という文のように，ある対象の状態や性質などを記述する**述語**を扱う．すなわち，Prolog以外の言語処理系では基本的に命題は真理値（true/falseの2値のどちらか）として扱われるのに対して，Prologで扱う述語論理の文は真理値を持つだけでなく，**対象**（複数の場合もある）と**述語**を保持する．

次のような例について考える．

命題論理の例. 命題記号 A, B, C に対して

$A := true$,
$B := false$,
$C := A\ and\ B$
と真理値を割り当てると，C には $false$ が割当てられる．

述語論理の例.

事実：「太郎は人である」
規則：「Aさんが人であるならば，Aさんはいずれ死ぬ」
　から
帰結：「太郎はいずれ死ぬ」　が真なる文として結論（推論）される．

上の命題論理の例はProlog以外の言語でも簡単に表現できるが述語論理の扱いはできない．次に，ここで挙げた述語論理の例をPrologで表現することを考える．

■ Prologによる推論の例

述語としての文をProlog言語で記述するための構文は次のようなものである．

述語（対象 1, **対象** 2,$\cdots$, **対象** n）

これは「”対象1”,”対象2”,$\cdots$,”対象n” は”述語”である」と解釈される文である．例えば

```
human(taro)
```

とProlog言語で記述すると,「tarooはhumanである」という文を記述したことになり，この文が1つの**事実**を表すことになる．

[1]エディンバラ大学がPrologを開発した当時に使用していた汎用計算機「DEC system 10」に由来する．

規則（推論規則）を Prolog 言語で記述するための構文は次のようなものである．

帰結 :- **条件**

これは「”条件”を満たす場合，”帰結”は正しい」という規則を意味する．例えば

```
mortal(A) :- human(A)
```

と記述すると,「A が human である場合，A は mortal である」という推論規則となる．すなわち，$P \Rightarrow Q$（P ならば Q）という推論規則は，$Q : -P$ という記述になる．

この規則の例にはアルファベット大文字の A が用いられているが，これは特定の対象に限定されない規則を記述する場合などに用いられる**変数**である．Prolog の変数に関しては後で詳しく解説する．

これらの事実と推論規則を Prolog のプログラムとして記述すると次のようになる．

```
human(taro).
mortal(A) :- human(A).
```

実際にはこのプログラムをテキストファイルとして作成し，それを Prolog 処理系に読み込むことで事実と規則を処理系に登録する．また，各文の終端にはピリオド ’.’ を置く．

事実や推論規則を登録した後は，処理系はユーザからの質問に答えることができる．例えば処理系に次のように入力してみる．

```
?- human(taro).
```
Enter

これは，処理系に対して「human(taro) は真ですか？」という質問をしたことになる．’?-’ で始まる文は**質問**である．これに対して処理系は

```
true.
```
（あるいは `yes.`）

と表示し，問われた文が真であると回答する．この例より，処理系が事実と照らし合わせて質問に答えていることがわかる．次に

```
?- mortal(taro).
```
Enter

と入力してみる．これは，処理系に対して「mortal(taro) は真ですか？」という質問をしたことになる．これに対して処理系は

```
true.
```
（あるいは `yes.`）

と表示する．この例より，処理系が推論規則に沿って質問に答えていることがわかる．

1.2 Prolog の動作

1.2.1 質問による処理の駆動

Prolog 処理系の基本的な動作は「質問に答えること」である．オペレーティングシステム上で起動された Prolog 処理系は多くの場合 ’?-’ というプロンプトを表示してユーザからの質問の入力を促す．

ソースプログラムの読込み，処理系の終了といった基本的な作業も，それらを実行するための「問い合わせ」を処理系に発行することで行う．

例えば，先に例示した述語論理のプログラム（ソースプログラム）がテキストファイルとしてオペレーティングシステム上に ’test01.pl’ というパス名で存在している場合，処理系に対して

```
?- consult('test01.pl').
```

という問い合わせを発行することで実際の読込み処理を行う．この consult という述語がソースプログラムを読込む機能を持っている．実行結果として

`true.`（あるいは `yes.`）

と表示し，読み込んだ事実と推論規則に沿って質問に答えることが可能となる．

ソースプログラムの読込み以外の入出力に関しても，そのための機能を持つ各種の述語が予め処理系に用意されており，それら述語を用いた節を処理系に対して「問い合わせる」ことで処理を実行する．

1.2.2 ここまでのまとめ：操作の例

先に説明した述語論理の推論処理を Prolog 処理系の上で実際に行う流れを示す．

(1) 事実と推論規則の定義（プログラム）をテキストファイルに用意する

テキストエディタを用いて次のような Prolog プログラムを作成する．

プログラム：test01.pl

```
1 human(taro).
2 mortal(A) :- human(A).
```

これは 'test01.pl' というファイル名でプログラムを作成した例である．

(2) プログラムを Prolog 処理系に読み込む

Prolog 処理系が起動した状態で consult 述語を使って 'test01.pl' を読み込む．

操作：

```
?- consult('test01.pl').  Enter    ← ユーザによる入力
true.        ← 処理系からの出力
?-           ← 次の質問を受け付けるプロンプトが表示される
```

(3) 処理系に対して質問する

操作：

```
?- human(taro).  Enter    ← ユーザによる入力
true.        ← 処理系からの出力
?- mortal(taro).  Enter   ← ユーザによる入力
true.        ← 処理系からの出力
?-           ← 次の質問を受け付けるプロンプトが表示される
```

1.2.2.1 定義を直接登録する方法（ファイルからではなくコンソールからの登録）

事実や推論規則をファイルからではなく，直接にキーボード（コンソール）から入力して登録することができる．Prolog のユーザインタラクションは **user モジュール** という**モジュール**[2] であるとみなされる．すなわち 'user' から定義を読み込むという形で，コンソールから定義を読み込むことができる．（次の例参照）

[2]「4.1 モジュール」で解説しているのでそちらを参照のこと．

例. コンソールからの定義の読込み（SWI-Prolog での例）

```
?- [user].  Enter          ← user モジュールからの読込み開始
|:  human( taro ).  Enter          ←定義の入力
|:  mortal( A ) :- human( A ).  Enter     ←定義の入力
|:  Ctrl+D     ← user モジュールからの入力を終了
%  user://1 compiled 0.00 sec, 2 clauses     ←コンパイルメッセージ
true.     ←評価結果
?-        ←問い合わせのプロンプト（トップレベル）に戻った
```

このように，Ctrl+D を押下することで問い合わせのプロンプトに戻る．（場合によっては Ctrl+C などの操作を組み合わせることが求められる）

1.2.3 単一化（unification）

先に挙げた例は文の真偽を検証するものであり，Prolog 処理系の動作を示す最も素朴な例である．Prolog 処理系には真偽の検証以外にも**単一化**（unification）の機能がある．先に挙げた例のプログラムが処理系に読み込まれた状態で次のように質問してみる．

```
?- human(A).
```

これは「A が human だとしたら A は何？」という質問であり，これに対して処理系は

```
A = taro.
```

と答える．次に

```
?- mortal(A).
```

と質問すると，今度は推論規則に沿って

```
A = taro.
```

と答える．

■ Prolog における変数

先の例の `?- human(A)` という質問において，アルファベット大文字の A は Prolog における**変数**であり，質問の文において変数を記述すると，処理系に定義された事実や推論規則に照らし合わせて，その文が真となるように変数に適切な値が自動的に割当てられる．これが Prolog 処理系に備わった単一化の機能である．

Prolog 処理系では，アルファベット大文字で始まる語は変数として扱われる．

「単一化」というのは論理プログラミングにおける用語であり,「与えられた 2 つの論理式が同一である」と見做すべく変数への割当てを行うことを意味する．先の例では，質問された文が与えられた事実や推論規則に照らし合わせて真となるように単一化が行われたことを示すものである．

■ 複数の値が変数に単一化する例

先のサンプルプログラム 'test01.pl' を次のように改変する．

改変したプログラム：test01.pl

```
1  human(taro).
2  human(jiro).
3  human(hanako).
4  human(yosiko).
5
6  mortal(A) :- human(A).
```

これは taro 以外にも，jiro,hanako,yosiko も human である旨を追加したものである．このプログラムを Prolog 処理系に読み込んで[3]，先と同様に単一化を試みる．

操作：

```
?- human(A).  Enter     ← ユーザによる入力
A = taro      ← 処理系からの出力（待機状態）
```

今度は，次の質問を受け付けるプロンプト '?-' が出ずに待機状態となる．この状態でセミコロン ';' をタイプすると，変数 A に当てはまる次の候補が表示される．(以下同様)

操作（つづき）：

```
?- human(A).  Enter     ← ユーザによる入力
A = taro ;      ← ユーザがセミコロンをタイプ
A = jiro ;      ← ユーザがセミコロンをタイプ
A = hanako ;    ← ユーザがセミコロンをタイプ
A = yosiko.      ← ここで候補が終了
?-               ← 次の質問を受け付けるプロンプトが表示される
```

課題. この例の状態で，?- mortal(A) と質問するとどうなるか確認せよ．

■ イコール記号 '=' による単一化

「左辺 = 右辺」は両辺を単一化するための文であり，この形の文を質問することができる．例えば，

```
?- a = a. Enter
```

と質問すると，両辺は等しいので単一化が成功して，

```
true.
```

と表示される．また，

```
?- X = a. Enter
```

と質問すると，両辺を同一のものとするために変数 X に　a が割当てられる．結果として，

```
X = a.
```

と表示される．

例題. p(X,f(X,b)) = p(a,Y) の単一化（文献 [3]）

```
?- p(X,f(X,b)) = p(a,Y). Enter
```

と質問すると，両辺を同一のものと見做すために次のような結果となる．

```
X = a,
Y = f(a, b).
```

この例題にある p，f は定義された述語ではなく，f(X,b)，p(X,f(X,b))，p(a,Y) も単にデータとして扱われる．すなわち，質問の対象となっている述語はイコール '=' であり，p(X,f(X,b)) と p(a,Y) は '=' 述語に対する引数で

[3]改変したプログラムを Prolog 処理系に再度読み込んで，以前に処理系に読み込んだ定義を置き換える場合は reconsult 述語を使用する．処理系によっては consult 述語で再度読込みとなる場合もある．詳しくは使用する Prolog 処理系の仕様を参照すること．

ある．イコールの両辺に記述された式は真理値を問われる文ではなく，単なるデータである．

課題． 実際の処理系で `p(f(P,Q),g(f(a,Z))) = p(X,g(X))` の単一化を試みて結果を確認せよ．また，何故そのような結果となるのか考えよ（文献 [3]）

参考）
両辺が単一化しないことを検証する '\=' もある．

1.2.4 Prolog 処理系の終了

Prolog 処理系を終了するには．次のように halt 述語を引数なしで質問する．

```
?- halt.
```

これにより Prolog 処理系は終了し，オペレーティングシステムのプロンプトが表示される．

1.3 サンプルプログラム

先に示した例を拡張したプログラム [4] について考える．

拡張の概要：

- 人間（human）以外にも pochi,john,nana,mimi という犬（dog）が存在する事実を追加する
- 人間も犬も性別があり，雄（male）もしくは，雌（female）である事実を追加する
- 雄と雌は結婚することができる（marriageable）という事実を追加する
- 犬もまた有限の寿命がある存在である（mortal）という事実を追加する

拡張したプログラム： test01.pl

```
1  human(taro).
2  human(jiro).
3  human(hanako).
4  human(yosiko).
5
6  dog(pochi).
7  dog(john).
8  dog(nana).
9  dog(mimi).
10
11 male(taro).
12 male(jiro).
13 male(pochi).
14 male(john).
15
16 female(hanako).
17 female(yosiko).
18 female(nana).
19 female(mimi).
20
21 marriageable(A,B) :- male(A),female(B).
22 marriageable(A,B) :- male(B),female(A).
23
24 mortal(A) :- human(A).
25 mortal(A) :- dog(A).
```

[4] ここで取り上げるサンプルプログラムでは「性別」や「結婚」といった概念が想定されている．このサンプルプログラムを作成するに当たって，筆者らは性的少数派の人たちに対する一切の差別意識を持っておらず，あくまで架空の世界設定の例としてこれら概念を用いていることを明言する．また同時に，異なる考え方を持つ全ての人々に対して筆者らは敬意を持っていることも明言する．

解説：

21 行目に推論規則として

```
marriageable(A,B) :- male(A),female(B).
```

とある．このように ':-' の右辺にコンマ ',' で連結された複数の項がある場合，それらは**連言**である．すなわち，

「$P :- Q, R$」　は　「Q, R が共に真ならば P は真である」

という推論規則を意味する．

課題．このプログラムを Prolog 処理系に読み込んで，考えられる限りの質問をせよ．

例 1. `?- marriageable(taro,hanako).`

例 2. `?- marriageable(jiro,A).`

例 3. `?- mortal(X).`

課題．このプログラムでは，人間の `taro` と犬の `mimi` が結婚できることになる．何故そのようになるのか理由を考えよ．また，人間は人間同士，犬は犬同士でしか結婚できないようにプログラムを改善せよ．

1.4　Prolog による論理プログラミング

冒頭でも述べたとおり，数理論理学上のアルゴリズムを実装することを「論理プログラミング」といい，Prolog は論理プログラミングを実現するための言語処理系である．ここでは，Prolog の基礎を与える論理プログラミングの要素について説明する．(Prolog の基礎となる論理プログラミングについてさらに詳細に学びたい場合は文献 [1] を参照のこと)

Prolog は**一階述語論理**という体系の上に構築された言語処理系であるが，一階述語論理の記述が全て Prolog 上で実現できるわけではない．

一階述語論理の式で記述した推論の手順を効率よく計算機上で実行するには様々な工夫が必要であり，Prolog 上でも扱える論理式や推論の手順が独特なものになっている．

1.4.1　ホーン節

推論を実現するための規則

$P \Rightarrow Q$　　(P ならば Q)

は論理和の形　$\neg P \vee Q$　として書くことができる．論理和の式は**節**（clause）であり，条件部の論理式 P が積の形であっても，推論規則の式は全体として節となる．(次式参照)

$$\begin{aligned} & \neg P_1 \vee \neg P_2 \vee \cdots \neg P_n \vee Q \\ = \quad & \neg(P_1 \wedge P_2 \wedge \cdots P_n) \vee Q \\ = \quad & P_1 \wedge P_2 \wedge \cdots P_n \Rightarrow Q \end{aligned}$$

この例より，「P_1 かつ P_2 かつ $\cdots P_n$ ならば Q」という推論規則が節になっていることがわかる．

このように 1 つの帰結を持つ節（あるいは帰結を持たない節）を**ホーン節**といい，Prolog で規則を記述するための基本形となっている．

より一般的には推論規則は，

$Q_1 \vee Q_2 \vee \cdots \vee Q_n \ :- \ P_1 \wedge P_2 \wedge \cdots \wedge P_m.$

すなわち，$P_1 \wedge P_2 \wedge \cdots \wedge P_m \Rightarrow Q_1 \vee Q_2 \vee \cdots \vee Q_n$ という形式であるが，Prolog では帰結部が選言で連結されていない（あるいは帰結部が無い）ものに限定されたホーン節を扱うことで，推論の処理を大幅に効率化している．

Prolog は，ホーン節の集合として記述したプログラムを，**線形導出**（linear resolution）と呼ばれる手法の内，特に**SLD 導出**（Selective Linear resolution for Definite clause）と呼ばれる手法で推論を実行する．

また，ホーン節による事実と推論規則の記述は**知識**の表現と見ることもでき，その意味で，Prolog 上に構築されたシステムは**知識ベースシステム**であると言うことができる．

2 Prologの文法と動作

2.1 事実と規則

Prolog のプログラムは**事実**と**規則**の集合である．

■ 事実

事実はある対象（あるいは対象の集合）についての状態や性質を記述したものであり，Prolog では次のように記述する．

文法：　述語（引数 1, 引数 2, … , 引数 n）

この形の式を**原子論理式**（atomic formula）と呼ぶ．**述語**には**アトム**を使用する．アトムとは，構造をもたない変数でないシンボルのことで，詳しくは後で説明する．

記述した事実は読み込んだ順序で処理系のデータベース[5] に登録される．処理系はデータベースに登録されている順序で事実や規則を照会し，ユーザからの質問に答える．

■ 規則（推論規則）

規則は**条件**と**帰結**からなるもので，より一般的には

条件 ⇒ **帰結**

と記述するが，Prolog では ':-' を用いて

帰結　:-　**条件**[6]

と記述する．帰結の部分に記述できるのは原子論理式である．条件の部分には**連言**や**選言**を用いた**論理式**が記述できる．

連言は複数の論理式をコンマ ',' で連結した形で記述する．選言は複数の論理式をセミコロン ';' で連結した形で記述する．

連言の例：　`f(a,b), g(c,d), h(e)`
選言の例：　`f(a,b); g(c,d); h(e)`

論理式の最小単位は原子論理式であるが，連言と選言を用いて入れ子の構造にすることができる．

複雑な論理式の例：　`(f(a,b);g(c,d)), p(w,x)`

2.1.1 否定（not 述語）と閉世界仮説

not 述語は否定である．すなわち，`not(P)` という節は `P` が真でないとき真となり，`P` が真のとき偽となる．

Prolog は**閉世界仮説**を前提として真偽の判定を行う．これは，事実や規則として登録された内容から推論によって真となる節のみが真であるということを意味している．Prolog は登録されていない事実は偽として扱う．[7]

---

[5] Prolog 処理系は内部に独自のデータベース機構を備えており，そこに事実や規則を格納する．
[6] **帰結**の部分は**頭部**（head）と呼ぶことがある．また，**条件**の部分は**目標**もしくは**ゴール**（goal）と呼ぶことがある．
[7] 処理系によっては，登録されれていない事実を偽とするための条件を設定できるものもある．詳しくは処理系の仕様を参照すること．

Prolog の not 述語に関して重要なことに，否定の否定（**二重否定**）の対象となる式は単一化されないという点が挙げられる．例えば，

```
?- A = 2.
```

と質問すると，

```
A = 2.
```

と単一化され，質問した文は真となるが，

```
?- not(not(A = 2)).
```

と二重否定の質問をすると，

```
true.
```

となるのみで，変数 A への単一化は起こらない．これは，Prolog でアプリケーションシステムを構築する際に十分注意すべき事柄である．

2.1.2 文の終端子

Prolog のプログラムとして事実や規則を記述する場合は，各文の終端にピリオド '.' を記述する．

2.1.3 同一の述語を持つ複数の節の定義

同一の述語を持つ複数の事実を定義したり，頭部の述語が同一である複数の規則を定義すると，それらは選言とみなされる．すなわち，

```
human(taro).
human(jiro).
human(hanako).
human(yosiko).
```

と事実を記述すると，Prolog のデータベースには

human(taro) ∨ human(jiro) ∨ human(hanako) ∨ human(yosiko)

という論理式として登録されることになる．規則の登録に関しても同様のことが言える．

2.1.4 Prolog の内部動作

事実と規則に沿って，Prolog がどのように推論するかについて，サンプルプログラムを挙げて考える．

サンプルプログラムの概要：

次のような事実と規則がある．

（事実） taro,hanako という human が存在する．
（事実） pochi,nana という dog が存在するが，nana は既に死んでいる．(deceased である)
（事実） ruru,tama という cat が存在する．

（規則） human,dog,cat は全て animal である．
（規則） animal の内，死んでいない（deceased でない）ものは寿命がある．(mortal である)

これをプログラムにしたものが 'test02.pl' である．

プログラム：test02.pl

```
1  human(taro).
2  human(hanako).
3
4  dog(pochi).
5  dog(nana).
6
7  deceased(nana).
8
9  cat(ruru).
10 cat(tama).
11
12 animal(A) :- human(A).
13 animal(A) :- dog(A).
14 animal(A) :- cat(A).
15
16 mortal(A) :- animal(A), not(deceased(A)).
```

動作の流れ

このプログラムを読み込んだ Prolog 処理系に対して

```
?- mortal(ruru).
```
Enter

と質問すると　true.　と答える．このときのシステムの動作を追うと次のようになる．

(1)　mortal(ruru) を検証する．そのために，
- (1)-1　animal(ruru) を検証する．そのために，
  - (1)-1-1　human(ruru) を検証したが，偽となるので次に，
  - (1)-1-2　dog(ruru) を検証したが，これも偽となるので次に，
  - (1)-1-3　cat(ruru) を検証するとこれは真となる．従って，
- (1)-2　animal(ruru) は真となる．次に，
- (1)-3　not(deceased(ruru)) を検証する．[8] そのために，
  - (1)-3-1　deceased(ruru) を検証したが偽となる．従って，
- (1)-4　not(deceased(ruru)) は真となる．従って，

(2)　mortal(ruru) は真である．(終了)

解説

この実行例からわかることは，節の条件部は左から順番に検証されるということである．すなわち，animal(A) の真偽が判定できた後で not(deceased(A)) の判定に移る．また animal(A) の真偽を判定するために，その定義に従って検証を行うという**深さ優先**の検証（探索）を行う．

上記の解説の (1)-1-1〜(1)-1-3 では，animal(ruru) が真になり得るかどうかを検証するために，animal(A) の定義を順番に検証して試行錯誤しているのがわかる．このように,「失敗→次の候補の検証」を繰り返すことを**バックトラック**（バックトラッキング）という．

2.1.5　true / fail

true は常に真となる述語（恒真式）で，引数はない．fail は常に偽となる述語で，引数はない．

特に fail は，推論処理の制御において重要な役割を果たす．それに関しては「2.7.5 繰り返し動作の実現」で詳しく解説する．

[8] not 述語に関しては 2.1.1 を参照のこと．

2.2 数値計算

数値の計算を行う述語に 'is' がある．例えば 1+2 の値を計算するには次のように質問する．

```
?- A is 1+2.
```
Enter

この結果

```
A = 3.
```

と表示される．is 述語は原子論理式の形式では「is(変数, 計算式)」であり,「' 変数' の値は' 計算式' である」と解釈される．この形式で質問しても多くの処理系において正しく機能する．

is 述語は処理系の**組込み述語**[9] であり，第 2 引数に変数を与えることはできない．例えば

```
?- 3 is 1+2.
```
Enter

と質問すると

```
true.
```

と答えるが,

```
?- 3 is A+2.
```

と質問すると結果はエラーとなる．すなわち，第 2 引数の評価値を第 1 引数に割り当てるというのが is 述語の仕様である．

is 述語の第 2 引数に使用できる演算子は，基本的には多くの言語処理系と共通である．Prolog 処理系で使用できる代表的な演算子を表 1 に示す．

表 1: Prolog で使用できる算術演算子

演算子	機　能	演算子	機　能	演算子	機　能
+	和	-	差	mod	剰余
*	積	/	商		

Prolog 処理系は基本的には浮動小数点数の数値演算も可能である．例えば

```
?- A is 1/3.
```
Enter

と質問すると

```
A = 0.3333333333333333.
```

と答える．扱える整数値の桁数の上限や，浮動小数点演算の精度に関しては各処理系の定めるところであり，詳しくは処理系の仕様を参照のこと．

is 述語の第 2 引数に各種の数学関数（三角関数，指数関数，対数関数など）が使用できる処理系もあるが，それらに関しても詳しくは処理系の仕様を参照のこと．

2.2.1 数値の型の検査

数値であるかどうかを検査する述語に integer, float, number がある．これら述語は引数を 1 つ与えることができる．integer は引数に与えたものが**整数**の場合に真，それ以外の場合は偽となる．float は引数に与えたものが**浮動小数点数**の場合に真，それ以外の場合は偽となる．number は引数に与えたものが整数もしくは浮動小数点数の場合に真，それ以外の場合は偽となる．

[9]処理系に予め組み込まれている機能のことであり，Prolog を起動した直後に使用できる述語である．

例. 数値の型の検査

```
?- integer(128).   Enter
true.
?- integer(1.2).   Enter
false.
?- integer(a).   Enter
false.
?- float(1.2).   Enter
true.
?- float(2).   Enter
false.
?- number(2.5).   Enter
true.
?- number(256).   Enter
true.
```

2.3 非単調推論のための機能

帰結の導出が単調でない推論のことを**非単調推論**という．Prolog は登録されている知識（事実，推論規則）に基いて推論してユーザからの質問に答えるシステムであるが，登録されている知識は随時変更（追加，削除）が可能であり，それを行うための組込み述語が用意されている．

当然であるが，事実や推論規則が変更されれば推論される帰結も異なったものになる．例えば，

```
bird(pipi)
```

という事実が 1 つだけ処理系に登録されている状況を考えてみる．この状態で処理系に対して

```
?- bird(pipi). Enter
```

と質問すると，

```
true.
```

と処理系は答える．また，この状態で処理系に対して

```
?- bird(hina). Enter
```

と質問すると，事実が登録されていないので

```
false.   （あるいは no.）
```

と答える．次にこの状態で，

```
?- assertz( bird(hina) ). Enter
```

と質問すると，bird(hina) という節が処理系に追加登録され，先と同じ質問

```
?- bird(hina). Enter
```

をすると，

```
true.   （あるいは yes.）
```

と答える．この例は assertz 述語の問い合わせによって，以降の推論結果が異なったものになることを示している．(ホーン節の登録が正常に終了すると assertz 述語の問い合わせは true となる)

すなわち，Porlog はプログラムの実行段階においてプログラム自体を改変できることを意味する．

assertz 述語は，既に処理系に登録されている節の末尾に新たな節を登録する．またこれと類似の asserta 述語も用意されており，この述語は既に処理系に登録されている節より前の位置に新たな節を登録する．

処理系に登録されている節を削除するための組込み述語 retract がある．先の例の状態で，

```
?- retract( bird(pipi) ).
```
Enter

と質問すると，節 `bird(pipi)` は処理系から取り除かれ，再度

```
?- bird(pipi).
```
Enter

と質問すると，

```
false.
```
（あるいは `no.`）

と処理系は答える．（ホーン節の削除が正常に終了すると retract 述語の問い合わせは true となる）

注意） 処理系によっては asserta,assertz,retract といった，事実や規則の定義を変更する述語が使えない場合がある．ただし，処理系の設定を変更することで使用可能となる．詳しくは「6.3.2 事実や規則の変更の可否」で説明する．

参考： 特定の述語で始まる定義を全て削除するための述語 abolish，retractall も用意されている．

2.4 データ構造

2.4.1 リスト

複数の要素を書き並べるための可変長のデータ構造として**リスト**がある．例えば `taro`，`jiro`，`hanako`，`yosiko` という 4 つの要素を書き並べたリストは次のようなものである．

```
[taro, jiro, hanako, yosiko]
```

リストは括弧 '[', ']' で要素の並びを括ったものであり，要素間はコンマ ',' で区切る．

Prolog のリストは Lisp 言語 [10] で扱うものと考え方は同じであり，リストの要素としてリストを与えることができる．例えば，要素として `[d,e,f]` を持つリスト

```
[a,b,c,[d,e,f],g,h]
```

を作ることができる．

リストは Prolog で**集合**を扱う場合などに使用できる構造である．ただし，リストは要素を記述した順の**順序構造**を持ち，同一の要素を複数持つことができる．これが集合論で扱う集合とリストが異なる点である．

■ 空リスト

要素を持たない空のリストは `[ ]` で表す．

■ リストの連結構造

リストは，左端の要素と残りのリストを '|' で連結して括弧 '[', ']' で括った構造である．すなわち，`[a,b,c]` は

```
[a|[b,c]]
```

として連結したものである．また，要素を 1 つだけ持つリストは，その要素と空リスト `[ ]` を'|' で連結したものである．すなわち，`[a]` は

[10] プログラム，データともにリスト構造で表現する言語である．'list processor' がその名の由来である．

```
[a|[]]
```

である．

リストは '|' によって先頭の要素とそれ以降のオブジェクトを連結した構造[11] である．

■ リストの構造を確認するための試み

リストの構造を確認するために，実際に Prolog 処理系で次のような質問をしてみる．

試み 1. `?- [A|B] = [a,b,c].`

```
A = a,
B = [b, c].
```

先頭要素と残りのリストが結合されている様子が確認できる．

試み 2. `?- [A|B] = [a].`

```
A = a,
B = [ ].
```

要素が 1 つのリストは，その要素と空リストを結合したものであることがわかる．

試み 3. `?- A = [a|[b|[c|[]]]].`

```
A = [a, b, c].
```

リストは '|' によって次々と先頭要素を追加した構造であることがわかる．

課題. `?- [A,B] = [a|b].` という単一化は失敗する．その原因を考えよ．

例題. リストの要素の数を求める述語 sizeOf[12] をつくる．

与えられたリストの要素の数（コンマで区切った要素の数）を求める述語 sizeOf の定義について考える．具体的には次のような仕様とする．

仕様： sizeOf(要素数を調べる対象のリスト, 要素数を単一化するための変数)

すなわち,

```
?- sizeOf([a,b,c,d],A).
```

と質問すると,

```
A = 4.
```

と答える述語をつくる．

考え方：

・空リストの要素数は 0 である．

・空リストでないリストの要素数は，そのリストの先頭以降のリストの要素数に 1 を加えたものである．

この述語の実装例を test03.pl に示す．

実装例：test03.pl

```
1 sizeOf( [ ], 0 ).
2 sizeOf( [A|B], N ) :- sizeOf( B, N2 ), N is N2 + 1.
```

[11] Lisp には，リストの先頭要素を取り出す関数 car と，それ以降のオブジェクトを取り出す関数 cdr がある．また，先頭要素を追加したリストを作る関数 cons も備わっており，Prolog の '|' がそれに相当する．

[12] Prolog にはリストの要素数を求める組込み述語 length がある．ここでは演習のための例題として sizeOf を実装する．

課題. リストの要素の存在を確認する述語 isMember を実装せよ.

与えられたリストに特定の要素が含まれることを確認する述語 isMember の定義を考えよ. 具体的には次のような仕様とする.

仕様： isMember(調べたい要素, 調査対象のリスト)

すなわち,

```
?- isMember(b,[a,b,c,d]).
```

と質問すると,

```
true.
```

と答える述語をつくる.

2.4.2 Prolog における文の基本構造

ここでは, Prolog 言語の文を構成するデータ構造について説明する. Prolog 言語の文は, 基本的には次の形を取る.

関数子（引数 1, 引数 2, ··· 引数 n）

もちろん原子論理式もこの形式であるが, 問い合わせや規則の定義に用いられる「?-」,「:-」, あるいは各種の演算子 (+,-,*,/,···) で構成される式を含む全ての式がこの形式で表現されている. このことは,「=..」演算子による単一化を実行するとよく理解できる.

■ 「=..」による式の分解と合成

「=..」は両辺に式を取る 2 項演算子である. この演算子で構成される式を問い合わせると, 左辺の式を分解した結果のリストを右辺と単一化する. 例えば.

```
?- f(x,y) =..  L.
```

と問い合わせると,

```
L = [f, x, y].
```

と答える. また右辺にリストを, 左辺に変数を与えた場合も同様の単一化が行われ, Prolog の式を合成（逆変換）することができる. (次の例を参照)

```
?- F =..  [f,x,y].  Enter        ← ユーザによる入力
F = f(x, y).                     ← システムからの返答
```

右辺のリストの 1 番目の要素には左辺の関数子が単一化される. 同様に右辺のリストの 2 番目以降の要素が左辺の引数の並びに単一化する.

このように「=..」による単一化を利用すると, Prolog の様々な式の構造を調べることができる. 以下にいくつかの例を挙げる.

例. 推論規則 「:-」

```
?- ( f(x) :- g(x) ) =..  L.  Enter
L = [:-, f(x), g(x)].
```

例. 質問 「?-」

```
?- ( ?- f(x) ) =..  L.  Enter
L = [?-, f(x)].
```

例. リスト

```
?- [a] =..  L.   Enter
L = ['[|]', a, []].
?- [a,b,c,d] =..  L.   Enter
L = ['[|]', a, [b, c, d]].
```

このように，リストは 2 項演算子 '[|]' [13] による連結構造であることがわかる．

例. 加算　「+」

```
?- a+b+c+d =..  L.   Enter
L = [+, a+b+c, d].
```

このように加算は 2 項演算であることがわかる．また，加算の連鎖は末尾に結合する形となる．すなわち，
a+b+c+d は (a+b+c) + d であり，(a+b+c) の末尾に d を結合したものである．

演算子の結合の順序や結合の強さに関しては「2.5 演算子」で説明する．

課題. 「=..」を用いて Prolog の様々な式の分解を試みよ．

参考. 式の引数を取り出す述語 arg もある．この述語は

arg(引数の位置, 式, 引数)

の形で実行（質問）する.「引数の位置」を整数で指定（最初の引数は 1）すると，与えた「式」の引数を取り出し，それが「引数」に単一化する．

結論.

Prolog で表現できる式（原子論理式や演算子で連結された式）は全て関数子と引数並びで構成される形式で表現される．例えば,「?- X is 1+2.」という質問を次のように記述して実行することもできる．

```
?- is(X,+(1,2)).
```

この結果

```
X = 3.
```

と表示される．[14]

2.4.3　アトム，シンボル，文字列リスト

構造を持たない**原子記号**を**アトム**という．アトムかどうかは，述語 atom を用いて調べることができる．

例. アトムかどうかの検査 (1)

```
?- atom(a).   Enter     ←記号「a」がアトムかどうかを調べる質問
true.                   ←アトムであることが判明
```

これと類似の述語 atomic もある．これは，空リスト '[ ]' もアトムとして判定する．

[13] '[|]' は SWI-Prolog 処理系における表記であるが，他の処理系では異なる表記になっている場合がある．詳しくは処理系の仕様を参照のこと．
[14] 質問する際にこのような表現を許さない処理系も存在する．

例. アトムかどうかの検査 (2)

```
?- atom([]).   Enter      ←空リストがアトムかどうかを調べる質問 (1)
false.        ←アトムとは認めない

?- atomic([]).   Enter      ←空リストがアトムかどうかを調べる質問 (2)
true.        ←アトムと認める
```

シングルクオート「'」で括られた記号列はアトムとして扱われるシンボルとなる．例えば，英大文字で始まる記号列は変数として扱われるが，シングルクオートで括られたものは変数ではなく通常のシンボルとして扱われる．

例. 英大文字で始まるシンボル

```
?- 'A' = 123.   Enter
false.
```

これは，'A' が変数ではない（ただのシンボルである）ので，数値である 123 と単一化することができず，質問した文が偽となる例である．

シングルクオートで括ると，特殊記号を含む多くの記号をシンボルとして扱うことができる．

ダブルクオート「"」（二重引用符）で括られた記号列は**文字列リスト**であり，その記号列を構成する文字の文字コードを要素とするリストとして扱われる．

例. 文字列リスト

```
?- A = "abc".   Enter
A = [97,98,99].
```

この例は，"abc" がそれを構成する文字（アルファベットの abc）の文字コード（10 進数で 97,98,99）を要素とするリストであることを示している．[15]

原子論理式やリストのように，アトムでない式を判定するための述語 compound がある．

例.

```
?- compound(a).   Enter
false.

?- compound(A).   Enter
false.

?- compound(f(a)).   Enter
true.

?- compound([a,b,c]).   Enter
true.
```

アトムの記号を構成する文字の文字コードを要素とするリストを得るには述語 name を用いると良い．

例. アトムの文字コードのリスト

```
?- name(abc,A).   Enter
A = [97, 98, 99].
```

[15] これは DEC-10 Prolog の仕様であるが，処理系によってはダブルクオートによる記述がリストとして扱われない場合もある．ただし，処理系の設定を変更するとリストとして扱われるようになるのが普通である．詳しくは「6.3.3 二重引用符の扱い」を参照のこと．

注）

処理系によっては，数値そのものを文字コードに変換した結果が異なることがある．すなわち，`name(123,A)` と `name('123,A)` の問い合わせにおいて A に得られる結果が処理系毎に異なる場合がある．ISO Prolog では name よりも安全に使用できる述語 atom_codes や number_codes がある．

2.4.4 変数

単一化に使用する変数は，英大文字で始まる記号列である．また，アンダーバー「_」で始まる記号列も変数として扱われる．

変数の例： A,　Xm,　_variable

参考． アンダーバーのみの記述も 1 つの変数として扱える．ただし，そのような変数は変数名で識別することができない「ダミー変数」である．

対象が変数かどうかを判定する述語 var が組込み述語として用意されている．

例．var による変数の判定

```
?- var(A).   Enter      ←「A は変数か？」
true.                   ←「変数である」

?- var(a).   Enter      ←「a は変数か？」
false.                  ←「変数ではない」
```

このように var は，対象が変数の場合に真となり，そうでない場合は偽となる．

var とは逆に，変数でないことを判定する組込み述語 nonvar も存在する．

2.5 演算子

Prolog における式の形式は「2.4.2 Prolog における文の基本構造」で述べたと通り，

関数子（引数 1, 引数 2, ··· 引数 n）

という形を基本とするが，利用者が独自の形式の演算子を定義することができる．

例えば，加算の演算子「+」と同じような連結構造をもつ演算子「plus」を定義する場合について考える．Prolog 処理系には「plus」なる演算子は用意されていないため，次のような単一化を試みるとエラーとなる．

```
?- A = 1 plus 2 plus 3.   Enter

ERROR: Syntax error:  Operator expected
ERROR: A = 1
ERROR: ** here **
ERROR: plus 2 plus 3 .
```

注． SWI-Prolog 処理系によるエラーメッセージの例

Prolog には述語 op が組込み述語として用意されており，これを利用してユーザが独自に演算子を定義できる．op 述語を用いて演算子「plus」を定義する例を次に示す．

```
?- op(500,yfx,plus).   Enter      ←ユーザによる入力
true.                             ←システムからの返答
```

この操作によって演算子「plus」が定義されたので，再度先の質問を試みると次のようになる．

```
?- A = 1 plus 2 plus 3.   Enter      ←ユーザによる入力
A = 1 plus 2 plus 3.                 ←システムからの返答
```

この例から，演算子「plus」による式が使用可能となっていることがわかる．

2.5.1 op 述語による演算子の定義

組込み述語 op の利用により，ユーザ独自の演算子が定義できる．このことは，ユーザが独自に定めた文法による式が扱えることを意味する．

■ 演算子の結合の強さと方向について

算術演算の計算式を例にして考えると,「+」,「*」の結合の強さに違いがあることがわかる．例えば，a+b+c*d という式があると，a+b+(c*d) と解釈され，c*d の部分が優先されて（先に）計算される．

また演算子には配置する位置があり，a-b と記述した場合，演算子「-」は左右に式を持つ**中置記法**であるという.「-」は負号として用いられることもあり，-a+b と記述した場合,「-」は a の前方に配置される**前置記法**であるという．更に，式の後ろに配置する**後置記法**の演算子もある．

op 述語で演算子を定義する際，その演算子を記すシンボルに加えて，結合の強さと方向（配置）を指定する．op 述語の問い合わせの書式は次のとおりである．

?- op(結合の強さ, 結合の方向, 演算子のシンボル)

「結合の強さ」は整数で指定し，値が小さいほどその演算子の優先度は高い（より結合が強い）．
「結合の方向」（演算子の配置）は次のように指定する．

● yfx

中置記法の指定．特に結合の強さが同じ演算子が連鎖した場合は，左側の式を優先的に結合した後で右端に式をつなげる形式となる．例えば加算の演算子「+」がこの形式であり，a+b+c と連鎖した場合は，(a+b)+c という優先度で結合する．すなわち，a+b に対して c を連結する形式である．

● xfy

中置記法の指定．特に結合の強さが同じ演算子が連鎖した場合は，右側の式を優先的に結合した後で左端に式をつなげる形式となる．例えばリストの連結を実現する演算子「'|'」[16] がこの形式であり，[a,b,c] と連鎖した場合は，[a|[b,c]] という優先度で結合する．すなわち，[b,c] の左側に a を連結する形式である．

● xfx

中置記法の指定．連鎖しない演算子を定義する場合に指定する．例えば推論規則の定義に使用する演算子「:-」がこの例である．

● fx

前置記法の指定．連鎖しない演算子を定義する場合に指定する．例えば問い合わせに使用する演算子「?-」がこの例である．

● fy

前置記法の指定として最も一般的なものである．負号「-」がこの例である．

算術演算に必要な演算子や，推論規則の定義に使用する「:-」，質問に使用する「?-」などは各処理系で予め定義されている．演算子の定義の詳細に関しては使用する処理系の仕様を参照のこと．

[16] SWI-Prolog 処理系の場合は '[|]' となる．

2.5.2 比較のための演算子

Prolog には比較のための演算子がいくつか予め用意されている.

■ 等値 ==

左辺と右辺が同じ値をであることを判定する演算子であり，述語である.

例.

```
?- 12 == 12.   Enter
true.
?- a == a.   Enter
true.
```

両辺が同じ数値である場合はもちろん，記号として同じものである場合も真となる．この演算子（述語）は単一化を行うためのものではない．このことは次の例でも確認できる.

```
?- A == 12.   Enter
false.
```

両辺が等しくないという理由で偽となる.

■ 等しい数値 =:=

左辺と右辺が同じ数値であることを判定する演算子であり，述語である．両辺には数式（is 述語で扱えるもの）を与えてもよい.

例.

```
?- 2 =:= 1+1.   Enter
true.
?- 1/2 =:= 0.5.   Enter
true.
```

■ 異なる数値 =\= [17]

右辺と左辺が異なる 数値 であることを判定する演算子であり，述語である.

例.

```
?- 12 =\= 24.   Enter
true.
```

■ 異なる式 \==

右辺と左辺が異なる 式 であることを判定する演算子であり，述語である.

例.

```
?- a \== b.   Enter
true.
?- a \== 2.   Enter
true.
```

[17] 多くの端末装置では，バックシュラッシュ「\」が円記号「¥」で代用されている.

■ **大小関係**　<, >, =<, >=

数値の大小関係を判定する演算子であり，述語である．

例.

```
?- 1 < 2.   Enter
true.
?- 1 =< 1.   Enter
true.
?- 2 > 1.   Enter
true.
?- 1 >= 1.   Enter
true.
```

※　SWI-Prolog，GNU Prolog で予約されている演算子に関しては巻末付録「B 予約されているオペレータ」を参照のこと．

2.6　入出力

多くのプログラム言語と同様に，Prolog にも入出力のための機能が備わっている．例えば，**標準入力**と**標準出力**はその代表的なものであり，質問の受付は標準入力に，質問への返答は標準出力に予め割当てられている．

2.6.1　出力

出力処理のための述語に write がある．この述語は，引数に指定された値を，**出力ストリーム**に出力する．出力ストリームは端末ディスプレイやファイルといった「出力先」のことであり，処理系の起動当初は標準出力に割当てられている．

例. 'Hello, world.' という文字列を標準出力に出力する．

```
?- write('Hello, world.').   Enter
Hello, world.
true.
```

この述語は正常に処理を終えると結果として「真」となる．

■ **ファイルへの出力**

出力ストリームを切り替えることで，write 述語で出力される内容をファイルに保存することができる．出力ストリームをファイルに接続するには tell 述語を使う．

例. 出力先をファイル 'test1.txt' にする．

```
?- tell('test1.txt').   Enter
true.
```

この後，出力先がファイル 'test1.txt' になり，write 述語による出力の内容がそのファイルに保存される．(ディスプレイには表示されない)　tell 述語は正常に処理を終えると真となる．

ファイルへの出力を終了し，出力ストリームを標準出力に戻すには，述語 told を使用する．

例. ファイルへの出力の終了

```
?- told.   Enter
true.
```

2.6.2　入力

入力処理のための述語に read がある．この述語は，**入力ストリーム**から Prolog の文を 1 つ読込み，引数に与えた変数にそれを単一化する．入力ストリームはキーボードやファイルといった「入力元」のことであり，処理系の起動当初は標準入力に割当てられている．

例. キーボードから 1 文を読み取る．

```
?- read(X).   Enter        ←ユーザによる入力
test_string.               ←ユーザによる入力
X = test_string.
```

この述語は正常に処理を終えると結果として「真」となる．

■ ファイルからの入力

入力ストリームを切り替えることで，ファイルに保存されている文を read 述語で読み取ることができる．入力ストリームをファイルに接続するには see 述語を使う．

例. 入力元をファイル 'test1.txt' にする．

```
?- see('test1.txt').   Enter
true.
```

この後，入力元がファイル 'test1.txt' になり，read 述語によってそのファイルから文を読み取ることができる．see 述語による入力元の切り替えが正常に終了すると真となる．

例. （続き）入力元ファイル 'test1.txt' から 1 文を読み取る．

```
?- read(X).   Enter
X = content_of_the_file.
```

これは，Prolog の文「content_of_the_file.」がファイル 'test1.txt' に保存されている場合の例であり，それを読み取った結果が変数 X に単一されていることを示すものである．

注意. read 述語が読み取るのは Prolog の文であるので，この例のように，ファイルに保存されている文の終端にはピリオド「.」が記述されている必要がある．

ファイルからの入力を終了し，入力ストリームを標準入力に戻すには，述語 seen を使用する．

例. ファイルからの入力の終了

```
?- seen.   Enter
true.
```

参考.

多バイト文字を含むデータを入出力で扱う場合，そのための設定を要する処理系がある．SWI-Prolog では，入出力の処理に先立って，扱うデータの文字コードを encoding フラグに設定しておく必要がある．(次の例参照)

実行例　SWI-Prolog の入出力で UTF-8 の多バイト文字を扱う場合の設定．

```
?- set_prolog_flag(encoding,utf8).   Enter     ← UTF-8 に設定
true.          ←設定処理が正常に終了した．
```

2.6.3 論理プログラミングにおける入出力処理の解釈

Prolog は論理プログラミングのための言語処理系であり，文の真偽を判定したり，文を真にするための単一化や，それによる探索をすることを主たる機能とする．

入出力のための処理は論理プログラミングの範疇にはなく，先に説明した入出力に関する各種の述語を含んだ文も，結果として真か偽となるものである．したがって，入力や出力の処理そのものは，それら述語を含んだ文の問い合わせに付随する 副作用 と見るべきである．

2.7 Prolog の動作の手続きとしての解釈

一階述語論理においては，事実や推論規則を**宣言的**に記述する．すなわち，事実や規則が定義された順序をあまり意識することなく，与えられたある文を検証する際に，事実や規則を「適宜」適用することでその文の真偽を検証する．しかし論理プログラミングにおいては，事実や規則の適用順序が検証処理の効率に大きく影響を与える．

Prolog は SLD 導出 [1] [18] に基いてプログラムを実行する．すなわち，事実や規則の適用は，それらが記述された順番で適用される．別の表現をするならば，assertz 述語などで処理系に登録された順番で，あるいは，ソースプログラムとして記述した順番で，事実や規則が適用されるということができる．また推論規則を適用する際も，ゴール部の記述の順番に従って検証される．詳述すると，

$$Q \ :- \ P_1, P_2, \cdots, P_n$$

と記述された規則の頭部 Q を検証するために，ゴール部は $P_1, P_2, \cdots, P_n$ の順番で検証される．

以上のことを理解するための例題について考える．

例題. 整数の値を順番に出力する

与えられた整数値 N から Max までを順番に出力する述語 loopTest01 について考える．この述語は

loopTest01(開始値, 終了値)

と問い合わせると,「開始値, 開始値+1, 開始値+2,···, 終了値」と出力するものとする．

この仕様を満たすプログラムの例を test04.pl に示す

プログラム：test04.pl

```
1 loopTest01(N,Max) :-
2     N < Max, write(N), write(','), N2 is N + 1, loopTest01(N2,Max).
3 loopTest01(N,N) :- write(N), nl.
```

考え方.

loopTest01(N,Max) を検証する際，$N < Max$ ならば N を出力したあと，loopTest01(N+1,Max) を検証するという再帰的なアルゴリズムとなっている．また，N と Max が等しい場合は N を出力するのみで，これを再帰的呼び出しの終了とする．

実行結果

```
?- loopTest01(1,10).   Enter
```

と問い合わせると，

```
1,2,3,4,5,6,7,8,9,10
true
```

と表示される．

[18] 文献 [1] の中では **SL レゾリューション**と呼んでいる．

動作の解説

プログラムは 1〜2 行目が $N < Max$ の場合の規則で，3 行目が $N = Max$ の場合の規則である．2 行目の記述のように，ゴール部の中に動作のための条件を記述する．すなわち 2 行目に $N < Max$ という記述があるので，$N \geq Max$ の場合は 1〜2 行目の規則の検証が偽となって失敗する．当然，2 行目の以後のゴール部は検証（実行）されず，検証が 3 行目に移る．また 2 行目の実行時に $N < Max$ の条件を満たす場合は，N の値とコンマ「,」の出力を行い，N+1 の値を N2 として，述語の再帰呼び出しに移る．

再帰呼び出しが次々と起こると，最終的に $N = Max$ となり，2 行目の実行が失敗する．この場合，述語の検証は 3 行目に移り，N の出力と改行出力（nl の実行 [19]）が行われ，検証全体が終了する．

課題. 上記プログラムにおいて $N > Max$ の条件で問い合わせるとどのような結果となるか考えよ．
また実際に実行して確かめよ．

2.7.1 プログラムの実行順序

結論として，Prolog のプログラムは「記述した順に実行される」．すなわち，事実や規則は記述した順番に，規則のゴール部は左から右へ順番に実行（検証）される．

2.7.2 仮引数部でのデータの検査

先のプログラム test04.pl の 3 行目における，$N = Max$ の判定方法について考える．この部分（3 行目）は

```
loopTest01(N,Max) :- N == Max, write(N), nl.
```

と記述しても同様の仕様を実現できる．これに対して先に挙げた例では 3 行目に

```
loopTest01(N,N) :- write(N), nl.
```

と記述している．このような記述をすることで，コーティングの量を少なく抑えることができるだけでなく，より重要なこととして，**仮引数部によるデータの検査が可能である**ということが言える．

Prolog には単一化の機能があるため，1 つの節の中で記述された同じ名前の変数は同じ値に束縛される．すなわち，`loopTest01(N,N)` と記述すると，第 1 番目の引数と第 2 番目の引数が同じ名前の変数であり，共に同じ値である場合にのみ `loopTest01(N,N)` は真となる．

Prolog では事実や規則の定義を記述する際，頭部の仮引数の記述のしかたによってデータの等値性の判定処理が実現できることを意味する．これは Prolog が強力な条件判定の機能を備えていることを意味する．例えば，引数に与えたデータがリストであるかどうかを判定する述語 isList について考える．サンプルプログラムを test05.pl に示す

プログラム：test05.pl [20]

```
1 isList([]).
2 isList([A|B]).
```

考え方.

データとしてのリストは次の 2 つのものしかない．

1) 空リスト
2) 空でなりリスト

サンプルプログラムはこの定義をそのままコーディングしたものである．

[19] 述語 nl を質問すると，改行コードが出力される．
[20] 2 行目の仮引数の中にある変数 A, B はゴール部で使用されない（そもそもゴールの記述がない）ため，このプログラムを読み込むと警告メッセージを発する処理系もある．それを解消するには 2 行目の記述をダミー変数（アンダーバー）を用いて `isList([_|_]).` とすると良い．

2.7.3 カット「！」

Prolog は問い合わせの検証において，登録されている適用可能な述語を全て試すが，バックトラックによる再検証を抑制するために **カット**と呼ばれる特殊な述語を使用することができる．カットは感嘆符「！」で表現される．

カットの動作と用法を理解するために，階乗を計算するプログラムについて考える．
階乗の定義は次の通りである．

$$factorial(n) = \begin{cases} n = 0 & \rightarrow \quad 1 \\ n > 0 & \rightarrow \quad n \times factorial(n-1) \end{cases}$$

これを Prolog のプログラムとして実装したものが test06-1.pl である．

プログラム：test06-1.pl

```
1 factorial(0,1).
2 factorial(N,F) :- N > 0, N2 is N - 1, factorial(N2,F2), F is F2 * N.
```

実行例. 10 の階乗を計算する

```
?- factorial(10,A).   Enter
A = 3628800
```

この表示の状態でシステムは動作を一時停止し，他の解の検証待ちとなる．セミコロン「；」をタイプすれば他の解を探す動作に入り，他に解がなけれ次のようにば `false` と表示される．

```
A = 3628800 ;      ←セミコロンをタイプ
false.             ←他の解がないことを通知
```

次にこのプログラムの実行効率の改善，すなわち，より少ないステップで処理系が計算を実行する方法を考えることで，カットの動作について説明する．

まずホーン節の記述の順番を変える．大きな数 N の階乗を計算するには，

$$N \times (N-1) \times (N-2) \times (N-3) \times \cdots \times 1$$

と計算することとなり，0 よりも大きい数の階乗の計算が圧倒的に多数を占めることになる．従って，プログラムを test06-1.pl のように書き換える．

プログラム：test06-1-1.pl

```
1 factorial(N,F) :- N > 0, N2 is N - 1, factorial(N2,F2), F is F2 * N.
2 factorial(0,1).
```

このようにすることで，1 回しか発生しない 0 の階乗の計算の優先順位を下げることができる．すなわち，0 より大きい数の階乗を計算する規則が最初に記述されているので，処理系がそれを優先的に実行することになる．先のプログラム test06-1.pl では `factorial` の再帰呼び出しの度に毎回 `factorial(0,1)` を検証することになるので余計な手順が増えることになる．

プログラムを test06-1-1.pl のように書き換えることで余計な検証の回数を減らすことができるが，処理系は，`factorial(0,1)` の検証も後で一応実行する．ここでカットを用いることで，更にこの余計な検証を省くことができる．次にプログラムを test06-2.pl のように書き換える．

プログラム：test06-2.pl

```
1 factorial(N,F) :- N > 0, N2 is N - 1, factorial(N2,F2), F is F2 * N, !.
2 factorial(0,1).
```

このプログラムの 1 行目の終わりの部分にカット「！」が挿入されている．カットが記述されていると，その節の検証が成功した（真となった）場合は以降の節の検証（バックトラック）を取りやめる．すなわち，1 行目の節が成功すると，2 行目の `factorial(0,1)` の検証は行わない．この例より，カットの記述によって不要な動作を排除できることが理解できる．

■ カットを用いる際の注意

カットを用いて定義された述語は，プログラマによって意図的に検証範囲が制限されている．従って，カットを用いて定義された述語は，一階述語論理による解釈の対象から逸脱していると考えるべきである．

実行効率の問題が深刻な場合のプログラミングにはカットの使用もやむを得ないが，カットを使用して定義した述語は，述語論理の推論系としての解釈を控え，1 つのブラックボックスとしてとらえるべきである．

参考： カットによって定義される「否定」

否定を実現する述語 not はカットを用いて定義[21] できる．次の例について考える．

```
not(P) :- P,!,fail.
not(_).
```

この例の動作を解釈すると次のようになる．

1 行目：「P が真ならば，not の検証はこの節で終了し，fail によって結果を偽とする」
2 行目：「それ以外ならば真とする」

この定義による not は論理学上の否定とは異なる．二重否定 $not(not(P))$ は P とは同一のものではなく，式 P に含まれる変数は単一化の対象とはならない．

課題. Prolog の二重否定によって，元の式は単一化しないことを確かめよ．

2.7.4 サンプルプログラム

仮引数部でデータを検査する方法，カットで探索を抑制する方法を用いてフィボナッチ数列を生成するプログラムについて考えるが，ここでは実行効率を順次改善する形でプログラムを例示する．

フィボナッチ数は次のように定義される．

$$F_0 = 0,$$
$$F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2} \quad (n \geq 2)$$

このような数で構成される数列を生成する．

プログラム： test07-1.pl

```
1 fib(N,[F,F1,F2|L]) :-
2     N > 1, N1 is N - 1, N2 is N - 2,
3     fib(N1,[F1,F2|L]), fib(N2,[F2|L]), F is F1 + F2,!.
4 fib(0,[0]) :- !.
5 fib(1,[1,0]).
```

解説

このプログラムはフィボナッチ数の定義を忠実に反映した形で実装されている．$n \geq 2$（$n > 1$）の場合の計算の頻度が高いため，その場合の定義の実装をプログラム冒頭（1〜3 行目）に記述している．また処理系の動作を最小限に

[21] 実際にそのように実装している処理系もある．

するためにカットが使用されている.

このプログラムの実行例を次に示す.

```
?- fib(10,A).   Enter                    ←ユーザによる入力
[55, 34, 21, 13, 8, 5, 3, 2, 1, 1, 0].   ← n = 10 までのフィボナッチ数のリストが表示される
```

問題なく計算が実行されていることがわかるが，次に計算量の改善について考える.

先の定義に従うと，1 つのフィボナッチ数の算出において 2 回の再帰呼出しを行う形となる．このことは，n が大きくなるにつれ計算量が幾何級数的に増大することを意味する．実際，n を大きくしてみると計算が実用的な時間範囲内で終了しないことがわかる.（あるいは，計算資源を使い果たして処理系が計算を打ち切ることもある）

次に計算量の削減のための改善例について説明する.

再帰的定義を忠実に実装したプログラムは，多くの場合に実行時の計算量が大きくなる．そこで**動的計画法** [22] などの手法を採用して実行時の計算量を改善するのが一般的である.

次に，動的計画法を応用して計算量の削減を実現する例について考える.

プログラム：test07-2.pl

```
1 fib(N,[F,F1,F2|L]) :-
2     N > 1, N2 is N - 1, fib(N2,[F1,F2|L]), F is F1 + F2,!.
3 fib(0,[0]) :- !.
4 fib(1,[1,0]).
```

解説

このプログラムでは，より小さな n に対するフィボナッチ数を次々と算出しながら，それらをリストに追加する形でフィボナッチ数列全体を生成している.

課題. 「エラトステネスのふるい」を用いて素数の列を生成するプログラムを作れ.

2.7.5 繰り返し動作の実現

先に「1.2.3 単一化（unification）」のところで，述語 human として複数の事実が登録されている場合に，human(A) の変数 A に個人名を単一化させる手法で全ての A の候補を挙げることが試みられた.
（プログラム test01.pl の最終版による試み）

そこでは，問い合わせに対する応答が表示される度に，キーボードからセミコロン「；」をタイプすることで，変数に単一化する値の候補を次々と表示するものであった.

ここでは，変数に単一化する値の候補を，ユーザのインタラクションを介すること無く自動的に列挙する方法について考える.

■ 「失敗」による繰り返しの実現

Prolog 処理系は，受けた質問を真にするべく単一化の動作をとる．すなわち，単一化の候補として取り上げた値を変数に当てはめて検証し，それが偽となった場合は，更に別の候補を変数に当てはめるという検証を続ける．プログラム test01.pl が読み込まれた状態で，次のように質問するとどうなるか考える.

```
?- human(A),write(A),nl,fail.   Enter
```

この質問に対する結果は単純である．質問文の最後に fail があることから，節は全体としては偽となる．ただし実行結果は次のようになる.

[22] 問題を小さな問題に分割して解決した結果を記録し，それらを利用しながら問題全体を解決する手法.

```
taro
jiro
hanako
yosiko
false.
```

この例からわかるように，変数 A に単一化が試みられた値が全て出力されている．これは，human(A) から nl までの検証が実行されたことによる出力であり，最後の fail によって偽となるので，処理系が可能な限りの A の候補を次々と挙げたことによる．処理系は，どの述語が偽となる結果をもたらしたのかを関知しないため，A に可能な限りの単一化を試みる．これが繰り返し処理をもたらした理由である．

ここまでのことを踏まえ，繰り返し処理を実現する述語 searchAll を定義するプログラム test08.pl を示す．

プログラム：test08.pl

```
1  human(taro).
2  human(jiro).
3  human(hanako).
4  human(yosiko).
5
6  searchAll :- human(A),write(A),nl,fail.
```

このプログラムを読み込んで，

```
?- searchAll.
```
Enter

と質問すると，先と同じように

```
taro
jiro
hanako
yosiko
false.
```

と表示される．

■ 発展. 試みた検証結果をまとめてデータとして取得する方法

先に説明した繰り返し処理は，単一化する候補を出力するのみであり，列挙された候補をデータとして取り出すには別の方法が必要となる．write や nl は副作用として出力処理を実現するものであるが，副作用としてデータを追記する働きを持つ述語が実現できれば，単一化候補をまとめてデータとして取り出すことが可能となる．例えば先のプログラム test08.pl を次のように改変する．

改変したプログラム：test08.pl

```
1  human(taro).
2  human(jiro).
3  human(hanako).
4  human(yosiko).
5
6  searchAll :- human(A),registerL(A),fail.
7
8  resultL([]).
9
10 registerL(A) :- resultL(L), retract(resultL(L)), assertz(resultL([A|L])).
```

解説.

改変前のプログラムと比べると，searchAll のゴールの中の write(A),nl の部分が registerL(A) に変わっている．この registerL(A) は副作用として A の値を追記する文である．registerL は，resultL の登録を書き換え

る [23] 形でデータを追記する．このプログラムを読み込んだ当初の段階では，

```
resultL([])
```

が処理系のデータベースに登録される．resultL の引数が空リストになっているのは，「まだデータが記録されていない」ということを意味する．この状態で

```
registerL(d1).  Enter
```

と質問すると，先の resultL([]) は書き換えられて resultL([d1]) となる．[24] 更に

```
registerL(d2).  Enter
```

と質問して d2 を追記すると resultL([d1]) の定義が書き換えられて resultL([d2,d1]) となる．以下同様に registerL の質問をする度に resultL の定義が書き換えられてデータが追記されてゆく．

このプログラムを処理系に読み込んで実行した結果を次に示す．

```
?- searchAll.  Enter      ←ユーザによる入力
false.                    ←システムの応答
```

この後，次のようにして resultL について質問する．

```
?- resultL(A).  Enter                ←ユーザによる入力
A = [yosiko, hanako, jiro, taro].    ←システムの応答
```

human(A) の A に該当するものが全てリストとして取得されることが確認できる．

課題． 先のサンプルプログラム test08.pl では，searchAll 述語は結果として偽となる．
これを結果として真となるように改変せよ．

参考） 失敗による繰り返しを起こすための組込み述語 repeat もある．

2.8 その他

2.8.1 問い合わせ結果の否定

論理式の検証結果を否定するための述語に \+ がある．

例． 単一化の “否定”

```
?- \+ a=b.  Enter
true.
```

[23] Prolog には事実や規則を書き換えるために asserta，assertz を始めとする組込み述語が用意されている．詳しくは「2.3 非単調推論のための機能」を参照のこと．

[24] 処理系によっては，ソースプログラムとして読み込んだ事実や規則を実行時に改変できないという初期設定になっているものもある．SWI-Prolog 処理系の場合は，test08.pl の先頭に「:- dynamic(resultL/1).」と記述して，resultL が事後に改変できるように指定しておく必要がある．

3 例題

3.1 経路探索

節点と辺で構成されるグラフ上の経路を探索するプログラムについて考える．
ここでは，図 1 に示すような 9 つの節点からなる格子状のグラフ上で，指定した 2 地点の間の経路を列挙するプログラムについて考える．

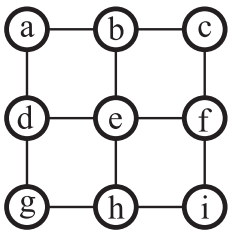


図 1: a〜i の 9 つの地点を結ぶ経路の図

グラフの表現

今回は各節点の連結関係を事実の形で記述することにする．例えば，a〜b 間の連結を

```
adjacent(a,b).
```

と記述する．

経路探索用の述語の仕様

isPath(X,Y,L) のような形で，指定した 2 地点 X, Y の間の経路をリスト L の形で求める述語 isPath を定義する．

考え方

開始地点 X の次の地点 Z から終点 Y までの経路のリスト L を再帰的に求め，X - Z 間の接続がある場合はリスト L に開始地点 X を加える．経路リストの作成に当たっては，同じ地点を通過しないようにする．

プログラム：Sample01.pl

```
 1 adjacent(a,b).
 2 adjacent(b,c).
 3 adjacent(d,e).
 4 adjacent(e,f).
 5 adjacent(g,h).
 6 adjacent(h,i).
 7 adjacent(a,d).
 8 adjacent(d,g).
 9 adjacent(b,e).
10 adjacent(e,h).
11 adjacent(c,f).
12 adjacent(f,i).
13
14 isPath(X,Y,[X,Y]) :- adjacent(X,Y);adjacent(Y,X).
15 isPath(X,Y,[X|L]) :-
16     isPath(Z,Y,L),
17     (adjacent(X,Z);adjacent(Z,X)), untrampled(X,L).
18
19 untrampled(_,[]) :- !.
20 untrampled(X,[X|_]) :- !,fail.
21 untrampled(X,[_|L]) :- untrampled(X,L).
```

解説

プログラム中の述語 untrampled は，再帰的に求めた経路リストに同じ地点を追加しないようにするためのもので，経路リスト L の中に地点 X が含まれていないことを untrampled(X,L) という形で判定する．untrampled の定義に当たっては，実行効率を優先するためにカット「！」を使用している．

このプログラムを処理系に読み込んで実行した例を次に示す.

```
?- isPath(a,i,L).   Enter
```

と入力すると,

```
L = [a, d, g, h, i]
```

と表示される．この後セミコロン「;」をタイプすると，次のように変数 L に経路の候補が次々と単一化される.

```
L = [a, d, g, h, i] ;
L = [a, d, e, h, i] ;
L = [a, b, e, h, i] ;
L = [a, d, e, f, i] ;
L = [a, b, e, f, i] ;
L = [a, b, c, f, i] ;
L = [a, b, e, d, g, h, i] ;
L = [a, b, c, f, e, h, i] ;
L = [a, d, g, h, e, f, i] ;
L = [a, d, e, b, c, f, i] ;
L = [a, b, c, f, e, d, g, h, i] ;
L = [a, d, g, h, e, b, c, f, i]
```

この状態（12 番目の候補が表示された状態）でセミコロンをタイプするとシステムからの応答が無くなる.

課題. システムからの応答がなくなる原因を考えよ．また 9 つの地点を全て含んだ経路の候補を全て列挙したところでバックトラック探索を終了するように先のプログラムを修正せよ.

3.2 3 賢人のパズル

複数の参加者による推理パズルとして「3 賢人」のパズルがある．ここでは，3 賢人のパズルを解く参加者（エージェント）達を Prolog で実装する例 [25] を挙げる.

3.2.1 パズルの概要

白か黒のどちらかの帽子をかぶった参加者が，互いに他人の帽子が見えるように配置されており，必ず白の帽子と黒の帽子の参加者が両方存在する.（図 2）このとき，参加者は自分の帽子を見ることができない．このような状況でそれぞれの参加者が自分の帽子の色を推理するのがこのパズルである．参加者は全員，十分に知的であるとする.

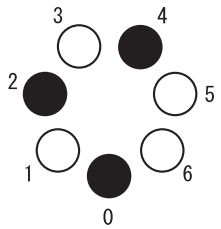


図 2: 3 賢人のパズル

パズル実行の流れ：

参加者とは別に司会者がおり，司会者が促したときに，自分の帽子の色が分かった参加者が自分の帽子の色を発表する．この時，自分の帽子の色が推理できない参加者は沈黙する．また司会者の 1 回の促しに対して複数の参加者が同時に発表することができ，発表は参加者全員に聞こえる．参加者全員が自分の帽子の色を当て終えるまで司会者は発表の促しを繰り返す.

[25] 特に本書では Prolog プログラミングの入門を意識しており，**知識と信念の論理** [4] に基づいた設計にはせず，状況に反応する単純なオートマトンが非単調推論を実行する例として提示する.

3.2.2　推理の方法

それぞれの参加者が自分の帽子の色を推理する方法を考えるために，1 人の参加者（0 番）の推理に注目し，パズルの開始時点でその参加者の視界に存在する「黒」の帽子の人数による判断について考えることとする．

開始時の状況 1) 黒の帽子が 1 人だけ存在する場合

パズルの開始時が図 3 のような状況であると想定する．

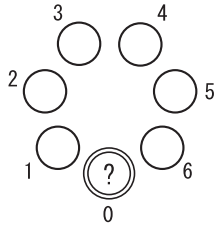


図 3: 0 から見て他全員の帽子が「白」の場合

これは 0 番の参加者の視界に，黒の帽子をかぶっている参加者が 1 人もいない状況である．この状況で 0 番の参加者は次のように考える．

「もしも自分の帽子が白ならば，参加者全員の帽子が白ということになり，それは『必ず白・黒両方存在する』という前提に反する．従って自分の帽子は黒である」

このような判断をした 0 番の参加者は，司会者の最初の発表の促しに対して，

「私の帽子は黒です」

と答える．また図 3 の状況では，0 番以外の参加者の視界には黒い帽子の参加者（0 番）が 1 人だけ存在することから，他にも黒の帽子の参加者がいる可能性があり,『自分も黒かもしれないし，白かもしれない』として，自分の帽子の色を断定できず，初回の発表時は司会者の促しに対して沈黙する．

初回の発表で 0 番の参加者が発言したことを受けて，0 番以外の全員は自分が「白」であると結論し，次回の発表でそれを発言する．

開始時の状況 2) 黒の帽子が 1 人だけ見える場合

パズルの開始時が図 4 のような状況であると想定する．

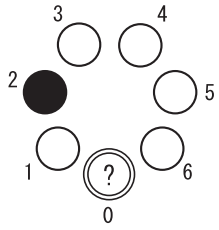


図 4: 0 から見て 2 の帽子のみが「黒」の場合

この状況で，0 番の参加者はまだ自分の帽子の色がわからないとする．そこで，0 番の帽子が「白」である場合と，「黒」である場合に分けて，0 番の参加者の推理の流れについて考える．まず，0 番の帽子が「白」である場合は，2 番の参加者が初回の発表時に自分の帽子の色を発言する．理由は「**開始時の状況 1)**」と同じ状況に置かれた 2 番の参加者の推理の方法である．

次に，0 番の帽子が「黒」である場合について考える．この場合は 2 番の参加者の視界にも「黒」が 1 人存在（0 番）することになり，初回の発表では 0 番と 2 番が共に沈黙する.（当然他の参加者も判断できず沈黙する）この瞬間に，0 番と 2 番の参加者は「2 人だけが黒である」と推理して，2 回目の発表時に自分たちの帽子の色を発言する．もちろんその次の発表では，残りの参加者も自分の帽子の色（白）を知っており，それを発言する．この判断の流れを図 5 に

示す．

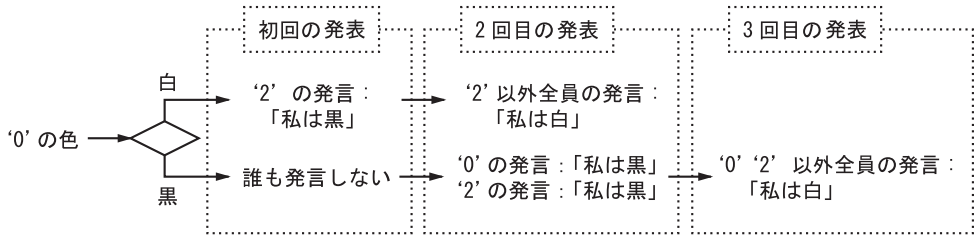


図 5: 図 4 の場合の運び

開始時の状況 3) 黒の帽子が 2 人見える場合

パズルの開始時が図 6 のような状況であると想定する．

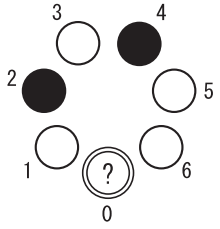


図 6: 0 から見て 2 と 4 の 2 人の帽子が「黒」の場合

先と同様に，開始時点では 0 番の参加者はまだ自分の帽子の色がわからないとする．そこで，0 番の帽子が「白」である場合と,「黒」である場合に分けて，0 番の参加者の推理の流れについて考える．まず，0 番の帽子が「白」である場合は，参加者の内 2 名（2 番と 4 番）が「黒」であることから「**開始時の状況 2)**」と同様の状況であり，初回発表時は誰も発言せず，2 回目の発表時に 2 番と 4 番が自分の帽子の色（黒）を発言する．更に 3 回目の発表時に残りの参加者が自分の帽子の色（白）を発言する．

次に，0 番の帽子が「黒」である場合について考える．この場合は 0 番，2 番，4 番の参加者が同様の立場に置かれており，それぞれが自分の帽子の色を場合分けに基いて推理することになる．まず初回発表時には，その 3 人は自分の帽子が「白」である可能性を念頭に置いて，自分以外の 2 人がその 2 回目の発表時に発言することを待つ．更に 2 回目の発表時にも自分以外の 2 人が沈黙するのを確認すると「黒の帽子は 2 名だけではない」と結論し，3 回目の発表時にその 3 人が同時に自分の帽子の色（黒）を発言する．これを受けて 4 回目の発表時には残りの参加者が自分の帽子の色（白）を発表する．この判断の流れを図 7 に示す．

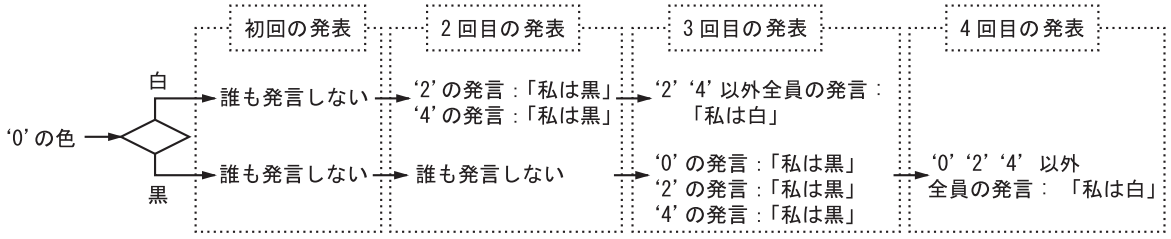


図 7: 図 6 の場合の運び

3.2.3 実装例

先に示した実行の流れに従って 3 賢人のパズルを解くシステムの実装例を示す．

設計

各エージェントが白／黒の帽子をかぶっている様子を述語 factWears により記述することにする．例えばエージェント 0 が黒の帽子をかぶっている様子を次のように記述する．

```
factWears(0,black)
```

エージェント 1〜7 についても同様に記述する．

現在何回目の発表であるか（何回目のターンであるか）を述語 factTurn により記述することにする．最初は 1 回目なので，

```
factTurn(1)
```

と記述する．

1 つのエージェントの視界に他のどのエージェントが見えるかを述語 perceptionVisual により記述することにする．例えばエージェント 0 からはエージェント 1〜7 が見えるがこれを次のように記述する．

```
perceptionVisual(0,[1,2,3,4,5,6,7])
```

これで，エージェント 0 から見える他のエージェントの集合がリスト [1,2,3,4,5,6,7] として定義される．
1 つのエージェントから見える他のエージェントのリストが perceptionVisual の第 2 引数として得られることから，先の factWears による事実の記述と合わせて，視界に見えるエージェント達の帽子の色のリストを取得することができる．そのために必要な述語 seeColor を次のように定義する．

```
seeColor([Af|Ar],[Cf|Cr]) :- factWears(Af,Cf), seeColor(Ar,Cr).
seeColor([],[]).
```

seeColor を用いることで，エージェントのリストから帽子の色のリストを得ることができる．

ここまでで，各エージェントの視界に見える他のエージェントたちの帽子の色が認識できることになる．また，リスト中の特定の要素の数を取得する述語 countMember を次のように定義すれば，視界に見える帽子の色を数えることが可能になる．

```
countMember(Mem,[Mem|L],N) :- countMember(Mem,L,N2), N is N2 + 1,!.
countMember(Mem,[_|L],N) :- countMember(Mem,L,N),!.
countMember(_,[],0).
```

例えば,「黒の帽子が視界に 1 つ見える」 ということを認識する述語 see1black を次のように定義できる．

```
see1black(N) :- perceptionVisual(N,L), seeColor(L,C),countMember(black,C,B),B==1.
```

これで，エージェント N の視界に黒の帽子が 1 つ見えることが判定できる．同様の定義により，黒の帽子がいくつ見えるかを判定する述語が実装できる．

各エージェントは，自分の帽子の色が分かれば次の発表の回でそれを皆に知らせる．この告知を announcement 述語を処理系に登録することで行うことにする．そのための述語 announce を次のように定義する．

```
announce(N,C) :- announcement(_,N,_),!.
announce(N,C) :- factTurn(T), assertz( announcement(T,N,C) ).
```

例えば，エージェント 3 が「自分の帽子が黒である」旨を皆に伝える場合，

```
announce(3,black)
```

を処理系に問い合わせればよい．これにより，例えば現在のターンが 2 の場合は，

```
announcement(2,3,black)
```

が処理系に登録される．他のエージェントはこれを参照することができる．

ここまでの定義で各エージェントは，

・ 視界にいる他のエージェントの帽子の色と数

・現在のターン
・過去の発表内容

を知ることができ，発表を重ねる中で自分の帽子の色を推論することができる．

実装全体をプログラム Sample02.pl に示す．

プログラム：Sample02.pl

```
/*********************************************************
 *  3賢人のパズル                                        *
 *********************************************************/

/*-----------------------------------------------*
 |  事実                                         |
 *-----------------------------------------------*/
/* エージェントが帽子をかぶっている様子 */
factWears(0,black).
factWears(1,white).
factWears(2,white).
factWears(3,black).
factWears(4,white).
factWears(5,white).
factWears(6,black).
factWears(7,white).

/* 現在N回目の発表 */
factTurn(1).

/*-----------------------------------------------*
 |  エージェントの視覚                           |
 *-----------------------------------------------*/
perceptionVisual(0,[1,2,3,4,5,6,7]).
perceptionVisual(1,[0,2,3,4,5,6,7]).
perceptionVisual(2,[0,1,3,4,5,6,7]).
perceptionVisual(3,[0,1,2,4,5,6,7]).
perceptionVisual(4,[0,1,2,3,5,6,7]).
perceptionVisual(5,[0,1,2,3,4,6,7]).
perceptionVisual(6,[0,1,2,3,4,5,7]).
perceptionVisual(7,[0,1,2,3,4,5,6]).

/*-----------------------------------------------*
 |  エージェントの認識                           |
 *-----------------------------------------------*/
/* エージェントNの視界に黒がいない */
see0black(N) :- perceptionVisual(N,L),seeColor(L,C), countMember(black,C,B),B==0.

/* エージェントNの視界に黒が1人いる */
see1black(N) :- perceptionVisual(N,L),seeColor(L,C), countMember(black,C,B),B==1.

/* エージェントNの視界に黒が2人いる */
see2black(N) :- perceptionVisual(N,L),seeColor(L,C), countMember(black,C,B),B==2.

/* エージェントNの視界に黒が3人いる */
see3black(N) :- perceptionVisual(N,L),seeColor(L,C), countMember(black,C,B),B==3.

/* 視界にいるエージェント達の帽子の色を知る */
seeColor([Af|Ar],[Cf|Cr]) :- factWears(Af,Cf), seeColor(Ar,Cr).
seeColor([],[]).

/*-----------------------------------------------*
 |  汎用述語                                     |
 *-----------------------------------------------*/
/* リスト中の特定の要素の個数を得る */
countMember(Mem,[Mem|L],N) :- countMember(Mem,L,N2), N is N2 + 1,!.
countMember(Mem,[_|L],N) :- countMember(Mem,L,N),!.
countMember(_,[],0).

/* 発表を次のターンに進める */
```

```
61 nextTurn :-
62     factTurn(N), N2 is N + 1,
63     retract( factTurn(N) ), assertz( factTurn(N2) ).
64
65 /* 自分の色を発表する */
66 announce(N,C) :- announcement(_,N,_),!.
67 announce(N,C) :- factTurn(T), assertz( announcement(T,N,C) ).
68
69 /*-----------------------------------------------*
70  |  エージェントの動作                           |
71  *-----------------------------------------------*/
72 /* 既に自分の色が分かっている人はそのまま */
73 actionAgent(N) :- factTurn(T),
74     announcement(T2,N,C), T2 < T,!,
75     write(N),write(' says '),write(C),nl.
76
77 /* 視界に黒がいなければ，私は黒である */
78 actionAgent(N) :- see0black(N),
79     announce(N,black),!,
80     write(N),write(' says '),write(black),nl.
81
82 /* 視界に黒が1人いて，
83     以前のターンで自分が黒であることが分かった人がいれば，
84     私は白である */
85 actionAgent(N) :- see1black(N),
86     factTurn(T),
87     announcement(T2,N2,black), T2 < T, N2 =\= N,
88     announce(N,white),!,
89     write(N),write(' says '),write(white),nl.
90
91 /* 視界に黒が1人いて，現在2回目以降のターンならば，
92     私は黒である */
93 actionAgent(N) :- see1black(N),
94     factTurn(T), T > 1,
95     announce(N,black),!,
96     write(N),write(' says '),write(black),nl.
97
98 /* 視界に黒が2人いて，
99     以前のターンで自分が黒であることが分かった人がいて，
100    現在のターンが3回目以降ならば，
101    私は白である */
102 actionAgent(N) :- see2black(N),
103    factTurn(T), T > 2,
104    announcement(T2,N2,black), T2 < T, N2 =\= N,
105    announce(N,white),!,
106    write(N),write(' says '),write(white),nl.
107
108 /* 視界に黒が2人いて，
109    現在のターンが3回目以降ならば，私は黒である */
110 actionAgent(N) :- see2black(N),
111    factTurn(T), T > 2,
112    announce(N,black),!,
113    write(N),write(' says '),write(black),nl.
114
115 /* 視界に黒が3人いて，
116    以前のターンで自分が黒であることが分かった人がいて，
117    現在のターンが4回目以降ならば，私は白である */
118 actionAgent(N) :- see3black(N),
119    factTurn(T), T > 3,
120    announcement(T2,N2,black), T2 < T, N2 =\= N,
121    announce(N,white),!,
122    write(N),write(' says '),write(white),nl.
123
124 /* 自分の色がまだ分からない */
125 actionAgent(N) :-
126    write(N),write(' says nothing'),nl.
127
128 /*-----------------------------------------------*
129  |  エージェントたちの発表                       |
```

```
130  *-----------------------------------------------*/
131 actionAll :-
132     factTurn(T), write('Turn: '),write(T),nl,
133     actionAgent(0), actionAgent(1), actionAgent(2), actionAgent(3),
134     actionAgent(4), actionAgent(5), actionAgent(6), actionAgent(7),
135     nextTurn.
```

各エージェントの認識動作を述語 actionAgent として定義している．
このプログラムを処理系に読み込んだ後，actionAll 述語を引数無しで問い合わせると，各エージェントの発表が表示される．この述語を問い合わせる度に発表のターンが進む．

このプログラムを実行した結果の例を次に示す．

1 回目	2 回目	3 回目	4 回目
?- actionAll.	?- actionAll.	?- actionAll.	?- actionAll.
Turn: 1	Turn: 2	Turn: 3	Turn: 4
0 says nothing	0 says nothing	0 says black	0 says black
1 says nothing	1 says nothing	1 says nothing	1 says white
2 says nothing	2 says nothing	2 says nothing	2 says white
3 says nothing	3 says nothing	3 says black	3 says black
4 says nothing	4 says nothing	4 says nothing	4 says white
5 says nothing	5 says nothing	5 says nothing	5 says white
6 says nothing	6 says nothing	6 says black	6 says black
7 says nothing	7 says nothing	7 says nothing	7 says white
true.	true.	true.	true.

0,3,6 のエージェントの帽子が黒の場合の実行例

発展課題

1. エージェントの数が 9 以上のケースでも実行できるようにせよ．
2. 黒の帽子の数が 4 以上のケースでも実行できるようにせよ．

3.3 数式処理システム

数式を記号的に処理するシステムは「数式処理システム」（CAS）[26] と呼ばれる．CAS の最も基本的な機能に，記号的代数式の簡単化があるが，ここでは，多項式を簡単化する機能を Prolog で実装する例を挙げる．
単項式の和，多項式の和の機能を順番に構築する形で説明する．

3.3.1 単項式の和

同類項の単項式同士の和を記号的に求める方法について考える．2 つの単項式 n_1x と n_2x の和は $(n_1+n_2)x$ として求められる．ただしここでは実装例を簡単なものとするため n_1, n_2 とも整数 $\mathbb{Z}$ に限定する．

単項式の整数の係数を取り出す述語 coeff を次のように定義する．

```
coeff( N*X, N, X) :- integer(N),!.
coeff( X, 1, X ).
```

coeff の第 1 引数に単項式を与えると第 2 引数に係数が，第 3 引数にそれを取り除いた部分が単一化する．
これとは逆の働きをする述語 makeMono を次のように定義する．

```
makeMono( 1, X, X ) :- !.
makeMono( 0, _, 0 ) :- !.
makeMono( N, X, N*X ).
```

[26] 数式処理システム（CAS：Computer algebra system）は実際にいくつかの実装が市販されている．

makeMono の第 1 引数に係数，第 2 引数に変数記号（式も可）を与えると，第 3 引数にそれらの積が単一化する．

先の 2 つの述語を利用して，単項式の和を求める述語 addMono を次のように定義する．

```
addMono( N1, N2, X ) :- integer(N1),integer(N2),!,X is N1 + N2.
addMono( X1, X2, Y ) :- coeff(X1,N1,X),coeff(X2,N2,X),
                          N is N1 + N2, makeMono(N,X,Y).
```

この述語は第 1 引数と第 2 引数に与えた単項式の和を簡単化し，それを第 3 引数に単一化する．ただし与えた単項式が同類項でない場合はこの述語の問い合わせは失敗する．

2 つの単項式の和として多項式を合成する述語 makePoly を次のように定義する．

```
makePoly(F,0,F) :- !.
makePoly(0,X,X) :- !.
makePoly(X,Y,Z) :- addMono(X,Y,Z),!.
makePoly(F,X,F+X).
```

この述語は第 1 引数と第 2 引数に与えた単項式の和を簡単化し，それを第 3 引数に単一化する．与えた単項式が同類項でない場合は '+' で結合したものを第 3 引数に単一化する．

3.3.2 多項式の和

先に定義した単項式同士の和を求める述語を利用して，多項式同士の和を求める方法について考える．まず，多項式に単項式を加える述語 addPolyMono を次のように定義する．

```
addPolyMono( F+X1,X2, Y ) :- addMono(X1,X2,X),!,makePoly(F,X,Y).
addPolyMono( F+X1,X2, Y ) :- addPolyMono(F,X2,Y2),makePoly(Y2,X1,Y),!.
addPolyMono( F, X, Y ) :- addMono(F,X,Y),!.
addPolyMono( F, X, Y ) :- makePoly(F,X,Y).
```

この述語は第 1 引数に与えた多項式と第 2 引数に与えた単項式の和を簡単化し，それを第 3 引数に単一化する．
記述からわかるように，'+' で結合された多項式を分解して単項式同士の和を求める手法を再帰的に適用するという方法で同類項を見つけ出し，結果として得られる多項式を簡単化している．

述語 addPolyMono を利用して多項式同士の和を求める述語 addPoly を次のように定義する．

```
addPoly(F,F1+F2,Y) :- addPolyMono(F,F2,Y2),addPoly(Y2,F1,Y),!.
addPoly(F,X,Y) :- addPolyMono(F,X,Y).
```

この述語は第 1 引数と第 2 引数に与えた多項式の和を簡単化し，それを第 3 引数に単一化する．
記述からわかるように，'+' で結合された多項式を分解して，多項式と単項式の和を求める手法を再帰的に適用するという方法により，結果として得られる多項式を簡単化している．

3.3.3 多項式の簡単化

先に定義した述語群を用いて，与えられた多項式を分解しながら再構成する手法で式全体を簡単化する述語 fsimple を次のように定義する．

```
fsimple(F+X,Y) :- fsimple(F,F2),addPoly(F2,X,Y),!.
fsimple(F,F).
```

プログラム全体を Sample03.pl に示す．

プログラム：Sample03.pl

```
1  /* 整数係数の取り出し */
2  coeff( N*X, N, X) :- integer(N),!.
3  coeff( X, 1, X ).
4
5  /* 単項式の生成 */
6  makeMono( 1, X, X ) :- !.
7  makeMono( 0, _, 0 ) :- !.
8  makeMono( N, X, N*X ).
9
10 /* 単項式の和 */
11 addMono( N1, N2, X ) :- integer(N1),integer(N2),!,X is N1 + N2.
12 addMono( X1, X2, Y ) :- coeff(X1,N1,X),coeff(X2,N2,X),
13                         N is N1 + N2, makeMono(N,X,Y).
14
15 /* 多項式の生成 */
16 makePoly(F,0,F) :- !.
17 makePoly(0,X,X) :- !.
18 makePoly(X,Y,Z) :- addMono(X,Y,Z),!.
19 makePoly(F,X,F+X).
20
21 /* 多項式と単項式の和 */
22 addPolyMono( F+X1,X2, Y ) :- addMono(X1,X2,X),!,makePoly(F,X,Y).
23 addPolyMono( F+X1,X2, Y ) :- addPolyMono(F,X2,Y2),makePoly(Y2,X1,Y),!.
24 addPolyMono( F, X, Y ) :- addMono(F,X,Y),!.
25 addPolyMono( F, X, Y ) :- makePoly(F,X,Y).
26
27 /* 多項式同士の和 */
28 addPoly(F,F1+F2,Y) :- addPolyMono(F,F2,Y2),addPoly(Y2,F1,Y),!.
29 addPoly(F,X,Y) :- addPolyMono(F,X,Y).
30
31 /* 多項式の簡単化 */
32 fsimple(F+X,Y) :- fsimple(F,F2),addPoly(F2,X,Y),!.
33 fsimple(F,F).
```

実行例

先のプログラムを処理系に読み込んで実行した例を示す.

例 1. 整数の和

```
?- fsimple(1+2,A).
A = 3.
```

例 2. 記号同士の和（簡単化できる場合）

```
?- fsimple(x+x,A).
A = 2*x.
```

例 3. 簡単化できない場合

```
?- fsimple(x+y,A).
A = x+y.
```

例 4. 簡単化できる場合 (2)

```
?- fsimple(x+y+x,A).
A = 2*x+y.
```

例 5. 少し複雑な場合

```
?- fsimple(x+y+x+z+x+s+x+l+y,A).
A = 4*x+l+s+z+2*y.
```

例 6. 消滅する項が存在する場合

```
?- fsimple(-1*x+y+x+y+z,A).
A = 2*y+z.
```

発展課題

1. 単項式同士の積を簡単化するように改良せよ.
2. 式の簡単化に伴い項が整列されるように改良せよ.
3. 項や係数として有理数が扱えるように改良せよ.
4. '-' や'/' で連結された式が扱えるように改良せよ.

4　アプリケーション構築のための技法

ここでは，Prolog でアプリケーションプログラムを作成するために有用と思われる各種の技法を紹介する．

4.1　モジュール

Prolog における**モジュール**は，いわゆる**名前空間**のようなものであり，定義された述語名などを指定されたモジュールの範囲内で局所的なものにすることができる．例えば wa/3 という述語が次のような 2 つの別々のファイルに別々に定義されているとする．

ソースファイル 1：module00_1.pl

```
1 :- module( mdA, [] ).
2
3 wa( N1, N2, Ans )   :- var( N1 ), N1 is Ans - N2,!.
4 wa( N1, N2, Ans )   :- var( N2 ), N2 is Ans - N1,!.
5 wa( N1, N2, Ans )   :- var( Ans ), Ans is N1 + N2.
```

ソースファイル 2：module00_2.pl

```
1 :- module( mdB, [] ).
2
3 wa( N1, N2, Ans )   :-
4     write(N1),write('+'),write(N2),write('='),write(Ans).
```

最初のファイルに定義されている wa/3 は加算を表現するものであり，2 つ目のファイルに定義されている wa/3 は '+' と '=' で加算の式を表現する．それぞれのファイルの冒頭に module の評価が記述されているが，これは，当該ファイルの中で定義した述語を指定したモジュール名で局所化するものである．これにより，同一の述語名が異なるモジュール内で定義されていても，その述語はモジュール毎に別のものとして扱われる．

● module 述語の書き方

:- module(**モジュール名**, **外部に公開する述語のリスト**)

module 述語の 2 番目の引数には，モジュール外に公開する述語のリストを記述する．この際，公開する述語の名前の後ろに '/**引数の数**' を付けること．

上の module00_1.pl ではモジュール名を mdA，module00_2.pl ではモジュール名を mdB としており，それぞれのモジュールにおいて wa/3 は別々のものとして扱われる．(次の実行例を参照のこと)

例. モジュール毎に異なる述語の扱い（SWI-Prolog の場合）

```
1 ?- consult('module00_1.pl'),consult('module00_2.pl').   Enter    ←ソースファイルの読込み
true.          ←読込み成功

2 ?- mdA : wa(1,2,A).   Enter    ←評価の先頭に「モジュール名：」を付ける
A = 3.          ←モジュール mdA のものとして評価されている．

3 ?- mdB : wa(1,2,3).   Enter    ←評価の先頭に「モジュール名：」を付ける
1+2=3          ←モジュール mdB のものとして評価されている．
true.          ←評価終了（成功）
```

この例のように，式の問い合わせ（評価）のときに，先頭に「モジュール名：」を付けることによって対象のモジュールを指定できる．

Prolog の評価サイクルそのものを特定のモジュールに設定することもできる．(次の例参照)

例. 評価サイクルのモジュールを変更する（先の例の続き）

```
4 ?- module(mdA).  Enter      ←モジュール mdA に切り替える.
true.              ←切り替え成功

mdA: 5 ?- wa(1,2,A).  Enter   ←プロンプトの前にモジュール名が表示される
A = 3.             ←モジュール mdA のものとして評価されている.

mdA: 6 ?- module(mdB).  Enter ←モジュール mdB に切り替える.
true.              ←切り替え成功

mdB: 7 ?- wa(1,2,3).  Enter   ←モジュール mdB として評価
1+2=3              ←モジュール mdB のものとして評価されている.
true.              ←評価終了（成功）
```

この例のように，module 述語に引数としてモジュール名（1 つ）を与えて評価することで，Prolog の評価サイクルをそのモジュールに設定することができる．システムの初期の評価サイクルのモジュール名は user である．(次の例参照)

例. 評価サイクルのモジュールを初期のものにする（先の例の続き）

```
mdB: 8 ?- module( user ).  Enter
true.              ←評価終了（成功）
9 ?-               ←評価サイクルが初期のモジュールに戻った
```

4.2 ソースファイルの分割

同一の述語の定義を複数のファイルに分けて記述し，それぞれのファイルを個別に consult で読み込む場合はその旨を宣言する必要がある．これには組み込み述語 multifile を用いる．例えば，3 個の引数を取る述語 pred1 の定義を複数のファイルに記述する場合，それぞれのファイルの冒頭に

```
:- multifile(pred1/3).
```

と記述すること．この宣言をしないと，Prolog 処理系に最後に読み込まれたファイルに記述された定義のみが有効となる．(定義が上書きされる)

4.2.1 include によるソースファイルの読込み

ソースファイルの中で include を用いると，別のソースファイルを読み込んで，当該ソースファイルの内容として加えることができる．例えば次に示すような 2 つのソースファイル module01_1.pl, module01_2.pl を別々に作っておき，それを include で読み込むことについて考える．

ソースファイル 1：module01_1.pl

```
1 wa( N1, N2, Ans )   :- var( N1 ), N1 is Ans - N2,!.
2 wa( N1, N2, Ans )   :- var( N2 ), N2 is Ans - N1,!.
3 wa( N1, N2, Ans )   :- var( Ans ), Ans is N1 + N2.
```

ソースファイル 2：module01_2.pl

```
1 seki( N1, N2, Ans ) :- var( N1 ), N1 is Ans / N2,!.
2 seki( N1, N2, Ans ) :- var( N2 ), N2 is Ans / N1,!.
3 seki( N1, N2, Ans ) :- var( Ans ), Ans is N1 * N2.
```

これらのファイルに記述されている述語は，加算のための述語 wa と乗算のための述語 seki であるが，2 つのファイルを include で読み込み 1 つのソースファイルと見做して統合する処理を module01.pl に示す．

ソースファイル 3：module01.pl

```
1 :- encoding( utf8 ).
2 :- module( md1, [wa/3,seki/3] ).
3
4 %--- ファイル群の読み込み ---
5 :- include( 'module01_1.pl' ).
6 :- include( 'module01_2.pl' ).
```

実際に操作としては，Prolog 処理系で module01.pl を consult 述語で読み込む.（次の例参照）

例. SWI-Prolog 処理系での実行例

```
1 ?- consult('module01.pl').   Enter     ←ソースファイルの読込み
true.          ←読込み成功

2 ?- wa(1,2,A).   Enter     ←加算の処理
A = 3.          ←計算結果が変数 A に単一化されている

3 ?- seki(2,3,A).   Enter     ←乗算の処理
A = 6.          ←計算結果が変数 A に単一化されている
```

この例は，プログラム全体を複数のファイルに分割して開発し，更に述語名をモジュールで局所化している例である．ただし wa と seki はモジュール外でも使用できるように公開する形（2 行目）にしている．

実用的なアプリケーションプログラムを開発する場合は，module 述語によるモジュール化，ソースファイルの分割と include，そして multifile 述語などを活用すると良い．

4.3 引数の双方向性の実現

Prolog においてプログラムの呼び出しは，述語と一連の引数を携えた原子論理式を問い合わせる形を取る．また，値を求めるようなプログラムを実行する場合は，問い合わせる式の中に変数を記述して，求める値を単一化によって取得するという形を取る．

この際に注意しなければならないこととして，

「特定の引数にのみ変数を与えることが許されているのが一般的である」

ということがある．典型的な例として is 述語が挙げられる．

is 述語は「is(A,1+2)」あるいは「A is 1+2」という形の問い合わせで数値演算の結果を求めるものであるが，この述語は第 1 引数にのみ変数を与えることが許されている．すなわち，

```
?- is(A,1+2).
```

という問い合わせをすることは可能であるが，

```
?- is(3,A+2).
```

は許されておらず，強いて実行しようとすると処理系はエラーを発生する．

結論

問い合わせ時にどの引数に変数を与えることができるかは，プログラマの意図によって定められた仕様である．

ここでは，「A は B と C の合計である」という言明を実現する述語 isSum の実装を例に挙げて，任意の引数に変数を与えることができる述語を実現する方法について説明する．

次のような仕様を考える．

仕様　isSum(A,B,C) ≡ A=B+C

どの引数が変数であるかを判定して場合に分けながら，計算処理を選択的に実行する．実装の考え方は次の通りである．

考え方

1. A（第 1 引数）に変数が与えられた場合は，`A is B+C` を問い合わせる．
2. B（第 2 引数）に変数が与えられた場合は，`B is A-C` を問い合わせる．
3. C（第 3 引数）に変数が与えられた場合は，`C is A-B` を問い合わせる．

引数に変数が与えられたかどうかは述語 var と nonvar を用いて判定する．実装例を次に示す．

実装例

```
isSum(A,B,C) :- var(A),nonvar(B),nonvar(C), A is B+C.
isSum(A,B,C) :- nonvar(A),var(B),nonvar(C), B is A-C.
isSum(A,B,C) :- nonvar(A),nonvar(B),var(C), C is A-B.
```

これを実行した例を次に示す．

実行例

```
?- isSum(A,1,2).   Enter     ←第 1 引数が変数
A = 3 .                      ←計算結果
?- isSum(3,B,2).   Enter     ←第 2 引数が変数
B = 1 .                      ←計算結果
?- isSum(3,1,C).   Enter     ←第 3 引数が変数
C = 2 .                      ←計算結果
```

任意の引数に変数が使えることがわかる．

4.4　書き換え可能な記憶

Prolog における変数は，他の言語の変数と比べて基本的な性質が異なる．C や Java といった言語では（あるいは Lisp においても），同一のスコープ内において変数には自由に値を格納することができ，最後に更新した内容が変数に記録されている．これに対して Prolog では，1 回の問い合わせの範囲内において 1 つの変数に束縛される値は 1 つのみであり，1 度割り当てられた変数の値が変更されることがない．なぜならば，1 度値が束縛された変数に対して値の変更が要求されるということは，「推論における矛盾の発生」を意味するからである．

論理プログラミングにおいて，値の変更を伴う状況の変化[27] は**非単調推論**として扱われ，推論処理の過程で値の変更が求められる場合もある．

値の変化を Prolog で実現するには，変数への値の束縛ではなく，事実の登録と変更 を用いるのが良い．

事実の登録と変更の処理によって値の登録と更新を実現するプログラムの例を Sample04.pl に示す．

プログラム：Sample04.pl

```
1 /* 値の更新 */
2 updateVal( Key, Val ) :- '#V'(Key,Val),!.
3 updateVal( Key, Val ) :- '#V'(Key,ValOld),
4     retract( '#V'(Key,ValOld) ), assertz( '#V'(Key,Val) ),!.
5 updateVal( Key, Val ) :- assertz( '#V'(Key,Val) ),updateKeys(Key).
6
7 /* 値の参照 */
```

[27]値の変更を伴う状況の変更を取り扱う推論として，例えば**状況演算**（Situation Calculus）などがある．(McCarthy,J.,et al.,“Actions and other events in situation calculus”, Proceedings of KR-2002, pp.615-628, 2002)

```
getVal( Key, Val ) :- '#V'(Key,Val).

/* キーリストの取得 */
getKeys( L ) :- '#K'(L),!.
getKeys( [] ).

/* キーリストの保存 */
setKeys( L ) :- '#K'(L),!.
setKeys( L ) :- '#K'(Lold),retract( '#K'(Lold) ),assertz( '#K'(L) ),!.
setKeys( L ) :- assertz( '#K'(L) ),!.

/* キーリストの更新 */
updateKeys( Key ) :- getKeys(L), isMember(Key,L),!.
updateKeys( Key ) :- getKeys(L), setKeys([Key|L]).

/* キーと値の一覧表示 */
showVals :- getKeys(L), write('*** Key/Value ***'),nl, showValsL(L).

showValsL( [K|L] ) :- showValsL1(K), showValsL(L).
showValsL( [] ).

showValsL1( Key ) :- '#V'(Key,Val),
    write(Key),write(' : '),write(Val),nl.

/* メンバーシップ検査 */
isMember( M, [M|_] ) :- !.
isMember( M, [_|L] ) :- isMember(M,L).
```

解説

このプログラムでは，**キー**と**値**の組を事実 '#V'(キー, 値) として保存する．キーに対して値を設定するには，述語 updateVal を問い合わせる．また，キーに設定された値を参照するには，述語 getVal を問い合わせる．

例. キー「x」に対する値の設定と参照

```
?- updateVal(x,2050).   Enter      ← x に 2050 を設定している
true.

?- ?- getVal(x,V).   Enter
V = 2050.            ← x に設定されている値を参照している
```

キーと値の関係を一覧表示するには述語 showVals を引数無しで問い合わせる．

例. キーと値の一覧表示

```
?- showVals.   Enter
*** Key/Value ***
x :  2050
c :  [d,e,f]
b :  31
a :  s+2
true.
```

この述語を実現するために，キーのリストを管理する機能を実装している．updateVal を問い合わせる度に，新出のキーを '#K'(L) の L に追加する方法を採用している．'#K'(L) を参照することで，**失敗の発生による処理の繰り返し**（「2.7.5 繰り返し動作の実現」を参照）に頼ることなく全てのキーに順番にアクセスすることが可能となる．

4.5 変数への単一化を遅延する方法

Prolog で扱う式をデータとして取り扱う場合，その式に含まれる変数の扱いには注意する必要がある．プログラマの意図しない単一化によって式に含まれる変数に値が束縛されてしまうと，式のデータとしての意味が変わってしまう．変数を含んだ式をデータとして扱う場合，式の中の変数記号を一旦別のもの（変数以外の式）に変更して単一化の対象から外しておき，後で変数記号に戻すという手法が必要になることがある．

ここでは，原子論理式の引数として指定された変数を別のデータ（変数以外の式）に変換する方法と，変換された部分を変数に戻す方法について説明する．

基本的な方法

変数記号を変数でない式に置き換える方法を採用する．例えば '$var'(番号) という式を生成し,「番号」の部分は変数記号を識別するための整数値とする．このような式を生成して，それを変換対象となる変数に単一化すれば，番号によって識別されるオブジェクトに変換することができる．これを実現するために，次に示すような述語 freezeVar を考える．

```
freezeVar( N1, T, N2 ) :- var(T),!, T = '$var'(N1), N2 is N1 + 1.
freezeVar( N, _, N).
```

freezeVar の第 1 引数には，変数記号の通し番号を意味する整数値を与え，第 2 引数には変換対象の記号（式）を与える．第 2 引数が変数記号の場合は，それに '$var'(番号) を単一化させ，第 1 引数の値に 1 を加えた値を第 3 引数に返す．第 2 引数が変数記号でない場合は，第 1 引数の値と同じ値を第 3 引数に返す．

原子論理式の引数に対して順番に freezeVar の処理を適用して，変数記号を '$var'(番号) に置き換えてゆく．

原子論理式の引数にある '$var'(番号) を変数記号に変換するには，同一の「番号」に同一の変数記号を対応させる必要がある．そのために，次のような述語 lookupTable を定義する．

```
lookupTable(Key,Val,[[Key,Val]|L],[[Key,Val]|L]) :- !.
lookupTable(Key,Val,[F|L],[F|L2]) :- lookupTable(Key,Val,L,L2),!.
lookupTable(Key,Val,[],[[Key,Val]]).
```

これは，“[キー, 値]”を要素とするリストを保持するものである．すなわち，第 3 引数に与えたリストの中から Key に対する値を検索して Val に単一化するものである．第 3 引数に与えたリストに Key を含む要素がなければ “[Key,Val]” を要素として付け加え，追加されたリストを第 4 引数に返す．この述語を利用して '$var'(番号) と変数記号の対応を管理しながら '$var'(番号) を変数記号に変換する述語 meltVar を次に示す．

```
meltVar('$var'(N),Vm,L1,L2) :- lookupTable('$var'(N),Vm,L1,L2),!.
meltVar(V,V,L,L).
```

meltVar の第 1 引数に '$var'(番号) が与えられれば，それを変数記号に置き換えたものを第 2 引数に返す．このとき，第 3 引数は '$var'(番号) と変数記号の対応を記録するリストを与え，更新されたリストが第 4 リストに得られる．

この述語を順次適用する形で，原子論理式の引数を変数記号に置き換えることができる．

以上の述語を用いて，変数記号を '$var'(番号) に変換する述語 freezeForm と，'$var'(番号) を変数記号に戻す述語 meltForm を定義する．その実装例を Sample05.pl に示す．

プログラム：Sample05.pl

```
1 /* 変数の固定 */
2 freezeForm( T ) :- var(T),!,T='$var'(1).
3 freezeForm( T ) :-
4     functor(T,_,M),
5     freezeFormSub(1,M,T,1).
```

```
6
7  freezeFormSub(M,M,F,N) :- arg(M,F,T), freezeVar(N,T,_),!.
8  freezeFormSub(M1,M2,F,N) :- arg(M1,F,T), freezeVar(N,T,N2),
9                              M3 is M1 + 1, freezeFormSub(M3,M2,F,N2).
10
11 freezeVar( N1, T, N2 ) :- var(T),!, T = '$var'(N1), N2 is N1 + 1.
12 freezeVar( N, _, N).
13
14 /* キーと値の保持 */
15 lookupTable(Key,Val,[[Key,Val]|L],[[Key,Val]|L]) :- !.
16 lookupTable(Key,Val,[F|L],[F|L2]) :- lookupTable(Key,Val,L,L2),!.
17 lookupTable(Key,Val,[],[[Key,Val]]).
18
19 /* 固定された変数の解凍 */
20 meltForm( '$var'(_), V ) :- var(V),!.
21 meltForm( Tf, Tm ) :-
22     functor(Tf,F,M), functor(Tm,F,M),
23     meltFormSub(1,M,Tf,Tm,[]).
24
25 meltFormSub(M,M,Tf,Tm,L) :- arg(M,Tf,Af),arg(M,Tm,Am), meltVar(Af,Am,L,_),!.
26 meltFormSub(M1,M2,Tf,Tm,L) :- arg(M1,Tf,Af),arg(M1,Tm,Am), meltVar(Af,Am,L,L2),
27     M3 is M1 + 1, meltFormSub(M3,M2,Tf,Tm,L2).
28
29 meltVar('$var'(N),Vm,L1,L2) :- lookupTable('$var'(N),Vm,L1,L2),!.
30 meltVar(V,V,L,L).
```

freezeForm と meltForm の実行例を次に示す.

実行例

```
?- F = f(V,w,X,y,z,V,W,X), freezeForm( F ), meltForm( F, F2 ).   Enter
F = f('$var'(1), w, '$var'(2), y, z, '$var'(1), '$var'(3), '$var'(2)),
V = '$var'(1),
X = '$var'(2),
W = '$var'(3),
F2 = f(_G1769, w, _G1771, y, z, _G1769, _G1775, _G1771).
```

F に与えた式の引数にある変数記号が '$var'(番号) に変換され，さらにそれらを変数記号に戻したものが F2 に得られている．当初 F に与えた変数記号と F2 にある変数記号は別のものであるが，記号の対応は同じである．

発展課題

式の入れ子がある場合にも，同様の機能を実現する方法を考えて実装せよ.

4.6 差分リスト

Prolog で高速なリスト処理を実現するには**差分リスト**を利用するとよい．通常のリストは

```
[a,b,c]
```

のような構造であるが，このままの構造でリストの連結処理を実装するには，例えば次のような実装となる.

```
appendL( [F|L1], L2, [F|L3] ) :- appendL(L1,L2,L3),!.
appendL( [], L, L ).
```

この述語 appendL は，第 1 引数と第 2 引数に与えられたリストを連結したものを第 3 引数に返す.
この実装の問題点に，第 1 引数に与えたリストの長さに比例する処理時間がかかるということがある.

Prolog は変数への単一化の処理が非常に高速であるため，その性質を利用することでリストの処理が高速化できることがある．例えばリストのデータを扱うために次のような式を使うことを考える.

```
difL([a,b,c|D],D)
```

このようなデータ構造を採用すると，リストの末尾に対応する部分が difL の第 2 引数としてリストの外部から容易にアクセスできる．このデータ構造を利用してリストの連結処理を記述すると

```
appendL( difL(L1,D1), difL(L2,D2), difL(L1,D2) ) :- D1 = L2.
```

となる．先の定義に比べて記述が短くなるだけでなく，実行にかかる時間もリストの長さに関係なく一定である．
この定義によるリストの連結処理の実行例を示す．

```
?- L1 = difL([a,b,c|D1],D1), L2 = difL([d,e,f|D2],D2), appendL(L1,L2,L3).   Enter
L1 = difL([a, b, c, d, e, f|D2], [d, e, f|D2]),
D1 = [d, e, f|D2],
L2 = difL([d, e, f|D2], D2),
L3 = difL([a, b, c, d, e, f|D2], D2).
```

変数 L1, L2 に与えたリストを連結したものが，変数 L3 に得られていることがわかる．[28]

[28] ここでは計算量の改善の例としてリストの連結を取り上げたが，同じ処理を行う組込み述語 append が存在する．

5　その他の組込み述語

ISO Prolog で予め提供されている組込みの述語のうちこれまでに紹介していないものもあり，有用と思われるものをここに挙げる．

5.1　入出力

現在の入力元や出力先を確認するための述語がある．

■ 現在の入力元の確認　seeing

see 述語で指定した入力元を確認する．

実行例

```
?- seeing(A).   Enter
A = user.
```

user は利用者が使用している端末装置で，多くの処理系の初期設定である．

■ 現在の出力先の確認　telling

tell 述語で指定した出力先を確認する．

実行例

```
?- telling(A).   Enter
A = user.
```

5.1.1　作業ディレクトリ

ソースプログラムの読込みやファイル入出力のための**作業ディレクトリ**を変更する，あるいは調べる方法について説明する．

【SWI-Prolog の場合】

組み込み述語 working_directory を用いて作業ディレクトリを変更する．この述語は 2 つの引数を取り，第一引数には変数を与え，第二引数に変更先のディレクトリのパスを文字列型で与える．第一引数には変更前の作業ディレクトリのパスが単一化する．

実行例

```
?- working_directory((A,'c:/Users/katsu').   Enter     ← c:/Users/katsu に移動
A = 'c:/program files/swipl/bin/'.     ←変更前のパスが変数 A に得られている．
```

この述語の両方の引数に同一の変数記号を与えると，現在の作業ディレクトリのパスを調べることができる．

実行例

```
?- working_directory(A,A).   Enter     ←両方の引数に同じ変数記号を与える．
A = 'c:/users/katsu/'.     ←現在の作業ディレクトリのパスが得られる．
```

【GNU Prolog の場合】

組み込み述語 change_directory を用いて作業ディレクトリを変更する．この述語は 1 つの引数を取り，変更先のディレクトリのパスを文字列型で与える．

実行例

```
?- change_directory('C:/Users/katsu').   Enter     ← C:/Users/katsu に移動
yes     ←処理が正常に終了した．
```

現在の作業ディレクトリを調べるには組み込み述語 working_directory を用いる．この述語は 1 つの引数を取り，ここに変数を与えることで作業ディレクトリのパスが単一化する．

実行例

```
?- working_directory(A).  Enter      ←引数に変数記号を与える．
A = 'C:\\Users\\katsu'      ←現在の作業ディレクトリのパスが得られる．
yes      ←処理が正常に終了した．
```

5.2 リスト処理

■ 要素の整列　sort

リストの要素の順序を整列する．

実行例

```
?- L=[d,z,c,y,b,x,a],sort(L,S).  Enter
L = [d, z, c, y, b, x, a],
S = [a, b, c, d, x, y, z].
```

5.3 検査

■ 式の記号的順序の判定

2 つの式 X，Y の記号的な順序[29] を判定する．

1. `X @< Y`　　X より Y が後である．
2. `X @> Y`　　X より Y が前である．
3. `X @=< Y`　　X より Y が後である．(同一も可)
4. `X @>= Y`　　X より Y が前である．(同一も可)

実行例

```
?- a @< b.  Enter
true.

?- f(a) @< f(b).  Enter
true.

?- f(a) @> f(b).  Enter
false.
```

5.4 シンボルに対する処理

■ アトムの長さ　atom_length

アトムの長さ（文字数）を取得する．

実行例

```
?- atom_length(abcde,A).  Enter
A = 5.
```

abcde の長さが 5 であることがわかる．

[29]式を文字列として見た場合の「文字コードに基づく順序」である．詳しくは処理系の仕様を参照のこと．

■ **アトムの連結**　atom_concat

2つのアトムを連結したものを取得する．

実行例

```
?- atom_concat( abc, def, A ).   Enter
A = abcdef.
```

abc と def を連結して abcdef を得ている．

■ **アトムの分解**　atom_chars

アトムを文字に分解したものを取得する．

実行例

```
?- atom_chars( abc, A ).   Enter
A = [a, b, c].
```

abc を分解して [a, b, c] を得ている．これと同様の処理を数値に対して行う number_chars もある．number_chars で得られたリストの要素は各桁を表す数値ではなく「文字」である．

1文字のシンボルの文字コードを得るための述語として char_code がある．

実行例

```
?- char_code( a, A ).   Enter
A = 97.
```

5.5　式の評価に関する処理

■ **明示的な問い合わせ**　call

式の評価を明示的に実行する．

実行例

```
?- call( A is 2+3 ).   Enter
A = 5.
```

■ **条件判定後の実行**　–>

条件部が真である場合に実行部を評価する機能で **条件部 –> 実行部** の形で問い合わせる．

実行例 条件部が真の場合

```
?- A is 4/2, B is 3+1 -> C is A+B.   Enter
A = 2,
B = 4,
C = 6.
```

実行例 条件部が偽の場合

```
?- A is 4/2, B is 3+1, fail -> C is A+B.   Enter
false.
```

■ **問い合わせ（検証）の中断**　abort

式の問い合わせ処理（検証）を実行中に中断するための述語である．例えば次のような商を求める述語を考える．

```
isQuotient( _, _, 0 ) :- write('divisor:0 !'),nl,!, abort.
isQuotient( Q, N, D ) :- Q is N / D.
```

この述語は第3引数にゼロが与えられた時に，商を求めることができないことから実行を中断するように実装されて

いる．

実行例

```
?- isQuotient( Q, 4, 2 ).   Enter
Q = 2.      ←商が得られる

?- isQuotient( Q, 4, 0 ).   Enter
divisor:0 !
%Execution Aborted   ←実行中断のメッセージ（SWI Prolog の場合の例）
```

■ 単一化の解の集合　bagof

指定した変数に単一化する値を全てリストにして取得することができる．例えば次のようなプログラムを考える

プログラム：test11.pl

```
1 human_gender(taro,male).
2 human_gender(jiro,male).
3 human_gender(hanako,female).
4 human_gender(yosiko,female).
```

このプログラムを処理系に読み込んで実行したすると例えば次のように単一化する．

```
?- human_gender(A,B).   Enter
A = taro,
B = male ;
A = jiro,
B = male ;
A = hanako,
B = female ;
A = yosiko,
B = female.
```

次に bagof を用いて単一化の解をまとめて取得する例を示す．

```
?- bagof(A,human_gender(A,X),L).   Enter
X = female,
L = [hanako, yosiko] ;
X = male,
L = [taro, jiro].
```

このように，bagof の第 1 引数に与えた変数に単一化する解をリストにしたものが第 3 引数に単一化する，

参考）　結果のリストを整列した形で得る述語 setof もある．

5.6　その他の処理

■ 式の複製　copy_term

式の複製を作る．このとき，式に含まれる変数は異なる変数記号として複製される．

実行例

```
?- copy_term( f(v,W,x,W,Y,y,Y,z), F ).   Enter
F = f(v, _G2879, x, _G2879, _G2882, y, _G2882, z).
```

式そのものをデータとして取り扱う際，その式に含まれている変数記号に余計な単一化が起こらないようにする必要がある．そのような場合にはこの述語を用いて式の完全な複製を取るのが良い．

■ **デバッグ**　trace

処理系を**トレースモード**にすることで，問い合わせを検証する過程を逐一表示させることができる．述語 trace を引数なしで問い合わせることで処理系がトレースモードに入る．

ここでは,「2.7.3 カット」で取り上げたサンプルプログラム test06-2.pl（階乗計算）をトレースモードで実行する例を挙げる

実行例　（SWI-Prolog の場合の例）

```
1 ?- trace.        ←トレースモードに入る操作
true.

[trace] 1 ?- factorial(3,A).
    Call:  (7) factorial(3, _G1454) ?  creep
    Call:  (8) 3>0 ?  creep
    Exit:  (8) 3>0 ?  creep
    Call:  (8) _G1532 is 3+ -1 ?  creep
    Exit:  (8) 2 is 3+ -1 ?  creep
    Call:  (8) factorial(2, _G1533) ?  creep
    Call:  (9) 2>0 ?  creep
    Exit:  (9) 2>0 ?  creep
    Call:  (9) _G1535 is 2+ -1 ?  creep
    Exit:  (9) 1 is 2+ -1 ?  creep
    Call:  (9) factorial(1, _G1536) ?  creep
    Call:  (10) 1>0 ?  creep
    Exit:  (10) 1>0 ?  creep
    Call:  (10) _G1538 is 1+ -1 ?  creep
    Exit:  (10) 0 is 1+ -1 ?  creep
    Call:  (10) factorial(0, _G1539) ?  creep
    Call:  (11) 0>0 ?  creep
    Fail:  (11) 0>0 ?  creep
    Redo:  (10) factorial(0, _G1539) ?  creep
    Exit:  (10) factorial(0, 1) ?  creep
    Call:  (10) _G1541 is 1*1 ?  creep
    Exit:  (10) 1 is 1*1 ?  creep
    Exit:  (9) factorial(1, 1) ?  creep
    Call:  (9) _G1544 is 1*2 ?  creep
    Exit:  (9) 2 is 1*2 ?  creep
    Exit:  (8) factorial(2, 2) ?  creep
    Call:  (8) _G1454 is 2*3 ?  creep
    Exit:  (8) 6 is 2*3 ?  creep
    Exit:  (7) factorial(3, 6) ?  creep
A = 6.

[trace] 2 ?- notrace. ←トレースモードを解除する操作 (1)
true.

[debug] 3 ?- nodebug. ←トレースモードを解除する操作 (2)
true.
```

SWI-Prolog 処理系では，notrace,nodebug によりトレースモードを解除する．（文献 [5] 参照）

参考）　特定の述語を呼び出したときのみ処理過程を表示するための述語 spy もある．

参考） ISO Prolog 準拠の機能ではないが，乱数を発生する述語 random が使える処理系（SWI-Prolog, GNU Prolog など）がある．

実行例

```
?- random(A).   Enter
A = 0.06889619586328002.
```

5.7 is 述語で使える演算子・関数・定数

is 述語で使用できる演算子，関数，定数として，先に説明したもの以外にも有用なものがいくつかあり，ここに紹介する．

表 2: is 述語で使える演算子・関数・定数

演算子・関数・定数	説明	対応処理系
abs(x)	x の絶対値（$\|x\|$）	SWI/GNU/ISO
x ** n	x^n	SWI/GNU/ISO
sqrt(x)	$\sqrt{x}$	SWI/GNU/ISO
exp(x)	e^x	SWI/GNU/ISO
e	自然対数の底（ネイピア数：e）	SWI/GNU
pi	円周率（π）	SWI/GNU/ISO
sin(x), cos(x), tan(x)	x の正弦関数，余弦関数，正接関数	SWI/GNU/ISO
asin(x), acos(x), atan(x)	x の逆正弦関数，逆余弦関数，逆正接関数	SWI/GNU/ISO
sinh(x), cosh(x), tanh(x)	x の双曲線正弦関数，双曲線余弦関数，双曲線正接関数	SWI/GNU
asinh(x), acosh(x), atanh(x)	x の逆双曲線正弦関数，逆双曲線余弦関数，逆双曲線正接関数	SWI/GNU
log(x)	x の自然対数（$\log_e x$）	SWI/GNU/ISO
log10(x)	10 を底とする x の対数（$\log_{10} x$）	SWI/GNU
round(x)	浮動小数点数 x を丸めた（整数にした）値	SWI/GNU/ISO
float(x)	数 x を浮動小数点数にした値	SWI/GNU/ISO

使用例

```
?- X is cos(pi/3),Y is sin(pi/3),R is sqrt(X**2+Y**2).   Enter
X = 0.5000000000000001,
Y = 0.8660254037844386,
R = 1.0.
```

6　Prolog 処理系

Prolog 処理系としては多くの実装があるが，ここでは入手と利用が容易な ISO 準拠の 2 つの実装を紹介する．

6.1　SWI-Prolog

SWI-Prolog はオランダのアムステルダム大学 [30] で開発と保守が行われている処理系であり，各種の OS（Microsoft Windows, Apple MacOS X, Linux など）の上で動作するフリーソフトウェア（LGPL ライセンス）である．同処理系はインターネットサイト http://www.swi-prolog.org/ で配布されている．

図 8: SWI-Prolog

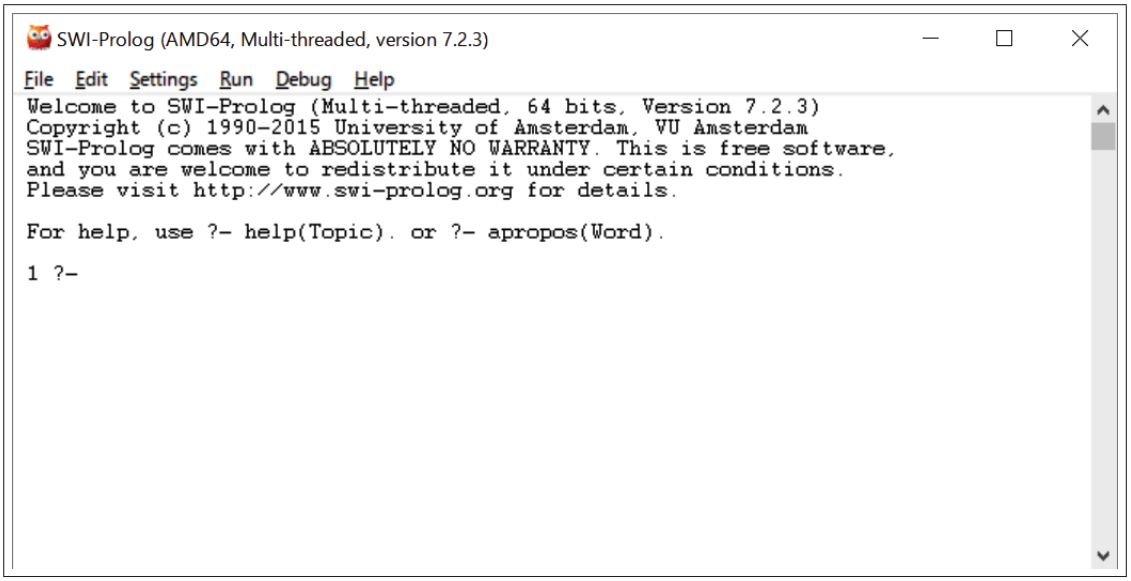


図 9: MS-Windows 版 SWI-Prolog を起動したところ

この処理系はマルチスレッドの問い合わせが実行できる．また整数，有理数の演算機能として GMP [31] を採用しており，多倍長精度の計算ができる．ただし，浮動小数点数の扱いに関しては，処理系を実装している C 言語の double の精度による．

「2.7.3 カット」で取り上げたサンプルプログラム test06-2.pl（階乗計算）を多倍長整数で演算する様子を次に示す．

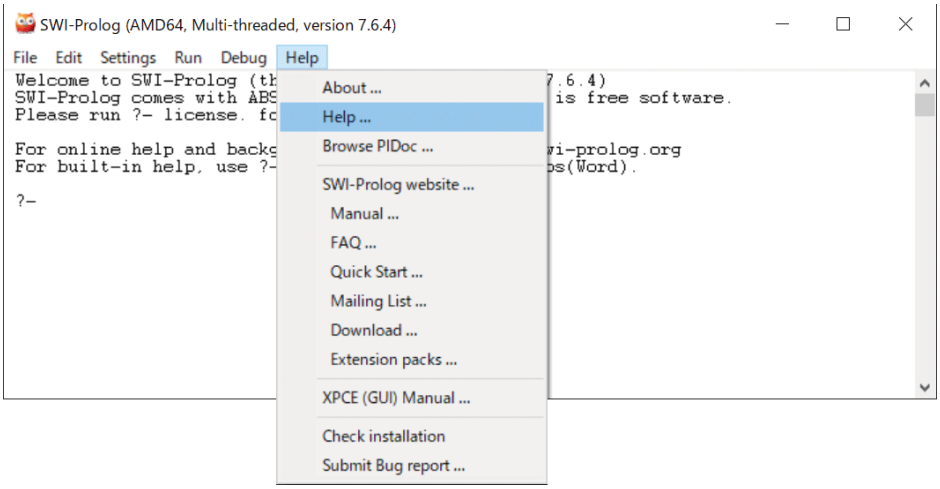
```
?- factorial(300,A).   Enter      ← 300! の計算

A = 306057512216440636035370461297268629388588804173576999416776741259476533176716867465
   515291422477573349939147888701726368864263907759003154226842927906974559841225476930 27
   195460400801221577625217685425596535690350678872526432189626429936520457644883038890 97
   539434896254360532259807765212708224376394491201286786753683057122936819436499564604 98
   166450227716500185176546469340112226034729724066333258583506870150169794168850353752 13
   755491028912640715715483028228493795263658014523523315693648223343679925459409527682 06
   080622328123873838808170496000000000000000000000000000000000000000000000000000000000 00
   000000000000000.
```

[30] **University of Amsterdam**：Kruislaan 419, 1098 VA Amsterdam The Netherlands

[31] GMP（GNU Multi-Precision Library）多倍長精度の整数と有理数の計算を実行するためのライブラリであり，LGPL（GNU Lesser General Public License）でライセンスされている．

SWI-Prolog には充実したヘルプ機能が備わっている．（図 10）



「Help」メニューから「Help」を選択　↓

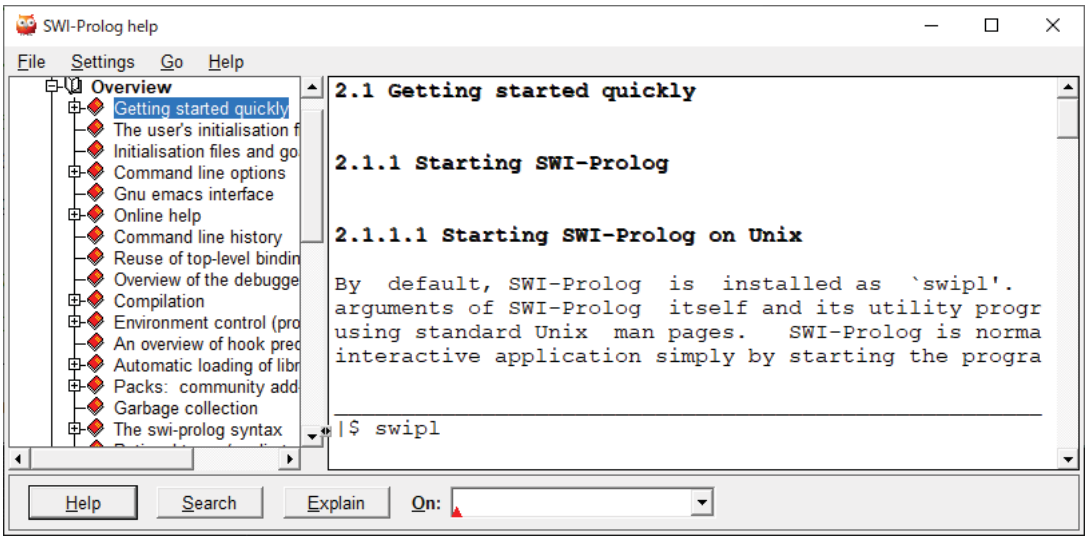


図 10: SWI-Prolog のヘルプ機能

SWI-Prolog には独自のグラフィックス作成機能（XPCE）が備わっている．同処理系に関する詳細は文献 [5] を参照のこと．

6.1.1　処理系の起動

図 8 のアイコンをダブルクリックすることで図 9 のような対話ウィンドウが開き，SWI-Prolog が利用できる．SWI-Prolog は端末ウィンドウからコマンドとして起動することもできる．

例. Windows 環境での SWI-Prolog の起動

```
C:￥Users￥katsu ＞ swipl  Enter          ←コマンド swipl を投入
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)     ←システム起動時のメッセージ
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license.  for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic).  or ?- apropos(Word).

1 ?-          ← Prolog の問い合わせプロンプト
```

この後，端末ウィンドウでのセションとなる．SWI-Prolog は swipl コマンドとして実行することができる．(swipl コマンドへのコマンドサーチパスの設定が必要)

6.1.2　処理系の初期設定

SWI-Prolog 処理系の起動時に実行する内容を 'swipl.rc' という名前のファイルに記述しておくことができる．このファイルは処理系を収めるディレクトリに保存しておく．

例. 処理系の入出力のエンコーディングを UTF-8 にする記述

```
:- set_prolog_flag(encoding,utf8).
```

Windows 環境では，ファイル 'swipl.rc' を

```
C:¥Program Files¥swipl
```

のディレクトリに保存しておく．

6.1.3　有理数の扱い

SWI-Prolog では is/2 による計算処理において**有理数**を扱うことができる．有理数は rdiv を用いて次のように記述する．

書き方：　N rdiv D　もしくは　**rdiv(N,D)**　　**N**：分子，**D**：分母

例. 有理数の扱い

```
?- X is 1 rdiv 2.   Enter        ← 1/2 の計算（その 1）
X = 1 rdiv 2.       ←計算結果
?- X is rdiv(1,2).  Enter        ← 1/2 の計算（その 2）
X = 1 rdiv 2.       ←計算結果
?- X is rdiv(1,2)+rdiv(1,3).  Enter   ← 1/2 + 1/3 の計算（その 1）
X = 5 rdiv 6.       ←計算結果
?- X is 1 rdiv 2 + 1 rdiv 3.  Enter   ← 1/2 + 1/3 の計算（その 2）
X = 5 rdiv 6.       ←計算結果
```

6.1.3.1　浮動小数点数と有理数との間の変換

浮動小数点数を有理数に変換するには rational もしくは rationalize を使用する．

例. 浮動小数点数を有理数に変換する

```
?- X is rational( 0.1 ).  Enter   ← 1.0 を有理数に変換する（その 1）
X = 3602879701896397 rdiv 36028797018963968.   ←変換結果
?- X is rationalize( 0.1 ).  Enter   ← 1.0 を有理数に変換する（その 2）
X = 1 rdiv 10.      ←変換結果
```

rational は浮動小数点数を正確に有理数に変換するが，rationalize は丸めを考慮した形で変換する．

有理数を浮動小数点数に変換するには float を使用する．

例. 有理数を浮動小数点数に変換する

```
?- X is float( 1 rdiv 4 ).  Enter   ← 1/4 を浮動小数点数に変換
X = 0.25.           ←変換結果
```

6.2 GNU Prolog

GNU Prolog は各種の OS（Microsoft Windows, Apple MacOS X, Linux など）の上で動作する Prolog 処理系であり，インターネットサイト http://www.gprolog.org/ で配布されている．

図 11: GNU Prolog のアイコン

図 12: MS-Windows 版 GNU Prolog を起動したところ

GNU Prolog には充実したヘルプ機能が備わっている．（図 13）

6.2.1 処理系の起動

図 11 のアイコンをダブルクリックすることで図 12 のような対話ウィンドウが開き，GNU Prolog が利用できる．

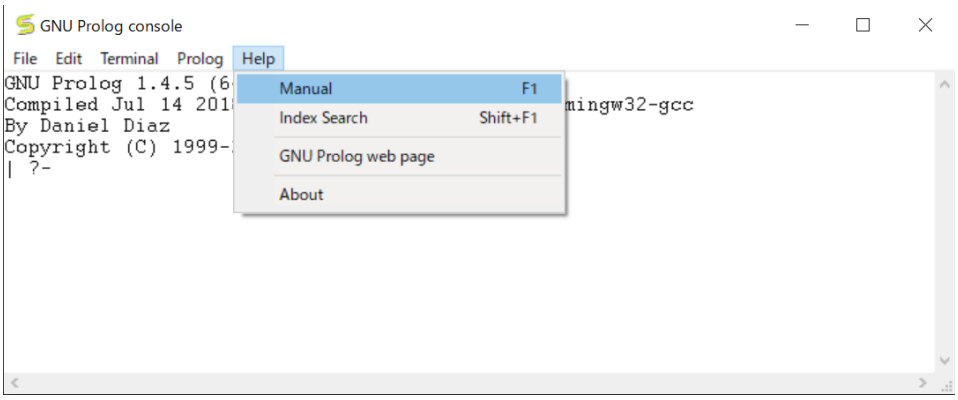
環境変数 LINEDIT に値 ‘gui=no’ を設定しておくと通常の端末のインターフェース（標準入出力）で GNU Prolog が使用できる．その場合は GNU Prolog 本体である gprolog コマンドを実行する．（gprolog コマンドへのコマンドサーチパスの設定が必要）

例. Windows 環境での GNU Prolog の起動（コマンド）

```
C:￥Users￥katsu＞ gprolog  Enter      ←コマンド gprolog を投入
GNU Prolog 1.4.5 (64 bits)      ←システム起動時のメッセージ
Compiled Jul 14 2018, 13:19:42 with x86_64-w64-mingw32-gcc
By Daniel Diaz
Copyright (C) 1999-2018 Daniel Diaz

| ?-            ← Prolog の問い合わせプロンプト
```

GNU Prolog には，Prolog で記述したプログラムをコンパイルして単独で動作するアプリケーションプログラムを生成する機能がある．同処理系の詳細に関しては文献 [6] を参照のこと．



「Help」メニューから「Manual」を選択　↓

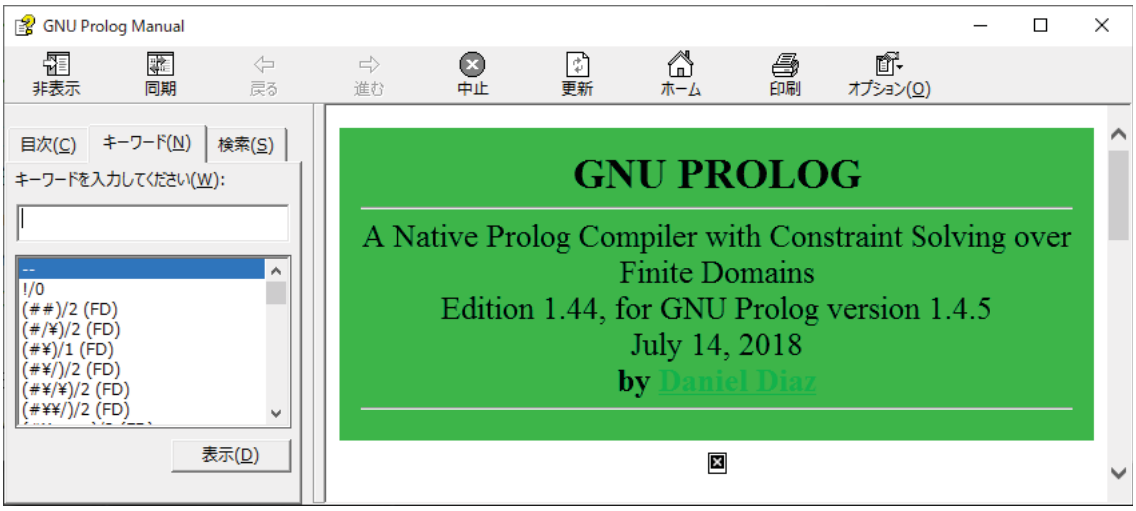


図 13: GNU Prolog のヘルプ機能

6.3　処理系の動作に関する設定

6.3.1　未定義述語の扱い

定義されていない事実や規則（未定義の述語）を問い合わせた場合にエラーが発生する処理系がある．ISO 準拠の Prolog には set_prolog_flag が組込み述語として用意されており，未定義の述語を頭部に持つゴールを偽として評価するための設定ができる．

実行例　（SWI-Prolog の場合）

```
?- predTest1(x).   Enter   ←定義されていない述語 predTest1 を問い合わせている
ERROR: toplevel:  Undefined procedure:  predTest1/1 (DWIM could not correct goal)
          （エラーとなって問い合わせが中断されている）

?- set_prolog_flag(unknown,fail).   Enter   ←エラー発生の抑止
true.

?- predTest1(x).   Enter   ←再度の問い合わせ
false.   ←偽となる
```

set_prolog_flag 述語の第 1 引数には**フラグ**[32] と呼ばれるものを指定する．これは処理系の設定項目とも言えるものであり，unknown は**未定義述語**を意味する．この例では第 2 引数に fail を指定して，未定義述語を偽として扱う設定をしている．

[32] **フラグ**には多くのものがある．詳しくは ISO Prolog の規格や処理系の仕様を参照のこと．

必要に応じて，Prolog のソースプログラムの冒頭に

```
:- set_prolog_flag(unknown,fail).
```

と記述しておくとよい．

フラグの設定値の取得

組込み述語 current_prolog_flag を使用することで，現在のフラグの設定値を取得することができる．

実行例. フラグ unknown の設定値を調べる

```
?- current_prolog_flag(unknown,V).   Enter
V = error.
```

実行例. 処理系の現在のフラグの設定値を全て調べる

```
?- current_prolog_flag(F,V).   Enter
F = save_history,
V = true ;
F = emulated_dialect,
V = swi ;
F = debug,
V = false ;
F = query_debug_settings,
V = debug(false, false) ;
F = double_quotes,
V = codes ;
F = unknown,
        ⋮
```

6.3.2 事実や規則の変更の可否

非単調推論を行うプログラムを実装するには，先に説明した asserta, assertz, retract などの組込み述語を使用して，事実や規則の登録や削除を行う．しかし多くの Prolog 処理系は，consult でプログラムを読み込む際に，動作速度向上のための最適化（コンパイル）を行っており，プログラムを読み込んだ後では定義を追加・削除することができないことがある．

consult で読み込んだ事実や規則を変更可能にするためには，変更可能とする対象の述語を最適化せずに “動的” な形にしておく必要があり，それを可能にするための組込み述語 dynamic がある．
テキストファイルとして記述する Prolog プログラムの冒頭で，処理系で読み込んだ後で変更可能とする述語を次のように指定する．

記述例

```
:- dynamic( editableFact/2 ).     ←「2 つの引数を持つ述語 editableFact」を編集可能とする設定
editableFact(taro,human).
editableFact(pochi,dog).
```

このように記述することで editableFact/2（2 つの引数を持つ述語 editableFact）は事後で編集が可能になる．

6.3.3 二重引用符の扱い

Prolog では二重引用符で囲った文字列は「文字コードのリスト」として扱われるが，処理系によっては，普通の意味での文字列（文字の列として表示されるもの）として扱われる場合もある．処理系による二重引用符の扱いの違いは，フラグ double_quotes の設定の違いによる．

先に紹介した 2 つの Prolog 処理系での実行結果を次に示す．

SWI-Prolog の場合

```
?- A = "abcde0123".   Enter
A = "abcde0123".      ←通常の意味での文字列

?- current_prolog_flag(double_quotes,V).   Enter      ←フラグを調べる
V = string.      ←「文字列」の扱いとなっている
```

GNU Prolog の場合

```
?- A = "abcde0123".   Enter
A = [97,98,99,100,101,48,49,50,51]      ←文字コードのリスト

?- current_prolog_flag(double_quotes,V).   Enter      ←フラグを調べる
V = codes      ←「文字コード」の扱いとなっている
```

SWI-Prolog 処理系においても，フラグ double_quotes を codes に設定することで，二重引用符で囲った文字列が文字コードのリストとして扱われる．

GNU Prolog において，二重引用符で囲った文字列をリストとしてではなく「文字列としてのアトム」として扱うには，フラグ double_quotes に atom を設定する．（文献 [6] 参照）

GNU Prolog における設定の例

```
?- set_prolog_flag(double_quotes,atom).   Enter      ←設定の変更
yes

?- A = "abc".   Enter
A = abc      ←アトムとして扱われている
```

6.3.4 その他

6.3.4.1 SWI-Prolog 処理系固有の設定

■ 表示されるリストの長さの制限

SWI-Prolog では表示されるリストの長さに制限があり，長いリストは簡略化して表示される．リストデータの連結処理などにおいて，得られるリストの長さが長い場合は，次に示す例のようにリストは簡略化されて表示される．

実行例 リストの表示の制限

```
?- L1=[a,b,c,d,e,f,g,h,i,j,k,l,m],L2=[n,o,p,q,r,s,t,u,v,w,x,y,z], append(L1,L2,A).   Enter
L1 = [a, b, c, d, e, f, g, h, i|...],
L2 = [n, o, p, q, r, s, t, u, v|...],
A = [a, b, c, d, e, f, g, h, i|...].
```

表示されるリストの長さの制限を変更するには，answer_write_options フラグの設定を変更するとよい．（次の例を参照）

実行例 リスト表示の長さの制限の変更

```
?- set_prolog_flag(answer_write_options,
    [quoted(true),portray(true),max_depth(256),spacing(next_argument)]).   Enter
true.

?- L1=[a,b,c,d,e,f,g,h,i,j,k,l,m],L2=[n,o,p,q,r,s,t,u,v,w,x,y,z],append(L1,L2,A).   Enter
L1 = [a, b, c, d, e, f, g, h, i, j, k, l, m],
L2 = [n, o, p, q, r, s, t, u, v, w, x, y, z],
A = [a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z].
```

この例のように，answer_write_options フラグに，設定内容のリストを与えるが，設定内容の中にある max_depth がリストの表示の長さの上限を指定する要素である．この設定に関する詳しいことは SWI-Prolog の仕様 [5] を参照のこと．

■ 文字コードの指定

ソースプログラムの中で使用する文字コードを指定するには，encoding 述語によって宣言する．例えば，utf-8 コード体系でソースプログラムを記述する場合は，当該ファイルの冒頭に，

```
:- encoding(utf8).
```

と記述する．この設定をせずに多バイト文字コードを使用したソースプログラムを読み込むと ‘Illegal multibyte Sequence’ という警告メッセージ（Warning）が表示される．

入出力で取り扱うデータとして多バイト文字コードを使用する場合は encoding フラグにコード体系を設定する．

実行例　入出力で UTF-8 の多バイト文字を扱う場合の設定．

```
?- set_prolog_flag(encoding,utf8).   Enter      ← UTF-8 に設定
true.          ←設定処理が正常に終了した．
```

付録

A　組込み述語

表 3: 組み込み述語： リスト処理関連

機能	式の書き方と解説
要素の取り出し	**nth0(要素位置, リスト, 該当要素)**
	「リスト」の「要素位置」（先頭は 0）の要素を「該当要素」に単一化する．GNU Prolog には類似の述語 nth があり，要素の開始位置を 1 とする．
要素の含有検査	**member(要素, リスト)**
	「要素」が「リスト」に含まれる場合に真となる．
リストの連結	**append(リスト 1, リスト 2, 連結結果)**
	「リスト 1」と「リスト 2」を連結したものが「連結結果」に単一化する．
末尾の要素	**last(リスト, 末尾の要素)**
	「リスト」の末尾の要素が「末尾の要素」に単一化する．
要素の順序の反転	**reverse(元のリスト, 反転したリスト)**
	「元のリスト」の順序を反転したものが「反転したリスト」に単一化する．

B　予約されているオペレータ

1200	xfx	--> , :-
1200	fx	:- , ?-
1150	fx	dynamic , discontiguous , initialization , meta_predicate , module_transparent , multifile , public , thread_local , thread_initialization , volatile
1100	xfy	; , |
1050	xfy	-> , *->
1000	xfy	,
990	xfx	:=
900	fy	\+
700	xfx	<, =, =.. , =@=, \=@=, =:= , =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, as , is , >:< , :<
600	xfy	:
500	yfx	+, - , /\ , \/ , xor
500	fx	?
400	yfx	*, / , // , div , rdiv , <<, >>, mod, rem
200	xfx	**
200	xfy	^
200	fy	+, - , \
100	yfx	.
1	fx	$

文献 [5] から引用

図 14: SWI-Prolog のオペレータ

Priority	Specifier	Operators
1200	xfx	:- -->
1200	fx	:-
1105	xfy	|
1100	xfy	;
1050	xfy	-> *->
1000	xfy	,
900	fy	\+
700	xfx	= \= =.. == \== @< @=< @> @>= is =:= =\= < =< > >=
600	xfy	:
500	yfx	+ - /\ \/
400	yfx	* / // rem mod div << >>
200	xfx	** ^
200	fy	+ - \

Priority	Specifier	Operators
750	xfy	#<=> #\<=>
740	xfy	#==> #\==>
730	xfy	## #\/ #\\/
720	yfx	#/\ #\/\
710	fy	#\
700	xfx	#= #\= #< #=< #> #>= #=# #\=# #<# #=<# #># #>=#
500	yfx	+ -
400	yfx	* / // rem
200	xfy	**
200	fy	+ -

文献 [6] から引用

図 15: GNU Prolog のオペレータ

参考文献

[1] R. コワルスキ（浦昭二監修, 山田眞市, 菊池光昭, 桑野龍夫 訳）,
「論理による問題の解法」, 培風館, 1987

[2] Leon Sterling, Ehud Shapiro 共著, （松田利夫 訳）,
「Prolog の技芸」, 構造計画研究所／共立出版, 1988

[3] 中村克彦,「Prolog と論理プログラミング」, オーム社,1985

[4] 東条敏,「言語・知識・信念の論理」, オーム社,2006

[5] Jan Wielemaker, 「SWI Prolog Reference Manual」, University of Amsterdam, 2017

[6] Daniel Diaz,
「GNU PROLOG : A Native Prolog Compiler with Constraint Solving over Finite Domains」,
Free Software Foundation, 2018

[7] ISO/IEC 13211-1:1995, Information technology
– Programming languages – Prolog – Part 1: General core

[8] ISO/IEC 13211-2:2000, Information technology
– Programming languages – Prolog – Part 2: Modules

索 引

「Prolog」 – 入門と演習

著者：中村勝則，平塚聡

発行：2020 年 4 月 14 日

テキストの最新版と更新情報

本書の最新版と更新情報を，プログラミングに関する情報コミュニティ Qiita で配信しています.

→ https://qiita.com/KatsunoriNakamura/items/6b6f599cdef098cc4e9d

上記 URL の QR コード

● 連絡先

nkatsu2012@gmail.com

中村勝則