初級 Prolog 講座

第零版

大黒学

初級 Prolog 講座・第零版
著者——大黒学

2014 年　9 月　1 日（月）　第零版発行

目次

第1章　プログラミングの基礎

1.1　データ

Q 1.1.1　データって何ですか。

A　「データ」(data) というのは、ものごとをあらわしている、物理的な存在に与えられた状態のことです。

ものごとを表現するためには、たとえば紙とインクのような、何らかの物理的な存在が必要です。そして、そのような物理的な存在に対して、形、位置、方向、個数、程度、周期、順序、組み合わせ、……というような状態を与えることによって、ものごとが表現されます。そのような状態のことを「データ」と呼ぶわけです。

たとえば、時計の針という物理的な存在に与えられた、それが向いている方向という状態は、現在の時刻というものをあらわしているデータだと考えることができます。

なお、データによってあらわされているもののことを、そのデータの「意味」(meaning) と呼びます。

Q 1.1.2　アナログのデータというのはどういうデータのことなんですか。

A　「アナログ」(analog) のデータというのは、ものごとをあらわしている連続的な量のことです。

たとえば、アルコール温度計という器具は、アルコールの体積という連続的な量によって温度をあらわしているわけですから、その場合のアルコールの体積というのはアナログのデータだと考えることができます。

また、紙やカンバスの上に絵具などで描かれた絵画も、その上の色や形は連続的な量ですから、やはりアナログのデータだと考えることができます。

Q 1.1.3　記号って何ですか。

A　「記号」(symbol) というのは、ほかのものから明確に区別することのできるひとつの状態のことです。

たとえば、日本語の文章で使われるひとつの文字というのは、ほかのものから明確に区別することのできるひとつの状態ですから、それはひとつの記号だと考えることができます。

Q 1.1.4　記号列って何ですか。

A　「記号列」(symbol sequence) というのは、重複を許して記号を並べることによってできる有限の長さの列のことです。

たとえば、重複を許して日本語の文字を並べることによってできる有限の長さの列は、記号列だと考えることができます。

Q 1.1.5　デジタルのデータというのはどういうデータのことなんですか。

A　「デジタル」(digital) のデータというのは、ものごとをあらわしている記号列のことです。

たとえば、日本語の文字から構成される記号列のうちで、何らかの意味を持っているものは、デジタルのデータだと考えることができます。

デジタルのデータは、アナログのデータと比較したとき、次のような二つの利点を持っています。

- デジタルのデータは、いくら複製を重ねても同一性が保持されます。それに対して、アナログのデータは、複製を重ねるたびに少しずつ品質が劣化していきます。
- デジタルのデータは、アナログのデータよりも、処理することによって得ることのできる結果の多様性が豊富です。

このような利点があることから、現在では、複製や処理の必要がある場合は、アナログではなくてデジタルのデータが使われることがほとんどです。

なお、「デジタルのデータ」の類義語として、「文書」(document) という言葉が使われることもあります。

Q 1.1.6　コードって何ですか。

A　「コード」(code) というのは、事物の集合から記号列の集合への 1 対 1 の関数のことです。

言い換えれば、コードというのは、ひとつの集合を構成するそれぞれの事物に対して異なる記号列を対応させる規則のことです。
たとえば、郵便番号簿に記載されている内容というのは、住所の集合から郵便番号の集合へのコードだと考えることができます。

Q 1.1.7　ビットって何ですか。

A　「ビット」(bit) というのは、二通りの状態のうちのどちらかであるような記号のことです。

たとえば、机の上に置かれたひとつのコインは、表が上になっているか、それとも裏が上になっているか、という二通りの状態を持つことができます。ですから、ひとつのコインが持っている、表か裏かという状態は、ひとつのビットだと考えることができます。
ビットがどちらの状態なのかということは、普通、0 と 1 という数字を使って書きあらわされます。

Q 1.1.8　ビット列って何ですか。

A　「ビット列」(bet sequence) というのは、重複を許してビットを並べることによってできる有限の長さの列のことです。

言い換えれば、ビット列というのは、記号としてビットを使っている記号列のことです。
なお、「ビット」(bit) という言葉は、ビット列の長さの最小単位という意味で使われることもあります。すなわち、「n ビット」というのは、n 個のビットから構成されるビット列の長さのことです。たとえば、0110001 というビット列の長さは 7 ビットです。

Q 1.1.9　文字列って何ですか。

A　「文字列」(string) というのは、重複を許して文字を並べることによってできる有限の長さの列のことです。

言い換えれば、文字列というのは、記号として文字を使っている記号列のことです。

Q 1.1.10　文字コードって何ですか。

A　「文字コード」(character code) というのは、文字の集合からビット列の集合へのコードのことです。

言い換えれば、文字コードというのは、ひとつの集合を構成するそれぞれの文字に対して異なるビット列を対応させる規則のことです。
文字列は、それを構成するそれぞれの文字を文字コードでビット列に変換したものを並べることによって、それをビット列に変換することができます。また、文字コードを使って文字列を変換することによってできたビット列は、同じ文字コードを使うことによって、もとの文字列に戻すことができます。
それでは、実際に文字コードを使って文字列をビット列に変換してみましょう。まず、

$$\{a, b, c, d, e, f\}$$

という、6 個の文字から構成される集合があるとします。そして、それらの文字に対してビット

列を対応させる文字コードとして、

$$
\begin{aligned}
a &\longrightarrow 000\\
b &\longrightarrow 001\\
c &\longrightarrow 010\\
d &\longrightarrow 011\\
e &\longrightarrow 100\\
f &\longrightarrow 101
\end{aligned}
$$

というものを定めたとします。そうすると、この文字コードを使うことによって、たとえば、

$cebbacf$

という文字列は、

01010000101001000010101

というビット列に変換することができます。そして、同じ文字コードを使うことによって、このビット列をもとの文字列に戻すことも可能です。

「文字コード」という言葉は、文字に対応しているビット列という意味で使われることもあります。つまり、c を文字だとするとき、それに対応しているビット列のことを c の「文字コード」と呼ぶこともある、ということです。

ビット列は、それを 2 進数だとみなすことによって、プラスの整数をあらわしていると考えることができます。たとえば、011010 というビット列は、それを 2 進数だとみなせば、26 という整数をあらわしていると考えることができます。

「文字コード」という言葉は、文字に対応しているビット列を 2 進数だとみなしたときにそれがあらわしている整数、という意味で使われることもあります。つまり、c を文字だとするとき、それに対応しているビット列を 2 進数だとみなしたときにそれがあらわしている整数のことを c の「文字コード」と呼ぶこともある、ということです。

Q 1.1.11 ASCIIって何ですか。

A ASCII（読み方は「アスキー」）というのは、実際に使われている文字コードのひとつです。

ASCII は、英字や数字などの文字を 7 ビットのビット列に対応させるコードです。たとえば、小文字の a という文字は、ASCII では、1100001 というビット列（2 進数だとみなせば 97 という整数）に対応しています。

ちなみに、ASCII という名前は、American National Standard Code for Information Interchange の略称です。

Q 1.1.12 テキストデータってなんですか。

A 「テキストデータ」(text data) というのは、ものごとをあらわしている文字列を、文字コードを使ってビット列に変換したもののことです。

言い換えれば、テキストデータというのは、文字に対応しているビット列から構成されるデータのことです。

テキストデータは、略して「テキスト」(text) と呼ばれることもあります。

Q 1.1.13 バイナリーデータって何ですか。

A 「バイナリーデータ」(binary data) というのは、ものごとをあらわしているビット列のうちで、テキストデータではないもののことです。

たとえば、デジタルの写真のデータやデジタルの音声のデータは、たいていの場合、バイナリーデータです。

バイナリーデータは、略して「バイナリー」(binary) と呼ばれることもあります。

1.2 プログラム

Q 1.2.1 コンピュータって何ですか。

A　「コンピュータ」(computer) というのは、データを処理する機械のことです。

コンピュータは、アナログのデータを処理するものとデジタルのデータを処理するものに分類することができて、前者は「アナログコンピュータ」(analog computer)、後者は「デジタルコンピュータ」(digital computer) と呼ばれます。ただし、現在では、アナログコンピュータというのはほとんど使われていません。

コンピュータは、データを外部から取り込んで、それを処理して、その結果として得られたデータを外部へ送り出す、という動作をする機械です。

コンピュータにデータを与えることを、データを「入力する」(input) と言います。そして、コンピュータがデータを外部へ送り出すことを、データを「出力する」(output) と言います。また、コンピュータがデータを外部から取り込むことを、データを「読み込む」(read) と言います。

なお、このチュートリアルのここから先の部分では、デジタルコンピュータのことを単に「コンピュータ」と呼ぶことにします。そして、デジタルのデータのことも、単に「データ」と呼ぶことにします。

Q 1.2.2　コンピュータの内部では、ものごとはどのような記号列によって表現されているのですか。

A　コンピュータの内部では、あらゆるものごとはビット列によって表現されています。

コンピュータの内部でものごとを表現するために使う記号列は、理論的にはどのようなものでもかまわないのですが、実際には、コンピュータの構造をもっとも単純にすることができるという理由で、2 種類の記号しか使わない記号列、つまりビット列しか使われていません。

Q 1.2.3　コンピュータは、データに対してどのような処理をするのですか。

A　コンピュータは、可能な限りどのようにでもデータを処理することができます。

Q 1.2.4　コンピュータの動作は何によって決定されるのですか。

A　コンピュータの動作は、どのように動作しなければならないのかということをあらわしているデータによって決定されます。

データをどのように処理するのかということを決定するものは、コンピュータという機械の構造の中に最初から刻み込まれているのではありません。コンピュータは、自分が実行するべき動作をあらわしているデータを外部から読み込んで、それにしたがって自分を動作させるのです。

ですから、コンピュータにデータを処理させるためには、そのデータをコンピュータに入力するのに先立って、コンピュータがどのように動作すればその処理を実現することができるのかということをあらわしているデータをコンピュータに入力することが必要です。

Q 1.2.5　プログラムって何ですか。

A　「プログラム」(program) というのは、コンピュータが実行するべき動作をあらわしているデータのことです。

つまり、目的とするデータの処理を実現するためにはコンピュータがどのように動作しなければならないのかということを記述したデータのことを「プログラム」と呼ぶわけです。

Q 1.2.6　ハードウェアとソフトウェアっていうのは何のことですか。

A　何らかのデータを利用するために必要となる物理的な装置は、そのデータを利用するための「ハードウェア」(hardware) と呼ばれます。そして、利用するために何らかの物理的な装置を必要とするデータは、その物理的な装置で利用できる「ソフトウェア」(software) と呼ばれます。

たとえば、DVD プレーヤーは、DVD に記録されている映画などのデータを利用するためのハードウェアです。そして、DVD に記録されているデータは、DVD プレーヤーで利用できるソフトウェアです。

同じように、コンピュータというのも、さまざまなデータを利用するためのハードウェアです。そして、利用するためにコンピュータを必要とするデータは、コンピュータのソフトウェアと呼ばれることになります。

Q 1.2.7 「コンピュータのソフトウェア」という言葉と「プログラム」という言葉は、同じ意味だと考えていいのですか。

A ほぼ同じ意味だと考えていいのですが、「コンピュータのソフトウェア」のほうが「プログラム」よりも広い意味の言葉だと考えるほうがいいでしょう。

プログラムというのはすべてコンピュータのソフトウェアですが、コンピュータのソフトウェアは、プログラムだけとは限りません。プログラムではないデータでも、それを利用するためにコンピュータを必要とするならば、それをコンピュータのソフトウェアと呼ぶことができます。

Q 1.2.8 プログラムは、どのような記号を使って作られるのですか。

A 人間がプログラムを作る場合は、普通、記号として文字が使われます。

文字というのは人間にとってもっとも扱いやすい記号ですから、人間は、プログラムを作るときにも記号として文字を使うのが普通です。ただし、コンピュータが何らかの処理の結果としてプログラムを出力する場合は、記号として文字以外のものが使われることもあります。
人間にとって、プログラムを作るという作業は、日記を書いたり手紙を書いたりするのと同様に、文字を並べていくという作業です。ですから、「プログラムを作る」という言い方よりも、「プログラムを書く」という言い方のほうが自然です。

Q 1.2.9 プログラミングって何ですか。

A 「プログラミング」(programming) というのは、プログラムの開発と、その改訂のために必要となるさまざまな作業の総称です。

プログラムというものを開発して、さらに、必要に応じてそれを改訂していくためには、さまざまな作業が必要になります。たとえば、プログラムの構造を設計する作業や、プログラムを書く作業や、プログラムが正しく動作するかどうかをテストする作業や、プログラムの不具合を修正する作業や、プログラムに関する文書を作成する作業などです。それらの作業は、総称して「プログラミング」と呼ばれます。

Q 1.2.10 プログラムの仕様って何ですか。

A プログラムの「仕様」(specification) というのは、そのプログラムが満足させるべき機能のことです。

プログラムを書くというのは、目標として存在している仕様に向かって、自分が書いているものの機能を少しずつ近付けていくという作業のことだと考えることができます。

Q 1.2.11 バグって何ですか。

A 「バグ」(bug) というのは、プログラムが満足させるべき仕様と、現状のプログラムによって実現されている機能との差異のことです。

余談ですが、プログラムというのは、その規模が大きくなるにつれて、バグが存在しないということを検証することが困難になっていきます。ですから、規模の大きなプログラムでは、すでに多くの人々によって使われているものでも、しばしばその中にバグが含まれていることがあります。

Q 1.2.12 デバッグって何ですか。

A 「デバッグ」(debug) というのは、プログラムからバグを取り除くことです。

Q 1.2.13 エラーって何ですか。

A 「エラー」(error) というのは、プログラムの中に含まれている間違いのことです。

「エラー」という言葉は、「バグ」という言葉とほとんど同じ意味だと考えていいのですが、間違いの種類によっては、「バグ」ではなくて「エラー」と呼ぶほうが自然なものもあります。
プログラムというものは、コンピュータがそれを実行することができるように作らないといけないわけですが、その中に間違いが含まれているために実行することができないということもあ

ります。そのような、コンピュータによる実行を阻害するような間違いは、「バグ」よりも「エラー」と呼ぶほうが自然です。

1.3　言語

Q 1.3.1　言語って何ですか。

A　「言語」(language) というのは、記号列を使って何かを表現するために定められた規則の集合のことです。

ひとつの言語を構成する規則は、大きく 2 種類のものに分類することができます。ひとつは「構文論」(syntax) と呼ばれる規則で、もうひとつは「意味論」(semantics) と呼ばれる規則です。

Q 1.3.2　構文論って何ですか。

A　構文論というのは、記号列の構造に関する規則のことです。

構文論は、「文法」(grammar) と呼ばれることもあります。

Q 1.3.3　意味論って何ですか。

A　意味論というのは、記号列と意味との対応に関する規則のことです。

Q 1.3.4　自然言語って何ですか。

A　「自然言語」(natural language) というのは、人間の社会の中で自然発生的に形成された言語のことです。

人間が読んだり聞いたりする記号列を作るための言語は、ほとんどすべて、自然言語だと考えていいでしょう。たとえば、アラビア語、中国語、日本語、スペイン語などは、自然言語に分類される言語です。

Q 1.3.5　人工言語って何ですか。

A　「人工言語」(artificial language) というのは、人間または人間の組織によって意図的に設計された言語のことです。

自然言語を構成する規則は、複雑で、かつ曖昧さを含んでいますので、それを使って作られた記号列は、コンピュータを使って機械的に処理することがきわめて困難です。ですから、コンピュータによって処理される記号列を作るための言語としては、自然言語ではなくて人工言語が使われるのが普通です。

Q 1.3.6　プログラミング言語って何ですか。

A　「プログラミング言語」(programming language) というのは、プログラムを書くために使われる言語のことです。

プログラムは、コンピュータによって処理される記号列ですから、プログラミング言語としては、自然言語ではなくて人工言語が使われるのが普通です。
プログラミング言語の実例としては、Fortran、Cobol、Pascal、C、awk、Perl、Tcl、Lisp、ML、Prolog、C++、Ruby、……というような、さまざまなものがあります。

Q 1.3.7　機械語って何ですか。

A　コンピュータ C がプログラミング言語 L で書かれたプログラムを実行することができるとき、L のことを、C の「機械語」(machine language) と言います。

どんなコンピュータも、自分の機械語を持っています。コンピュータが実行することができるのは、そのコンピュータの機械語で書かれているプログラムだけです。

Q 1.3.8　人間は、かならず、動作させたいコンピュータの機械語を使ってプログラムを書かないといけないのですか。

A いいえ、人間がプログラムを書くときに使うプログラミング言語は、それを実行するコンピュータとは無関係に選ぶことができます。

Q 1.3.9 セマンティックギャップって何ですか。

A 「セマンティックギャップ」(semantic gap) というのは、人間がプログラムを書くために使う言語とコンピュータの機械語とがどれだけ離れているかという距離のことです。

人間によって書かれたプログラムをコンピュータに実行させることができるためには、それらのあいだにあるセマンティックギャップがゼロである必要があります。しかし、人間が最初から機械語でプログラムを書くのでない限り、セマンティックギャップはゼロではあり得ません。ですから、それをゼロにするために、何らかの工夫が必要になります。

セマンティックギャップをゼロにするための工夫は、プログラムをコンピュータに近付ける方法と、コンピュータをプログラムに近付ける方法とに分類することができます。それらの方法は、どちらもそのためのプログラムを必要とします。プログラムをコンピュータに近付けるためのプログラムは「コンパイラ」(compiler) と呼ばれ、コンピュータをプログラムに近付けるためのプログラムは「インタプリタ」(interpreter) と呼ばれます。

Q 1.3.10 コンパイラって何ですか。

A コンパイラというのは、プログラムを読み込んで、それと同じ意味を持つ機械語のプログラムを出力するプログラムのことです。

コンパイラを使えば、プログラムをコンピュータに近付けるという方法でセマンティックギャップをゼロにすることができます。人間が書いたプログラムをコンパイラに処理させれば、それと同じ意味を持つ機械語のプログラムを得ることができるわけですから、その結果として、人間が書いたプログラムをコンピュータに実行させることができるわけです。

Q 1.3.11 インタプリタって何ですか。

A インタプリタというのは、プログラムの意味を解釈して、それがあらわしている動作をコンピュータに実行させるプログラムのことです。

別の言い方をすれば、インタプリタというのは、コンピュータの本来の機械語とは異なるプログラミング言語を機械語とする仮想的なコンピュータを作り出すプログラムのことだ、と言うこともできます。

インタプリタを使えば、コンピュータをプログラムに近付けるという方法でセマンティックギャップをゼロにすることができます。インタプリタを使って、人間がプログラムを書くために使ったプログラミング言語を機械語とする仮想的なコンピュータを作り出せば、その結果として、人間が書いたプログラムをコンピュータに実行させることができるわけです。

Q 1.3.12 コンパイラとインタプリタの両方を使うことができる場合、それらはどのように使い分ければいいのですか。

A プログラムの開発段階ではインタプリタを使って、プログラムが完成したのちにコンパイラを使うといいでしょう。

インタプリタを使うと、プログラムの一部分だけの動作を容易に確認することができますので、プログラムの開発段階では、コンパイラよりもインタプリタのほうが便利です。

しかし、インタプリタはプログラムを解釈しながらコンピュータに実行させますので、インタプリタによるプログラムの実行は、コンピュータが機械語のプログラムを実行する場合に比べると、時間的な効率が悪くなります。ですから、完成したプログラムは、インタプリタを使って実行するよりも、コンパイラを使って機械語に変換したものを実行するほうがいいわけです。

Q 1.3.13 言語処理系って何ですか。

A 「言語処理系」(language processor) というのは、プログラムを処理の対象とするプログラムのことです。

日本語では、誤解のおそれがある場合を除いて、言語処理系のことを、「言語」を省略して単に「処理系」と呼ぶことが多いようです。

コンパイラとインタプリタは、いずれも、プログラムを処理の対象とするプログラムですから、処理系の一種だと言えます。

ちなみに、コンパイラよりも少しだけ一般的な概念をあらわす言葉として、「トランスレーター」(translator) という言葉があります。これは、プログラムを読み込んで、それを別の言語に翻訳した結果を出力するプログラムのことです。コンパイラというのは、出力するプログラムの言語が何らかのコンピュータの機械語であるようなトランスレーターのことです。

Q 1.3.14　対話型のプログラムっていうのはどういうプログラムのことですか。

A　「対話型の」(interactive) プログラムというのは、人間が入力したデータを読み込んで、そのデータを処理する、ということを何回も繰り返すように作られているプログラムのことです。

インタプリタには、対話型のものとそうでないものとがあります。対話型のインタプリタは、人間が入力したプログラム（またはプログラムの一部分）を読み込んで、それを実行する、ということを何回も繰り返すように作られています。

Q 1.3.15　プロンプトって何ですか。

A　「プロンプト」(prompt) というのは、対話型のプログラムが、人間による入力が可能な状態になっているということを人間に知らせるために画面に表示する文字列のことです。

対話型のプログラムは、ほとんどすべて、人間による入力が可能な状態になっているということを人間に知らせるために、プロンプトを画面に表示するように作られています。ですから、対話型のプログラムというのは、

(1) プロンプトを表示する。
(2) 人間が入力したデータを読み込む。
(3) 読み込んだデータを処理する。

という 3 段階の動作を延々と繰り返すと考えていいでしょう。

1.4　計算モデル

Q 1.4.1　アルゴリズムって何ですか。

A　「アルゴリズム」(algorithm) というのは、問題を解くための機械的な手順のことです。

人間が出会うさまざまな問題のうちには、それを解くための機械的な手順、すなわちアルゴリズムが存在するものが少なからずあります。

たとえば、知恵の輪をはずすという問題は、アルゴリズムが存在する問題の典型的な例のひとつです。それを解くアルゴリズムを知らないあいだは、それをはずすためには試行錯誤をする必要がありますが、ひとたびアルゴリズムが得られれば、それを実行することによって、いとも簡単にそれをはずすことができるようになります。

アルゴリズムは、人間だけではなくてコンピュータに実行させることも可能です。プログラムというのは、プログラミング言語を使ってアルゴリズムを記述したものだと考えることができます。

Q 1.4.2　計算って何ですか。

A　「計算」(computation) というのは、アルゴリズムを実行することです。

つまり、すでに解き方が分かっている問題を解くために人間やコンピュータが実行していることを、「計算」と呼ぶわけです。

Q 1.4.3　計算モデルって何ですか。

A　「計算モデル」(computation model) というのは、アルゴリズムを実行する主体となるメカニズムのことです。

アルゴリズム自体は、それを実行する主体から独立した抽象的な知識です。しかし、アルゴリズムを記述するためには、そのアルゴリズムを実行する計算モデルを想定する必要があります。

たとえば、アルゴリズムを人間に伝えるために自然言語を使ってそれを記述する場合は、人間というメカニズムを計算モデルとして想定することになります。

プログラミング言語を使ってアルゴリズムを記述する場合、計算モデルは、使用するプログラミング言語によって提供されます。たとえば、何らかのコンピュータの機械語は、そのコンピュータを計算モデルとして提供します。

プログラミング言語が提供する計算モデルは、それぞれのプログラミング言語ごとに異なっています。ですから、計算モデルの種類はプログラミング言語の種類と同じ数だけあるわけですが、計算モデルをいくつかの大きな種類に分類することも可能です。そのような計算モデルの大きな種類としては、「関数型計算モデル」(functional computation model)、「論理型計算モデル」(logic computation model)、「オブジェクト指向計算モデル」(object-oriented computation model)、「手続き型計算モデル」(procedural computation model) などがあります。

プログラミング言語も、どのような計算モデルを提供するのかということによって、大きく分類することができます。関数型計算モデルを提供するプログラミング言語は「関数型言語」(functional language) と呼ばれ、論理型計算モデルを提供するプログラミング言語は「論理型言語」(logic language) と呼ばれ、オブジェクト指向計算モデルを提供するプログラミング言語は「オブジェクト指向言語」(object-oriented language) と呼ばれ、手続き型計算モデルを提供するプログラミング言語は「手続き型言語」(procedural language) と呼ばれます。たとえば、ML は関数型言語、Prolog は論理型言語、Ruby はオブジェクト指向言語、C は手続き型言語に分類されるプログラミング言語です。

Q 1.4.4 なぜ、プログラミング言語には枚挙にいとまがないほどさまざまなものがあるのですか。

A 表現されるべきアルゴリズムの性質に応じて、異なる計算モデルが必要になるからです。

プログラミング言語にはさまざまなものがあるという事実は、けっしてひとつの理由だけで説明できるものではありませんが、おそらく最大の理由は、万能の計算モデルというものが存在しないから、ということでしょう。

プログラムによって表現されるアルゴリズムには、さまざまな性質のものがあります。そして、アルゴリズムを表現するのに適した計算モデルも、アルゴリズムの性質ごとにさまざまです。つまり、プログラミング言語の多様性は、アルゴリズムの多様性を反映したものだと考えることができるわけです。

Q 1.4.5 関数って何ですか。

A 「関数」(function) というのは、データを受け取って、そのデータを処理して、その結果として得られたデータを返す、という動作をあらわしているデータのことです。

関数が受け取るデータは「引数」(argument) と呼ばれ、関数が返すデータは「戻り値」(returned value) と呼ばれます。

関数にデータを処理させることを、関数をデータに「適用する」(apply) と言います。

Q 1.4.6 関数型計算モデルっていうのはどんな計算モデルなんですか。

A 関数型計算モデルというのは、関数をデータに適用することによって戻り値を得るということを基本的な動作とする計算モデルのことです。

Q 1.4.7 命題って何ですか。

A 「命題」(proposition) というのは、何らかの方法によって、正しいか正しくないかということを判定することのできる言明のことです。

自然言語で作られたひとつの文は、平叙文、疑問文、命令文、感動文のいずれかに分類されます。それらのうちで、平叙文があらわしているもの、つまり疑問や命令や感動ではないものは、命題をあらわしていると考えることができます。

命題が正しいことを「真」(true) と言い、正しくないことを「偽」(false) と言います。また、命題が正しいか正しくないかということを、その命題の「真偽値」(truth value) と言います。

Q 1.4.8 証明って何ですか。

A　命題 P が真であるということを、すでに真であることがわかっている命題を使って論証することを、P の「証明」(proof) と言います。

Q 1.4.9　論理型計算モデルっていうのはどんな計算モデルなんですか。

A　論理型計算モデルというのは、命題を証明するということを基本的な動作とする計算モデルのことです。

Q 1.4.10　オブジェクトって何ですか。

A　「オブジェクト」(object) というのは、データと、そのデータに対するさまざまな処理とを一体にしたもののことです。

ひとつのオブジェクトの中には、データと、そのデータに対する処理を実行するさまざまなものとが入っています。オブジェクトの中にある、データに対する処理を実行するものは、「メソッド」(method) と呼ばれます。

オブジェクトは、「メッセージ」(message) と呼ばれるデータを別のオブジェクトに送ることができます。オブジェクトとオブジェクトとのあいだの相互作用は、メッセージのやり取りだけに限定されています。

メッセージを受け取ったオブジェクトは、そのメッセージにしたがって、自分の中にあるメソッドを動作させます。

Q 1.4.11　オブジェクト指向計算モデルっていうのはどんな計算モデルなんですか。

A　オブジェクト指向計算モデルというのは、オブジェクトがオブジェクトにメッセージを送るということを基本的な動作とする計算モデルのことです。

Q 1.4.12　記憶領域って何ですか。

A　「記憶領域」(memory area) というのは、データを保持することと、それによって保持されているデータを変更することが可能であるようなもののことです。

Q 1.4.13　手続き型計算モデルっていうのはどんな計算モデルなんですか。

A　手続き型計算モデルというのは、記憶領域によって保持されているデータを変更するということを基本的な動作とする計算モデルのことです。

Q 1.4.14　プログラミングパラダイムって何ですか。

A　「プログラミングパラダイム」(programming paradigm) というのは、アルゴリズムの抽象性のレベルを、計算モデルが実行することのできるレベルにまで具象化するための思考法のことです。

人間がプログラムを書くためには、特定の計算モデルにもとづいてアルゴリズムを具象化するという作業が必要になります。プログラミングパラダイムというのは、その作業を進めるためにはどのように思考しなければならないか、という考え方のことです。

どのようなプログラミングパラダイムが必要になるかというのは、計算モデルによって異なります。ですから、プログラミングパラダイムも、計算モデルやプログラミング言語と同じように分類することが可能です。関数型計算モデルで必要となるプログラミングパラダイムは「関数プログラミング」(functional programming) と呼ばれ、論理型計算モデルで必要となるプログラミングパラダイムは「論理プログラミング」(logic programming) と呼ばれ、オブジェクト指向計算モデルで必要となるプログラミングパラダイムは「オブジェクト指向プログラミング」(object-oriented programming) と呼ばれ、手続き型計算モデルで必要となるプログラミングパラダイムは「手続き型プログラミング」(procedural programming) と呼ばれます。

第 2 章　Prolog の基礎

2.1 Prolog の基礎の基礎

Q 2.1.1 Prologって、誰が作ったプログラミング言語なんですか。

A Prolog を開発したのは、Alain Colmerauer という人です。

フランスにあるマルセイユ大学で教授を務めていた Colmerauer は、1971 年に、SYSTEM Q と呼ばれるプログラミング言語を開発しました。この言語は、そののち、イギリスにあるエディンバラ大学の Robert A. Kowalski という人によって論理学との関係が明らかにされた結果、Prolog と改名されました。ちなみに、Prolog という名前は、フランス語の programmation en logique（英語に訳すと programming in logic）という語句を縮めることによって作られたものです。

Q 2.1.2 Prolog に標準規格っていうのはあるんですか。

A はい、あります。

1995 年に、ISO (International Organization for Standardization) によって Prolog の標準規格が制定されています。

Q 2.1.3 事実って何ですか。

A 「事実」(fact) というのは、真だということがすでにわかっている命題のことです。

Q 2.1.4 規則って何ですか。

A 「規則」(rule) というのは、二つの命題のあいだに、一方が真のときはかならず他方も真になるという関係が成り立っている、という事実のことです。

規則を自然言語で表現すると、それは、二つの文を「ならば」(if) という接続詞で組み合わせた形の文になります。たとえば、「増雄が働き者ならば恵子は幸せである」という文は、「増雄は働き者である」という命題が真のときはかならず「恵子は幸せである」という命題も真になる、という規則をあらわしています。

一般的には規則というのも事実の一種ですが、Prolog では、規則と事実とは別々のものだと考えます。ですから、Prolog での「事実」という言葉の定義は、一般的な「事実」の定義に対して、「ただし規則は含まない」という但し書を付けたものだと考えることができます。

Q 2.1.5 三段論法って何ですか。

A 「三段論法」(syllogism) というのは、ひとつの規則とひとつの事実を前提として、そこからひとつの結論を導き出す、という推論のことです。

三段論法の前提となる規則は「大前提」(major premise) と呼ばれ、前提となる事実は「小前提」(minor premise) と呼ばれます。

A と B が命題で、「A ならば B」が大前提で、A が小前提だとすると、

大前提　A ならば B

小前提　A

結論　B

という三段論法によって、B もまた事実だということを推論することができます。もう少し具体的な例としては、

大前提　増雄が働き者ならば恵子は幸せである。

小前提　増雄は働き者である。

結論　恵子は幸せである。

という推論も三段論法です。

Q 2.1.6 Prologっていうのは、どんな計算モデルを提供するプログラミング言語なんですか。

A　Prolog が提供するのは、三段論法を使って命題を証明する、ということを基本的な動作とする計算モデルです。

命題を証明するということを基本的な動作とする計算モデルは、Q 1.4.9 で説明したように、「論理型計算モデル」と呼ばれます。Prolog が提供する計算モデルも論理型計算モデルの一種で、それは、命題を証明する方法として三段論法を使います。

Q 2.1.7　公理系って何ですか。

A　「公理系」(axiomatic system) というのは、命題を証明するための前提として使われる事実または規則の集合のことです。

ちなみに、公理系に含まれるそれぞれの事実や規則は、「公理」(axiom) と呼ばれます。

Q 2.1.8　Prolog のプログラムっていうのは、何を記述したものなんですか。

A　Prolog のプログラムというのは、公理系を記述したものです。

つまり、Prolog のプログラムというのは、命題を証明するための前提として使われるいくつかの事実または規則を記述したものだということです。

Q 2.1.9　質問って何ですか。

A　「質問」(query) というのは、Prolog のインタプリタに与えられる、証明することが要請される命題のことです。

Prolog のインタプリタに Prolog のプログラムを与えたとしても、それだけではインタプリタは何の動作もしません。Prolog のインタプリタにプログラムを実行させるためには、プログラムだけではなくて、証明することが要請される命題、つまり質問を与える必要があります。
プログラムと質問を Prolog のインタプリタに与えると、インタプリタは、プログラムがあらわしている公理系にもとづいて、三段論法を使って質問の証明を試みます。この、「質問の証明を試みる過程」というのが、「Prolog のプログラムの実行」ということなのです。

Q 2.1.10　述語って何ですか。

A　「述語」(predicate) というのは、ものの性質、またはものとものとのあいだの関係のことです。

たとえば、「地球は天体である」という文は、「地球」というものが「天体である」という性質を持っているという命題をあらわしているわけですが、このときの「天体である」という性質は、ひとつの述語だと考えることができます。
また、「伸之は聡子の父親である」という文は、「伸之」という人と「聡子」という人とのあいだに、前者が後者の「父親である」という関係が成り立っているという命題をあらわしているわけですが、このときの「父親である」という関係も、ひとつの述語だと考えることができます。

Q 2.1.11　述語の引数って何のことですか。

A　述語の「引数」(argument) というのは、述語が適用される対象のことです。

言い換えれば、命題を自然言語で記述したときに、述語以外の主語や目的語などで指示されているものが引数です。
たとえば、「地球は天体である」という命題では、「地球」というのが、「天体である」という述語の引数です。
また、「伸之は聡子の父親である」という命題では、「伸之」と「聡子」のそれぞれが、「父親である」という述語の引数です。

Q 2.1.12　述語の項数って何のことですか。

A　述語の「項数」(arity) というのは、述語が適用される引数の個数のことです。

項数は、述語によって決定されます。言い換えれば、それぞれの述語は自分の項数を持っている、ということです。たとえば、「地球は天体である」という命題で使われている「天体である」という述語は、1 という項数を持っています。

種類	説明	詳細
アトム (atom)	ものごとの名前として使われる項	第 2.3 節
数値定数 (number constant)	特定の数値をあらわす項	第 2.4 節
変数 (variable)	「何でもかまわない何か」をあらわす項	第 3.4 節
複合項 (compound term)	2 個以上の項を組み合わせて作られた項	第 3.1 節

表 2.1: 項の種類

述語の名前（述語をあらわす言葉）は同じだけれども項数が異なる、といういくつかの述語は、それぞれ異なる述語だとみなされます。たとえば、「伸之は父親である」という命題で使われている、項数が 1 の「父親である」と、「伸之は聡子の父親である」という命題で使われている、項数が 2 の「父親である」とは、別々の異なる述語だとみなされます。

Q 2.1.13 述語を定義するって、どういうことですか。

A 述語を「定義する」(define) というのは、名前と項数との組み合わせに対して、ひとつの述語を意味として与えることです。

ひとつの述語は、それに関するいくつかの事実または規則を記述することによって定義されます。この観点から言えば、Prolog のプログラムというのは、述語を定義する記述がいくつか集まったもののことだと考えることができます。

ちなみに、Prolog のプログラムのうちで、ひとつの述語を定義している部分は、「述語定義」(predicate definition) と呼ばれます。

Q 2.1.14 項って何ですか。

A 「項」(term) というのは、Prolog のプログラムを構成する文法上の単位のひとつです。

項には、表 2.1 に示されているように、四つの種類があります。

なお、アトムと数値定数は、どちらも特定のものごとをあらわす項ですので、それらを合わせたものを定数 (constant) と呼ぶこともあります。

2.2 ホワイトスペース

Q 2.2.1 ホワイトスペースって何ですか。

A Prolog では、空白 (space)、水平タブ (horizontal tab)、改行 (newline)、注釈 (comment) という 4 種類の文字や文字列を総称して、「ホワイトスペース」(white space) と呼びます。

空白と水平タブと改行は、主として、人間がプログラムを見たときに、その構造が視覚的にわかりやすくなるように、その形を整えるために使われます。

注釈については次の質問と回答を参照してください。

Q 2.2.2 注釈って何ですか。

A 「注釈」(comment) というのは、プログラムの中に書かれた文字列のうちで、処理系によって無視されるもののことです。

ほとんどすべてのプログラミング言語は、その中に、処理系によって無視される文字列を書くための規則を持っています。そのような規則にしたがって書かれた文字列は、「注釈」と呼ばれます。

Q 2.2.3 プログラムの中に注釈を書く目的って、何ですか。

A プログラムの中に注釈を書く目的のうちでもっとも重要なのは、プログラムを読む人間に対して、それを理解するためのヒントを提供することです。

プログラムというのは、処理系によって処理されるデータであると同時に、人間によって読まれる文書でもあります。ですから、プログラムというものは、処理系がそれを理解することがで

きるように書かなければならないということはもちろんですが、同時に、それを読む人間にとって理解しやすくなるように書かなければならないということも重要です。

人間にとって理解しやすいプログラムを書くための技法は多岐に渡りますが、そのうちのひとつが、プログラムの中に注釈を書くということです。つまり、プログラムを理解する上で役に立つヒントが注釈の形でプログラムの中に書き込まれていれば、プログラムはそれだけ理解しやすいものになるわけです。

Q 2.2.4 Prolog も、注釈を書くための規則を持っているのですか。

A はい、Prolog も、注釈を書くための規則を持っています。

Prolog は、注釈を書くための規則を二つ持っています。ひとつはパーセント (percent sign, `%`) を使う規則で、もうひとつはスラッシュ(slash, `/`) とアスタリスク (asterisk, `*`) を使う規則です。

Q 2.2.5 パーセントを使って注釈を書く規則というのは、どのようなものなんですか。

A プログラムの中にパーセントを書くと、そこから最初の改行までが注釈とみなされます。

たとえば、プログラムの中に、

```
utsukushii(kimiko).  % Kimiko is beautiful.
```

という行を書いたとすると、その中にあるパーセントから、行の末尾にある改行までが注釈とみなされることになります。

Q 2.2.6 スラッシュとアスタリスクを使って注釈を書く規則というのは、どのようなものなんですか。

A スラッシュアスタリスク (`/*`) で始まってアスタリスクスラッシュ(`*/`) で終わる文字列は、注釈とみなされます。

たとえば、プログラムの中に、

```
utsukushii(kimiko/* This name may not right. */).
```

という文字列を書いたとすると、その中に含まれている、

```
/* This name may not right. */
```

という文字列は、注釈とみなされることになります。

`/*` と `*/` による注釈は、その中に改行を含んでいてもかまいません。ですから、複数行にまたがる文字列も、その全体を `/*` と `*/` で囲むことによって注釈にすることができます。たとえば、

```
/* All quotations from the Encyclopedia Galactica here
reproduced are taken from the 116th Edition published in
1020 F.E. by the Encyclopedia Galactica Publishing Co.,
Terminus, with permission of the publishers. */
```

という文字列は、ひとつの注釈になります。

なお、`/*` と `*/` による注釈の中にパーセントが含まれている場合、そのパーセントは、最初の改行までを注釈にするという機能を失います。たとえば、

```
utsukushii(/*%*/kimiko).
```

という文字列の中の注釈は `/*%*/` という部分だけで、その右側にある `kimiko).` は注釈の一部分にはなりません。

Q 2.2.7 `/*` と `*/` による注釈の中には、どんな文字列を書いてもかまわないのですか。

A いいえ、`*/` という文字列を書くことができないという例外があります。

`/*` と `*/` による注釈の中に `*/` を書くことができない理由は、それが注釈の途中にあると、注釈がそこで終わっていると解釈されてしまうからです。たとえば、

```
/*namako*/hitode*/
```

という文字列は、`/*namako*/` という前半だけが注釈だと解釈されて、その右側の `hitode*/` は注釈ではないと解釈されます。

Q 2.2.8 コメントアウトって何ですか。

A 本来は処理系によって処理されるべきプログラムの一部分を、何らかの理由で一時的に注釈にすることを、その部分を「コメントアウトする」(comment out) と言います。

プログラムをデバッグするときには、しばしば、プログラムの一部分を取り除くとどうなるかを試す必要が生じることがあります。その場合は普通、その部分を本当に削除してしまうのではなくて、一時的に注釈にする、つまりコメントアウトするというテクニックが使われます。

Q 2.2.9 パーセントによる注釈と、`/*` と `*/` による注釈とは、どのように使い分ければいいのですか。

A プログラムを読む人のためのヒントにはパーセントを使って、コメントアウトには `/*` と `*/` を使うのがいいでしょう。

なぜなら、ヒントの注釈が `/*` と `*/` を使って書かれていると、それらの注釈を含むプログラムの一部分を `/*` と `*/` で囲むことによってコメントアウトするということができなくなってしまうからです。ヒントの注釈がパーセントを使って書かれていれば、プログラムの一部分を自由にコメントアウトすることができるようになります。

2.3 アトム

Q 2.3.1 アトムって何ですか。

A 「アトム」(atom) というのは、ものごとの名前として使われる項のことです。

ちなみに、「アトム」という言葉は、「これ以上は分割できないもの」を意味するギリシア語の言葉に由来します。

アトムの用途は、何かを指示するための名前として使うというというものですが、実は、アトムの用途はそれだけではありません。アトムのもうひとつの用途は、データとして扱われる特定の文字列を Prolog のプログラムの中に書くために使う、というものです。

アトムには、次の四つの種類があります。

- 英数字アトム (alphanumeric atom)
- 非英数字アトム (non-alphanumeric atom)
- 引用アトム (quoted atom)
- 空リスト (empty list)

英数字アトムについては Q 2.3.2、非英数字アトムについては Q 2.3.3、引用アトムについては Q 2.3.4、空リストについては Q 2.3.10 を参照してください。

Q 2.3.2 英数字アトムって、どんなアトムなんですか。

A 英数字アトムというのは、英数字（英字または数字）を使って作られたアトムのことです。

英数字アトムを作るために使うことのできる文字は、63 種類あります。英字の大文字と小文字、数字、そしてアンダースコア (underscore, _) です。英数字アトムは、これらの文字を 1 個以上並べて作られた列です。ただし、先頭の 1 文字は英字の小文字でないといけません。

それでは、英数字アトムの例をいくつか挙げてみましょう。たとえば、

```
a  namako  kusetsu30nen  tadashii_eisuuji_atom
```

などは正しい英数字アトムの例です。2 個以上の単語から構成される英数字アトムを作るときは、普通、4 番目の例のようにアンダースコアを使ってそれらの単語を区切ります。

Prolog 以外の言語では、2 個以上の単語から構成される名前を作るための方法として、しばしば、

```
tadashiiEisuujiAtom
```

というように、2 番目以降の単語の先頭を大文字にするという方法が使われます。この方法は、小文字の列の中に混ざった大文字がラクダの瘤のように見えるので、camel case と呼ばれます。Prolog でも camel case を使うことは可能ですが、あまり好まれていないようです。

次に、正しくない英数字アトムの例も挙げてみます。たとえば、

```
uni@ikura  Yoshiko  36kasen  _umiushi
```

などは正しくない英数字アトムの例です。1 番目の例が正しくない理由は、@ という、英字でも数字でもアンダースコアでもない文字が含まれているからです。そして、2、3、4 番目の例が正しくない理由は、先頭の文字が英字の小文字ではないからです。

Q 2.3.1 で説明したように、アトムには、何かを指示するための名前として使うという用途のほかに、データとして扱われる特定の文字列を記述するという用途もあります。後者の用途のために、アトムは、特定の文字列をあらわしていると解釈することができるようになっています。

英数字アトムがあらわしている文字列は、それ自身です。たとえば、namako という英数字アトムは、namako という文字列をあらわしていると解釈することができます。

Q 2.3.3 非英数字アトムって、どんなアトムなんですか。

A 非英数字アトムというのは、英数字ではない文字を使って作られたアトムのことです。

非英数字アトムを作るために使うことのできる文字は、17 種類あります。その 17 種類というのは、

```
  #  $  &  *  +  -  .  /  :  <  =  >  ?  @  ^  ~  \
```

です。非英数字アトムは、これらの文字を 1 個以上並べて作られた列です。

たとえば、

```
#  @:?*&=$  +-+-+  ===>
```

などは正しい非英数字アトムの例です。それに対して、

```
  @@y@@
```

は、正しくない非英数字アトムの例です。これが正しくない理由は、その中に y という英字が含まれているからです。

非英数字アトムも、英数字アトムと同じように、それ自身という文字列をあらわしていると解釈することができます。たとえば、===> という非英数字アトムは、===> という文字列をあらわしていると解釈することができます。

Q 2.3.4 引用アトムって、どんなアトムなんですか。

A 引用アトムというのは、一重引用符で始まって一重引用符で終わるアトムのことです。

任意の文字列の前後に一重引用符 (single quote, ’) という文字を書いたものは、引用アトムになります（ただし特殊な例外はありますが)。たとえば、

```
’uni@ikura’  ’Yoshiko’  ’36kasen’  ’_umiushi’
’@@y@@’      ’a  b  c’  ’       ’  ’’
```

などは正しい引用アトムの例です。ちなみに、最後の ’’ という例は、一重引用符を連続して二つ並べたものですが、これも正しい引用アトムです。

文字列を一重引用符で囲んだにもかかわらず引用アトムにならない例としては、たとえば、

```
  ’I don’t know.’
```

というのがあります。これが引用アトムにならない理由は、文字列の中に一重引用符が含まれているからです。また、

```
  ’I am a very
  very long string.’
```

というのも引用アトムにならない例です。これが引用アトムにならない理由は、文字列の中に改行が含まれているからです。

英数字アトムや非英数字アトムが、何かを指示するための名前として使われることが多いのに対して、引用アトムは、主として、データとして扱われる特定の文字列を Prolog のプログラムの中に書くために使われます。

引用アトムは、その先頭と末尾の一重引用符のあいだにある文字列をあらわしていると解釈することができます。たとえば、’c@t’ という引用アトムは、c@t という文字列をあらわしていると解釈することができます。

Q 2.3.5 一重引用符を含む文字列をあらわしている引用アトムを作りたいときって、どうすればいいんですか。

A 一重引用符を含む文字列をあらわしている引用アトムは、一重引用符を二つ続けて書くことによって作ることができます。

引用アトムの中には、2 個の連続する一重引用符を書くことができて、それは、1 個の一重引用符とみなされます。たとえば、

```
'I don''t know.'
```

という引用アトムを作ることができて、これは、

```
I don't know.
```

という文字列をあらわしていると解釈されます。

Q 2.3.6 ひとつの引用アトムを 2 行以上に分けて書きたいときって、どうすればいいんですか。

A 改行 (newline) という文字の直前にバックスラッシュ(backslash, \) という文字を書くことによって、ひとつの引用アトムを 2 行以上に分けて書くことができます。

引用アトムの中では、バックスラッシュの直後に改行を書くと、その改行は無視されます。ですから、

```
'I am a very very long string.'
```

という引用アトムは、バックスラッシュを使うことによって、

```
'I am a very \
very long string.'
```

というように、2 行に分けて書くことができます。

Q 2.3.7 改行という文字を含む文字列をあらわしている引用アトムを作りたいときって、どうすればいいんですか。

A 改行 (newline) という文字を含む文字列をあらわしている引用アトムを作りたいときは、「エスケープシーケンス」(escape sequence) と呼ばれるものを使います。

引用アトムの中にはさまざまな文字を書くことができるわけですが、まったく例外がないわけではありません。そのままでは引用アトムの中に書くことができない文字というのも存在します。たとえば、改行というのは、そのままでは引用アトムの中に書くことができない文字のひとつです。しかし、エスケープシーケンスと呼ばれるものを使うことによって、そのままでは書くことができない文字を含む文字列をあらわしている引用アトムを作る、ということが可能になります。

エスケープシーケンスというのは、バックスラッシュ(backslash, \) という文字で始まる、1 個の文字を意味する文字列のことです。たとえば、改行という文字は、バックスラッシュと小文字の `n` とを並べることによってできる、`\n` というエスケープシーケンスによってあらわすことができます。ですから、

```
'123\n456'
```

と書くことによって、`123` と `456` とのあいだに改行を含んでいる文字列をあらわしている引用アトムを作ることができます。

エスケープシーケンスには、改行を意味する `\n` のほかに、次のようなものがあります。

`\a` 警告音 (alert)。
`\b` バックスペース (backspace)。
`\f` フォームフィード (formfeed)。
`\r` キャリッジリターン (carriage return)。
`\t` 水平タブ (horizontal tab)。
`\v` 垂直タブ (vertical tab)。
`\\` バックスラッシュ(backslash)。
`\`n`\` 8 進数 n であらわされるビット列を文字コードとする文字。例 `\155\`

`\x`n`\` 16 進数 n であらわされるビット列を文字コードとする文字。例 `\x6d\`

Q 2.3.8 空文字列って何ですか。

A 「空文字列」(empty string) というのは、0 個の文字から構成される文字列のことです。

2 個の連続する一重引用符、つまり `''` は、空文字列をあらわしている引用アトムです。

Q 2.3.9 英数字アトムまたは非英数字アトムを一重引用符で囲んで引用アトムを作ったとすると、その引用アトムは、もとのアトムとは違うものになるんですか。

A いいえ、それらは同じアトムだとみなされます。

たとえば、`namako` という英数字アトムと、それを一重引用符で囲んだ `'namako'` という引用アトムとは、同じアトムだとみなされます。同じように、`===>` という非英数字アトムと、それを一重引用符で囲んだ `'===>'` という引用アトムも、同じアトムだとみなされます。

Q 2.3.10 空リストって、どんなアトムなんですか。

A 空リストというのは、`[]` というアトムのことです。

つまり、空リストというのは、左角括弧 (left bracket, `[`) という文字の右側に右角括弧 (right bracket, `]`) という文字を書いたもののことです。
空リストは、「リスト」と呼ばれるデータを作るときに使われるアトムです。リストについては、第 6 章を参照してください。

2.4 数値定数

Q 2.4.1 数値を扱うプログラムを Prolog で書くことって、可能なんですか。

A はい、可能です。

Prolog のプログラムが取り扱うことのできる数値は、整数と浮動小数点数に分類することができます。

Q 2.4.2 整数って何ですか。

A 「整数」(integer) というのは、小数点以下の端数を持たない数値のことです。

つまり、整数というのは、

$$\cdots, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, \cdots$$

という数値のことです。

Q 2.4.3 浮動小数点数って何ですか。

A 「浮動小数点数」(floating-point number) というのは、1 個の数字列（数字を並べることによってできる列）と、その数字列の小数点の位置を移動させる機能を持つ 1 個の数字列、という 2 個の数字列の組によってあらわされる数値のことです。

浮動小数点数を表現するための 2 個の数字列は、それぞれ「仮数部」(mantissa) と「指数部」(exponent) と呼ばれます。指数部は、仮数部の小数点の位置を移動させる数字列です。
仮数部を m、指数部を e として、仮数部が r 進法で表現されているとするとき、m と e の組によってあらわされる浮動小数点数は、

$$m \times r^e$$

という計算の結果として求まる数値です。
たとえば、仮数部が 4.826900252 という 10 進数で、指数部が 33 という 10 進数であるような数字列の組は、

$$4.826900252 \times 10^{33} = 4826900252000000000000000000000000$$

という浮動小数点数をあらわします。

同じように、仮数部が 4.061590294 という 10 進数で、指数部が −23 という 10 進数であるような数字列の組は、

$$4.061590294 \times 10^{-23} = 0.00000000000000000000004061590294$$

という浮動小数点数をあらわします。

Q 2.4.4 数値定数って何ですか。

A 「数値定数」(number constant) というのは、特定の数値をあらわす項のことです。

Prolog のプログラムが取り扱うことのできる数値が整数と浮動小数点数に分類されることを反映して、数値定数も、特定の整数をあらわす「整数定数」(integer constant) と、特定の浮動小数点数をあらわす「浮動小数点定数」(floating-point number constant) に分類されます。

Q 2.4.5 整数定数って、どんなふうに書けばいいんですか。

A 数字列、またはその左側にマイナス (minus, `-`) という文字を書いたものは、整数定数になります。

たとえば、`3017` や `-2568` などは整数定数の例です。これらの整数定数は、10 進法で整数をあらわしていると解釈されます。左側にマイナスという文字が書かれた数字列は、マイナスの整数をあらわす整数定数になります。

Q 2.4.6 10 進法以外の記数法で整数をあらわす整数定数を書くことって、可能ですか。

A はい、整数定数は、10 進法だけではなくて、2 進法、8 進法、16 進法で書くことも可能です。

10 以外の基数による記数法で整数をあらわす整数定数を書きたいときは、数字（16 進法の場合は、数字に、`a` から `f` までの英小文字と、`A` から `F` までの英大文字が加わります）の列の左側に、基数をあらわす次のような接頭辞を書きます。

基数	接頭辞	整数定数の例
2	`0b`	`0b1011011111000101`
8	`0o`	`0o133705`
16	`0x`	`0xb7c5`

10 以外の基数による記数法で、マイナスの整数をあらわす整数定数を書きたいときは、接頭辞のさらに左側にマイナスという文字を書きます。たとえば、`-0b1110` は、マイナスの整数をあらわす整数定数の例です。

Q 2.4.7 文字コード定数って何ですか。

A 「文字コード定数」(character code constant) というのは、文字コードを使って整数をあらわす定数のことです。

文字コード定数は、`0'` という接頭辞の右側に 1 個の文字を書いたものです。たとえば、`0'a`、`0'8`、`0'@` などは文字コード定数の例です。

文字コード定数は、`0'` という接頭辞の右側に書かれた文字に対応している文字コードを 2 進数とみなして解釈することによって得られる整数をあらわしていると解釈されます。ですから、文字コード定数があらわしている整数は、使われている文字コードに依存して決定されることになります。たとえば、文字コードとして ASCII が使われているとすると、`a` という英小文字は 1100001 というビット列（2 進数とみなすと 97）に対応していますので、`0'a` という文字コード定数は、97 という整数をあらわすことになります。

`0'` という接頭辞の右側には、どんな文字でも書くことができます。そして、その場所に書かれた文字は、その本来の意味を失って、文字コード定数の一部分になります。たとえば、`0'%` という文字コード定数を書くことができて、その中のパーセントは注釈の開始という意味には解釈されません。

Q 2.4.8 浮動小数点定数って、どんなふうに書けばいいんですか。

A　ドット (dot, `.`) という文字の左と右の両側に数字列を書いたもの、またはそのような記述の左側にマイナス (minus, `-`) という文字を書いたものは、浮動小数点定数になります。

たとえば、`2.037` や `-855.4` などは浮動小数点定数の例です。これらの浮動小数点定数は、10 進法で浮動小数点数をあらわしていると解釈されます。浮動小数点定数の中のドットは、小数点の位置を示します。マイナスという文字で始まる浮動小数点定数は、マイナスの浮動小数点数をあらわします。

なお、浮動小数点定数のドットの左右には、かならず 1 個以上の数字を書く必要があります[1]。たとえば、`.4603` や `8117.` のような記述は、正しい浮動小数点定数ではありません。

Q 2.4.9　ものすごく巨大な浮動小数点数とか、ものすごく微小な浮動小数点数をあらわす定数って、数字をものすごくたくさん並べないといけないんですか。

A　いいえ、`e` または `E` という文字を含む浮動小数点定数を書くことによって、巨大な浮動小数点数や微小な浮動小数点数を簡潔に記述することができます。

整数定数、または、`e` や `E` を含まない浮動小数点定数と同じ構文を持つ記述を a として、整数定数と同じ構文を持つ記述を b とするとき、

a`e`b または a`E`b

という記述を書くと、それは、

$a \times 10^b$

を意味する浮動小数点定数になります。

それでは、`e` または `E` を含む浮動小数点定数の例を、いくつか書いてみましょう。

例	意味
`3e8`	3×10^8
`-7E6`	-7×10^6
`4e-7`	4×10^{-7}
`-9.2E24`	-9.2×10^{24}
`-3.01e-16`	-3.01×10^{-16}

2.5　練習問題

2.1　次の文字の列はアトムとして正しいですか。正しくないものについては、その理由も答えてください。

(a) `m`
(b) `M`
(c) `hitode`
(d) `Hitode`
(e) `_hitode`
(f) `ika/tako`
(g) `ika_tako`
(h) `ika tako`
(i) `ikaTako`
(j) `naoki35`
(k) `226jiken`
(l) `:=:=:`
(m) `>>8>>`
(n) `''`
(o) `'Hitode'`

[1] ドットの左側に数字のないものを浮動小数点定数として認めているプログラミング言語もたくさんありますが、Prolog では認められていません。

(p) `'ika tako'`

(q) `'No, I'm not.'`

(r) `''''`

(s) `[]`

2.2 次の文字の列は数値定数として正しいですか。正しくないものについては、その理由も答えてください。

(a) `87233`

(b) `-89471`

(c) `55090-`

(d) `0b101110101100`

(e) `0s30261`

(f) `-0x25ca`

(g) `0'=`

(h) `31.457`

(i) `4.38.06`

(j) `.29501`

(k) `81103.`

(l) `-87.504`

(m) `7e8`

(n) `-3e-6`

(o) `4.802e7`

(p) `6e4.3`

第3章　節

3.1　複合項

Q 3.1.1　複合項って何ですか。

A　「複合項」(compound term) というのは、2 個以上の項を組み合わせることによって作られた項のことです。

もう少し厳密に言うと、複合項というのは、1 個のアトム、1 個以上の項、丸括弧 (parenthesis, `()`)、そしてコンマ (comma, `,`) を、

アトム (項 , ···)

という形で組み合わせたもののことです。丸括弧の中には、コンマで区切って何個でも好きなだけ項を並べることができます（ただし 0 個というのはだめですが)。たとえば、

```
a(b)
a(b,c,d)
*=*(>@<,+/+)
'Prolog'(ebi,kani)
```

などは、正しい複合項の例です

Q 3.1.2　関数子って何のことですか。

A　「関数子」(functor) というのは、複合項の先頭に書かれているアトムのことです。

たとえば、

```
tango(warau,doushi,godan)
```

という複合項の関数子は、tango です。

Q 3.1.3　複合項の引数って何のことですか。

A　複合項の「引数」(argument) というのは、複合項の丸括弧の中に並べられているそれぞれの項のことです。

たとえば、

```
tango(taberu,doushi,shimoichidan)
```

という複合項の引数は、taberu と doushi と shimoichidan です。

Q 3.1.4　複合項の引数として数値定数を書くことって、可能ですか。

A　はい、可能です。

数値定数は項の一種ですので、数値定数を複合項の引数として書くことは、可能です。たとえば、

```
a(4037)
point(501,-7401)
object(6.08e17,2.33e-5,-4.18e12)
```

などは、正しい複合項の例です。

ちなみに、複合項の関数子というのは、かならずアトムでないといけませんので、数値定数を関数子とする複合項というものを書くことはできません。つまり、

```
8503(namako)
```

という記述は複合項にはならない、ということです。

Q 3.1.5　複合項の中にホワイトスペースを書くことって、可能ですか。

A　はい、可能です。

複合項を書くとき、それぞれの引数の前後には、ホワイトスペースを何個でも書くことができます。たとえば、

```
ningen(onna,36,sapporo)
```

という複合項は、

```
ningen(  onna  ,  36  ,  sapporo  )
```

と書くこともできますし、

```
ningen(
    onna,
    36,
    sapporo
)
```

と書くこともできますし、

```
ningen(onna,     % seibetsu
       36,       % nenrei
       sapporo)  % juusho
```

と書くこともできます。

ただし、複合項の関数子と左丸括弧とのあいだにはホワイトスペースを書くことができません。ですから、

```
ningen   (onna,36,sapporo)
```

という記述は、正しい複合項ではありません。

なお、複合項の中にホワイトスペースを書いたとしても、それによって複合項の意味が変化することはありません。

Q 3.1.6 複合項も項の一種だとすると、複合項の引数として複合項を書いてもかまわないんですか。

A はい、かまいません。

複合項というのは、項の一種です。したがって、複合項の引数として複合項を書くということも可能です。ですから、

```
a(b(c))
a(b(c), d(e), f(g))
a(b(c, d), e(f, g))
a(b(c(d(e(f)))))
```

などは、正しい複合項の例です。

Q 3.1.7 複合項の項数って何のことですか。

A 複合項の「項数」(arity) というのは、複合項に含まれている引数の個数のことです。

たとえば、

```
ningen(otoko, 64, akashi)
```

という複合項の項数は 3 です。

複合項の項数を数えるとき、引数として複合項が書かれている場合は、その引数の全体を 1 と数えます。ですから、

```
a1(b1(c1, c2, c3), b2(c4, c5, c6, c7))
```

という複合項の項数は 2 になります。7 ではないという点に注意してください。

Q 3.1.8 演算子記法って何ですか。

A 「演算子記法」(operator notation) というのは、項数が 1 または 2 の複合項を、丸括弧もコンマも使わないで書く書き方のことです。

演算子記法は、

- 前置記法 (prefix notation)
- 後置記法 (postfix notation)
- 中置記法 (infix notation)

という三種類に分類することができます。

前置記法と後置記法は、

functor (*argument*)

という項数が 1 の複合項の特別な書き方で、前置記法は、

functor *argument*

というように関数子を前に書き、後置記法は、

argument *functor*

というように関数子を後ろに書きます。そして、中置記法というのは、

functor (*argument1* , *argument2*)

という項数が 2 の複合項の特別な書き方で、

argument1 *functor* *argument2*

というように、関数子を中央に置いて、その左右に引数を書きます。

Q 3.1.9 項数が 1 または 2 の複合項って、どんなものでも演算子記法で書くことができるんですか。

A いいえ、項数が 1 または 2 の複合項のうちで、演算子記法で書くことができるのは、その関数子が「演算子」(operator) と呼ばれる特別なアトムであるものだけです。

種類	演算子							
前置	:-	?-	\+	-	\			
中置	:-	-->	=	\=	==	\==	@>	@<
	@>=	@=<	is	=:=	=\=	>	<	>=
	=<	=..	**	;	->	,	^	+
	-	/\	\/	*	/	//	rem	mod
	<<	>>						

表 3.1: 既定義演算子

演算子というのは、それを関数子とする複合項を演算子記法で書くことができるアトムのことです。

演算子は、どの演算子記法のためのものなのかということによって分類することができます。前置記法のための演算子は「前置演算子」(prefix operator) と呼ばれ、後置記法のための演算子は「後置演算子」(postfix operator) と呼ばれ、中置記法のための演算子は「中置演算子」(infix operator) と呼ばれます。ただし、ひとつの演算子が二つ以上の種類に分類されることもあります（たとえば、ひとつの演算子が前置演算子と中置演算子とを兼ねる、というような場合です）。

Q 3.1.10 既定義演算子って何ですか。

A 「既定義演算子」(predefined operator) というのは、処理系によって最初から演算子として使えるように設定されているアトムのことです。

ちなみに、ISO による Prolog の標準規格は、表 3.1 に示されているアトムを既定義演算子として定めています。

たとえば、\+ というアトムは前置演算子ですので、

```
\+(a)
```

という項数が 1 の複合項は、

```
\+ a
```

という前置記法で書くことも可能です。

また、= というアトムは中置演算子ですので、

```
=(a, b)
```

という項数が 2 の複合項は、

```
a = b
```

という中置記法で書くことも可能です。

なお、:- と - は、前置演算子として使うことも中置演算子として使うことも可能です。

Q 3.1.11 既定義演算子じゃないアトムを演算子にすることって、可能ですか。

A はい、既定義演算子ではないアトムを演算子にすることは可能です。

既定義演算子以外のアトムについても、プログラムの中にそのための記述を書くことによって、演算子にすることができます。

アトムを演算子にするための記述の具体的な書き方については、Q 8.6.1 を参照してください。

Q 3.1.12 複合項を演算子記法で書くとき、演算子と引数とのあいだには、かならずホワイトスペースを入れる必要があるんですか。

A 演算子と引数とのあいだのホワイトスペースは、必要な場合と必要ではない場合とがあります。

演算子と引数との間にホワイトスペースが必要なのは、もしもホワイトスペースがなかったとすると、演算子と引数との境目がわからなくなってしまう場合です。

複合項を演算子記法で書くとき、英数字アトムと英数字アトムが連続したり、非英数字アトムと非英数字アトムが連続したりする場合は、それらのあいだを 1 個以上のホワイトスペースで区切らないと、複数のアトムがつながってひとつになってしまいます。たとえば、`hikkoshi` というアトムが中置演算子だとするとき、

```
hikkoshi(matsuyama, utsunomiya)
```

という複合項を中置記法で書く場合、ホワイトスペースで区切らなければ、

```
matsuyamahikkoshiutsunomiya
```

というひとつのアトムになってしまいますので、この場合はホワイトスペースが不可欠です。それに対して、`===>` というアトムが中置演算子だとするとき、

```
===>(matsuyama, utsunomiya)
```

という複合項を中置記法で書く場合、ホワイトスペースで区切らないで、

```
matsuyama===>utsunomiya
```

と書いたとしても、演算子と引数との境目を識別することが可能ですので、問題はありません。

3.2 演算子の優先順位と結合性

Q 3.2.1 演算子の優先順位って何のことですか。

A 演算子の「優先順位」(priority, precedence) というのは、それぞれの演算子が持っている、自分と引数とが結合する強さの順位という属性のことです。

演算子の優先順位は、2 個以上の演算子を含む複合項を演算子記法で書いたものを解釈するために使われます。たとえば、$\circ$ と $\bullet$ が中置演算子だとするとき、

$$A \circ B \bullet C$$

という複合項は、

$$\bullet(\circ(A, B), C)$$

と解釈するべきなのか、それとも、

$$\circ(A, \bullet(B, C))$$

と解釈するべきなのかというのは、$\circ$ と $\bullet$ のそれぞれが持っている優先順位によって決定されます。

演算子の優先順位は、1 個のプラスの整数によって示されます。その整数が小さいほど、優先順位が高いということ、つまり演算子と引数とが強く結合するということを示します。

もしも、$\circ$ の優先順位が 500 で、$\bullet$ の優先順位が 1000 だとすると、$\circ$ のほうが $\bullet$ よりも左右の項と強く結合しますので、

$$A \circ B \bullet C$$

という複合項は、

$$\bullet(\circ(A, B), C)$$

と解釈されることになります。それとは逆に、$\circ$ の優先順位が 1000 で、$\bullet$ の優先順位が 500 だとすると、$\bullet$ のほうが $\circ$ よりも左右の項と強く結合しますので、上の複合項は、

$$\circ(A, \bullet(B, C))$$

と解釈されることになります。

表 3.2 は、既定義演算子の優先順位を示しています。なお、`-` の優先順位は、中置演算子の場合は 500 で、前置演算子の場合は 200 です。

Q 3.2.2 演算子の結合性って何のことですか。

A 演算子の「結合性」(associativity) というのは、それぞれの演算子が持っている属性のひとつで、同じ優先順位を持つ演算子が自分の左右にあるときに複合項をどのように解釈するべきかということを決定する属性です。

優先順位	演算子							
1200	`:-`	`?-`	`-->`					
1100	`;`							
1050	`->`							
1000	`,`							
900	`\+`							
700	`=`	`\=`	`==`	`\==`	`@>`	`@<`	`@>=`	`@=<`
	`is`	`=:=`	`=\=`	`>`	`<`	`>=`	`=<`	`=..`
600	`:`							
500	`+`	`-`	`/\`	`\/`				
400	`*`	`/`	`//`	`rem`	`mod`	`<<`	`>>`	
200	`**`	`^`	`\`	`-`				

表 3.2: 既定義演算子の優先順位

同一の優先順位を持つ 2 個以上の演算子を含む複合項を演算子記法で書いたものを解釈するときには、それらの演算子が持っている結合性が使われます。たとえば、$\circ$ と $\bullet$ が中置演算子で、それらの優先順位がどちらも 500 だとすると、

$A \circ B \bullet C$

という複合項を解釈するためには、それらの演算子が持っている結合性が必要になります。

演算子は、自分の結合性として、

- 左結合性 (left-associativity)
- 右結合性 (right-associativity)
- 無結合性 (non-associativity)

のいずれかを持っています。

$\circ$ と $\bullet$ が同一の優先順位を持つ中置演算子だとするとき、

$A \circ B \bullet C$

という複合項は、$\circ$ と $\bullet$ の両方が左結合性を持っているとすると、

$\bullet(\circ(A, B), C)$

と解釈されます。それに対して、$\circ$ と $\bullet$ の両方が右結合性を持っているとすると、

$\circ(A, \bullet(B, C))$

と解釈されます。そして、$\circ$ と $\bullet$ の両方が無結合性を持っているとすると、文法的なエラーとみなされます。

Q 3.2.3 演算子の記述子って何のことですか。

A 演算子の「記述子」(specifier) というのは、演算子の種類（前置、中置、後置）と結合性との組み合わせをあらわしているアトムのことです。

演算子の記述子としては、次の 7 種類のものがあります。

記述子	種類	結合性
`fx`	前置	無結合性
`fy`	前置	右結合性
`xfx`	中置	無結合性
`xfy`	中置	右結合性
`yfx`	中置	左結合性
`xf`	後置	無結合性
`yf`	後置	左結合性

記述子	演算子							
fx	:-	?-						
fy	\+	-	\					
xfx	:-	-->	=	\=	==	\==	@>	@<
	@>=	@=<	is	=:=	=\=	>	<	>=
	=<	=..	**					
xfy	;	->	,	^				
yfx	+	-	/\	\/	*	/	//	rem
	mod	<<	>>					

表 3.3: 既定義演算子の記述子

ちなみに、記述子で既定義演算子を分類すると、表 3.3 のようになります。

Q 3.2.4 2 個以上の演算子を含んでいる演算子記法の複合項を、優先順位とか結合性とかで解釈するんじゃなくて、プログラムを書いた人が意図したとおりに解釈してほしいときって、どうすればいいんですか。

A そんなときは、ひとつの複合項だと解釈してほしい部分を丸括弧 (parenthesis, `()`) で囲みます。

項を丸括弧で囲んだものは、ひとつの項になります。そしてそれは、その中の項と同じものだとみなされます。たとえば、`a` というアトムを丸括弧で囲んだ `(a)` という記述はひとつの項になって、それは `a` というアトムと同じものだとみなされます。

演算子記法で書かれた複合項の一部分を丸括弧で囲むと、左丸括弧から右丸括弧までの部分は、演算子の優先順位や結合性とは無関係に、ひとつの項だとみなされます。

$\circ$ と $\bullet$ が中置演算子で、$\bullet$ のほうが $\circ$ よりも優先順位が高いとすると、

$$A \circ B \bullet C$$

という複合項は、

$$\circ(A, \bullet(B, C))$$

と解釈されるわけですが、

$$(A \circ B) \bullet C$$

というように、複合項の一部分を丸括弧で囲むと、優先順位とは無関係に、

$$(A \circ B)$$

という部分がひとつの項だとみなされますので、

$$\bullet(\circ(A, B), C)$$

と解釈されることになります。

3.3 事実と規則の書き方

Q 3.3.1 Prolog では、命題はどんなふうに書けばいいんですか。

A Prolog では、ひとつの命題はひとつの項によって記述されます。

項数がゼロの述語を使った命題は、その述語に名前として与えられたアトムによって記述されます。たとえば、「いい天気である」という項数がゼロの述語を `iitenki` というアトムであらわすとすれば、その述語を使った「いい天気である」という命題は、`iitenki` というアトムのみによって記述することができます。

項数が 1 以上の述語を使った命題は、述語の名前が関数子で、述語が適用される対象が引数であるような複合項によって記述されます。たとえば、「伸之は聡子の父親である」という命題は、述語と引数のそれぞれを、

父親である chichioya
伸行 nobuyuki
聡子 satoko

であらわすとすれば、

```
chichioya(nobuyuki, satoko)
```

という複合項によって記述することができます。
引数の順番は、プログラムを書く人が自由に決めてかまいませんが、ひとつのプログラムの中で首尾一貫している必要があります。「誰々は誰々の父親である」という命題を書く場合に、上の例のように 1 個目は父親で 2 個目は子供と決めたならば、同じプログラムの中ではその順番を守らないといけません。

Q 3.3.2 Prolog では、事実はどんなふうに書けばいいんですか。

A Prolog では、事実を記述したいときは、まず命題をあらわす項を書いて、その右側にドット (dot, .) という文字を書いて、そしてそのさらに右側に 1 個以上のホワイトスペースを書きます。

つまり、事実というのは、Prolog では、

項	.	1 個以上のホワイトスペース

という形のものによってあらわされる、ということです。たとえば、「仲之は聡子の父親である」という命題を、事実として Prolog のプログラムの中に記述したいというときは、その命題をあらわす項の右側に 1 個のドットと 1 個以上のホワイトスペースを書いたもの、つまり、

```
chichioya(nobuyuki, satoko).
```

という形のものを書けばいいわけです。

Q 3.3.3 Prolog では、規則はどう書けばいいんですか。

A Prolog では、規則を記述したいときは、まず :- というアトムを関数子とする項数が 2 の複合項を書いて、その右側にドットを書いて、そしてそのさらに右側に 1 個以上のホワイトスペースを書きます。

A と B が命題をあらわしている項だとするとき、「A ならば B」という規則を、Prolog では、

```
:-(B, A).
```

と書きます。 :- というアトムは中置演算子ですので、この規則は、

```
B :- A.
```

と書くこともできます。いずれの場合も、日本語とは語順が逆になるという点に注意が必要です。
たとえば、「増雄が働き者ならば恵子は幸せである」という規則は、述語と引数のそれぞれを、

働き者である hatarakimono
幸せである shiawase
増雄 masuo
恵子 keiko

であらわすとすれば、

```
shiawase(keiko) :- hatarakimono(masuo).
```

と記述することができます。

Q 3.3.4 規則の頭部とか本体とかっていうのは何のことですか。

A 規則の「頭部」(head) というのは、規則をあらわしている複合項の 1 個目の引数のことで、規則の「本体」(body) というのは、規則をあらわしている複合項の 2 個目の引数のことです。

つまり、規則を演算子記法で書いた場合、その規則の頭部というのは `:-` の左側の項のことで、本体というのは `:-` の右側の項のことだということです。たとえば、

```
shiawase(keiko) :- hatarakimono(masuo).
```

という規則では、`shiawase(keiko)` が頭部で `hatarakimono(masuo)` が本体です。

Q 3.3.5 節って何ですか。

A 「節」(clause) というのは、事実または規則をあらわしている記述のことです。

Q 2.1.8 で述べられているように、Prolog のプログラムというのは、いくつかの事実または規則を記述したもののことです。事実または規則をあらわしている記述は「節」と呼ばれるわけですから、Prolog のプログラムはいくつかの節から構成されることになります。

Prolog のプログラムは基本的には節から構成されるわけですが、節だけではなくて、「命令」(directive) と呼ばれるものをプログラムに中に書くこともできます。命令については、Q 8.6.3 を参照してください。

Q 3.3.6 述語指示子って何ですか。

A 「述語指示子」(predicate indicator) というのは、特定の述語を指示するために書かれる、述語の名前と項数とを組み合わせた記述のことです。

Q 2.1.12 で説明したように、述語の名前は同じだけれども項数が異なる、といういくつかの述語は、それぞれ異なる述語だとみなされます。ですから、Prolog に関する文章や Prolog のプログラムの中で特定の述語に言及する場合には、名前と項数とを組み合わせた記述、すなわち述語指示子が必要になります。

述語指示子は、述語名とスラッシュ(slash, `/`) と項数をこの順序で並べることによって作られます。つまり、

述語名 `/` 項数

というように書くわけです。たとえば、`bagof/3` というのは名前が `bagof` で項数が 3 の述語という意味で、`halt/0` というのは名前が `halt` で項数が 0 の述語という意味です。

非英数字アトムを名前にしている述語を指示する述語指示子を書く場合は、その名前とスラッシュとが何らかの方法で分離されていないと、人間はそれを正しく解釈できたとしても、Prolog の処理系はそれを正しく解釈することができません。たとえば、`=` という名前で項数が 2 の述語を指示するために `=/2` という記述を書いたとすると、Prolog の処理系はそれを、`=/` というアトムの右側に `2` という数値定数を書いたものだと解釈してしまいます。

非英数字アトムとスラッシュとを分離するための書き方としては、`(=)/2` のように丸括弧 (parenthesis, `()`) を使う書き方や、`'='/2` のように一重引用符 (single quote, `'`) を使う書き方などがあります。ちなみに、このチュートリアルでは丸括弧を使う書き方を採用しています。

Q 3.3.7 述語定義っていうのは、どう書けばいいんですか。

A 定義したい述語に関する節をいくつか並べて書けば、その全体がひとつの述語定義になります。

定義したい述語に関する節というのは、その述語を使った事実をあらわす節か、または、その述語を使った命題を頭部とする規則をあらわす節のことです。たとえば、

`hoge(` *argument1* `,` *argument2* `).`

という形の事実をあらわす節か、または、

`hoge(` *argument1* `,` *argument2* `) :-` *body* `.`

という形の規則をあらわす節をいくつか並べて書けば、その全体は、項数が 2 の `hoge` という述語（つまり `hoge/2`）を定義する述語定義になります。

Q 3.3.8 規則の本体の中に「*A* かつ *B*」という形の命題を書きたいときは、どうすればいいんですか。

A そんなときは、コンマ (comma, `,`) という中置演算子を使います。

「A かつ B」、つまり A と B の両方が真であるという命題は、

A，B

という複合項によってあらわされます。たとえば、「君香がしとやかでかつ鈴香がおてんばならば、正晴は幸せである」という規則は、

```
shiawase(masaharu) :- shitoyaka(kimika), otenba(suzuka).
```

という節を書くことによって記述することができます。

ちなみに、コンマを関数子とする複合項を、演算子記法ではない書き方で書く場合は、その関数子を、’,’という引用アトムにする必要があります。つまり、

A，B

という複合項を、演算子記法ではない書き方で書くためには、

’,’(A，B)

と書く必要がある、ということです。

Q 3.3.9　規則の本体の中に「A かつ B かつ C」という形の命題を書きたいときは、どうすればいいんですか。

A　「A かつ B かつ C」という形の命題を書きたいときは、A、B、C のそれぞれをコンマで連結した形の複合項を書きます。

「A かつ B かつ C」、つまり A と B と C がすべて真であるという命題は、

A，B，C

という複合項によってあらわされます。これは、

’,’(A，’,’(B，C))

という複合項を演算子記法で記述したものです。

同じように、n 個の命題がすべて真であるという命題、つまり、「A_1 かつ A_2 かつ･･･かつ A_n」という命題も、

A_1，A_2，･･･，A_n

という複合項によって記述することができます。

Q 3.3.10　ゴールって何ですか。

A　「ゴール」(goal) というのは、証明することが要請される命題をあらわしている項のことです。

「A ならば B」という規則は、「A を証明すれば、B を証明したことになる」という意味だと解釈することができます。たとえば、「増雄が働き者ならば恵子は幸せである」という規則は、「増雄が働き者だということを証明すれば、恵子が幸せだということを証明したことになる」という意味です。

したがって、「A ならば B」の A、すなわち規則の本体というのは、証明することが要請される命題をあらわしている項、すなわちゴールだと考えることができます。

ゴールが、「かつ」を意味するコンマによって何個かの項を結合した形になっている場合、コンマによって結合されているそれぞれの項もゴールです。なぜなら、その全体を証明するためには、「かつ」によって結合されたすべての命題を証明することが要請されるからです。

Q 3.3.11　ゴールを実行するっていうのはどういうことですか。

A　ゴールの証明を試みることを、そのゴールを「実行する」(execute) と言います。

ゴールを実行するというのは、ゴールを証明することではなくて、ゴールの証明を試みることです。つまり、ゴールを実行したとしても、かならずしもそのゴールが証明されるとは限らないわけです。ゴールが証明されることを、ゴールが「成功する」(succeed) と言います。それに対して、ゴールの証明を試みたけれども証明することができなかったということを、ゴールが「失敗する (fail)」と言います。

Q 3.3.12 述語を呼び出すっていうのはどういうことですか。

A 何らかの述語を使った命題の証明を試みることを、その述語を「呼び出す」(call) と言います。

たとえば、

```
hatarakimono(masuo)
```

というゴールを実行することは、hatarakimono という述語を使った命題の証明を試みることですから、このことを、「hatarakimono を呼び出す」と表現することができます。

3.4 変数

Q 3.4.1 変数って何ですか。

A 「変数」(variable) というのは、「何でもかまわない何か」をあらわす項のことです。

Q 3.4.2 変数っていうのは、どんなふうに書けばいいんですか。

A 変数は、英字、数字、アンダースコア (underscore, _) から構成される列です。ただし、先頭の 1 文字は、英字の大文字かまたはアンダースコアでないといけません。

たとえば、

```
A  Namako  Kusetsu30nen  Tadashii_hensuu  _a  _5
```

などは正しい変数の例です。

Q 3.4.3 変数を使うと、どんなことができるんですか。

A 変数を使うことによって、一般的な規則を記述することができます。

「増雄が働き者ならば恵子は幸せである」という規則は、それほど一般的なものではありませんので、変数を使わなくても記述することができます。しかし、もっと一般的な規則を記述するためには、変数が必要になります。

たとえば、「それがカラスならばそれは黒い」という規則は、かなり一般的な規則です。なぜなら、この規則は特定のカラスについてだけ成り立つものではなくて、すべてのカラスについて成り立つからです。

「それがカラスならばそれは黒い」という規則をあらわす文に含まれている「それ」という代名詞は、特定のものではなくて、「何でもかまわない何か」を意味しています。そして、その文の中に含まれている二つの「それ」は、どちらも同じものをあらわしていると解釈されます。

変数も、一般的な規則をあらわす文の中で使われる「それ」という代名詞と同じ機能を持っています。すなわち、変数は「何でもかまわない何か」を意味していて、ひとつの節の中に出現する何個かの同じ変数は、すべて同じものをあらわしていると解釈されるのです。

ですから、「カラスである」という述語を karasu というアトムで、「黒い」という述語を kuroi というアトムであらわすとすれば、「それがカラスならばそれは黒い」という規則は、

```
kuroi(X) :- karasu(X).
```

という節で書きあらわすことができます。

なお、Q 2.1.4 で、「規則を自然言語で表現すると、それは、二つの文を「ならば」(if) という接続詞で組み合わせた形の文になる」と書きましたが、自然言語では、一般的な規則は「ならば」を使わない文であらわすのが普通です。たとえば、「それがカラスならばそれは黒い」という規則は、普通、「カラスは黒い」という文によってあらわされます。

Q 3.4.4 変数が、「何でもかまわない何か」というひとつの意味しか持たないものだとすると、なぜ、異なる変数をいくつも作ることができるようになってるんですか。

A いくつかの異なる変数を使うことによって、いくつかの異なる「何でもかまわない何か」を扱う規則を記述することができるからです。

ひとつの規則の中で扱われる「何でもかまわない何か」は、ひとつだけとは限りません。「何でもかまわない何か」がひとつの規則の中に 2 個以上含まれている場合、それぞれの「何でもかまわない何か」は、それぞれに異なっていてもかまいません。

ひとつの節の中には、いくつかの異なる変数を書くことができます。ひとつの節の中に出現する異なる変数は、異なるものをあらわすことができます。

いくつかの異なる変数を必要とする規則の例として、「子供の子供は孫である」という規則について考えてみましょう。この規則は、「子供である」という述語を kodomo というアトムで、「孫である」という述語を mago というアトムであらわすとすれば、

```
mago(C, A) :- kodomo(B, A), kodomo(C, B).
```

という節で書きあらわすことができます。この節に出現する、A、B、C という変数のそれぞれは、異なったものをあらわしていてもかまわないと解釈されます。

Q 3.4.5　変数の有効範囲って何のことですか。

A　変数の「有効範囲」(scope) というのは、同じ変数が同じものを意味する、プログラムの字句の上での範囲のことです。

Q 3.4.3 で説明したように、ひとつの節の中に何個かの同じ変数が出現する場合、それらの変数は、「何でもかまわないけれどもすべて同じもの」を意味しています。

しかし、同じ変数が同じものを意味していると解釈されるのは、節という字句の範囲の中だけのことです。つまり、いくつかの節のそれぞれに同じ変数が使われている場合、それぞれの変数は、それぞれの節ごとに違ったものを意味していてもかまわないということです。たとえば、

```
kuroi(X) :- karasu(X).
shiroi(X) :- kamome(X).
```

という二つの節の両方に含まれている X という変数は、節ごとに違ったものを意味していてもかまわないわけです。

このように、同じ変数が同じものを意味しているという解釈の有効性には、節という範囲があります。この範囲のことを、変数の「有効範囲」と呼ぶわけです。

Q 3.4.6　匿名変数って何ですか。

A　「匿名変数」(anonymous variable) というのは、アンダースコア (underscore, _) という文字を 1 個だけ書くことによってでできる変数のことです。

匿名変数、つまり _ という変数は、それ以外の変数とは機能が少し異なっています。普通の変数と匿名変数とで共通しているのは、それが「何でもかまわない何か」を意味しているという点だけです。

普通の変数と匿名変数とで違っているところは、ひとつの節の中に同じ変数がいくつか含まれている場合に、それらがどのように解釈されるか、という点にあります。

普通の変数に関しては、同じ節の中にある同じ変数は同じものを意味していると解釈されるわけですが、匿名変数に関しては、異なるものを意味していてもかまわないと解釈されるのです。

普通の変数は、ひとつの節の中に同じものを何個か書くことによって、「何でもかまわないけれども、こことここは同じものを意味している」ということを示すために使われます。しかし、しばしば、「こことここは同じものを意味している」ということを示す必要がなくて、ただ単に「何でもかまわない」ということだけを示すことができさえすればそれでいい、という場合もあります。そのような場合には、普通の変数を使っても間違いではありませんが、匿名変数を使うことによって、「別の場所に同じ意味のものは出現しない」ということを積極的に示すことができます。

たとえば、「何かを愛している者は幸せである」という規則を Prolog の節として記述することを考えてみましょう。この規則は、「愛している」という述語を aisuru というアトムで、「幸せである」という述語を shiawase というアトムであらわすとすれば、

```
shiawase(X) :- aisuru(X, Y).
```

という節で書きあらわすことができます。しかし、この節の中に書かれている Y という変数は、「こことここは同じものを意味している」ということを示すために使われているわけではありません。

このような場合は、普通の変数ではなくて匿名変数を使うことによって、「別の場所に同じ意味のものは出現しない」ということを積極的に示したほうがいいでしょう。つまり、

```
shiawase(X) :- aisuru(X, _).
```

と書くほうがいい、ということです。

同じように、「友達を持っていて、かつ子供を持っている者は幸せである」という規則は、「友達である」という述語を tomodachi というアトムで、「子供である」という述語を kodomo というアトムで、「幸せである」という述語を shiawase というアトムであらわすとすれば、

```
shiawase(X) :- tomodachi(X, _), kodomo(_, X).
```

という節で書きあらわすことができます。この節の中には 2 箇所に匿名変数が書かれていますが、同じ節の中に書かれている匿名変数のそれぞれは、同じものを意味しているとは限らないと解釈されます。

3.5 練習問題

3.1 次の文字の列は複合項として正しいですか。正しくないものについては、その理由も答えてください。

(a) `a(b)`
(b) `a[b]`
(c) `a(7)`
(d) `7(b)`
(e) `a()`
(f) `a(b,c,d)`
(g) `a(5,3,2)`
(h) `a(b(c),d(e),f(g))`
(i) `a(b(c,d(e,f(g,h))`
(j) `a(b,c)(d,e,f)`
(k) `a(b , c , d)`
(l) `a (b,c,d)`

3.2 既定義演算子を使った次の複合項を、演算子記法ではない複合項に書き換えてください。もしも文法的なエラーになる場合は、そのことを指摘してください。なお、既定義演算子の優先順位と記述子は表 3.2 と表 3.3 のとおりだとします。

(a) `a + b`
(b) `a + b * c`
(c) `a / b - c`
(d) `a / b * c`
(e) `a - b + c`
(f) `a < b < c`
(g) `- a`
(h) `- a + b`
(i) `(a + b) * c`
(j) `a / (b - c)`
(k) `a - (b + c)`
(l) `- (a + b)`

3.3 次の文字の列は変数として正しいですか。正しくないものについては、その理由も答えてください。

(a) `m`
(b) `M`
(c) `hitode`
(d) `Hitode`

(e) `_hitode`
(f) `Ika/Tako`
(g) `Ika_Tako`
(h) `Ika Tako`
(i) `IkaTako`
(j) `Naoki35`
(k) `226Jiken`
(l) `'Hitode'`

3.4　次の事実をあらわす節を書いてください。

(a) 秀吉は関白である。
(b) 富士山は火山である。
(c) 浩則は美穂子を愛している。
(d) 知美の趣味は編物である。
(e) オーストラリア (Australia) の首都はキャンベラ (Canberra) である。
(f) 晴彦は加奈子からノートを借りている。
(g) 今日子は智弘のことをトモタンと呼んでいる。

3.5　次の規則をあらわす節を書いてください。

(a) 正雄が元気ならば貴子は幸せである。
(b) 和幸が博美を愛しているならば博美は幸せである。
(c) 茂雄が大富豪でかつ茂雄が真理子を愛しているならば、真理子は幸せである。
(d) 男はつらい。
(e) 固体の水は氷である。
(f) 子供の子供の子供は曾孫である。
(g) 何か趣味を持っている者は幸せである。
(h) 誰かを愛していてかつ誰かに愛されている者は幸せである。

第 4 章　Prolog のインタプリタ

4.1　インタプリタの使い方の基礎

Q 4.1.1　Prolog のインタプリタは、どうやって起動すればいいんですか。

A　Prolog のインタプリタを起動する方法は、インタプリタのそれぞれの実装ごとに異なりますので、インタプリタに付属している文書を参照してください。

Q 4.1.2　Prolog のインタプリタを起動したら、何行かのメッセージを表示したのちに停止してしまいました。これはどういう状態なんですか。

A　それは、人間による質問の入力を待っている状態です。

Prolog のインタプリタは、起動すると、自分の名前やバージョンや著作権表示などのメッセージを表示したのち、プロンプトを出力して、人間によって質問が入力されるのを待ちます。なお、プロンプトとしてどのような文字列を出力するかというのは、インタプリタの実装ごとに異なります。

Prolog のインタプリタは、対話型のプログラムです。それは、

(1) プロンプトを出力する。
(2) 質問を読み込む。
(3) 質問の証明を試みる。
(4) 質問の証明が成功したかどうかを出力する。

ということを何回も繰り返すように作られています。

Q 4.1.3 質問って、どんなふうに入力すればいいんですか。

A 質問を入力したいときは、まず、?- というアトムが関数子で、ゴールが引数であるような、項数が 1 の複合項を入力して、次に 1 個のドット (.) を入力して、最後に改行を入力します。

A が項だとするとき、

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

という質問をインタプリタに入力すると、インタプリタは A をゴールとみなして、その証明を試みます。たとえば、

```
?-(shiawase(keiko)).
```

という質問を入力することによって、 shiawase(keiko) というゴールの証明をインタプリタに試みさせることができます。

?- というアトムは前置演算子ですので、質問は、

```
?- A.
```

という形で入力することもできます。ですから、先ほどの質問の例は、

```
?- shiawase(keiko).
```

と入力してもかまいません。

なお、インタプリタの実装のうちには、プロンプトの末尾に ?- という文字列を含んでいるものもあります。そのような実装では、質問として入力する必要があるのは、インタプリタに証明させたいゴールだけです。

Q 4.1.4 インタプリタは、質問の証明が成功したかどうかを、どのような形で出力するんですか。

A インタプリタは、質問の証明が成功した場合は yes を出力して、失敗した場合は no を出力します。

Q 4.1.5 組み込み述語って何ですか。

A 「組み込み述語」(built-in predicate) というのは、Prolog の処理系の中に最初から組み込まれている述語のことです。

述語は、基本的にはプログラムの中に書かれた述語定義によって作られるわけですが、Prolog の処理系には、述語定義を与えなくても使うことのできる多数の述語が組み込まれていて、それらの述語は「組み込み述語」と呼ばれます。

Q 4.1.6 組み込み述語って、たとえばどんなものがあるんですか。

A 組み込み述語の例としては、たとえば atom/1 というのがあります。この述語は、引数がアトムならば成功して、そうでなければ失敗します。

たとえば、

```
?- atom(namako).
```

という質問をインタプリタに入力すると、 namako はアトムですので証明は成功して、 yes と出力されます。それに対して、

```
?- atom(a(b)).
```

という質問をインタプリタに入力すると、 a(b) はアトムではなくて複合項ですので証明は失敗して、 no と出力されます。

Q 4.1.7 Prolog のインタプリタを終了させたいときはどうすればいいんですか。

A Prolog のインタプリタを終了させたいときは、 halt/0 という組み込み述語を呼び出します。

つまり、

```
?- halt.
```

という質問をインタプリタに入力することによって、インタプリタを終了させることができる、ということです。

Q 4.1.8　質問を入力するとき、ドットを入力するのを忘れて改行を入力したら、インタプリタの動作が停止してしまいました。こんなときはどうすればいいんですか。

A　そんなときは、ドットを入力してから、もう一度、改行を入力してください。

ドットを入力するのを忘れて改行を入力した場合、インタプリタの動作が停止してしまったかのような状態になりますが、それはただ単に、質問の入力の続きを待っているだけです。ですから、ドットと改行を入力すれば、質問の入力を完了させることができます。

インタプリタは、質問の入力の途中で改行が入力されたとしても、ドットと改行が入力されるまでは、まだ質問の入力が続いていると判断します。ですから、ひとつの質問を何行かに分けて入力することも可能です。つまり、

```
?- atom(a(b)).
```

という質問を、

```
?- atom(
    a(
        b
    )
)
.
```

と入力してもかまわないわけです。

Q 4.1.9　質問の中に「A かつ B」という形の命題を書きたいときは、どうすればいいんですか。

A　そんなときは、規則の本体を書く場合と同じように、コンマ (,) という中置演算子を使います。

つまり、

?- A, B.

という質問を入力することによって、「A かつ B」という形の命題の証明をインタプリタに試みさせることができます。たとえば、

```
?- atom(:==:), atom('X').
```

という質問を入力すると、インタプリタは、「`:==:` はアトムでありかつ `'X'` もアトムである」という命題の証明を試みます。

同じように、

?- A_1, A_2, $\cdots$, A_n.

という質問を入力することによって、「A_1 かつ A_2 かつ$\cdots$かつ A_n」という命題の証明をインタプリタに試みさせることができます。

4.2　プログラムの実行

Q 4.2.1　Prolog のプログラムって、どんなソフトを使って書けばいいんですか。

A　Prolog のプログラムは、「テキストエディター」(text editor) と呼ばれるソフトを使って書きます。

テキストエディター（単に「エディター」(editor) と呼ばれることもあります）というのは、テキストデータを書いたり修正したりするためのソフトのことです。Prolog のプログラムというのはテキストデータですので、テキストエディターを使うことによって、書いたり修正したりすることができます。

Q 4.2.2　Prolog のプログラムをファイルに保存する場合、ファイル名の拡張子は何にすればいいんですか。

A Prolog のプログラムを保存するファイルには、`.pl` という拡張子を持つファイル名を付けます。

Q 4.2.3 データベースって何ですか。

A Prolog のインタプリタの内部にある、実行の対象となるプログラムが置かれる場所は、「データベース」(database) と呼ばれます。

「データベース」という言葉は、通常は、「系統的に管理できる形で蓄積されたデータ」というような意味で使われますが、Prolog のインタプリタに関連して使われる「データベース」という言葉は、そのような通常の意味とは少し違っていて、データではなくて、データが置かれる場所を意味しています。

Q 4.2.4 プログラムをコンサルトするっていうのはどういうことですか。

A プログラムを「コンサルトする」(consult) というのは、Prolog のインタプリタがファイルからプログラムを読み込んで、それをデータベースに置くことです。

Prolog のインタプリタを使ってプログラムを実行するためには、まず、そのプログラムをコンサルトする必要があります。そののち、インタプリタに質問を入力すると、インタプリタは、データベースの中にあるプログラムを使って、その質問の証明を試みます。

Q 4.2.5 プログラムをコンサルトしたいときはどうすればいいんですか。

A プログラムをコンサルトしたいときは、`consult/1` という組み込み述語を使います。

`consult/1` を呼び出すゴールには、引数として、Prolog のプログラムが格納されているファイルのパス名をあらわすアトムを書きます。そうすると、`consult/1` は、そのパス名で指定されたファイルからプログラムを読み込んで、そのプログラムをデータベースに置きます。たとえば、

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

という質問を入力することによって、`namako.pl` というパス名で指定されるファイルの内容をコンサルトすることができます。

それでは、実際にプログラムをコンサルトしてみましょう。

`shiawase.pl` というファイルに、次のようなプログラムが保存されているとします。

```
shiawase(keiko) :- hatarakimono(masuo).
hatarakimono(masuo).
```

このプログラムのファイルがあるのと同じディレクトリがカレントディレクトリになっていると仮定すると、Prolog のインタプリタに、

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

という質問を入力することによって、このプログラムをコンサルトすることができます。ですから、そののち、

```
?- shiawase(keiko).
```

という質問を入力すれば、インタプリタはその証明を試みて、その結果として `yes` を出力します。

Q 4.2.6 データベースの内容を調べたいときは、どうすればいいんですか。

A データベースの内容を調べたいときは、`listing/0` または `listing/1` という組み込み述語を使います。

`listing/0` は、データベースの中にあるプログラムをすべてモニターに出力する述語です。

それに対して、`listing/1` は、特定の述語の定義だけを出力する述語です。P がアトムだとするとき、`listing(`P`)` というゴールを実行すると、データベースの中にあるプログラムのうちで、P を述語名とする述語の定義だけが出力されます。たとえば、

```
?- listing(namako).
```

という質問を入力することによって、namako という述語の定義だけを出力させることができます。項数が異なる何個かの namako が存在する場合は、それらの述語の定義がすべて出力されます。

listing/1 を呼び出すゴールの引数として、アトムではなくて述語指示子を書くことによって、述語名と項数を限定して述語の定義を出力させることができます。たとえば、

```
?- listing(namako/2).
```

という質問を入力することによって、項数が 2 の namako の定義だけを出力させることができます。

Q 4.2.7　文字列をモニターに出力したいときは、どうすればいいんですか。

A　文字列をモニターに出力したいときは、write/1 という組み込み述語を使います。

write/1 は、引数をモニターに出力する述語です。引数は、どんな項でもかまいません。たとえば、

```
?- write(a(b)).
```

という質問をインタプリタに入力すると、a(b) という複合項が出力されます。

write/1 を呼び出すゴールの引数として引用アトムを書くことによって、任意の文字列をモニターに出力させることができます。たとえば、

```
?- write('vita brevis, ars longa.').
```

という質問を入力することによって、

```
vita brevis, ars longa.
```

という文字列を出力させることができます。

Q 4.2.8　コンマで結合されたそれぞれのゴールは、どんな順番で実行されるんですか。

A　コンマで結合されたそれぞれのゴールは、基本的には、左から右へという順番で実行されます。

ですから、

```
?- write(a), write(b), write(c).
```

という質問をインタプリタに入力すると、abc という文字列が出力されることになります。

規則の本体が、コンマで結合された何個かのゴールから構成されている場合も、それぞれのゴールは、基本的には左から右へという順番で実行されます。ですから、

```
order :- write(c), write(a), write(t).
```

という規則によって定義される order/0 という述語を呼び出すと、cat という文字列が出力されます。

Q 4.2.9　改行をモニターに出力したいときは、どうすればいいんですか。

A　改行をモニターに出力したいときは、nl/0 という組み込み述語を使います。

たとえば、

```
?- write(namako), nl, write(hitode).
```

という質問をインタプリタに入力すると、

```
namako
hitode
```

という文字列が出力されます。

ちなみに、改行だけではなくて、改行を含む文字列をモニターに出力したいときは、わざわざ nl/0 を呼び出すよりも、改行をあらわすエスケープシーケンス（Q 2.3.7 参照）を使うほうが自然です。たとえば、

```
write('Take care of the sense.\n')
```

と書くことによって、末尾に改行を含んでいる文字列をモニターに出力することができます。

Q 4.2.10 hello, worldって何ですか。

A hello, world というのは、プログラミング言語の解説書の冒頭などに掲載される、きわめて単純な動作をするプログラムのことです。

プログラミング言語の解説書の多くは、プログラムの最初の例として、

'Hello, world.\n' という文字列をモニターに出力する。

というような単純な動作をするものを掲載しています。そのようなプログラムは、hello, world と呼ばれます。

Prolog で hello, world を書くと、たとえば次のようなものになります。

```
hello :- write('Hello, world.\n').
```

このプログラムをコンサルトして、

```
?- hello.
```

という質問で実行すると、

```
Hello, world.
```

という文字列がモニターに出力されます。

4.3 単一化

Q 4.3.1 項と項とを単一化するって、どういうことですか。

A 項と項とを「単一化する」(unify) というのは、二つの項を同じものにしようと試みることです。

項と項とを単一化すると、それらの項が最初から同じものであれば成功して、最初から違うものであれば失敗します。たとえば、namako というアトムと namako というアトムとを単一化すると成功して、namako というアトムと hitode というアトムとを単一化すると失敗します。

同じように、a(b,c) という複合項と a(b,c) という複合項とを単一化すると成功して、a(b,c) という複合項と a(b,d) という複合項とを単一化すると失敗します。

ちなみに、unify という英単語の品詞は動詞です。「単一化すること」を意味する名詞は unification になります。

Q 4.3.2 単一化を実行する組み込み述語って、あるんですか。

A はい、単一化を実行する (=)/2 という組み込み述語があります。

= という中置演算子を名前とする組み込み述語は、二つの引数を単一化するという動作をします。たとえば、

```
?- namako = namako.
```

という質問を入力すると、yes という結果が出力され、

```
?- namako = hitode.
```

という質問を入力すると、no という結果が出力されます。

Q 4.3.3 単一化する項のうちの一方が変数の場合、その単一化はどうなるんですか。

A 単一化する項のうちの一方が変数の場合、その変数は他方の項と同一のものに変化しますので、その単一化は成功します。

たとえば、X という変数と namako というアトムとを単一化すると、X が namako に変化することによって、その単一化は成功します。

それでは、= を使って実際に試してみましょう。たとえば、

```
?- X = namako.
```

という質問を入力すると、インタプリタは、

```
X = namako
```

と出力します。これは、X という変数が namako というアトムに変化したことを意味しています。なお、このとき、その出力の右側にカーソルが表示された状態でインタプリタの動作が停止しますが、改行を入力すれば、yes が出力されて、次のプロンプトが出力されます。

Q 4.3.4　変数が項に具体化するって、どういうことですか。

A　変数が変数ではない項に変化することを、変数が項に「具体化する」(instantiate) と言います。

たとえば、X という変数が namako というアトムに変化することを、「X が namako に具体化する」と言います。

Q 4.3.5　変数の値って何のことですか。

A　変数が変数ではない項に具体化しているとき、その項のことを、その変数の「値」(value) と呼びます。

たとえば、X という変数が namako というアトムに具体化しているとすると、X の値は namako だということになります。

Q 4.3.6　変数の値を利用したいときって、どうすればいいんですか。

A　変数の値を利用したいときは、ただ単にその変数を書けばいいだけです。

変数は、それがすでに具体化している場合は、変数としてではなくて、その値として扱われます。たとえば、

```
?- X = namako, write(X).
```

という質問を入力したとしましょう。すると、まず X と namako とが単一化されて、X は namako に具体化します。そうすると、そののち X は、変数としてではなくて namako という値として扱われますので、write(X) というゴールの実行によって、namako というアトムが出力されることになります。

Q 4.3.7　変数と変数とを単一化するとどうなるんですか。

A　変数と変数とを単一化すると、それらの変数は同じものになります。

変数と変数とを単一化した場合、そののちに一方の変数が変数ではない項に具体化したときに、他方も自動的に同じ項に具体化します。
たとえば、

```
?- X = Y, Y = namako.
```

という質問を入力したとしましょう。すると、まず X と Y とが単一化されて、X と Y とが同じものになります。ですから、そののち Y と namako とが単一化されたとき、Y が namako に具体化すると同時に、X も namako に具体化することになります。

Q 4.3.8　引数として変数を含んでいる複合項と、別の項とを単一化すると、どうなるんですか。

A　引数として変数を含んでいる複合項と、別の項との単一化は、その変数を具体化することによって両者が同じものになるならば成功します。

たとえば、a(X) と a(b) との単一化は、X が b に具体化することによって両者が同じものになりますので、成功します。同じように、a(X) と a(b(c,d)) との単一化も、X が b(c,d) に具体化することによって両者が同じものになりますので、やはり成功します。

Q 4.3.9　a(X) と a(b,c) との単一化って、成功するんですか。

A　いいえ、成功しません。

複合項の引数を区切るためのコンマというのは、あくまで複合項の構文の一部分であって、演算子のコンマとは異なる別のものです。つまり、a(b,c) の引数を、b,c というひとつの項だと解釈することはできない、ということです。ですから、a(X) と a(b,c) との単一化は、失敗することになります。

ちなみに、a(X) ではなくて a(X,Y) ならば、a(b,c) との単一化は、X が b に、Y が c に具体化することによって、成功します。
一般に、複合項と複合項との単一化が成功するためには、両者の項数が同一である必要があります。

Q 4.3.10 a(X,X) と a(b,c) との単一化って、成功するんですか。

A いいえ、成功しません。

同一の変数を 2 個以上含んでいる複合項と、別の複合項との単一化は、それらの同一の変数が同一の項に具体化したときに両者が同じものになる場合だけ成功して、そうでなければ失敗します。
たとえば、a(X,X) と a(b,b) との単一化は、X が b に具体化することによって両者が同じものになりますので、成功します。しかし、a(X,X) と a(b,c) との単一化は、X が何に具体化したとしても両者を同じものにすることができませんので、失敗することになります。

Q 4.3.11 a(_,_) と a(b,c) との単一化って、成功するんですか。

A はい、成功します。

匿名変数ではない普通の変数に関しては、ひとつの複合項の中に 2 個以上の同一の変数が含まれている場合、それらの変数を具体化する項は、同一でないといけません。それに対して、ひとつの複合項の中に 2 個以上の匿名変数が含まれている場合、それぞれの匿名変数は、それぞれ異なる項に具体化することができます。
ですから、a(_,_) と a(b,c) との単一化は、1 個目の _ が b に具体化して、2 個目の _ が c に具体化することによって両者が同じものになりますので、成功することになるわけです。
このように、ひとつの複合項の中に 2 個以上の匿名変数が含まれている場合、それぞれの匿名変数は、あたかも別々の異なる変数であるかのように扱われます。そのため、匿名変数に関しては、その値を利用するということはできなくなっています。

Q 4.3.12 単一化って、いったい何のためにあるんですか。

A 単一化の存在理由は、インタプリタがゴールを実行する上で必要だからということです。

インタプリタにゴールを実行させると、インタプリタは、そのゴールと、事実または規則の頭部とを単一化します。
ゴールと事実とを単一化して、それが成功した場合、そのゴールは成功したことになります。たとえば、

```
chichioya(nobuyuki, satoko).
```

という事実がデータベースに置かれているとするとき、

```
?- chichioya(nobuyuki, satoko).
```

という質問を入力すると、インタプリタは、その質問の中のゴールと事実とを単一化します。この単一化は成功しますので、質問の中のゴールは成功したことになります。
ゴールと規則の頭部とを単一化して、それが成功した場合、インタプリタは次に規則の本体を実行して、それが成功したならば、そのゴールは成功したことになります。たとえば、

```
shiawase(keiko) :- hatarakimono(masuo).
hatarakimono(masuo).
```

という規則と事実がデータベースに置かれているとするとき、

```
?- shiawase(keiko).
```

という質問を入力すると、インタプリタは、まず、その質問の中のゴールと規則の頭部とを単一化します。この単一化は成功しますので、インタプリタは次に規則の本体を実行します。そうすると、その本体の実行も成功しますので、質問の中のゴールは成功したことになります。
規則の頭部が変数を含んでいる場合、その変数は単一化にともなって具体化します。そして本体が実行されるとき、具体化した変数は、変数としてではなくてその値として扱われます。
たとえば、

```
twice(X) :- write(X), write(X).
```

という規則がデータベースに置かれているとするとき、

```
?- twice(hoge).
```

という質問を入力したとしましょう。

インタプリタは、まず、その質問の中のゴールと規則の頭部とを単一化します。この単一化は成功して、X という変数は hoge というアトムに具体化します。

インタプリタは、次に規則の本体を実行するわけですが、本体の中に含まれている X という変数は、変数としてではなくて hoge という値として扱われます。ですから、規則の本体を実行した結果として、hogehoge という文字列が出力されることになります。

4.4　バックトラック

Q 4.4.1　バックトラックするって、どういうことですか。

A　「バックトラックする」(backtrack) というのは、これまでの過程を逆にたどって、まだ試していない選択肢を試してみる、ということです。

迷路に迷い込んだ人間が出口を発見するためには、バックトラックを実行する必要があります。つまり、進んでいる道が行き止まりだった場合は、最後の分岐点まで引き返して、別の道を選択しないといけません。そして、最後の分岐点のすべての道が行き止まりだった場合は、最後から二番目の分岐点まで引き返して、別の道を選択しないといけません。

Prolog のプログラムというのは迷路に似ています。そして、それが実行される方法は、迷路に迷い込んだ人間が出口を探す方法に似ています。つまり、Prolog のインタプリタも、進んでいる道が行き止まりだった場合は、分岐点まで引き返して別の道を選択する、という動作をするのです。

Q 4.4.2　Prolog のプログラムを迷路にたとえたとき、分岐点から分かれていくそれぞれの道って、何に相当するんですか。

A　分岐点から分かれていくそれぞれの道は、ひとつの述語定義を構成するそれぞれの節に相当します。

ひとつの述語定義は、1 個以上の節から構成されます。ゴールを実行したとき、それらの節は、かならずしもすべてが試されるとは限りません。ゴールを成功させる節が発見された場合、まだ試していない節が残っていたとしても、それらを残したまま次のゴールの実行へ移ります。そして、バックトラックによって引き返してきたときには、試されないで残っていた節を試すわけです。

Q 4.4.3　選択点って何ですか。

A　「選択点」(choice point) というのは、バックトラックによってそこへ引き返すことのできる、まだ試されていない節の列のことです。

述語を呼び出したとき、その定義を構成している節のうちのいくつかが試されないで残ったまま、その実行が成功した場合、その述語はひとつの選択点を生成することになります。そして、選択点は、それを構成しているすべての節がバックトラックによって試されたのちに消滅します。

Q 4.4.4　Prolog のプログラムを迷路にたとえたとき、行き止まりって、何に相当するんですか。

A　行き止まりは、ゴールの失敗に相当します。

たとえば、

A :- B_1, B_2, B_3, B_4, B_5.

という規則がデータベースに置かれているときに、A というゴールを実行したとしましょう。そうすると、その規則の本体が左から右へ順番に実行されていくことになります。ここで、B_5 の実行が失敗したとします。その場合、A の実行が直ちに失敗するわけではありません。インタプリタはバックトラックを実行するのです。

もしも B_4 を実行したときに選択点が生成されていたとすると、インタプリタはその選択点へバックトラックして、試されないで残っていた節を試してみて、それによって B_4 が成功したな

らば、ふたたび B_5 を実行します。

もしも B_4 に選択点が存在していないならば、インタプリタは、さらに左のゴールの選択点へバックトラックします。このように、選択点へバックトラックする順番は、通常のゴールの実行とは逆になります。

そして、もしも、

A :- B_1, B_2, B_3, B_4, B_5.

という規則を構成している B_1 から B_4 までのすべてのゴールについて選択点がまったく存在していないときに、B_5 が失敗したとすると、そのときは、規則の本体の実行が失敗したことになります。

Q 4.4.5 Prolog のインタプリタは、どんな順番で、それぞれの節を試していくんですか。

A Prolog のインタプリタは、上から下へ、つまりプログラムの中に書かれているのと同じ順番で、それぞれの節を試していきます。

たとえば、

```
bijin(yoshiko).
bijin(sayaka).
bijin(michiko).
bijin(akina).
bijin(kiyomi).
```

という述語定義がデータベースに置かれているとしましょう。このとき、

```
?- bijin(X).
```

という質問を入力すると、インタプリタは述語定義を構成しているそれぞれの節を上から順番に試しますので、

```
X = yoshiko
```

という結果を出力します。そしてインタプリタは、その出力の右側にカーソルを表示して、人間からの入力を待つわけですが、このときにセミコロン (semicolon, ;) と改行を入力することによって、インタプリタに強制的にバックトラックを実行させることができます。ですから、セミコロンと改行を次々と入力していくことによって、述語定義を構成しているそれぞれの節を上から下へ順番に試していくことができます。つまり、

```
X = yoshiko ;
X = sayaka ;
X = michiko ;
X = akina ;
X = kiyomi ;
no
```

というような結果が得られるわけです。

Q 4.4.6 インタプリタに強制的にバックトラックを実行させる組み込み述語って、あるんですか。

A はい、`fail/0` という組み込み述語を呼び出すことによって、インタプリタに強制的にバックトラックを実行させることができます。

`fail/0` というのは、かならず失敗する、という単純な機能を持つ組み込み述語です。この述語を呼び出すゴールを実行すると、それはかならず失敗するわけですから、インタプリタは必然的にバックトラックすることになります。

それでは、`fail/0` を使ってプログラムを書いてみましょう。

```
write_bijin :- bijin(X), write(X), nl, fail.
write_bijin.

bijin(yoshiko).
bijin(sayaka).
bijin(michiko).
bijin(akina).
bijin(kiyomi).
```

このプログラムの中で定義されている write_bijin/0 という述語を呼び出すと、

```
yoshiko
sayaka
michiko
akina
kiyomi
```

というものが出力されます。

ちなみに、write/1 と nl/0 という組み込み述語は、どちらも常に成功する述語で、選択点は生成しません。

また、write_bijin/0 の定義の中にある、

```
write_bijin.
```

という事実は、write_bijin/0 を呼び出すゴールをかならず成功させるために書かれているものです。

もうひとつ、似たようなプログラムを書いてみます。

```
pair :- joshi(A), danshi(B), write(pair(A, B)), nl, fail.
pair.

joshi(shiori).
joshi(risako).
joshi(marina).

danshi(hitoshi).
danshi(tsunehiko).
danshi(yoshio).
```

このプログラムの中で定義されている pair/0 という述語を呼び出すと、

```
pair(shiori, hitoshi)
pair(shiori, tsunehiko)
pair(shiori, yoshio)
pair(risako, hitoshi)
pair(risako, tsunehiko)
pair(risako, yoshio)
pair(marina, hitoshi)
pair(marina, tsunehiko)
pair(marina, yoshio)
```

というものが出力されます。この実行結果から判るとおり、

```
pair :- joshi(A), danshi(B), write(pair(A, B)), nl, fail.
```

という規則の中の fail を実行したとき、インタプリタは、これまでの過程を逆の順序でたどって、もっとも近い位置にある選択点へバックトラックします。つまり、まず danshi(B) の選択点へバックトラックして、それが消滅したのち、joshi(A) の選択点へバックトラックするわけです。

Q 4.4.7　バックトラックって、実行中の規則の選択点だけじゃなくて、別の規則の選択点へ引き返すこともあるんですか。

A　はい、あります。

たとえば、A というゴールを実行したことによって、

A :- B_1, B_2.

という規則の本体が実行されて、その結果として、

B_1 :- C.

という規則の本体が実行されたとしましょう。そして、C は選択点を生成して成功したとしましょう。

そのとき、もしも B_2 の実行が失敗したとすると、インタプリタは C の選択点へバックトラックします。このように、バックトラックは、実行中の規則の選択点だけではなくて、別の規則の選択点へ引き返すこともあるわけです。

ですから、Q 4.4.6 で紹介した pair/0 という述語は、次のように定義を書いたとしても、同じ動作をすることになります。

```
pair :- pair(A, B), write(pair(A, B)), nl, fail.
pair.

pair(A, B) :- joshi(A), danshi(B).

joshi(shiori).
joshi(risako).
joshi(marina).

danshi(hitoshi).
danshi(tsunehiko).
danshi(yoshio).
```

4.5 基本的な組み込み述語

Q 4.5.1 項の種類を判定する組み込み述語って、あるんですか。

A はい、あります。

組み込み述語の中には、項の種類を判定する、次のような述語が含まれています。

atom(T)	T はアトムである。
atomic(T)	T はアトムまたは数値定数である。
number(T)	T は数値定数である。
integer(T)	T は整数定数である。
float(T)	T は浮動小数点定数である。
compound(T)	T は複合項である。
var(T)	T は具体化されていない変数である。
nonvar(T)	T は変数ではない。

それでは、これらの述語を呼び出すゴールをいくつか書いてみましょう。

atom(a)	成功
atom(3)	失敗
atom(a(b))	失敗
atomic(a)	成功
atomic(3)	成功
atomic(a(b))	失敗
number(3)	成功
number(a)	失敗
compound(a(b))	成功
compound(a)	失敗
var(X)	成功
var(a)	失敗
X = a, var(X)	失敗
nonvar(a)	成功
nonvar(X)	失敗
X = a, nonvar(X)	成功

Q 4.5.2 二つのアトムを連結する組み込み述語って、あるんですか。

A はい、あります。

atom_concat/3 という組み込み述語を使うことによって、二つのアトムを連結することができます。

A と B がアトムだとするとき、atom_concat(A, B, C) というゴールを実行すると、A の右側に B を連結することによって作られたアトムと C とが単一化されます。たとえば、

```
?- atom_concat(mike, neko, X).
```

という質問を入力したとすると、

```
X = mikeneko
```

と出力されます。

atom_concat/3 によるアトムの連結というのは、もう少し厳密に言うと、アトムによってあらわされている文字列を連結するということです。ですから、'8' という引用アトムの右側に '7' という引用アトムを連結した結果は、'8''7' ではなくて、'87' になります。

atom_concat/3 は、連結とは逆の結果を求めたいときにも使うことができます。たとえば、

```
?- atom_concat(X, neko, mikeneko).
```

という質問を入力すると、

```
X = mike
```

と出力されます。同じように、

```
?- atom_concat(mike, X, mikeneko).
```

という質問を入力すると、

```
X = neko
```

と出力されます。

Q 4.5.3　二つの項が同一のものかどうかを調べる組み込み述語って、あるんですか。

A　はい、あります。

(==)/2 という組み込み述語を使うことによって、二つの項が同一のものかどうかを調べることができます（== というアトムは中置演算子です）。

A と B が項だとするとき、A == B というゴールの実行は、A と B が同一ならば成功して、そうでなければ失敗します。たとえば、

```
a(b, c) == a(b, c)
```

というゴールの実行は成功して、

```
a(b, c) == a(b, y)
```

というゴールの実行は失敗します。

(=)/2 とは違って、(==)/2 には、二つの項を単一化するという機能はありません。ですから、X という変数が、まだどんな項にも具体化していないとするとき、

```
a(b, c) == X
```

というゴールの実行は失敗します。

Q 4.5.4　(==)/2 とは逆に、二つの項が同一のものだと失敗して、同一ではない場合に成功する組み込み述語って、あるんですか。

A　はい、あります。

(\==)/2 という組み込み述語は、(==)/2 とは逆の動作をします（\== というアトムは中置演算子です）。

A と B が項だとするとき、A \== B というゴールの実行は、A と B が同一ならば失敗して、そうでなければ成功します。たとえば、

```
a(b, c) \== a(b, c)
```

というゴールの実行は失敗して、

```
a(b, c) \== a(b, y)
```

というゴールの実行は成功します。

4.6 式

Q 4.6.1 式って何ですか。

A 「式」(expression) というのは、数値定数、または、数値に対する処理をあらわす複合項のことです。

Prolog では、式を書くことによって、数値に対するさまざまな処理を記述することができるようになっています。

Q 4.6.2 式を評価するって、どういうことですか。

A 式を「評価する」(evaluate) というのは、式があらわしている処理を実行することです。

Q 4.6.3 式の値って何のことですか。

A 式の「値」(value) というのは、式を評価した結果として得られた数値のことです。

Q 4.6.4 数値定数を評価すると、どんな値が得られるんですか。

A 数値定数を評価すると、その値として、その数値定数があらわしている数値が得られます。

たとえば、`8703` という数値定数を評価すると、8703 という整数が値として得られます。

Q 4.6.5 算術関数子って何ですか。

A 「算術関数子」(arithmetic functor) というのは、式を書くときに関数子として使われる、数値に対する処理をあらわすアトムのことです。

算術関数子のうちの主要なものとしては、次のようなものがあります。

A `+` B	A と B とを加算した結果
A `-` B	A から B を減算した結果
A `*` B	A と B とを乗算した結果
A `/` B	A を B で除算した結果
N `//` M	整数の範囲で整数 N を整数 M で除算した結果
N `mod` M	整数 N を整数 M で除算したときのあまり
A `**` B	A の B 乗
`-` A	A の符号を反転した結果
`abs(`A`)`	A の絶対値
`sqrt(`A`)`	A の平方根
`exp(`A`)`	指数関数（自然対数の底 e の A 乗）
`log(`A`)`	A の自然対数
`sin(`A`)`	A のサイン（A の単位はラジアン。以下同様）
`cos(`A`)`	A のコサイン
`tan(`A`)`	A のタンジェント
`atan(`A`)`	A のアークタンジェント（単位はラジアン）
`truncate(`A`)`	A の小数点以下を切り捨てることによって得られる整数
`round(`A`)`	A にもっとも近い整数
`ceiling(`A`)`	A を下回らない最小の整数
`floor(`A`)`	A を超えない最大の整数

Q 4.6.6 算術関数子のオペランドって何のことですか。

A 算術関数子の「オペランド」(operand) というのは、算術関数子によってあらわされる処理の対象となる式（またはその値）のことです。

言い換えれば、オペランドというのは、式であるような複合項の引数（またはその値）のことです。

たとえば、5+3 という式に含まれている + という算術関数子のオペランドは、5 と 3 です。

Q 4.6.7　式を評価する組み込み述語って、あるんですか。

A　はい、あります。

is/2 という組み込み述語を使うことによって、式を評価することができます（is というアトムは中置演算子です）。

E が式だとするとき、V is E というゴールを実行すると、E を評価することによって得られた値と V とが単一化されます。たとえば、

```
?- X is 5+3.
```

という質問を入力すると、

```
X = 8
```

と出力されます。

次の述語は、A が式だとするとき、square(A, B) というゴールで呼び出すと、A の値の 2 乗をあらわす数値定数と B とを単一化します。

```
square(A, B) :- B is A * A.
```

この述語を使うことによって、次のように、数値の 2 乗を求めることができます。

```
?- square(7, X).
X = 49
yes
```

Q 4.6.8　式の値の大きさを比較する組み込み述語って、あるんですか。

A　はい、あります。

式の値の大きさを比較する組み込み述語としては、次のようなものがあります。

A > B　　A は B よりも大きい

A < B　　A は B よりも小さい

A >= B　　A は B よりも大きいかまたは等しい

A =< B　　A は B よりも小さいかまたは等しい

A =:= B　　A と B とは等しい

A =\= B　　A と B とは等しくない

これらの組み込み述語は、左右の引数を両方とも式として評価して、それらの値のあいだに、上に書いたような関係が成り立っているならば成功して、そうでなければ失敗します。たとえば、

```
?- 5+3 > 3*2.
```

という質問を入力すると yes が出力されて、

```
?- 5+3 > 3*3.
```

という質問を入力すると no が出力されます。

次の述語は、N が式だとするとき、even(N) というゴールで呼び出すと、N の値が偶数ならば成功して、そうでなければ失敗します。

```
even(N) :- N mod 2 =:= 0.
```

この述語を使うことによって、次のように、整数が偶数かどうかを判定することができます。

```
?- even(6).
yes
?- even(7).
no
```

次の述語は、A が式だとするとき、sign(A, B) というゴールで呼び出すと、A の値がプラスならば positive というアトムと B とを単一化して、A の値がマイナスならば negative というアトムと B とを単一化して、A の値がゼロならば zero というアトムと B とを単一化します。

```
sign(A, positive) :- A > 0.
```

```
sign(A, negative) :- A < 0.
sign(_, zero).
```

この述語を使うことによって、次のように、数値の符号を判定することができます。

```
?- sign(7, X).
X = positive
yes
?- sign(-7, X).
X = negative
yes
?- sign(0, X).
X = zero
yes
```

4.7 練習問題

4.1 次の単一化は成功しますか。匿名変数ではない変数が項に含まれていて、かつ成功するものについては、変数の値も答えてください。

(a) `hibari = hibari`

(b) `hibari = suzume`

(c) `a(b,c,d) = a(b,c,d)`

(d) `a(b,c,d) = a(e,f,g)`

(e) `a(b,c,d) = a(b,c)`

(f) `X = kamome`

(g) `X = a(b,c,d)`

(h) `X = Y, Y = Z, Z = tsubame`

(i) `a(X) = karasu`

(j) `a(X) = a(karasu)`

(k) `a(X) = a(b(c,d,e))`

(l) `a(X,Y,Z) = a(b,c)`

(m) `a(X,Y,Z) = a(b,c,d)`

(n) `a(X,Y,Z) = a(b,c,d,e)`

(o) `a(X,c) = a(b,Y)`

(p) `a(X,Y,Y) = a(b,c,d)`

(q) `a(X,Y,Y) = a(b,c,c)`

(r) `a(_,_,_) = a(b,c,d)`

4.2 次のプログラムを、

```
?- suteki.
```

という質問で実行すると、どんなものが出力されますか。

```
suteki :- suteki(A), write(A), nl, fail.
suteki.

suteki(A) :- handsome(A), yasashii(A).

handsome(tomohiro).
handsome(masao).
handsome(hideki).
handsome(yoshihiko).
handsome(nobuyuki).

yasashii(hitoshi).
yasashii(sadayoshi).
yasashii(yoshihiko).
```

```
yasashii(takashi).
yasashii(masao).
```

4.3 次のプログラムを、

```
?- mago.
```

という質問で実行すると、どんなものが出力されますか。

```
mago :- mago(A, B), write(mago(A, B)), nl, fail.
mago.

mago(C, A) :- kodomo(B, A), kodomo(C, B).

kodomo(masayo, haruka).
kodomo(takayuki, haruka).
kodomo(tomoko, haruka).
kodomo(hiroko, takayuki).
kodomo(nobuo, takayuki).
kodomo(midori, hiroko).
kodomo(kunihiko, hiroko).
kodomo(naomi, midori).
```

4.4 次のプログラムを、

```
?- couple.
```

という質問で実行すると、どんなものが出力されますか。

```
couple :- couple(A, B), write(couple(A, B)), nl, fail.
couple.

couple(A, B) :- aisuru(A, B), aisuru(B, A).

aisuru(hiroyuki, sayaka).
aisuru(shigeo, kumiko).
aisuru(shigeru, haruka).
aisuru(yoshihiko, manami).
aisuru(akira, mina).
aisuru(mina, mitsumasa).
aisuru(kumiko, shigeo).
aisuru(haruka, tomoki).
aisuru(sayaka, shinji).
aisuru(manami, yoshihiko).
```

4.5 like というゴールで呼び出すと、

```
I like the world.\n
```

という文字列を出力する、like/0 という述語の定義を書いてください。

実行例

```
?- like.
I like the world.
yes
```

4.6 A が項だとするとき、like(A) というゴールで呼び出すと、

```
I like A.\n
```

という文字列を出力する、like/1 という述語の定義を書いてください。

実行例

```
?- like('medieval history').
I like medieval history.
yes
```

4.7 A と B が項だとするとき、like(A, B) というゴールで呼び出すと、

```
I like A rather than B.\n
```

という文字列を出力する、like/2 という述語の定義を書いてください。

実行例

```
?- like('Aristotle', 'Plato').
```

```
I like Aristotle rather than Plato.
yes
```

4.8 A が項だとするとき、type(A, T) というゴールで呼び出すと、A がアトムならば atom というアトムと T とを単一化して、A が数値定数ならば number というアトムと T とを単一化して、A が複合項ならば compound というアトムと T とを単一化する、type/2 という述語の定義を書いてください。

実行例
```
?- type(miho, X).
X = atom
yes
?- type(7041, X).
X = number
yes
?- type(a(b), X).
X = compound
yes
```

4.9 A がアトムだとするとき、twice(A, T) というゴールで呼び出すと、二つの A を連結した結果と T とを単一化する、twice/2 という述語の定義を書いてください。

実行例
```
?- twice(neko, X).
X = nekoneko
yes
```

4.10 H と M_1 が、H 時間 M_1 分という時間の長さをあらわす整数定数だとするとき、

hmtom(H, M_1, M_2)

というゴールで呼び出すと、H 時間 M_1 分と同じ時間の長さを、分を単位としてあらわしている整数定数を求めて、その結果と M_2 とを単一化する、hmtom/3 という述語の定義を書いてください。

実行例
```
?- hmtom(2, 30, X)
X = 150
yes
```

4.11 M_1 が、分を単位として時間の長さをあらわしている整数定数だとするとき、

mtohm(M_1, H, M_2)

というゴールで呼び出すと、M_1 分という長さを何時間何分という形式に変換して、時間の部分をあらわしている整数定数と H とを単一化して、分の部分をあらわしている整数定数と M_2 とを単一化する、mtohm/3 という述語の定義を書いてください。

実行例
```
?- mtohm(150, X, Y)
X = 2
Y = 30
yes
```

4.12 N がプラスの整数をあらわす定数だとするとき、sumint(N, M) というゴールで呼び出すと、1 から N までのすべての整数を加算した結果と M とを単一化する、sumint/2 という述語の定義を書いてください。

実行例
```
?- sumint(100, X).
X = 5050
yes
```

ヒント n がプラスの整数だとするとき、1 から n までのすべての整数を加算した結果は、

$$\frac{n \times (n+1)}{2}$$

という計算の結果と等しくなります。

4.13 N と M が整数定数だとするとき、measure(N, M) というゴールで呼び出すと、N が M の約数ならば成功して、そうでなければ失敗する、measure/2 という述語の定義を書いて

ください。

実行例
```
?- measure(7, 28).
yes
?- measure(7, 30).
no
```

第5章 再帰

5.1 再帰の基礎

Q 5.1.1 再帰的な構造っていうのはどういう構造のことなんですか。

A 「再帰的な」(recursive) 構造というのは、その一部分として、全体と同じ構造を持っているものを含んでいる構造のことです。

向かい合わせに置かれた2枚の鏡に映し出される映像は、再帰的な構造を持っているものの例のひとつです。その場合、正面の鏡の中には背後の鏡が映っていて、背後の鏡の中には正面の鏡が映っています。正面の鏡の映像を全体とすると、それと同じものが、その一部分として映っていますので、それは再帰的な構造を持っているということになるわけです。

「再帰的な」(recursive) という形容動詞 (英語では形容詞) だけではなくて、「再帰する」(recurse) という動詞や、「再帰」(recursion) という名詞が使われることもあります。「再帰する」という動詞は「一部分を作るために全体を利用する」という意味で使われ、「再帰」という名詞は「再帰的であること」または「再帰すること」という意味で使われます。

Q 5.1.2 入れ子構造って何のことですか。

A 「入れ子構造」(nested structure) というのは、その一部分として、全体と同じ性質を持っているものを含んでいる構造のことです。

たとえば、中に箱が入っている箱や、演劇を上演する場面を含んでいる演劇や、絵画が描かれている絵画などは、入れ子構造を持っているものの例です。

Q 5.1.3 再帰的な構造と入れ子構造とは同じものだと考えていいんですか。

A いいえ、再帰的な構造は、入れ子構造の特殊なものです。

再帰的な構造というのは入れ子構造の一種ですから、全体と同じ性質を持っているものを一部分として含んでいます。再帰的な構造は、「全体と同じ性質を持っているものを一部分として含んでいる」という性質を、全体と部分とが共有している、という点で、入れ子構造よりも特殊な構造です。

箱の場合で考えてみましょう。入れ子構造の箱というのは、全体と部分とが「箱である」という性質を共有している箱のことです。それに対して、再帰的な構造の箱というのは、全体と部分とが「中に箱が入っている箱である」という性質を共有している箱のことです。

ですから、再帰的な構造というのは、何らかの歯止めがない限り、部分の中に全体を入れるということが無限に続いていくような構造のことだと考えることができます。

Q 5.1.4 基底って何ですか。

A 「基底」(basis) というのは、再帰が無限に続いていかないで、何らかの歯止めに到達した場合に、その構造のもっとも内側にあるもののことです。

再帰が無限に続いている構造を持つものは、思考の対象として存在することは可能ですが、現実的に存在することはできません。つまり、現実的には、再帰というのはかならず何らかの歯止めに到達して終了しなければならないということです。再帰が歯止めに到達した場合、そこには、全体と同じ構造を持つものではなく、それに代わるものがあります。それが基底です。

Q 5.1.5 再帰的な定義って何のことですか。

A 再帰的な定義というのは、定義の対象となっているものを使って何かを説明している定義のことです。

再帰的に定義されたものは、かならず再帰的な構造を持つことになります。たとえば、「無限箱が中に入っている箱のことを『無限箱』と呼ぶ」という記述によって再帰的に定義された無限箱は、再帰的な構造を持っています。

Q 5.1.6 何らかの歯止めに到達して終了するような再帰的な構造を定義したいときって、どうすればいいんですか。

A そのようなときは、再帰する場合と再帰しない場合とを選択することができるようにします。

たとえば、「無限箱が中に入っている箱か、または中に何も入っていない箱のことを『無限箱』と呼ぶ」と定義された無限箱は、中に何も入っていない箱を選択することによって、再帰を終了することができます。

ちなみに、再帰しない選択肢というのは、再帰的な構造の基底のことです。

Q 5.1.7 関係の推移的閉包って何のことですか。

A 関係の「推移的閉包」(transitive closure) というのは、その関係をたどっていくことによって到達できるもののあいだに成り立つ関係のことです。

たとえば、「子孫である」という関係は、「子供である」という関係の推移的閉包です。なぜなら、二人の人間のあいだに子孫であるという関係が成り立っているとすると、一方の人間から出発して、子供であるという関係をたどっていくことによって他方の人間に到達することができるからです。

Q 5.1.8 推移的閉包を定義したいときって、どうすればいいんですか。

A 推移的閉包は、再帰を使うことによって定義することができます。

たとえば、「子孫である」という関係は、次のように定義することができます。

- B が A の子供であるならば、B は A の子孫である。
- C が A の子供であって、かつ B が C の子孫であるならば、B は A の子孫である。
- 以上の記述から導かれるもの以外は子孫ではない。

Prolog でも、同じように再帰を使うことによって推移的閉包を定義することができます。たとえば、

`kodomo(`B`, `A`)`

というゴールで、B が A の子供かどうかを調べることができるとするとき、

`shison(`Y`, `X`)`

というゴールで、Y が X の子孫かどうかを調べることができるようにしたいとしましょう。そのために必要となる `shison/2` という述語の定義は、再帰を使って、次のように書くことができます。

```
shison(B, A) :- kodomo(B, A).
shison(B, A) :- kodomo(C, A), shison(B, C).
```

Q 5.1.9 再帰的なアルゴリズムって、どんなものなんですか。

A 再帰的なアルゴリズムというのは、問題の一部分を解くために、その問題を解くためのアルゴリズムの全体を利用するアルゴリズムのことです。

再帰的なアルゴリズムの一例として、括弧列を作るアルゴリズム、というものを紹介しましょう。

n が 0 またはプラスの整数だとするとき、n 個の左括弧の右側に n 個の右括弧を並べたもののことを、「n 重の括弧列」と呼ぶことにします。たとえば、`(((())))` という文字列は 3 重の括弧列です。0 重の括弧列は、空文字列です。

n が 1 以上のとき、n 重の括弧列というのは、

($\boxed{(n-1)\text{ 重の括弧列}}$)

という文字列のことだと考えることができます。つまり、括弧列は、丸括弧で囲まれた内側にさらに括弧列があるという再帰的な構造を持っているわけです。

n 重の括弧列は、次のような再帰的なアルゴリズムを実行することによって作ることができます。

- n が 0 ならば、n 重の括弧列は空文字列である。
- n が 1 以上ならば、n 重の括弧列は、1 個の左括弧と $(n-1)$ 重の括弧列と 1 個の右括弧を、この順番で連結したものである。

このアルゴリズムは、その一部分として $(n-1)$ 重の括弧列を作るという作業を含んでいますが、その作業は、括弧列を作るアルゴリズムの全体を利用することによって実現することができます。この点が、このアルゴリズムが再帰的だと考えられる理由です。

次の kakkoretsu/2 という述語の定義は、括弧列を作るアルゴリズムを Prolog で記述したものです。

```
kakkoretsu(0, '').
kakkoretsu(N, K) :-
    N >= 1, N1 is N - 1, kakkoretsu(N1, K1),
    atom_concat('(', K1, ')', K).

atom_concat(A, B, C, D) :-
    atom_concat(A, B, E), atom_concat(E, C, D).
```

N が 0 またはプラスの整数だとするとき、kakkoretsu(N, K) というゴールを実行すると、N 重の括弧列をあらわすアトムと K とが単一化されます。たとえば、

```
kakkoretsu(5, X)
```

というゴールを実行したとすると、'((((()))))' というアトムが X の値になります。

5.2　自然数

Q 5.2.1　後継関数って何ですか。

A　「後継関数」(successor function) というのは、「引数の次のもの」を意味する、項数が 1 の複合項のことです。

後継関数の関数子としては、普通、s が使われます。ですから、N の次のものを意味する後継関数は、

s(N)

と書くことになります。

Q 5.2.2　後継関数って、何をするためにあるんですか。

A　後継関数の目的は、自然数を書きあらわすことです。

Prolog では、整数定数を使って自然数を書きあらわすことができます。しかし、Prolog で自然数を書きあらわす方法としては、そのほかに、後継関数を使うという方法もあります。

後継関数による自然数の記述法では、ゼロは、0 という整数定数によって書きあらわされます。

1 という自然数は、「0 の次のもの」ですから、s(0) という後継関数で書きあらわすことができます。同じように、2 という自然数は、「s(0) の次のもの」ですから、s(s(0)) という後継関数で書きあらわすことができます。

後継関数による自然数の記述は、再帰的な構造を持っています。ですから、後継関数を使って自然数 n をあらわす項は、次のように再帰的に定義することができます。

- n が 0 ならば、n をあらわす項は 0 である。
- n が 0 ではないならば、n をあらわす項は、$n-1$ をあらわす項を引数とする後継関数である。

Q 5.2.3　項が後継関数で記述された自然数かどうかを判定するアルゴリズムって、あるんですか。

A　はい、あります。

N が項だとするとき、次のアルゴリズムを実行することによって、N が後継関数で記述された自然数かどうかを判定することができます。

- N が0ならば、N は後継関数で記述された自然数である。
- N がs(N_1)ならば、N が後継関数で記述された自然数かどうかは、N_1 が後継関数で記述された自然数かどうかを判定した結果と等しい。

次のnatural/1という述語の定義は、項が後継関数で記述された自然数かどうかを判定するアルゴリズムをPrologで記述したものです。

```
natural(0).
natural(s(N)) :- natural(N).
```

この述語を使うことによって、次のように、項が後継関数で記述された自然数かどうかを判定することができます。

```
?- natural(s(s(s(s(s(s(s(0)))))))).
yes
?- natural(s(s(s(s(s(s(s(8)))))))).
no
```

Q 5.2.4 後継関数で記述された二つの自然数を加算するアルゴリズムって、あるんですか。

A はい、あります。

N と M が後継関数で記述された自然数だとするとき、次のアルゴリズムを実行することによって、それらを加算した結果を求めることができます。

- M が0ならば、N と M とを加算した結果は N である。
- M がs(M_1)ならば、N と M とを加算した結果は、s(N)と M_1 とを加算した結果と等しい。

次のadd/3という述語の定義は、後継関数で記述された二つの自然数を加算するアルゴリズムをPrologで記述したものです（自然数かどうかの判定は、Q 5.2.3で紹介したnatural/1を使っています）。

```
add(N, 0, N) :- natural(N).
add(N, s(M), s(A)) :- add(N, M, A).
```

この述語を使うことによって、二つの自然数を加算した結果を求めることができます。たとえば、

```
?- add(s(s(s(0))), s(s(s(s(s(0))))), X).
X = s(s(s(s(s(s(s(0)))))))
yes
```

というように、3と5を加算させると、結果として8が得られます。

面白いことに、このadd/3という述語は、加算だけではなくて減算を実行することもできます。3個目の引数が変数になっているゴールで呼び出した場合は加算が実行されるわけですが、1個目または2個目の引数が変数になっているゴールで呼び出した場合は、減算が実行されるのです。たとえば、

```
?- add(s(s(s(0))), X, s(s(s(s(s(s(s(0))))))).
X = s(s(s(s(0))))
yes
```

というように、add/3を使って、8から3を減算した結果を求めることができます。

Prologのインタプリタの中には、変数の値を完全な形で出力しないで、

```
X = s(s(s(s(s(s(s(...)))))))
```

というように、一部分を省略した形で出力するものもあります。しかし、そのようなインタプリタを使っている場合でも、

```
?- add(s(s(s(0))), s(s(s(s(s(0))))), X), write(X).
```

というようにwrite/1を使うことによって、変数の値を完全な形で出力させることができます。

Q 5.2.5 後継関数で記述された二つの自然数を乗算するアルゴリズムって、あるんですか。

A はい、あります。

N と M が後継関数で記述された自然数だとするとき、次のアルゴリズムを実行することによって、それらを乗算した結果を求めることができます。

- M が `0` ならば、N と M とを乗算した結果は `0` である。
- M が `s(`M_1`)` ならば、N と M とを乗算した結果は、N と M_1 とを乗算した結果と N とを加算した結果と等しい。

次の `multiply/3` という述語の定義は、後継関数で記述された二つの自然数を乗算するアルゴリズムを Prolog で記述したものです（自然数かどうかの判定は、Q 5.2.3 で紹介した `natural/1` を使っています。また、加算は、Q 5.2.4 で紹介した `add/3` を使っています）。

```
multiply(N, 0, 0) :- natural(N).
multiply(N, s(M), P) :- multiply(N, M, P1), add(P1, N, P).
```

この述語を使うことによって、二つの自然数を乗算した結果を求めることができます。たとえば、

```
?- multiply(s(s(0)), s(s(s(s(s(0))))), X).
X = s(s(s(s(s(s(s(s(s(s(0))))))))))
yes
```

というように、2 と 5 を乗算させると、結果として 10 が得られます。

Q 5.2.6　N と M が後継関数で記述された自然数だとするとき、N の M 乗を求めるアルゴリズムって、あるんですか。

A　はい、あります。

N と M が後継関数で記述された自然数だとするとき、次のアルゴリズムを実行することによって、N の M 乗を求めることができます。

- M が `0` ならば、N の M 乗は `s(0)` である。
- M が `s(`M_1`)` ならば、N の M 乗は、N の M_1 乗と N とを乗算した結果と等しい。

次の `power/3` という述語の定義は、後継関数で記述された自然数のべき乗を求めるアルゴリズムを Prolog で記述したものです（自然数かどうかの判定は、Q 5.2.3 で紹介した `natural/1` を使っています。また、乗算は、Q 5.2.5 で紹介した `multiply/3` を使っています）。

```
power(N, 0, s(0)) :- natural(N).
power(N, s(M), P) :- power(N, M, P1), multiply(P1, N, P).
```

この述語を使うことによって、自然数のべき乗を求めることができます。たとえば、

```
?- power(s(s(0)), s(s(s(0))), X).
X = s(s(s(s(s(s(s(s(0))))))))
yes
```

というように、2 の 3 乗を求めさせると、結果として 8 が得られます。

5.3　二分木

Q 5.3.1　木って何ですか。

A　「木」(tree) というのは、1 個の根 (root) を出発点として、そこから枝 (branch) が分かれていって、最後に葉 (leaf) に到達して終わる、という構造のことです。

たとえば、コンピュータの中にあるディレクトリ（フォルダ）とファイルが作る構造は、木になっています。つまり、ルートディレクトリが根で、空ではないディレクトリが枝の分岐点で、空のディレクトリまたはファイルが葉に相当するわけです。

Prolog の複合項の構造も木です。複合項では、関数子が根で、引数として書かれた複合項の関数子が枝の分岐点で、複合項ではない項が葉に相当します。

なお、木の枝の分岐点、および葉は、「節点」(node) と呼ばれます。根も節点のひとつです。

Q 5.3.2　二分木って何ですか。

A　「二分木」(binary tree) というのは、木の一種で、どの節点についても、そこから出る枝の本数が 2 本以下であるもののことです。

二分木は、次のように再帰的に定義することができます。

- 空の木は二分木である。
- 根から 2 本の枝が出ていて、それぞれの枝の先に二分木があるものは、二分木である。
- 以上の記述から導かれるもの以外は二分木ではない。

T が空ではない二分木だとするとき、T の根から出ている 2 本の枝の先にある二分木のそれぞれは、T の「左部分木」(left subtree)、T の「右部分木」(right subtree) と呼ばれます。

項数が 1 または 2 の複合項と、複合項ではない項とを組み合わせることによって作られた項は、二分木になります。たとえば、

```
a(b(c, d(e)), f(g(h, i), j(k)))
```

という複合項は、ひとつの二分木を記述しています。

このような、項数が 1 または 2 の複合項を使って二分木を記述するという方法には、次のような欠点があります。

- アトム以外の項を節点にすることができない。
- 関数子が一定ではないので、それを処理する述語を定義することが困難である。

しかし、このような欠点は、項数が 3 の複合項を使うことによって克服することができます。

Q 5.3.3 項数が 3 の複合項を使って二分木を記述する方法っていうのは、どういうものなんですか。

A 項数が 3 の複合項を使って二分木を記述する方法というのは、節点、左部分木、右部分木のそれぞれを引数とする複合項を書くというものです。

項数が 3 の複合項を使って二分木を記述する場合には、あらかじめ、空の二分木をあらわすアトムと、複合項の関数子にするアトムを決めておく必要があります。ここでは、`v` というアトムで空の二分木をあらわし、`t` というアトムを複合項の関数子にすると決めておくことにします。

そうすると、たとえば、`a` という根から出ている枝の先に `b` という葉と `c` という葉がある、という二分木は、

```
t(a, t(b, v, v), t(c, v, v))
```

という複合項で書きあらわすことができます。

Q 5.3.4 項が二分木かどうかを判定するアルゴリズムって、あるんですか。

A はい、あります。

T が項だとするとき、次のアルゴリズムを実行することによって、T が二分木かどうかを判定することができます。

- T が `v` ならば、T は二分木である。
- T が `t(_, L, R)` であり、かつ L が二分木であり、かつ R も二分木であるならば、T は二分木である。

次の `binary/1` という述語の定義は、このアルゴリズムを Prolog で記述したものです。

```
binary(v).
binary(t(_, L, R)) :- binary(L), binary(R).
```

この述語を使うことによって、次のように、項が二分木かどうかを判定することができます。

```
?- binary(t(a, t(b, v, v), t(c, t(d, v, v), t(e, v, v)))).
yes
?- binary(t(a, t(b, v, v), t(c, t(d, v, v), t(e, 8, v)))).
no
```

Q 5.3.5 与えられた項が二分木の中に節点として含まれているかどうかを判定するアルゴリズムって、あるんですか。

A はい、あります。

M が項で、T が二分木だとするとき、次のアルゴリズムを実行することによって、M が T の中に節点として含まれているかどうかを判定することができます。

- T が t(M, _, _) ならば、M は T に含まれている。
- T が t(_, L, _) のとき、M が L に含まれているならば、M は T に含まれている。
- T が t(_, _, R) のとき、M が R に含まれているならば、M は T に含まれている。

次の tree_member/2 という述語の定義は、このアルゴリズムを Prolog で記述したものです。

```
tree_member(M, t(M, _, _)).
tree_member(M, t(_, L, _)) :- tree_member(M, L).
tree_member(M, t(_, _, R)) :- tree_member(M, R).
```

この述語を使うことによって、次のように、与えられた項が二分木に節点として含まれているかどうかを判定することができます。

```
?- tree_member(qq, t(a, t(b, v, v), t(c, v, t(qq, v, v)))).
yes
?- tree_member(qq, t(a, t(b, v, v), t(c, v, t(xx, v, v)))).
no
```

Q 5.3.6　二分木の中に含まれている節点の個数を求めるアルゴリズムって、あるんですか。

A　はい、あります。

T が二分木だとするとき、次のアルゴリズムを実行することによって、T の中に含まれている節点の個数を求めることができます。

- T が v ならば、T の節点の個数は 0 である。
- T が t(_, L, R) ならば、T の節点の個数は、1 と、L の節点の個数と、R の節点の個数とを加算した結果である。

次の count_tree/2 という述語の定義は、このアルゴリズムを Prolog で記述したものです。

```
count_tree(v, 0).
count_tree(t(_, L, R), C) :-
    count_tree(L, CL), count_tree(R, CR), C is 1 + CL + CR.
```

この述語を使うことによって、次のように、二分木の中に含まれている節点の個数を求めることができます。

```
?- count_tree(t(a, t(b, v, v), t(c, t(d, v, v), v)), X).
X = 4
yes
```

Q 5.3.7　二分木の中に含まれているすべての葉を刈り取ることによってできる二分木を求めるアルゴリズムって、あるんですか。

A　はい、あります。

T が二分木だとするとき、次のアルゴリズムを実行することによって、T の中に含まれているすべての葉を刈り取ることによってできる二分木を求めることができます。

- T が v ならば、T の葉を刈り取った結果は v である。
- T が t(_, v, v) ならば、T の葉を刈り取った結果は v である。
- T が t(N, L, R) ならば、T の葉を刈り取った結果は、N を根、L の葉を刈り取った結果を左部分木、R の葉を刈り取った結果を右部分木とする二分木である。

次の prune_leaves/2 という述語の定義は、このアルゴリズムを Prolog で記述したものです。

```
prune_leaves(v, v).
prune_leaves(t(_, v, v), v).
prune_leaves(t(N, L, R), t(N, PL, PR)) :-
    prune_leaves(L, PL), prune_leaves(R, PR).
```

この述語を使うことによって、次のように、二分木の中に含まれているすべての葉を刈り取ることによってできる二分木を求めることができます。

```
?- prune_leaves(t(a, t(b, t(c, v, v), t(d, v, v)),
```

```
                        t(e, t(f, v, v), t(f, v, v))), X).
X = t(a, t(b, v, v), t(e, v, v))
yes
```

5.4 練習問題

5.1 A がアトムで、N が 0 またはプラスの整数だとするとき、multi_atom(A, N, M) というゴールで呼び出すと、N 個の A を連結することによってできるアトムと M とを単一化する、multi_atom/3 という述語の定義を書いてください。

実行例
```
?- multi_atom(neko, 7, X).
X = nekonekonekonekonekonekoneko
yes
```

5.2 N が 0 またはプラスの整数だとするとき、number_to_binary(N, B) というゴールで呼び出すと、N を 2 進数であらわしているアトムと B とを単一化する、number_to_binary/2 という述語の定義を書いてください。

実行例
```
?- number_to_binary(83, X).
X = '1010011'
yes
```

ヒント n が 0 またはプラスの整数だとするとき、次のアルゴリズムを実行することによって、n をあらわす 2 進数を求めることができます。

- n が 0 ならば、n をあらわす 2 進数は 0 である。
- n が 1 ならば、n をあらわす 2 進数は 1 である。
- n が 2 以上ならば、n をあらわす 2 進数は、n を 2 で除算したときの商をあらわす 2 進数の右側に、n を 2 で除算したときのあまりをあらわす 2 進数を連結した結果である。

5.3 N が後継関数で記述された自然数だとするとき、even(N) というゴールで呼び出すと、N が偶数ならば成功して、そうでなければ失敗する、even/1 という述語の定義を書いてください。

実行例
```
?- even(s(s(s(s(s(s(0)))))).
yes
?- even(s(s(s(s(s(s(s(0)))))))).
no
```

5.4 N が後継関数で記述された自然数だとするとき、odd(N) というゴールで呼び出すと、N が奇数ならば成功して、そうでなければ失敗する、odd/1 という述語の定義を書いてください。

実行例
```
?- odd(s(s(s(s(s(s(0)))))).
yes
?- odd(s(s(s(s(s(s(s(0)))))))).
no
```

5.5 n が自然数だとするとき、n から 1 までの整数をすべて乗算した結果、つまり、

$$n \times (n-1) \times (n-2) \times \cdots \times 1$$

という計算の結果のことを、n の「階乗」(factorial) と呼んで、$n!$ と書きあらわします。ただし、$0!$ は 1 だと定義します。

たとえば、$5!$ は、

$$5 \times 4 \times 3 \times 2 \times 1$$

という計算をすればいいわけですから、120 ということになります。

N が後継関数で記述された自然数だとするとき、factorial(N, F) というゴールで呼び出すと、N の階乗と F とを単一化する、factorial/2 という述語の定義を、Q 5.2.5 で紹介した multiply/3 を使って書いてください。

実行例
```
?- factorial(s(s(s(0))), X).
X = s(s(s(s(s(0)))))
yes
```

ヒント $n!$ は、

$$\begin{cases} & 0! = 1 \\ n \geq 1 \text{ ならば} & n! = n \times (n-1)! \end{cases}$$

というように再帰的に定義することができます。

5.6 第0項と第1項が1で、第2項以降はその直前の2項を足し算した結果である、という数列のことを、「フィボナッチ数列」(Fibonacci sequence) と言います。フィボナッチ数列の第0項から第12項までを表にすると、次のようになります。

n	0	1	2	3	4	5	6	7	8	9	10	11	12
第 n 項	1	1	2	3	5	8	13	21	34	55	89	144	233

N が後継関数で記述された自然数だとするとき、fibonacci(N, F) というゴールで呼び出すと、フィボナッチ数列の第 N 項と F とを単一化する、fibonacci/2 という述語の定義を、Q 5.2.4 で紹介した add/3 を使って書いてください。

実行例
```
?- fibonacci(s(s(s(s(s(s(0)))))), X).
X = s(s(s(s(s(s(s(s(s(s(s(s(s)))))))))))
yes
```

ヒント フィボナッチ数列の第 n 項 (F_n) は、

$$\begin{cases} & F_0 = 1 \\ & F_1 = 1 \\ n \geq 2 \text{ ならば} & F_n = F_{n-2} + F_{n-1} \end{cases}$$

というように再帰的に定義することができます。

5.7 N と M が後継関数で記述された自然数だとするとき、greater(N, M) というゴールで呼び出すと、N のほうが M よりも大きいならば成功して、そうでなければ失敗する、greater/2 という述語の定義を、Q 5.2.3 で紹介した natural/1 を使って書いてください。

実行例
```
?- greater(s(s(s(s(s(0))))), s(s(s(0)))).
yes
?- greater(s(s(s(0))), s(s(s(s(s(0)))))).
no
?- greater(s(s(s(s(0)))), s(s(s(s(0))))).
no
```

5.8 N と M が後継関数で記述された自然数で、M は 0 ではないとするとき、divide(N, M, D) というゴールで呼び出すと、N を M で除算した結果と D とを単一化する、divide/3 という述語の定義を、Q 5.2.4 で紹介した add/3 と練習問題 5.7 で書いた greater/2 を使って書いてください。

実行例
```
?- divide(s(s(s(s(s(s(s(0))))))), s(s(s(0))), X).
X = s(s(0))
yes
```

5.9 N と M が後継関数で記述された自然数で、M は 0 ではないとするとき、mod(N, M, R) というゴールで呼び出すと、N を M で除算したときのあまりと R とを単一化する、mod/3 という述語の定義を、Q 5.2.4 で紹介した add/3 と練習問題 5.7 で書いた greater/2 を使って書いてください。

実行例
```
?- mod(s(s(s(s(s(s(s(s(0))))))), s(s(s(0))), X).
X = s(s(0))
yes
```

5.10 S が後継関数で記述された自然数だとするとき、ston(S, N) というゴールで呼び出すと、S と同じ自然数をあらわしている数値定数と N とを単一化する、ston/2 という述語の定義を書いてください。

実行例
```
?- ston(s(s(s(s(s(s(s(s(0))))))), X).
```

```
X = 8
yes
```

5.11 N が自然数をあらわしている数値定数だとするとき、ntos(N, S) というゴールで呼び出すと、N があらわしている自然数を後継関数で記述したものと S とを単一化する、ntos/2 という述語の定義を書いてください。

実行例
```
?- ntos(8, X).
X = s(s(s(s(s(s(s(s(0)))))))
yes
```

5.12 n が 0 ではない自然数で、m が自然数だとするとき、n と m の両方に共通する約数のうちで最大のもののことを、n と m の「最大公約数」(greatest common measure, GCM) と呼びます（m が 0 の場合は、n と m の最大公約数は n だと定義します)。たとえば、100 と 36 の最大公約数は 4 です。

N と M が後継関数で記述された自然数で、N は 0 ではないとするとき、gcm(N, M, G) というゴールで呼び出すと、N と M の最大公約数と G とを単一化する、gcm/3 という述語の定義を、Q 5.2.3 で紹介した natural/1 と練習問題 5.9 で書いた mod/3 を使って書いてください。

実行例
```
?- gcm(s(s(s(s(s(s(0)))))), s(s(s(s(s(s(s(s(s(0)))))))), X).
X = s(s(s(0)))
yes
```

ヒント n が 0 ではない自然数で、m が自然数だとするとき、n と m の最大公約数は、「ユークリッドの互除法」(Euclidean algorithm) と呼ばれるアルゴリズムを実行することによって求めることができます。それは次のようなアルゴリズムです。

- m が 0 ならば、n と m の最大公約数は n である。
- m が 0 ではないならば、n と m の最大公約数は、n を m で除算したときのあまりが r だとすると、m と r の最大公約数である。

5.13 S と T が木だとするとき、それらが同じものであるか、または T の一部分として S が含まれているならば、S は T の「部分木」(subtree) であると言われます。

S と T が二分木だとするとき、subtree(S, T) というゴールで呼び出すと、S が T の部分木ならば成功して、そうでなければ失敗する、subtree/2 という述語の定義を書いてください。

実行例
```
?- subtree(t(d, v, v),
           t(a, t(b, v, v), t(c, t(d, v, v), t(e, v, v)))).
yes
?- subtree(t(x, v, v),
           t(a, t(b, v, v), t(c, t(d, v, v), t(e, v, v)))).
no
```

5.14 T_A と T_B が二分木だとするとき、それらが、次の三つの条件のうちのいずれかを満足しているならば、T_A と T_B は「同形である」(isomorphic) と言われます。

- T_A と T_B はどちらも空である。
- T_A の根と T_B の根が同一で、かつ、T_A の左部分木と T_B の左部分木が同形で、かつ、T_A の右部分木と T_B の右部分木が同形である。
- T_A の根と T_B の根が同一で、かつ、T_A の左部分木と T_B の右部分木が同形で、かつ、T_A の右部分木と T_B の左部分木が同形である。

T_A と T_B が二分木だとするとき、isotree(T_A, T_B) というゴールで呼び出すと、T_A と T_B が同形ならば成功して、そうでなければ失敗する、isotree/2 という述語の定義を書いてください。

実行例
```
?- isotree(t(a, t(b, v, v), t(c, t(d, v, v), t(e, v, v))),
           t(a, t(c, t(d, v, v), t(e, v, v)), t(b, v, v))).
yes
?- isotree(t(a, t(b, v, v), t(c, t(d, v, v), t(e, v, v))),
           t(a, t(c, v, v), t(b, t(d, v, v), t(e, v, v)))).
```

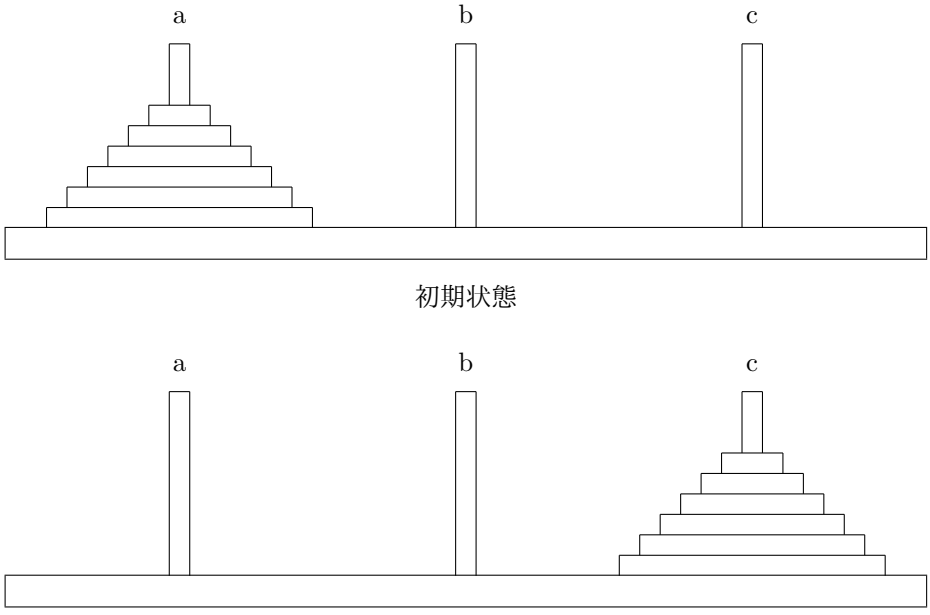

図 5.1: ハノイの塔

```
no
```

5.15　T が二分木で、その節点はすべて数値定数だとするとき、sum_tree(T, S) というゴールで呼び出すと、T に含まれているすべての節点を加算した結果と S とを単一化する、sum_tree/2 という述語の定義を書いてください。

実行例
```
?- sum_tree(t(3, t(7, v, v), t(4, v, t(6, v, v))), X).
X = 20
yes
```

5.16　「ハノイの塔」(Tower of Hanoi) と呼ばれるパズルがあります。このパズルでは、3 本の棒が垂直に立っている台と、直径が少しずつ違う何枚かの円盤が道具として使われます。それらの円盤は、中央に穴が開いていて、台の上の棒にはめ込むことができるようになっています。

ハノイの塔は、円盤を動かしていくことによって初期状態から最終状態へ移行させるための手順を求めてください、というパズルです。初期状態と最終状態というのは、図 5.1 のような状態のことです。つまり、初期状態では、すべての円盤が棒 a にはまっていて、しかも下にある円盤ほど直径が大きいという順番になっています。そして最終状態では、すべての円盤が棒 c にはまっていて、そして初期状態と同じように、下にある円盤ほど直径が大きいという順番になっていないといけません。

円盤を動かすときには、次の二つの規則にしたがう必要があります。

- 1 回の操作で実行できるのは、どれかの棒にはめ込まれている円盤のうちでもっとも上にある 1 枚を棒から抜き取って、それを別の棒にはめ込む、ということだけである。
- すでに棒にはめ込まれている円盤よりも直径の大きな円盤をその棒にはめ込むことはできない。

N が 0 またはプラスの整数だとするとき、hanoi(N) というゴールで呼び出すと、円盤の枚数が N 枚のハノイの塔を解くための手順を出力する、hanoi/1 という述語の定義を書いてください。

実行例
```
?- hanoi(3).
a -> c
a -> b
c -> b
a -> c
```

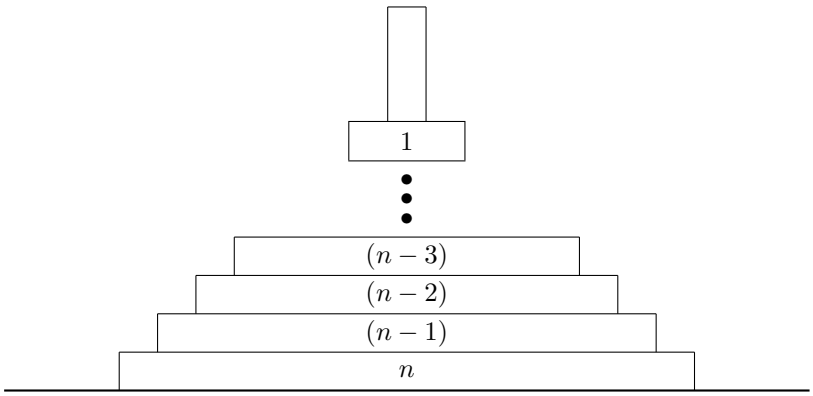

図 5.2: 円盤の番号

```
b -> a
b -> c
a -> c
yes
```

ヒント　円盤の枚数が n 枚だとするとき、図 5.2 のように、直径の大きな円盤から順番に、n、$(n-1)$、$(n-2)$、$(n-3)$、……、1、という番号がそれぞれの円盤に与えられているとします。そして、n 番目から 1 番目までの円盤を、上へ行くほど小さくなるように重ねたもののことを、「n 円錐」と呼ぶことにします。そうすると、n 円錐から n 番目の円盤を取り除いた部分は、「$(n-1)$ 円錐」と呼ばれることになります。

ハノイの塔の 3 本の棒のそれぞれは、「出発点」、「待避所」、「目的地」という 3 種類の役割を持っていると考えることができます。出発点というのは、n 円錐がそこから出発していく棒のことで、目的地というのは、移動が終わったときに n 円錐がはめ込まれている棒のことです。そして待避所というのは、n 番目の円盤を移動させるために $(n-1)$ 円錐を待避させておくための棒のことです。

ただし、どの棒がどの役割なのかという関係は、固定されていません。パズルの全体としては、棒 a が出発点で、棒 b が待避所で、棒 c が目的地ですが、パズルを解いていく過程で、棒と役割の関係は変化します。

ハノイの塔は、次のアルゴリズムを実行することによって解くことができます。

ステップ 1　$(n-1)$ 円錐を出発点から目的地経由で待避所へ移動させる。
ステップ 2　n 番目の円盤を出発点から目的地へ移動させる。
ステップ 3　$(n-1)$ 円錐を待避所から出発点経由で目的地へ移動させる。

図 5.3 は、このアルゴリズムを図で示したものです。

このアルゴリズムの中のステップ 1 とステップ 3 は、全体と同じ構造ですので、ハノイの塔を解くアルゴリズムを再帰的に使うことによって解くことができます。

第 6 章　リスト

6.1　リストの基礎

Q 6.1.1　列って何ですか。

A　「列」(sequence) というのは、何個かのものが 1 列に並んでいるという構造のことです。

Prolog の複合項は、ひとつの列を記述したものだと考えることができます。たとえば、

```
sequence(a, b, c, d, e, f, g)
```

という複合項は、`a`、`b`、`c`、`d`、`e`、`f`、`g` がこの順序で並んでいるという列をあらわしています。

しかし、このような、単独の複合項を使って列を記述するという方法は、長さが決まっていない列を処理する述語を定義することが困難であるという欠点を持っています。ですから、Prolog

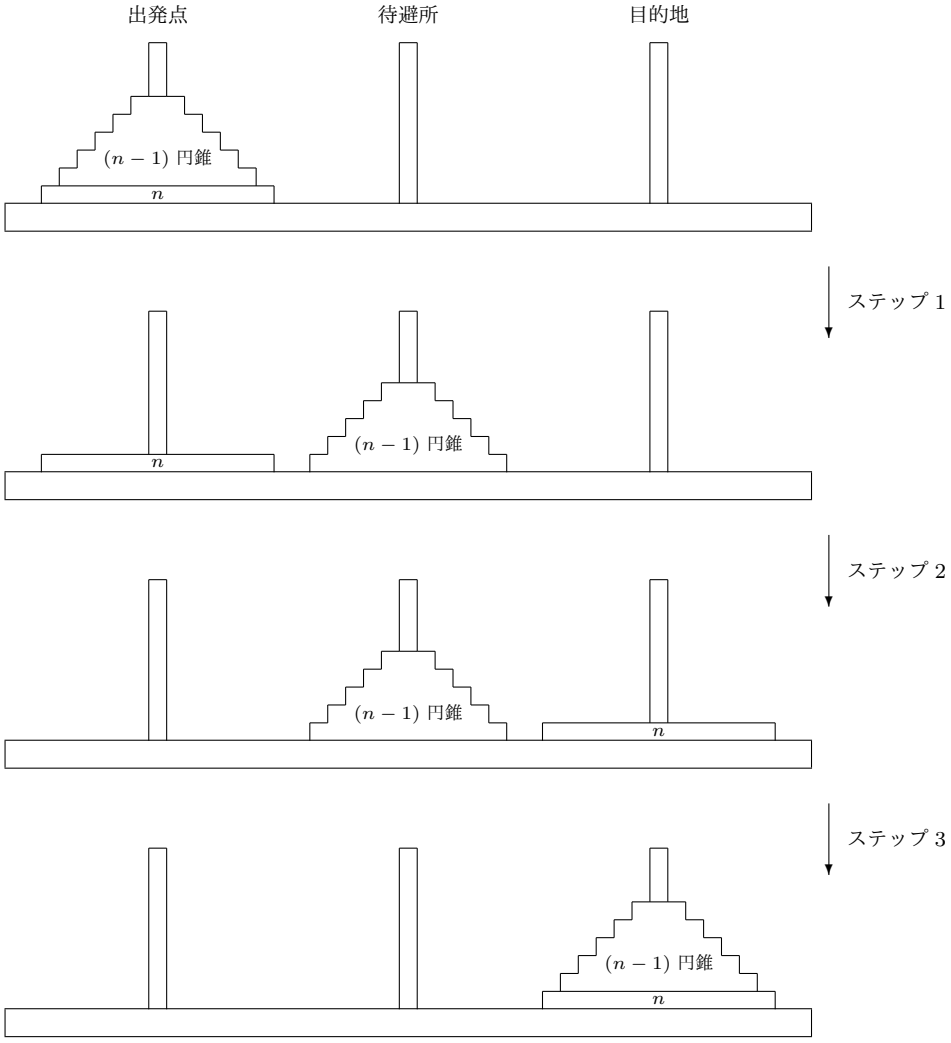

図 5.3: ハノイの塔の解法

では普通、列を記述するときは、単独の複合項を使うという方法ではなくて、項数が 2 の複合項を組み合わせるという方法が使われます。

Q 6.1.2　リストって何ですか。

A　「リスト」(list) というのは、項数が 2 の複合項を組み合わせることによって列を記述したもののことです。

リストを書くためには、あらかじめ、空の列をあらわすアトムと、複合項の関数子にするアトムを決めておく必要があります。

空の列をあらわすアトムとして、Prolog では、Q 2.3.10 で紹介した空リストというアトム、つまり `[]` というアトムを使います。そして、リストを書くための複合項の関数子として、Prolog では、ドット (.) というアトムを使います。

リストは、空リストか、またはドットを関数子とする項数が 2 の複合項です。リストを使って列を記述したいときは、複合項の 1 個目の引数として列の先頭を書いて、2 個目の引数として列の残りを書きます。たとえば、`a`、`b`、`c`、`d`、`e`、`f`、`g` がこの順序で並んでいるという列は、

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

というリストを書くことによって記述することができます。

リストは、次のように再帰的に定義することができます。

- 空リストはリストである。

- ドットを関数子とする項数が 2 の複合項で、2 個目の引数がリストであるものは、リストである。
- 以上の記述から導かれるもの以外はリストではない。

Q 6.1.3 リストの頭部とか尾部とかって、何のことですか。

A L が空リストではないリストだとするとき、L の 1 個目の引数のことを「頭部」(head) と呼び、L の 2 個目の引数のことを「尾部」(tail) と呼びます。

たとえば、

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

というリストの場合、頭部は a というアトムで、尾部は、

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

というリストです。

Q 6.1.4 リストを頭部と尾部に分解したいときって、どうすればいいんですか。

A リストを頭部と尾部に分解したいときは、頭部と尾部のそれぞれに変数を書いたリストと、分解したいリストとを単一化します。

たとえば、

```
.(X, Y) = .(a, .(b, .(c, .(d, .(e, .(f, .(g, []))))))) 
```

という単一化によって、右側のリストの頭部が X の値になって、尾部が Y の値になります。

Q 6.1.5 リストの要素って、何のことですか。

A リストの「要素」(element) というのは、リストによって記述された列を構成しているそれぞれの項のことです。

たとえば、

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

というリストの要素は、a、b、c、d、e、f、g、というアトムです。

Q 6.1.6 リストから要素を取り出したいときって、どうすればいいんですか。

A リストから要素を取り出したいときは、要素として変数を書いたリストと、要素を取り出したいリストとを単一化します。

たとえば、

```
.(_, .(_, .(X, _))) = .(a, .(b, .(c, .(d, ,(e, [])))))
```

という単一化によって、右側のリストが持っている 3 番目の要素が X の値になります。

Q 6.1.7 リストの長さって、何のことですか。

A リストの「長さ」(length) というのは、そのリストを構成している要素の個数のことです。

たとえば、

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

というリストの長さは、7 です。
ちなみに、空リスト ([]) は、長さが 0 のリストです。

Q 6.1.8 リストの略記法って、どんな書き方なんですか。

A リストの略記法というのは、要素をコンマで区切って並べて、その全体を角括弧 (square bracket, []) で囲む、というリストの書き方のことです。

たとえば、

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

というリストは、略記法を使って書かれた、

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

というリストと同じものだとみなされます。

Q 6.1.9　略記法を使ってリストを書くとき、その中にホワイトスペースを書くことって、可能ですか。

A　はい、可能です。

略記法を使ってリストを書くとき、左角括弧の右側、右角括弧の左側、そしてコンマの左右には、ホワイトスペースを書くことができます。たとえば、

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

というリストは、

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

と書くこともできますし、

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

と書くこともできます。

Q 6.1.10　尾部として変数を書いたリストを略記法で書きたいときって、どうすればいいんですか。

A　尾部として変数を書いたリストを略記法で書きたいときは、縦棒 (vertical line, |) という文字を使います。

リストの略記法としては、コンマを使って要素を列挙する書き方のほかに、もうひとつ別の書き方があります。それは、

[頭部|尾部]

というように、中央に縦棒を書いて、その左右に頭部と尾部を書く、という書き方です（頭部と縦棒とのあいだ、縦棒と尾部とのあいだにはホワイトスペースを書いてもかまいません)。たとえば、

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

というリストは、縦棒を使うことによって、

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

と書き換えることができます。
尾部として変数を書いたリストを略記法で書きたいときは、縦棒の右側にその変数を書きます。たとえば、

```
.(X, Y)
```

というリストは、縦棒を使うことによって、

```
[X|Y]
```

と書き換えることができます。

Q 6.1.11　行儀の悪いリストって、どんなリストのことなんですか。

A　「行儀の悪いリスト」(improper list) というのは、リストの基底にある空リストを、空リスト以外の項に置き換えたもののことです。

リストという再帰的な構造を持つ複合項の基底には、かならず空リストを書く必要があります。ですから、

```
.(a, .(b, .(c, .(d, .(e, .(f, .(g, z))))))
```

のような、基底として空リスト以外の項が書かれている複合項をリストと呼ぶことはできません。このような、基底が空リストではないリストもどきのことを、「行儀の悪いリスト」と呼びます。

Q 6.1.12 行儀の悪いリストを略記法で書きたいときって、どうすればいいんですか。

A 行儀の悪いリストを略記法で書きたいときは、縦棒 (|) を使います。

Q 6.1.10 で説明したように、縦棒は、その左側がリストの頭部で、右側がリストの尾部だということを示す文字です。この文字を使えば、行儀の悪いリストを略記法で書くことが可能になります。たとえば、

```
.(a, z)
```

という行儀の悪いリストは、縦棒を使うことによって、

```
[a|z]
```

と書き換えることができます。同じように、

```
.(a, .(b, .(c, .(d, .(e, .(f, .(g, z)))))))
```

という行儀の悪いリストは、縦棒を使うことによって、

```
[a, b, c, d, e, f, g|z]
```

と書き換えることができます。

6.2 リストの処理

Q 6.2.1 リストの末尾の要素を求めるアルゴリズムって、あるんですか。

A はい、あります。

L が、長さが 1 以上のリストだとするとき、次のアルゴリズムを実行することによって、L の末尾の要素を求めることができます。

- L が [H] ならば、L の末尾の要素は H である。
- L が [H] ではなくて [H|T] ならば、T の末尾の要素が L の末尾の要素である。

次の last_elem/2 という述語の定義は、このアルゴリズムを Prolog で記述したものです。

```
last_elem([H], H).
last_elem([_|T], L) :- last_elem(T, L).
```

この述語を使うことによって、次のように、リストの末尾の要素を求めることができます。

```
?- last_elem([a, b, c, d, e, f, g], X).
X = g
yes
```

Q 6.2.2 リストの長さを求めるアルゴリズムって、あるんですか。

A はい、あります。

L がリストだとするとき、次のアルゴリズムを実行することによって、L の長さを求めることができます。

- L が [] ならば、L の長さは 0 である。
- L が [H|T] ならば、L の長さは、T の長さに 1 を加算した結果である。

次の list_length/2 という述語の定義は、このアルゴリズムを Prolog で記述したものです。

```
list_length([], 0).
list_length([_|T], L) :- list_length(T, LT), L is LT + 1.
```

この述語を使うことによって、次のように、リストの長さを求めることができます。

```
?- list_length([a, b, c, d, e, f, g], X).
X = 7
yes
```

Q 6.2.3　N が 1 以上の整数だとするとき、リストの N 番目の要素を求めるアルゴリズムって、あるんですか。

A　はい、あります。

L がリストで、N が 1 以上の整数だとするとき、次のアルゴリズムを実行することによって、L の N 番目の要素を求めることができます。

- L が [H|_] で、N が 1 ならば、L の N 番目の要素は H である。
- L が [_|T] で、N が 2 以上ならば、L の N 番目の要素は、T の $N-1$ 番目の要素である。

次の nth/3 という述語の定義は、このアルゴリズムを Prolog で記述したものです。

```
nth([H|_], 1, H).
nth([_|T], N, E) :- N >= 2, N1 is N - 1, nth(T, N1, E).
```

この述語を使うことによって、次のように、リストの N 番目の要素を求めることができます。

```
?- nth([a, b, c, d, e, f, g], 4, X).
X = d
yes
```

Q 6.2.4　二つのリストを連結するアルゴリズムって、あるんですか。

A　はい、あります。

二つのリストを連結するというのは、たとえば、

```
[a, b, c, d, e]
```

というリストと、

```
[1, 2, 3, 4]
```

というリストが与えられたとするときに、一方の末尾に他方を追加した、

```
[a, b, c, d, e, 1, 2, 3, 4]
```

というリストを作るという処理のことです。

L_L と L_R がリストだとするとき、次のアルゴリズムを実行することによって、L_L の右側に L_R を連結したリストを求めることができます。

- L_L が [] ならば、L_L の右側に L_R を連結したリストは L_R である。
- L_L が [H|T] で、T の右側に L_R を連結したリストが A ならば、L_L の右側に L_R を連結したリストは [H|A] である。

次の appand_list/3 という述語の定義は、このアルゴリズムを Prolog で記述したものです。

```
append_list([], L, L).
append_list([H|T], L, [H|A]) :- append_list(T, L, A).
```

この述語を使うことによって、次のように、二つのリストを連結することができます。

```
?- append_list([a, b, c, d, e], [1, 2, 3, 4], X).
X = [a, b, c, d, e, 1, 2, 3, 4]
yes
```

後継関数で記述された二つの自然数を加算する述語を使って自然数を減算した結果を求めることができるのと同じように、リストを連結する述語を使って、リストの右の部分を取り除いたり左の部分を取り除いたりすることができます。

```
?- append_list(X, [e, f, g], [a, b, c, d, e, f, g]).
X = [a, b, c, d]
yes
?- append_list([a, b, c, d], X, [a, b, c, d, e, f, g]).
X = [e, f, g]
yes
?- append_list(X, Y, [a, b, c, d, e, f, g]),
   write(X), write(' '), write(Y), nl, fail.
[] [a, b, c, d, e, f, g]
[a] [b, c, d, e, f, g]
[a, b] [c, d, e, f, g]
```

```
[a, b, c] [d, e, f, g]
[a, b, c, d] [e, f, g]
[a, b, c, d, e] [f, g]
[a, b, c, d, e, f] [g]
[a, b, c, d, e, f, g] []
no
```

Q 6.2.5 リストの要素を逆の順序で並べ替えたリストを求めるアルゴリズムって、あるんですか。

A はい、あります。

L がリストだとするとき、次のアルゴリズムを実行することによって、L の要素を逆の順序で並べ替えたリストを求めることができます。

- L が [] ならば、L の要素を逆の順序で並べ替えたリストは [] である。
- L が $[H|T]$ で、T の要素を逆の順序で並べ替えたリストが R_T で、R_T の右側に $[H]$ を連結したリストが R ならば、L の要素を逆の順序で並べ替えたリストは、R である。

次の reverse_list/2 という述語の定義は、このアルゴリズムを Prolog で記述したものです（リストの連結は、Q 6.2.4 で紹介した append_list を使っています）。

```
reverse_list([], []).
reverse_list([H|T], R) :-
    reverse_list(T, RT), append_list(RT, [H], R).
```

この述語を使うことによって、次のように、リストの要素を逆の順序で並べ替えることができます。

```
?- reverse_list([a, b, c, d, e, f, g], X).
X = [g, f, e, d, c, b, a]
yes
```

Q 6.2.6 累算器って何ですか。

A 「累算器」(accumulator) というのは、事実または規則の頭部の中に書かれた変数のうちで、処理の対象でも処理の結果でもなく、処理の途中経過と単一化されるもののことです。

累算器は、リストを処理する述語を定義する上で絶対に必要なものだとは言えませんが、累算器を使うことによって効率を改善することができる場合もあります。

累算器を使って述語を定義する場合は、その述語よりも項数の多い補助的な述語を定義する必要があります。たとえば、リストの長さを求める list_length/2 という述語の定義を累算器を使って書くと、次のようになります。

```
list_length(L, N) :- list_length(L, 0, N).

list_length([], A, A).
list_length([_|T], A, N) :-
    A1 is A + 1, list_length(T, A1, N).
```

このように、項数が 2 の述語を定義するために、項数が 3 の述語を補助的に定義するわけです。項数が 3 の述語の定義の中で使われている A という変数が累算器です。A と単一化される整数定数は、述語が再帰的に呼び出されるたびに 1 ずつ大きくなっていきます。たとえば、

```
list_length([a, b, c, d], 0, X)
```

というゴールで list_length/3 を呼び出したとすると、1 番目の引数と 2 番目の引数は、次のように変化していきます。

1 番目の引数	2 番目の引数
[a, b, c, d]	0
[b, c, d]	1
[c, d]	2
[d]	3
[]	4

このようにして、1 番目の引数が [] になったとき、2 番目の引数は最初のリストの長さになります。そのとき、2 番目の引数は、

```
list_length([], A, A).
```

という事実によって、3 番目の引数へ移されます。

次の述語定義は、リストの要素を逆の順序で並べ替えたリストを求める述語の定義を、累算器を使って書いたものです。

```
reverse_list(L, R) :- reverse_list(L, [], R).

reverse_list([], A, A).
reverse_list([H|T], A, R) :- reverse_list(T, [H|A], R).
```

項数が 3 の reverse_list の定義の中で使われている A という変数が累算器です。A と単一化されるリストは、述語が再帰的に呼び出されるたびに、左側に要素が追加されていきます。たとえば、

```
reverse_list([a, b, c, d], [], X)
```

というゴールで reverse_list/3 を呼び出したとすると、1 番目の引数と 2 番目の引数は、次のように変化していきます。

1 番目の引数	2 番目の引数
[a, b, c, d]	[]
[b, c, d]	[a]
[c, d]	[b, a]
[d]	[c, b, a]
[]	[d, c, b, a]

このようにして、1 番目の引数が [] になったとき、2 番目の引数は、最初のリストの要素を逆の順序で並べ替えたリストになります。そのとき、2 番目の引数は、

```
reverse_list([], A, A).
```

という事実によって、3 番目の引数へ移されます。

リストの長さを求める述語の場合は、累算器を使ったからと言って、それを使わない方法よりも効率がよくなるわけではありません。それに対して、リストの要素を逆の順序で並べ替える述語の場合は、累算器を使うことによって、二つのリストを連結するという処理が必要ではなくなりますので、効率がかなり改善されます。

6.3　写像

Q 6.3.1　リストの写像って、何のことですか。

A　リストの「写像」(mapping) というのは、リストを構成しているそれぞれの要素に対して何らかの処理を実行して、その結果から構成されるリストを作る、という処理のことです。

たとえば、数値から構成されるリストが与えられたときに、そのリストを構成しているそれぞれの数値を 2 乗して、その結果から構成されるリストを作る、という処理は、写像の一例です。たとえば、

```
[7, 3, 10, 4, 6]
```

というリストに対してこの写像を実行すると、

```
[49, 9, 100, 16, 36]
```

というリストが得られます。

次の述語定義によって定義される square_list/2 という述語は、リストの要素を 2 乗する写像を実行します。

```
square_list([], []).
square_list([H|T], [SH|ST]) :-
    SH is H * H, square_list(T, ST).
```

この述語を使うことによって、次のように、リストの要素を 2 乗したリストを作ることができます。

```
?- square_list([7, 3, 10, 4, 6], X).
X = [49, 9, 100, 16, 36]
yes
```

Q 6.3.2 要素に対する処理を引数で指定することのできる汎用的な写像の述語って、定義できるんですか。

A はい、できます。

たとえば、数値の 2 乗を求める square/2 という述語が、

```
square(N, S) :- S is N * N.
```

と定義されているとするとき、

```
?- map_list(square, [7, 3, 10, 4, 6], X).
X = [49, 9, 100, 16, 36]
yes
```

というように、リストのそれぞれの要素を square/2 で処理した結果から構成されるリストを作る、 map_list/3 という汎用的な写像の述語を定義することは、可能です。
ただし、そのためには、「ユニブ」と呼ばれる処理を実行する必要があります。

Q 6.3.3 ユニブって何ですか。

A 「ユニブ」(univ) というのは、複合項が与えられたときに、その関数子を頭部、その引数から構成されるリストを尾部とするリストを求めるという処理、または、それとは逆に、アトムを頭部とするリストが与えられたときに、その頭部を関数子、その尾部の要素を引数とする複合項を求めるという処理のことです。

たとえば、

```
a(b, c, d, e, f, g)
```

という複合項と、

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

というリストとは、ユニブを実行することによって、相互に他方へ変換することができます。

Q 6.3.4 ユニブを実行する組み込み述語って、あるんですか。

A はい、あります。

=.. という中置演算子を名前とする、(=..)/2 という組み込み述語は、左側の複合項と右側のリストとのあいだでユニブを実行します。たとえば、

```
a(b, c, d, e, f, g) =.. X
```

というゴールを実行すると、 X の値として、

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

というリストが得られます。また逆に、

```
X =.. [a, b, c, d, e, f, g]
```

というゴールを実行すると、 X の値として、

```
a(b, c, d, e, f, g)
```

という複合項が得られます。

Q 6.3.5 汎用的な写像の述語を定義するとき、どうしてユニブが必要になるんですか。

A 汎用的な写像の述語を定義するためには、引数として受け取った述語名を関数子とする複合項を作る必要があります。その複合項を作るためにユニブが必要なのです。

たとえば、汎用的な写像を実行する map_list/3 という述語を、

```
map_list(square, [7, 3, 10, 4, 6], X)
```

というゴールで呼び出したとすると、map_list/3 は、リストの頭部を処理するために、

```
square(7, S)
```

というゴールを実行しないといけません。そのゴールを作るために、ユニブが必要になるのです。

汎用的な写像を実行する map_list/3 は、ユニブを実行する組み込み述語の (=..)/2 を使うことによって、次のように定義を書くことができます。

```
map_list(_, [], []).
map_list(P, [H|T], [PH|PT]) :-
    G =.. [P, H, PH], G, map_list(P, T, PT).
```

それでは、map_list/3 を使って、リストの写像を実行してみましょう。

map_list/3 を使ってリストの写像を実行するためには、それぞれの要素を処理する項数が 2 の述語が必要です。その述語は、1 個目の引数を処理した結果と 2 個目の引数とを単一化するものでないといけません。たとえば、1 個目の引数を式として評価して得られた値と 2 個目の引数とを単一化する eval/2 という述語を、

```
eval(E, V) :- V is E.
```

と定義したとしましょう。そうすると、map_list/3 を使うことによって、

```
?- map_list(eval, [3+4, 100//8, 100 mod 8, 6*(15-7)], X).
X = [7, 12, 4, 48]
yes
```

というように、リストのそれぞれの要素を eval/2 で評価した結果から構成されるリストが得られます。

組み込み述語の中にも、map_list の 1 個目の引数としてその名前を書くことによって、それによる写像を実行することのできるものがあります。たとえば (=..)/2 は、そのような組み込み述語のひとつです。

(=..)/2 は、1 個目（左側）の引数をリストに変換した結果と 2 個目（右側）の引数とを単一化しますから、それを使って、リストを構成しているそれぞれの複合項をリストに変換した結果から構成されるリストを作るという写像を実行することができます。つまり、

```
?- map_list(=.., [a(b, c, d), e(f), g(h, i, j, k)], X).
X = [[a, b, c, d], [e, f], [g, h, i, j, k]]
yes
```

というような写像を実行することができるということです。

6.4 文字列

Q 6.4.1 文字コードリストって何ですか。

A 「文字コードリスト」(character code list) というのは、文字列を記述しているリストのことで、文字列を構成しているそれぞれの文字の文字コードをあらわす整数定数を、その順番のとおりに並べたものです。

たとえば、namako という文字列は、

```
[0'n, 0'a, 0'm, 0'a, 0'k, 0'o]
```

という文字コードリストによって記述することができます。文字コードとして ASCII が使われているとすれば、このリストは、

```
[110, 97, 109, 97, 107, 111]
```

と書き換えることもできます。

なお、これから先、具体的な文字コードの例を書くときは、文字コードとして ASCII が使われているという断り書きを省略することにします。

Q 6.4.2 文字コードリストって、どんなときに使われるものなんですか。

A 文字コードリストは、文字列の内部を処理したいときに使われるものです。

Prolog では、アトムを使って文字列を記述することができます。しかし、アトムを使って記述された文字列は、その内部を処理するということができません。文字列の内部を処理するためには、アトムではなくて、文字コードリストを使ってそれを記述する必要があります。

文字コードリストを使って文字列を記述すれば、リストを処理するための述語を使うことによって、その文字列の内部を処理することができます。たとえば、リストの N 番目の要素を求める述語を使うことによって文字列の N 番目の文字を求めることができますし、リストの要素を逆の順序で並べ替える述語を使うことによって文字列の文字を逆の順序で並べ替えることができます。

Q 6.4.3 文字コードリストの略記法って、どんな書き方なんですか。

A 文字コードリストの略記法というのは、文字列を二重引用符 (double quote, `"`) で囲む、という書き方のことです。

たとえば、`namako` という文字列を記述している、

```
[0'n, 0'a, 0'm, 0'a, 0'k, 0'o]
```

という文字コードリストは、略記法を使うことによって、

```
"namako"
```

と書きあらわすことができます。

基本的には、任意の文字列を二重引用符で囲んだものはすべて略記法の文字コードリストになると考えていいのですが、そうならない例外もあります。たとえば、

```
"I saw "Hamlet" last night."
```

というような、二重引用符を含んでいる文字列を二重引用符で囲んだものや、

```
"I am a very
very long string."
```

というような、改行を含んでいる文字列を二重引用符で囲んだものは、略記法の文字コードリストとはみなされません。

Q 6.4.4 二重引用符を含んでいる文字列を文字コードリストの略記法で書きたいときって、どうすればいいんですか。

A 二重引用符を含んでいる文字列を文字コードリストの略記法で書きたいときは、二重引用符を二つ続けて書きます。

文字コードリストを略記法で書くとき、2 個の連続する二重引用符をその中に書くことができて、それは 1 個の二重引用符とみなされます。たとえば、

```
"I saw ""Hamlet"" last night."
```

という文字コードリストを書くことができて、これは、

```
I saw "Hamlet" last night.
```

という文字列を記述したものだとみなされます。

Q 6.4.5 略記法の文字コードリストを 2 行以上に分けて書きたいときって、どうすればいいんですか。

A 改行 (newline) という文字の直前にバックスラッシュ(backslash, `\`) という文字を書くことによって、略記法の文字コードリストを 2 行以上に分けて書くことができます。

略記法の文字コードリストの中では、バックスラッシュの直後に改行を書くと、その改行は無視されます。ですから、

```
"I am a very very long string."
```

という略記法の文字コードリストは、バックスラッシュを使うことによって、

```
"I am a very \
very long string."
```

というように、2 行に分けて書くことができます。

Q 6.4.6　略記法の文字コードリストを使って、改行という文字を含む文字列を記述したいときって、どうすればいいんですか。

A　略記法の文字コードリストを使って、改行 (newline) という文字を含む文字列を記述したいときは、エスケープシーケンス (escape sequence) を使います。

エスケープシーケンスというのは、そのままでは引用アトムの中に書くことのできない文字（たとえば改行）をあらわす、バックスラッシュ(backslash, \) という文字で始まる文字列のことです（詳細は Q 2.3.7 を参照してください)。たとえば、\n は、改行という文字をあらわすエスケープシーケンスです。

略記法の文字コードリストの中にエスケープシーケンスを書くと、それによってあらわされる文字がその位置にあると解釈されます。たとえば、

```
"ebi\nkani"
```

という記述は、

```
[101, 98, 105, 10, 107, 97, 110, 105]
```

という文字コードリストの略記法になります（10 というのが、ASCII での改行の文字コードです)。

Q 6.4.7　アトムを文字コードリストに変換したり、文字コードリストをアトムに変換したりしたいときって、どうすればいいんですか。

A　atom_codes/2 という組み込み述語を呼び出すことによって、アトムを文字コードリストに変換したり、文字コードリストをアトムに変換したりすることができます。

atom_codes/2 の引数は、1 個目がアトムで、2 個目が文字コードリストです。アトムを文字コードリストに変換したいときは、ゴールの 1 個目の引数としてアトムを書いて、2 個目の引数として変数を書きます。そうすると、そのアトムを文字コードリストに変換した結果と変数とが単一化されます。たとえば、

```
atom_codes(namako, X)
```

というゴールで atom_codes/2 を呼び出すと、

```
[110, 97, 109, 97, 107, 111]
```

という文字コードリストと X とが単一化されます。

文字コードリストをアトムに変換したいときは、ゴールの 1 個目の引数として変数を書いて、2 個目の引数として文字コードリストを書きます。そうすると、その文字コードリストをアトムに変換した結果と変数とが単一化されます。たとえば、

```
atom_codes(X, [0'u, 0'm, 0'i, 0'u, 0's, 0'h, 0'i])
```

というゴールで atom_codes/2 を呼び出すと、umiushi というアトムと X とが単一化されます。

Q 6.4.8　数値定数を文字コードリストに変換したり、文字コードリストを数値定数に変換したりしたいときって、どうすればいいんですか。

A　number_codes/2 という組み込み述語を呼び出すことによって、数値定数を文字コードリストに変換したり、文字コードリストを数値定数に変換したりすることができます。

number_codes/2 の引数は、1 個目が数値定数で、2 個目が文字コードリストです。数値定数を文字コードリストに変換したいときは、ゴールの 1 個目の引数として数値定数を書いて、2 個目の引数として変数を書きます。そうすると、その数値定数を文字コードリストに変換した結果と変数とが単一化されます。たとえば、

```
number_codes(387, X)
```

というゴールで number_codes/2 を呼び出すと、

```
[51, 56, 55]
```

という文字コードリストと X とが単一化されます。

文字コードリストを数値定数に変換したいときは、ゴールの 1 個目の引数として変数を書いて、2 個目の引数として文字コードリストを書きます。そうすると、その文字コードリストを数値定数に変換した結果と変数とが単一化されます。たとえば、

```
number_codes(X, [0'3, 0'8, 0'7])
```

というゴールで number_codes/2 を呼び出すと、387 という数値定数と X とが単一化されます。

数値変数を具体的にどのような文字コードリストに変換するのか、また文字コードを具体的にどのような数値変数に変換するのかという点については、Prolog のインタプリタの実装ごとに多少の相違があります。たとえば、

```
[0'8, 0'e, 0'5]
```

という文字コードリストは、8e5 という数値変数に変換されるかもしれませんし、800000 という数値変数に変換されるかもしれません。

6.5 練習問題

6.1 E が項で、N が 0 またはプラスの整数だとするとき、generate_list(E, N, L) というゴールで呼び出すと、E を N 個だけ並べることによってできるリストと L とを単一化する、generate_list/3 という述語の定義を書いてください。

実行例
```
?- generate_list(a, 10, X).
X = [a, a, a, a, a, a, a, a, a, a]
yes
```

6.2 E が項で、L がリストだとするとき、member_list(E, L) というゴールで呼び出すと、L の中に E が要素として含まれているならば成功して、そうでなければ失敗する、member_list/2 という述語の定義を書いてください。

実行例
```
?- member_list(d, [a, b, c, d, e, f, g]).
yes
?- member_list(m, [a, b, c, d, e, f, g]).
no
```

6.3 L がリストで、E が項だとするとき、delete_elem(L, E, D) というゴールで呼び出すと、L の中に要素として含まれているすべての E を L から取り除くことによってできるリストと D とを単一化する、delete_elem/3 という述語の定義を書いてください。

実行例
```
?- delete_elem([a, y, b, y, y, c, y, d, y, e, f], y, X).
X = [a, b, c, d, e, f]
yes
```

6.4 L がリストだとするとき、unique(L, U) というゴールで呼び出すと、L の中に重複して含まれている要素をひとつだけ残して削除することによってできるリストと U とを単一化する、unique/2 という述語の定義を書いてください。

実行例
```
?- unique([b, a, d, b, a, b, c, a, c, b, c, c, d, b, b], X).
X = [a, c, d, b]
yes
```

6.5 L がアトムから構成されるリストだとするとき、concat_list(L, C) というゴールで呼び出すと、L を構成しているそれぞれのアトムを、その順番のとおりに連結することによってできるアトムと C とを単一化する、concat_list/2 という述語の定義を書いてください。

実行例
```
?- concat_list([foo, bar, baz, corge, grault, plugh], X).
X = foobarbazcorgegraultplugh
yes
```

6.6 L が数値定数から構成されるリストだとするとき、sum_list(L, S) というゴールで呼び出すと、L を構成しているすべての数値定数の合計と S とを単一化する、sum_list/2 という述語の定義を書いてください。

実行例
```
?- sum_list([8, 3, 7, 2, 4, 6], X).
X = 30
yes
```

6.7　L が数値定数から構成されるリストだとするとき、max_list(L, M) というゴールで呼び出すと、L を構成している数値定数のうちでもっとも大きなものと M とを単一化する、max_list/2 という述語の定義を書いてください。なお、L が空リストの場合は失敗するようにしてください。

実行例
```
?- max_list([-3, 4, -6, 2, -8, 7, 0, -1], X).
X = 7
yes
```

6.8　N が 1 以上の整数だとするとき、n_to_one(N, L) というゴールで呼び出すと、

[N, $N-1$, $N-2$, ･･･, 1]

というリストと L とを単一化する、n_to_one/2 という述語の定義を書いてください。

実行例
```
?- n_to_one(10, X).
X = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
yes
```

6.9　N が 1 以上の整数だとするとき、one_to_n(N, L) というゴールで呼び出すと、

[1, 2, 3, ･･･, N]

というリストと L とを単一化する、one_to_n/2 という述語の定義を、累算器を使って書いてください。

実行例
```
?- one_to_n(10, X).
X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
yes
```

6.10　T が項だとするとき、count_atomic(T, N) というゴールで呼び出すと、T の中に含まれているアトムと数値定数の個数を数えて（ただし、空リストと、リストの関数子になっているドットは数えません）、その結果と N とを単一化する、count_atomic/2 という述語の定義を書いてください。

実行例
```
?- count_atomic([], X).
X = 0
yes
?- count_atomic(a, X).
X = 1
yes
?- count_atomic([a, [b, [c, d], [e, f], g], h], X).
X = 8
yes
?- count_atomic(a(7, b(c(6, 4), d(8, 10)), 12), X).
X = 10
yes
```

6.11　T が項だとするとき、atomiclist(T, L) というゴールで呼び出すと、T の中に含まれているアトムと数値定数から構成されるリスト（ただし、空リストと、リストの関数子になっているドットは除外します）を作って、その結果と L とを単一化する、atomiclist/2 という述語の定義を書いてください。

実行例
```
?- atomiclist(a, X).
X = [a]
yes
?- atomiclist([a, [b, [c, d], [e, f], g], h], X).
X = [a, b, c, d, e, f, g, h]
yes
?- atomiclist(a(7, b(c(6, 4), d(8, 10)), 12), X).
X = [a, 7, b, c, 6, 4, d, 8, 10, 12]
yes
```

6.12 P がアトムで、T が項だとするとき、deepmap(P, T, M) というゴールで呼び出すと、T に含まれているアトムと数値定数（ただし、空リストと、複合項の関数子になっているアトムは除外します）のそれぞれを、P を名前とする項数が 2 の述語を使って処理して、それらの結果から構成される T と同じ構造の項と M とを単一化する、deepmap/3 という述語の定義を書いてください。

実行例
```
?- listing(square/2).
square(A, B) :-
        B is A*A.
yes
?- deepmap(square, [12, [4, [10, 8], [7, 20], 15], 16], X).
X = [144, [16, [100, 64], [49, 400], 225], 256]
yes
?- deepmap(square, a(14, b(6, c(3, 2), d(7, 5), 32), 4), X).
X = a(196, b(36, c(9, 4), d(49, 25), 1024), 16)
yes
```

6.13 データの列が与えられたとき、何らかの大小関係にもとづいて、小さなものから大きなものへ、または大きなものから小さなものへ、という順序になるように、その列を並べ換えることを、データの列を「ソートする」(sort) と言います。なお、小さなものから大きなものへという順序のことを「昇順」(ascending order) と言い、大きなものから小さなものへという順序のことを「降順」(descending order) と言います。

L が数値定数から構成されるリストだとするとき、sort_list(L, S) というゴールで呼び出すと、数値の大小関係にもとづいて L を昇順にソートした結果と S とを単一化する、sort_list/2 という述語の定義を書いてください。

実行例
```
?- sort_list([53, 22, 8, 6, 10, 72, 3, 28, 41, 7, 18], X).
X = [3, 6, 7, 8, 10, 18, 22, 28, 41, 53, 72]
yes
```

ヒント

データの列をソートするアルゴリズムにはさまざまなものがあるのですが、ここでは、「クイックソート」(quick sort) と呼ばれるアルゴリズムを紹介しておきたいと思います。

L がリストだとするとき、クイックソートを使って L を昇順にソートするというのは、次のようなアルゴリズムを実行することです。

- L が空リストならば、L をソートした結果は空リストである。
- L が空リストではないならば、まず、L を A と B という 2 個のリストに分割する。ただし、A と B は、A に含まれるすべての要素は B に含まれるどの要素よりも小さい、という条件を満足していないといけない。次に、クイックソートを使って A と B をソートする。そして、A をソートした結果の右側に B をソートした結果を連結する。そうすると、その結果が、L をソートした結果である。

6.14 明確なものを明確に集めたものという数学上の概念は「集合」(set) と呼ばれます。集合を構成しているそれぞれのものは、その集合の「要素」(element) と呼ばれます。

Prolog には集合を記述するための特別な規則は定義されていませんが、リストを使って集合を記述することは可能です。任意のリストは、要素が並んでいる順番と要素の重複を無視すれば、集合をあらわしていると解釈することができます。

A と B が集合で、A のすべての要素が B にも含まれているとするとき、A のことを、B の「部分集合」(subset) と呼びます。

L_A と L_B がリストだとするとき、is_subset(L_A, L_B) というゴールで呼び出すと、L_A と L_B が集合をあらわしていると解釈したときに、L_A が L_B の部分集合になっているならば成功して、そうでなければ失敗する、is_subset/2 という述語の定義を書いてください。

実行例
```
?- is_subset([a, b, a], [x, y, a, x, b, y, b]).
yes
?- is_subset([a, b, a], [x, y, z, y, b, y, b]).
no
```

6.15 A と B が集合だとするとき、A と B の両方に共通して含まれているすべての要素を集め

ることによって作られた集合のことを、A と B の「共通部分」(intersection) と呼びます。

L_A と L_B がリストだとするとき、`inter_sets(`L_A`,` L_B`,` I`)` というゴールで呼び出すと、L_A と L_B が集合をあらわしていると解釈したときのそれらの共通部分をあらわすリストを求めて、その結果と I とを単一化する、`inter_sets/3` という述語の定義を書いてください。

実行例
```
?- inter_sets([a, b, c, d, c, d], [d, c, d, c, f, e], X).
X = [c, d, c, d]
yes
```

6.16　A が集合だとするとき、A のすべての部分集合から構成される集合のことを、A の「べき集合」(power set) と呼びます。たとえば、

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

というリストによってあらわされる集合のべき集合は、

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

というリストによってあらわされる集合です。

L がリストだとするとき、`power_set(`L`,` P`)` というゴールで呼び出すと、L が集合をあらわしていると解釈したときの L のべき集合をあらわすリストを求めて、その結果と P とを単一化する、`power_set/2` という述語の定義を書いてください。

実行例
```
?- power_set([a, b, b, c], X).
X = [[], [c], [b], [b, c], [a], [a, c], [a, b], [a, b, c]]
yes
```

第 7 章　実行の制御

7.1　論理演算

Q 7.1.1　論理演算って何ですか。

A　「論理演算」(logic operation) というのは、1 個以上の命題について、それぞれの命題の真偽値の組み合わせに対して 1 個の真偽値を対応させるという操作のことです。

論理演算には、「連言」(conjunction)、「選言」(disjunction)、「否定」(negation) などがあります。

Q 7.1.2　連言っていうのはどんな論理演算なんですか。

A　連言というのは、2 個の命題について、それらの両方が真という組み合わせに対しては真を対応させて、それ以外の組合せに対しては偽を対応させるという操作のことです。

日本語の「かつ」や英語の and は、連言をあらわしています。A と B が命題だとするとき、「かつ」を使うことによってそれらの命題を組み合わせた「A かつ B」という命題は、A と B の両方が真のときだけ全体が真になります。

数学や論理学では、A と B の連言を、$\wedge$ という記号を使って、

$$A \wedge B$$

という式で記述します。

Q 7.1.3　連言を実行する組み込み述語って、あるんですか。

A　はい、あります。

Q 3.3.8 で、コンマ (comma, `,`) という中置演算子を使うことによって「A かつ B」という形の命題を書くことができると説明しましたが、実は、そのときに紹介したコンマというのが、連言を実行する組み込み述語です。つまり、`(,)/2` という組み込み述語を呼び出すことによって、連言を実行することができるわけです。なお、この述語の名前になっている `,` というアトムは、優先順位が 1000 で記述子が `xfy` の演算子です。

A と B がゴールだとするとき、

A`,` B

というゴールで (,)/2 を呼び出すと、(,)/2 は、まず A を実行します。そして、A が成功したならば B を実行して、B も成功したならば自分自身も成功します。どちらかのゴールが失敗した場合は自分自身も失敗します。なお、A が失敗した場合、B は実行しません。

(,)/2 を呼び出す質問をインタプリタに入力してみると、次のようになります。

```
?- 5 == 5, 6 == 6.
yes
?- 5 == 5, 6 == 7.
no
?- 5 == 4, 6 == 6.
no
?- 5 == 4, 6 == 7.
no
```

Q 7.1.4 真理値表って何ですか。

A 「真理値表」(truth table) というのは、表の形で論理演算を書きあらわしたもののことです。

真理値表は、命題の真偽値の組み合わせと、その組み合わせに対する論理演算の結果をあらわす行から構成されます。たとえば、A と B の連言 $(A \wedge B)$ は、

A	B	$A \wedge B$
真	真	真
真	偽	偽
偽	真	偽
偽	偽	偽

という真理値表によって書きあらわされます（表の中の真偽値を、真は T、偽は F という略称で書くこともあります）。

Q 7.1.5 選言っていうのはどんな論理演算なんですか。

A 選言というのは、2 個の命題について、それらのうちのどちらか一方または両方が真という組み合わせに対しては真を対応させて、両方が偽という組み合わせに対しては偽を対応させるという操作のことです。

日本語の「または」や英語の or は、選言をあらわしています。A と B が命題だとするとき、「または」を使うことによってそれらの命題を組み合わせた「A または B」という命題は、A と B のうちのどちらか一方または両方が真のときに全体が真になります。

数学や論理学では、A と B の選言を、$\vee$ という記号を使って、

$$A \vee B$$

という式で記述します。

A と B の選言 $(A \vee B)$ は、

A	B	$A \vee B$
真	真	真
真	偽	真
偽	真	真
偽	偽	偽

という真理値表によって書きあらわされます。

Q 7.1.6 選言を実行する組み込み述語って、あるんですか。

A はい、あります。

(;)/2 というのが、選言を実行する組み込み述語です。この述語の名前になっている ; というアトムは、優先順位が 1100 で記述子が xfy の演算子です。

A と B がゴールだとするとき、

A; B

というゴールで (;)/2 を呼び出すと、(;)/2 は、まず A を実行します。そして、A が失敗したならば B を実行して、B も失敗したならば自分自身も失敗します。どちらかのゴールが成功した場合は自分自身も成功します。なお、A が成功した場合、B は実行しません。

(;)/2 を呼び出す質問をインタプリタに入力してみると、次のようになります。

```
?- 5 == 5; 6 == 6.
yes
?- 5 == 5; 6 == 7.
yes
?- 5 == 4; 6 == 6.
yes
?- 5 == 4; 6 == 7.
no
```

; は、, よりも優先順位が低い演算子ですから、; を使って組み合わせた命題と、さらに別の命題とを、, を使って組み合わせる場合は、丸括弧が必要になります。たとえば、

「A または B」かつ C

という命題をあらわす Prolog のゴールは、丸括弧を使って、

(A; B), C

と書かないといけません。

```
?- 5 == 5; 5 == 4, 5 == 6.
yes
?- (5 == 5; 5 == 4), 5 == 6.
no
```

(;)/2 は、Prolog でプログラムを書く上で絶対に必要になる述語というわけではありませんし、それを使うことはプログラムを読みにくくしますので、あまり推奨されていません。もしも、

```
a :- (b; c), d.
```

という規則を書きたい場合は、b; c という部分を述語として独立させて、

```
a :- e, d.

e :- b.
e :- c.
```

と書くほうがいいでしょう。

Q 7.1.7　否定っていうのはどんな論理演算なんですか。

A　否定というのは、1 個の命題について、それが真ならば偽を対応させて、偽ならば真を対応させるという操作のことです。

日本語の「ではない」や英語の not は、否定をあらわしています。A が命題だとするとき、それに「ではない」を付け加えた「A ではない」という命題は、A が真のときは偽になって、偽のときは真になります。

数学や論理学では、A の否定を、¬ という記号を使って、

$\neg A$

という式で記述します（∼ という記号が使われることもあります）。

A の否定 ($\neg A$) は、

A	$\neg A$
真	偽
偽	真

という真理値表によって書きあらわされます。

Q 7.1.8　否定を実行する組み込み述語って、あるんですか。

A はい、あります。

(\+)/1 というのが、否定を実行する組み込み述語です。この述語の名前になっている \+ というアトムは、優先順位が 900 で記述子が fy の演算子です。

A がゴールだとするとき、

```
\+ A
```

というゴールで (\+)/1 を呼び出すと、(\+)/1 は、A を実行して、それが成功したならば自分自身は失敗して、失敗したならば自分自身は成功します。

(\+)/1 を呼び出す質問をインタプリタに入力してみると、次のようになります。

```
?- \+ 5 == 5.
no
?- \+ 5 == 4.
yes
```

否定を定義するためには、何を根拠として命題が偽だという判断を下せばいいかということが問題となります。Prolog の (\+)/1 は、引数を実行して、それが失敗したならば成功するように定義されています。つまり、失敗することが偽だということの根拠として使われているわけです。このような、失敗を偽とみなすことによって定義された否定の述語は、「失敗による否定」(negation as failure) と呼ばれます。

Q 7.1.9 閉世界仮説って何ですか。

A 「閉世界仮説」(closed world assumption) というのは、プログラムの中に書かれていない命題は偽とみなすという前提のことです。

Prolog の (\+)/1 は失敗によって否定を定義しているわけですが、それが可能なのは、Prolog が閉世界仮説を前提としているからです。

閉世界仮説というものについて、具体例を使って少し考えてみましょう。「人間である」ということを意味する human/1 という述語が、次のように定義されているとします。

```
human(kumiko).
human(yoshiaki).
human(tomomi).
human(atsushi).
human(sayaka).
```

このとき、human(junko) というゴールを実行してみます。すると、

```
human(junko).
```

という事実がプログラムの中に書かれていませんので、ゴールの実行は失敗します。

次に、\+ human(junko) というゴールを実行してみます。すると、このゴールの実行は成功します。つまり、「純子は人間ではない」という命題が真だということが証明されてしまうわけです。

このことは少し奇妙な印象を与えるかもしれませんが、プログラムの中に書かれていない命題は偽とみなすという前提、つまり閉世界仮説を Prolog が採用している以上、当然の結果なのです。つまり、human/1 という述語は、人間が 5 人しかいない閉じた世界の中で定義されているわけです。

Q 7.1.10 論理型言語を設計するに当たって、閉世界仮説とは異なる前提を採用することって、可能なんですか。

A はい、可能です。

たとえば、論理型言語を設計するに当たって、プログラムの中に書かれていない命題の真偽はわからないという前提を採用してもかまいません。そのような前提は、「開世界仮説」(open world assumption) と呼ばれます。

ただし、開世界仮説を採用している論理型言語では、失敗によって否定を定義するということはできません。

7.2　選択

Q 7.2.1　選択って何ですか。

A　「選択」(selection) というのは、いくつかの動作のうちからひとつを選んで実行するという動作のことです。

「選択」という言葉が日常的に使われる場合、選択の対象はかならずしも動作とは限らないわけですが、プログラミングに関連する文脈では、選択の対象について何も言及されていなければ、この言葉は動作の選択を意味することになります。

Q 7.2.2　選択を実行する組み込み述語って、あるんですか。

A　はい、あります。

(->)/2 というのが、選択を実行する組み込み述語です。この述語の名前になっている -> というアトムは、優先順位が 1050 で記述子が xfy の演算子です。

A と B がゴールだとするとき、

A -> B

というゴールで (->)/2 を呼び出すと、(->)/2 は、まず A を実行して、それに成功した場合にのみ B を実行します。つまり、A が成功したかどうかということによって、B を実行するかどうかを選択するわけです。

(->)/2 を呼び出す質問をインタプリタに入力してみると、次のようになります。

```
?- 5 == 5 -> write(selection), nl.
selection
yes
?- 5 == 4 -> write(selection), nl.
no
```

(->)/2 を、

A -> B

というゴールで呼び出した場合、ゴール全体の実行が成功するのは、A が成功して、さらに B も成功した場合だけです。それ以外の場合、ゴール全体の実行は失敗します。

(->)/2 も、(;)/2 と同じように、Prolog でプログラムを書く上で絶対に必要な述語というわけではありません。(->)/2 を使うことによって、それを使わない場合よりもプログラムを簡潔にすることができるというメリットがあるのは事実ですが、この述語の使用は、あまり好まれない傾向にあります。

(->)/2 の使用が好まれない理由のひとつは、この述語が論理プログラミングとは異質な存在だからということです。つまり、その述語を使うことによって論理プログラミングとは異質な考え方が混入したプログラムができてしまうことを「美しくない」と感じる人が多い、ということです。

Q 7.2.3　二つの動作のうちのどちらか一方を選択して実行するゴールって、どう書けばいいんですか。

A　(->)/2 と (;)/2 を組み合わせて使うことによって、二つの動作のうちのどちらか一方を選択して実行するゴールを書くことができます。

A と B と C がゴールだとするとき、(->)/2 と (;)/2 を呼び出す、

A -> B; C

というゴールを実行すると、まず A が実行されて、それが成功したならば B が実行されて、失敗したならば C が実行されます。つまり、A が成功したかどうかということによって、B と C のどちらかが選択されて実行されるわけです。

(->)/2 と (;)/2 を呼び出す質問をインタプリタに入力してみると、次のようになります。

```
?- (5 == 5 -> write(equal); write('not equal')), nl.
equal
yes
```

```
?- (5 == 4 -> write(equal); write('not equal')), nl.
not equal
yes
```

7.3 カット

Q 7.3.1 カットって何ですか。

A 「カット」(cut) というのは、選択点を消去するということです。

「選択点」(choice point) というのは、Q 4.4.3 で説明したように、バックトラックによってそこへ引き返すことのできる、まだ試されていない節の列のことです。

選択点は、普通は、それを構成しているすべての節がバックトラックによって試された場合に消滅するわけですが、それらの節を試すことなく消去するということができて、そのことを、選択点を「カットする」と言います。

人間が迷路の中で出口を探している場合で言えば、カットというのは、その人がこれまでに歩いてきた道を消去するということに相当します。カットを実行すると、その時点までに歩いてきた道を引き返すということができなくなります。

Q 7.3.2 カットを実行する組み込み述語って、あるんですか。

A はい、あります。

`(!)/0` というのが、カットを実行する組み込み述語です。

A_1、A_2、$\cdots$、A_n、B がゴールだとするとき、

$$A_1,\ A_2,\ \cdots,\ A_n,\ !,\ B$$

というゴールを実行したとしましょう。そうすると、もしも、A_1、A_2、$\cdots$、A_n が実行される過程で選択点が生成されたとしても、それらの選択点は、`(!)/0` が呼び出された時点で消去されます。ですから、もしも B の実行が失敗したとすると、バックトラックするべき選択点が存在しませんので、ゴール全体が失敗することになります。なお、`(!)/0` は、常に成功する述語です。

具体的な例を使って試してみましょう。たとえば、

```
human :- human(A), write(A), nl, fail.

human(risako).
human(tsunehiko).
human(shiori).
```

というプログラムの中で定義されている `human/0` という述語を呼び出すと、

```
risako
tsunehiko
shiori
```

と出力されます。つまり、バックトラックによって `human/1` のすべての節が試されるわけです。しかし、`human/0` を定義している規則の中に、

```
human :- human(A), write(A), nl, !, fail.
```

というようにカットを実行するゴールを書き加えて、そののち `human/0` を呼び出すと、`human/1` を呼び出したときに生成された選択点がカットされますので、`risako` だけしか出力されません。

`(!)/0` が選択点をカットする範囲は、それを呼び出すゴールを含んでいる規則の本体だけではありません。`(!)/0` は、自分を呼び出したゴールを定義の中に含んでいる述語によって生成された選択点もカットします。

それについても、具体的な例を使って試してみましょう。たとえば、

```
human :- write(risako), nl, fail.
human :- write(tsunehiko), nl, fail.
human :- write(shiori), nl.
```

というプログラムの中で定義されている `human/0` という述語を呼び出すと、

```
risako
tsunehiko
```

```
shiori
```

と出力されます。つまり、バックトラックによって human/0 のすべての節が試されるわけです。しかし、human/0 を定義している規則の最初のひとつに、

```
human :- write(risako), nl, !, fail.
```

というようにカットを実行するゴールを書き加えて、そののち human/0 を呼び出すと、human/0 によって生成された選択点がカットされますので、risako だけしか出力されません。

Q 7.3.3　カットっていうのは、どんなときに必要になるんですか。

A　カットが必要になるのは、バックトラックして試すことが望ましくない選択点が生成されることを防ぎたい場合です。

例として、rank/2 という述語の定義について考えてみましょう。

rank/2 は、N が数値だとするとき、rank(N, R) というゴールで呼び出すと、N が 80 以上ならば a と R とを単一化して、N が 50 以上 80 未満ならば b と R とを単一化して、N が 20 以上 50 未満ならば c と R とを単一化して、20 未満ならば d と R とを単一化する述語です。

rank/2 の定義は、たとえば、

```
rank(N, a) :- N >= 80.
rank(N, b) :- N >= 50.
rank(N, c) :- N >= 20.
rank(_, d).
```

と書くことができます。そうすると、

```
?- rank(90, X).
X = a
yes
?- rank(60, X).
X = b
yes
?- rank(30, X).
X = c
yes
?- rank(10, X).
X = d
yes
```

というように、正しい結果が得られます。しかし、このように定義された rank/2 は、バックトラックして試すことが望ましくない選択点を生成します。つまり、

```
?- rank(90, X), write(X), nl, fail.
a
b
c
d
no
```

というように、rank/2 が生成した選択点へのバックトラックは、望ましくない動作を引き起こすということです。

このような場合は、選択点をカットする必要があります。カットを使って rank/2 の定義を改良すると、次のようになります。

```
rank(N, a) :- N >= 80, !.
rank(N, b) :- N >= 50, !.
rank(N, c) :- N >= 20, !.
rank(_, d).
```

Q 7.3.4　組み込み述語の (\+)/1 と同じ動作をする述語を自分で定義することは可能ですか。

A　はい、可能です。

それでは、mynot という名前で、組み込み述語の (\+)/1 と同じ動作をする述語を定義してみましょう。

mynot/1 は、引数をゴールとして実行して、それが成功したならば失敗して、失敗したならば成功すればいいわけです。ですから、次のように定義を書くことができます。

```
mynot(G) :- G, !, fail.
mynot(_).
```

このように定義すれば、mynot/1 は、

```
?- mynot(5 == 5).
no
?- mynot(5 == 4).
yes
```

というように、組み込み述語の (\+)/1 と同じ動作をします。

7.4 繰り返し

Q 7.4.1 繰り返しって何ですか。

A 「繰り返し」(repetition) というのは、同じ動作を何回も実行することです。

Q 7.4.2 繰り返しを実行する述語を定義したいときって、どうすればいいんですか。

A 繰り返しを実行する述語を定義したいときは、再帰またはバックトラックを使います。

Prolog では、繰り返しは二通りの方法で実現することができます。ひとつは再帰を使う方法で、もうひとつはバックトラックを使う方法です。

Q 7.4.3 指定された回数だけ同じ動作を繰り返すっていう述語を、バックトラックを使って定義したいときって、どうすればいいんですか。

A そのような述語は、指定された回数だけ成功する述語を使うことによって定義することができます。

指定された回数だけ成功する述語を定義する方法については次の Q 7.4.4 で説明することにして、ここでは、その述語を使って同じ動作を繰り返す述語を定義する方法について説明します。

times/1 という述語は、引数で指定された回数だけ成功する述語だ、と仮定します。たとえば、この述語を times(5) というゴールで呼び出したとしましょう。そうすると、この述語は、1 個の節から構成される選択点を生成して成功します。そののち、その選択点へバックトラックすると、ふたたび 1 個の節から構成される選択点を生成して成功します。2 回目、3 回目、4 回目のバックトラックも同様です。しかし、5 回目にバックトラックしたときは、選択点を構成する節が失敗してしまいます。このようにして、バックトラックではない最初の実行も含めると、5 回だけ成功することになります。

times/1 を使うことによって、指定された回数だけ同じ動作を繰り返す述語の定義を書くことができます。

それでは、次のような動作をする times/2 という述語の定義を、times/1 を使って書いてみましょう。

```
?- times(5, (write('I will be written five times.'), nl)).
I will be written five times.
I will be written five times.
I will be written five times.
I will be written five times.
I will be written five times.
yes
```

このように、times/2 は、N が 1 以上の整数で、G がゴールだとするとき、times(N, G) というゴールで呼び出すと、G の実行を N 回だけ繰り返す述語です。

指定された回数だけ同じ動作を繰り返す述語を定義したいときは、指定された回数だけ成功するゴールと fail/0 を呼び出すゴールとのあいだに、繰り返したい動作を書きます。

ですから、times/2 の定義は、times/1 を使うことによって次のように書くことができます。

```
times(N, G) :- times(N), G, fail.
times(_, _).
```

Q 7.4.4　指定された回数だけ成功する述語って、定義を書くことは可能なんですか。

A　はい、可能です。

N が 1 以上の整数だとするとき、`times(`N`)` というゴールの実行が N 回だけ成功する、`times/1` という述語は、次のように定義を書くことができます。

```
times(_).
times(N) :- N > 1, N1 is N - 1, times(N1).
```

Q 7.4.5　同じ動作を無限に繰り返す述語を定義することって、可能ですか。

A　はい、可能です。

無限に成功し続ける述語というものを使えば、同じ動作を無限に繰り返す述語を定義することができます。無限に成功し続ける述語を呼び出すゴールと `fail/0` を呼び出すゴールとのあいだに何らかの動作を書けば、その動作は無限に繰り返されることになるからです。

ちなみに、無限に成功し続ける述語は、組み込み述語として Prolog のインタプリタに組み込まれています。それは `repeat/0` という述語です。この述語は、

```
repeat.
repeat :- repeat.
```

と定義されていると考えることができます。

`repeat/0` を使うことによって、同じ動作を無限に繰り返す述語を定義することができます。たとえば、G がゴールだとするとき、`infinite(`G`)` というゴールで呼び出すと、G の実行を無限に繰り返す、`infinite/1` という述語の定義は、

```
infinite(G) :- repeat, G, fail.
```

と書くことができます。

7.5　例外

Q 7.5.1　例外って何ですか。

A　「例外」(exception) というのは、プログラムの中にある位置のうちで、インタプリタが実行しようとしているところ、というものをあらわすデータのことです。

例外は、ボールのようなものだと考えることができます。述語は、それがあたかもボールであるかのように、例外を投げたり捕獲したりすることができます。例外を投げると、インタプリタが実行しようとする位置は、その例外を捕獲した述語へ移動します。

投げられた例外は、述語が述語を呼び出すという系列を、逆の順序で飛び越えていきます。述語は、自分の頭の上を通過する例外を捕獲することができます。

P、Q、R、S のそれぞれを述語だとしましょう。そして、P が Q を呼び出して、Q が R を呼び出して、R が S を呼び出したとしましょう。そのとき、S が例外を投げたとすると、その例外は、どの述語も捕獲しないとすれば、R、Q、P という順序で、それぞれの述語の頭上を飛び越えていきます。ですから、R と Q がその例外を捕獲しないとすれば、P は、その例外を捕獲することができます。

Q 7.5.2　もしも、どの述語も例外を捕獲しなかったとすると、その例外はどうなるんですか。

A　述語によって捕獲されなかった例外は、インタプリタによって捕獲されます。

インタプリタは、自分が実行しているプログラムの中で投げられた例外が、そのプログラムの中の述語によって捕獲されなかった場合、その例外を自分が捕獲して、そのことを知らせるメッセージを出力します。その場合、そのプログラムの実行は、その時点で終了することになります。

Q 7.5.3　例外って、どんなときに投げるものなんですか。

A　例外が投げられるのは、何らかの不都合な事態が発生した場合です。

プログラムの実行中に何らかの不都合な事態が発生した場合、そのプログラムは、その事態に対処する処理を実行する必要があります。しかし、不都合な事態に対処する処理を実行するためには、プログラムの実行の自然な順序から離れて、その処理が書かれている位置へ一足飛びに飛

んでいく必要があります。そのような場合に、インタプリタが実行する位置を移動させるために例外というものを投げるわけです。

Q 7.5.4 組み込み述語が例外を投げることって、あるんですか。

A はい、あります。

たとえば、is/2という述語は、右側の引数が正しい式ではなかった場合や、ゼロによる除算が実行された場合などに例外を投げます。

is/2に例外を投げさせる質問をインタプリタに入力してみると、次のようになります（出力されるメッセージは、インタプリタの実装によって異なります）。

```
?- X is 5 + a.
ERROR: Arithmetic: ‘a/0’ is not a function
?- X is 5 / 0.
ERROR: //2: Arithmetic: evaluation error: ‘zero_divisor’
```

Q 7.5.5 例外を投げたいときって、どうすればいいんですか。

A throw/1という組み込み述語を呼び出すことによって、例外を投げることができます。

Tが項だとするとき、throw(T)というゴールでthrow/1を呼び出すと、throw/1は例外を投げます。

throw/1の引数は、例外に付加される項です。例外を捕獲した述語は、それに付加された項を取り扱うことができます。ですから、例外を投げる原因となったデータや、それがどうして不都合なのかという理由などを例外に付加するといいでしょう。

throw/1を呼び出す質問をインタプリタに入力してみると、次のようになります（出力されるメッセージは、インタプリタの実装によって異なります）。

```
?- write(namako), nl, throw(umiushi), write(hitode).
namako
ERROR: Unhandled exception: umiushi
```

次に、不都合な事態が発生した場合に例外を投げる述語を定義してみましょう。

Nが0またはプラスの整数だとするとき、factorial(N, F)というゴールで呼び出すと、Nの階乗とFとを単一化する、factorial/2という述語の定義は、

```
factorial(0, 1).
factorial(N, F) :-
    N >= 1, N1 is N - 1, factorial(N1, F1), F is N * F1.
```

と書くことができます。この述語を呼び出すことによって、

```
?- factorial(5, X).
X = 120
yes
```

というように、階乗を求めることができます。

factorial/2が上のように定義されている場合、それにマイナスの整数の階乗を求めさせようとすると、

```
?- factorial(-5, X).
no
```

というように、ただ単に失敗するだけです。しかし、階乗というのはマイナスの整数に対しては定義されていませんので、1個目の引数がマイナスだった場合は、失敗して終わるよりも例外を投げるほうがいいでしょう。

factorial/2の定義は、次のように改良することができます。

```
factorial(0, 1).
factorial(N, F) :-
    N >= 1, N1 is N - 1, factorial(N1, F1), F is N * F1.
factorial(N, _) :- throw(factorial(N, 'not defined')).
```

このように定義しておけば、factorial/2は、1個目の引数がマイナスの整数だった場合は、

```
?- factorial(-5, X).
```

```
ERROR: Unhandled exception: factorial(-5, 'not defined')
```

というように、例外を投げることになります（出力されるメッセージは、インタプリタの実装によって異なります）。

Q 7.5.6　例外を捕獲したいときって、どうすればいいんですか。

A　catch/3 という組み込み述語を呼び出すことによって、例外を捕獲することができます。

G と H がゴールで、T が項だとするとき、catch(G, T, H) というゴールで catch/3 を呼び出したとしましょう。そうすると、catch/3 は、まず G を実行します。

G の実行中に、どの述語も例外を投げなかった場合は、G の実行の終了とともに catch/3 の実行も終了します。G が成功したならば catch/3 も成功して、G が失敗したならば catch/3 も失敗します。

G の実行中に述語が例外を投げた場合、catch/3 は、その例外に付加された項と T とを単一化します。そして、その単一化に成功した場合は、その例外を捕獲して、H を実行します。

catch/3 を呼び出す質問をインタプリタに入力してみると、次のようになります（例外に付加される項などは、インタプリタの実装によって異なります）。

```
?- catch(X is 5*3, A, (write(A), nl)).
X = 15
A = _G363
yes
?- catch(X is 5/0, A, (write(A), nl)).
error(evaluation_error(zero_divisor), context(/ /2, _G458))
X = _G360
A = error(evaluation_error(zero_divisor), context(/ /2, _G458))
yes
```

Q 7.5.5 で、階乗を求める factorial/2 という述語の定義を書きました。その述語は、1 個目の引数がマイナスの数値だった場合、例外を投げます。そこで、その述語が例外を投げたならばその例外を捕獲する、factorial/1 という述語を定義してみましょう。たとえば、次のような定義を書くことができます。

```
factorial(N) :- catch(write_factorial(N), E, write_error(E)).

write_factorial(N) :-
    factorial(N, F), write(factorial(N, F)), nl.

write_error(E) :- write('error: '), write(E), nl.
```

factorial/1 を呼び出す質問をインタプリタに入力してみると、次のようになります。

```
?- factorial(5).
factorial(5, 120)
yes
?- factorial(-5).
error: factorial(-5, not defined)
yes
```

7.6　練習問題

7.1　T が項だとするとき、not_atom(T) というゴールで呼び出すと、T がアトムではないならば成功して、アトムならば失敗する、not_atom/1 という述語の定義を、組み込み述語の atom/1 と (\+)/1 を使って書いてください。

実行例
```
?- not_atom(3807).
yes
?- not_atom(a(b)).
yes
?- not_atom(adam).
no
```

7.2　A と B がゴールだとするとき、disjunction(A, B) というゴールで呼び出すと、組み込み述語の (;)/2 と同じ動作をする（つまり、A と B の選言が真ならば成功して、そうでなけ

れば失敗する)、disjunction/2 という述語の定義を書いてください。

実行例
```
?- disjunction(5 == 5, 6 == 6).
yes
?- disjunction(5 == 5, 6 == 7).
yes
?- disjunction(5 == 4, 6 == 6).
yes
?- disjunction(5 == 4, 6 == 7).
no
```

7.3 次の真理値表であらわされる論理演算 $\rightarrow$ は、「含意」(implication) と呼ばれます。

A	B	$A \rightarrow B$
真	真	真
真	偽	偽
偽	真	真
偽	偽	真

A と B がゴールだとするとき、implication(A, B) というゴールで呼び出すと、$A \rightarrow B$ が真ならば成功して、そうでなければ失敗する、implication/2 という述語の定義を書いてください。

実行例
```
?- implication(5 == 5, 6 == 6).
yes
?- implication(5 == 5, 6 == 7).
no
?- implication(5 == 4, 6 == 6).
yes
?- implication(5 == 4, 6 == 7).
yes
```

7.4 G と V がゴールだとするとき、truth_value(G, V) というゴールで呼び出すと、G が真ならば true というアトムと V とを単一化して、そうでなければ false というアトムと V とを単一化する、truth_value/2 という述語の定義を書いてください。

実行例
```
?- truth_value(5 == 5, X).
X = true
yes
?- truth_value(5 == 4, X).
X = false
yes
```

7.5 次の真理値表であらわされる論理演算 $\leftrightarrow$ は、「対等」(equivalence) と呼ばれます。

A	B	$A \leftrightarrow B$
真	真	真
真	偽	偽
偽	真	偽
偽	偽	真

A と B がゴールだとするとき、equivalence(A, B) というゴールで呼び出すと、$A \leftrightarrow B$ が真ならば成功して、そうでなければ失敗する、equivalencde/2 という述語の定義を、練習問題 4 で定義を書いた truth_value/2 を使って書いてください。

実行例
```
?- equivalence(5 == 5, 6 == 6).
yes
?- equivalence(5 == 5, 6 == 7).
no
?- equivalence(5 == 4, 6 == 6).
no
?- equivalence(5 == 4, 6 == 7).
yes
```

7.6　次の真理値表であらわされる論理演算 $\oplus$ は、「排他的選言」(exclusive disjunction) と呼ばれます。

A	B	$A \oplus B$
真	真	偽
真	偽	真
偽	真	真
偽	偽	偽

A と B がゴールだとするとき、exdisjunc(A, B) というゴールで呼び出すと、$A \oplus B$ が真ならば成功して、そうでなければ失敗する、exdisjunc/2 という述語の定義を、練習問題 4 で定義を書いた truth_value/2 を使って書いてください。

実行例
```
?- exdisjunc(5 == 5, 6 == 6).
no
?- exdisjunc(5 == 5, 6 == 7).
yes
?- exdisjunc(5 == 4, 6 == 6).
yes
?- exdisjunc(5 == 4, 6 == 7).
no
```

7.7　C、T、E がゴールだとするとき、if(C, T, E) というゴールで呼び出すと、まず C を実行して、C が成功した場合は T を実行して、C が失敗した場合は E を実行する、if/3 という述語の定義を、組み込み述語の (->)/2 を使わないで書いてください。

実行例
```
?- if(5 == 5, write(equal), write('not equal')), nl.
equal
yes
?- if(5 == 3, write(equal), write('not equal')), nl.
not equal
yes
```

7.8　Q 7.4.3 で定義を書いた times/2 を、1 個目の引数 N が 1 よりも小さい場合は、

```
times(N, 'less than one')
```

という項が付加された例外を投げるように改良してください。

実行例
```
?- times(3, (write('Good wine needs no bush.'), nl)).
Good wine needs no bush.
Good wine needs no bush.
Good wine needs no bush.
yes
?- times(-3, (write('They have no wine.'), nl)).
ERROR: Unhandled exception: times(-3, 'less than one')
```

7.9　N が 1 以上の整数だとするとき、asterisk(N) というゴールで呼び出すと、N 個のアスタリスク (*) を出力して、最後に改行を出力する、asterisk/1 という述語の定義を、練習問題 8 で改良した times/2 を使って書いてください。ただし、times/2 が例外を投げた場合は、それを捕獲して、それに付加されている項を出力するようにしてください。

実行例
```
?- asterisk(40).
****************************************
yes
?- asterisk(-40).
error: times(-40, less than one)
yes
```

第 8 章　入出力

8.1 ストリーム

Q 8.1.1 ストリームって何ですか。

A 「ストリーム」(stream) というのは、プログラムがそこからデータを読み込んだりそこへデータを書き込んだりすることのできる対象のことです。

ストリームというのは、ファイル（またはファイルと同様に扱うことのできる対象）を抽象化したもののことです。プログラムは、ファイル自体ではなく、それに対応するストリームに対して読み書きを実行します。

ファイルに対する読み書きを実行するプログラムは、読み書きに先立って、そのファイルに対応するストリームを生成する必要があります。ストリームを生成することを、ストリームを「オープンする」(open) と言います。

また、ストリームに対する読み書きが終了した場合は、そのストリームを消滅させる必要があります。ストリームを消滅させることを、ストリームを「クローズする」(close) と言います。

Q 8.1.2 ストリームの現在位置って何のことですか。

A ストリームの「現在位置」(current position) というのは、現在の時点で読み込みや書き込みを実行するときにその対象となる、ストリームの中の位置のことです。

ストリームに対するデータの読み込みや書き込みは、そのストリームの現在位置に対して実行されます。そして、その実行が終了すると、現在位置は、次に読み込みや書き込みを実行するべき位置へ移動します。

Q 8.1.3 ストリームをオープンしたいときって、どうすればいいんですか。

A ストリームをオープンしたいときは、`open/3` という組み込み述語を使います。

`open/3` を呼び出すゴールの中に書く引数は、1 個目がファイル名、2 個目が入出力モード、3 個目が変数です。

ここで「ファイル名」(file name) と呼んでいるのは、特定のファイルを指定するアトムのことです。ファイル名として具体的にどのようなものを書けばいいのかというのは処理系に依存しますが、普通は、ファイルを指定するパス名を書きます。

「入出力モード」(I/O mode) というのは、ファイルに対してどのような操作をするためにストリームをオープンするのかということをあらわすアトムのことです。入出力モードとしては、次のようなものがあります。

`read` ファイルの先頭からデータを読み込む。この場合、対象となるファイルはすでに存在していなければならない。

`write` ファイルへデータを書き込む。対象となるファイルがすでに存在している場合は、書き込みに先立ってその内容を空にする。対象となるファイルが存在していない場合は空のファイルを新しく作る。

`append` ファイルへデータを書き込む。対象となるファイルがすでに存在している場合は、書き込みを開始する位置として、ファイルの内容の末尾を設定する。対象となるファイルが存在していない場合は空のファイルを新しく作る。

`open/3` は、1 個目の引数で指定されたファイルに対応するストリームを、2 個目の引数で指定された入出力モードでオープンして、そのストリームと 3 個目の引数とを単一化します。たとえば、

```
open('/home/junko/data.txt', read, S)
```

というゴールで `open/3` を呼び出したとすると、`open/3` は、

```
/home/junko/data.txt
```

というファイルに対応するストリームを、`read` という入出力モードでオープンして、そのストリームをあらわす項と `S` という変数とを単一化します。

ストリームは、かならずオープンすることができるとは限りません。ストリームをオープンすることができなかった場合、`open/3` は例外を投げることになります。

Q 8.1.4　ストリームをクローズしたいときって、どうすればいいんですか。

A　ストリームをクローズしたいときは、close/1 という組み込み述語を使います。

close/1 は、引数で指定されたストリームをクローズします。たとえば、ストリームをあらわす項と S とが単一化されているとするとき、

```
close(S)
```

というゴールで close/1 を呼び出すと、close/1 は S をクローズします。

Q 8.1.5　ストリームのエイリアスって、何のことですか。

A　ストリームの「エイリアス」(alias) というのは、ストリームに名前として与えられたアトムのことです。

ストリームには、それをオープンするときに、その名前として 1 個のアトムを与えることができます。
ストリームに対して読み書きやクローズを実行するためには、基本的には、それをオープンしたときに得られた、それをあらわす項が必要になります。しかし、ストリームにエイリアスが与えられている場合、ストリームをあらわす項を使う代わりに、そのエイリアスを使うことによって、読み書きやクローズを実行することができます。
たとえば、あるストリームに対して、エイリアスとして wakana というアトムが与えられているとするならば、

```
close(wakana)
```

というゴールで close/1 を呼び出すことによって、そのストリームをクローズすることができます。

Q 8.1.6　ストリームにエイリアスを与えたいときって、どうすればいいんですか。

A　ストリームにエイリアスを与えたいときは、open/4 という組み込み述語を使います。

open/4 も、open/3 と同じように、ストリームをオープンする組み込み述語です。
open/4 を呼び出すゴールの中に書く引数のうち、最初の 3 個は、open/3 と同じです。つまり、1 個目はファイル名、2 個目は入出力モード、3 個目は変数です。そして、4 個目の引数は、「オープンオプションリスト」(open option list) と呼ばれるリストです。
オープンオプションリストというのは、ストリームをどのようにオープンするのかということに関する細かな指定をあらわす、「オープンオプション」(open option) と呼ばれる項から構成されるリストのことです。
ストリームにエイリアスを与えたいときは、オープンオプションとして、

```
alias( アトム )
```

という複合項を書きます。そうすると、この複合項の引数として書かれたアトムが、エイリアスとしてストリームに与えられます。たとえば、

```
open('/home/junko/data.txt', read, _, [alias(wakana)])
```

というゴールで open/4 を呼び出すことによって、オープンしたストリームに対して wakana というアトムをエイリアスとして与えることができます。

Q 8.1.7　ストリームの終わりって、何のことですか。

A　「ストリームの終わり」(end of stream) というのは、ストリームの内容の末尾、という位置のことです。

Q 8.1.8　読み込みの現在位置がストリームの終わりに到達しているかどうかを調べたいときって、どうすればいいんですか。

A　at_end_of_stream/1 という組み込み述語を使うことによって、読み込みの現在位置がストリームの終わりに到達しているかどうかを調べることができます。

at_end_of_stream/1 を呼び出すゴールには、引数として、読み込みの入出力モードでオープンされたストリームをあらわす項またはエイリアスを書きます。

at_end_of_stream/1 は、引数で指定されたストリームからの読み込みの現在位置がストリームの終わりに到達しているならば成功して、そうでなければ失敗します。たとえば、読み込みの入出力モードでオープンされているストリームに wakana というエイリアスが与えられているとするとき、

```
at_end_of_stream(wakana)
```

というゴールの実行は、 wakana からの読み込みの現在位置がストリームの終わりに到達しているならば成功して、そうでなければ失敗します。

Q 8.1.9 カレントストリームプロパティーって、何のことですか。

A 「カレントストリームプロパティー」(current stream property) というのは、ストリームが持っている属性のことです。

カレントストリームプロパティーとしては、ファイル名、入出力モード、エイリアス、現在位置などがあります。

Q 8.1.10 カレントストリームプロパティーを調べたいときって、どうすればいいんですか。

A カレントストリームプロパティーを調べたいときは、 stream_property/2 という組み込み述語を使います。

stream_property/2 を呼び出すゴールの中に書く 1 個目の引数は、ストリームをあらわす項です。そして 2 個目の引数は、1 個の変数を引数とする複合項です。この複合項の関数子は、調べたいカレントストリームプロパティーをあらわすアトムです。

stream_property/2 は、1 個目の引数で指定されたストリームについて、2 個目の引数の関数子で指定されたカレントストリームプロパティーを調べます。そして、その結果と、2 個目の引数の中の変数とを単一化します。

2 個目の引数としては、たとえば次のような複合項を書くことができます。

file_name(F)	F はファイル名（実装に依存する）。
mode(M)	M は入出力モードをあらわすアトム。
alias(A)	A はエイリアス。
position(P)	P はストリームの現在位置をあらわす項（実装に依存する）。
end_of_file(E)	E は、ストリームの現在位置がストリームの終わりならば at、そこを通り過ぎているならば past、まだストリームの終わりに到達していないならば not。

たとえば、 S がストリームをあらわす項だとするとき、

```
stream_property(S, mode(M))
```

というゴールを実行すると、その中の M という変数は、 read 、 write 、 append のいずれかと単一化されます。

8.2 読み込み

Q 8.2.1 ストリームから項を読み込みたいときって、どうすればいいんですか。

A ストリームから項を読み込みたいときは、 read/2 という組み込み述語を使います。

read/2 を呼び出すゴールの中には、1 個目の引数としてストリームをあらわす項またはストリームのエイリアス、2 個目の引数として変数を書きます。そうすると、 read/2 は、1 個目の引数で指定されたストリームの現在位置から 1 個の項を読み込んで、その項と 2 個目の引数とを単一化します。たとえば、 S がストリームをあらわす項だとするとき、

```
read(S, T)
```

というゴールを実行すると、 read/2 は、 S の現在位置から 1 個の項を読み込んで、その項と T とを単一化します。

なお、read/2 を使ってファイルから項を読み込むためには、その項の後ろに、1 個のドット (.) と 1 個以上のホワイトスペースが書かれている必要があります。

それでは、read/2 を使ってファイルから項を読み込む述語を定義してみましょう。

```
read_file(P) :-
    open(P, read, S), read(S, T), write(T), nl, close(S).
```

この述語定義によって定義される read_file/1 という述語は、引数で指定されたファイルから先頭の項を読み込んで、その項を出力します。たとえば、

```
/home/yukiko/terms.txt
```

というファイル中に、

```
namako.
3427.
yadokari(583, 43, 81).
[mimizu, namekuji, marumushi].
```

というように、いくつかの項が格納されているとします。このとき、そのファイルのパス名を指定して read_file/1 を呼び出すと、

```
?- read_file('/home/yukiko/terms.txt').
namako
yes
```

というように、そのファイルの先頭の項が出力されます。

Q 8.2.2　現在位置がストリームの終わりになっているときに read/2 を呼び出すと、どうなるんですか。

A　read/2 は、現在位置がストリームの終わりになっている場合、end_of_file というアトムと 2 個目の引数とを単一化します。

Q 8.2.3　ストリームからの読み込みをストリームの終わりに到達するまで繰り返したいときって、どうすればいいんですか。

A　組み込み述語の repeat/0 を使うことによって、ストリームからの読み込みをストリームの終わりに到達するまで繰り返すことができます。

repeat/0 は、Q 7.4.5 で説明したように、無限に成功し続ける組み込み述語です。G がゴールだとするとき、

```
repeat, G, fail
```

というゴールは、G の実行を無限に繰り返します。

repeat/0 と fail/0 を使うと、ゴールの実行が無限に繰り返されることになるわけですが、fail/0 ではなくて、繰り返しを終了する条件をあらわす述語を使うことによって、終了の条件が成り立っていないあいだだけ動作を繰り返す、ということができます。

read/2 によって読み込まれたものが end_of_file というアトムと等しいかどうかを判定するゴールは、ストリームからの読み込みがストリームの終わりに到達した、という条件をあらわします。

それでは、ストリームからの読み込みをストリームの終わりに到達するまで繰り返す述語を定義してみましょう。

```
read_file(P) :-
    open(P, read, S), repeat_read_write(S), close(S).

repeat_read_write(S) :-
    repeat, read(S, T), write_nl(T), T == end_of_file.

write_nl(end_of_file) :- !.
write_nl(T) :- write(T), nl.
```

このプログラムの中で定義されている read_file/1 という述語は、引数で指定されたファイルからすべての項を読み込んで、それらの項を出力します。たとえば、

```
/home/yukiko/terms.txt
```

というファイル中に、

```
namako.
3427.
yadokari(583, 43, 81).
[mimizu, namekuji, marumushi].
```

というように、いくつかの項が格納されているとするとき、そのファイルのパス名を指定して read_file/1 を呼び出すと、

```
?- read_file('/home/yukiko/terms.txt').
namako
3427
yadokari(583, 43, 81)
[mimizu, namekuji, marumushi]
yes
```

というように、そのファイルに格納されているすべての項が出力されます。
次のプログラムは、上のプログラムをエイリアスを使って書き換えたものです。

```
read_file(P) :-
    open(P, read, _, [alias(istream)]),
    repeat_read_write,
    close(istream).

repeat_read_write :-
    repeat, read(istream, T), write_nl(T), T == end_of_file.

write_nl(end_of_file) :- !.
write_nl(T) :- write(T), nl.
```

Q 8.2.4 ストリームから 1 個の文字を読み込みたいときって、どうすればいいんですか。

A ストリームから 1 個の文字を読み込みたいときは、get_char/2 という組み込み述語を使います。

get_char/2 を呼び出すゴールには、1 個目の引数としてストリームをあらわす項またはエイリアス、2 個目の引数として変数を書きます。そうすると、get_char/2 は、指定されたストリームの現在位置から 1 個の文字を読み込んで、その文字をあらわすアトムと 2 個目の引数とを単一化します。たとえば、S がストリームをあらわす項だとするとき、

```
get_char(S, C)
```

というゴールを実行すると、get_char/2 は、S の現在位置から 1 個の文字を読み込んで、その文字をあらわすアトムと C とを単一化します。

get_char/2 は、現在位置がストリームの終わりになっている場合、end_of_file というアトムと 2 個目の引数とを単一化します。
それでは、get_char/2 を使って、ファイルからすべての文字を読み込む述語を定義してみましょう。

```
read_file(P) :-
    open(P, read, S), repeat_get_write(S), close(S).

repeat_get_write(S) :-
    repeat, get_char(S, C), write_char(C), C == end_of_file.

write_char(end_of_file) :- !.
write_char(C) :- write(C).
```

このプログラムの中で定義されている read_file/1 という述語は、引数で指定されたファイルからすべての文字を読み込んで、それらの文字を出力します。たとえば、

```
/home/arika/friends.txt
```

というファイルの中に、

```
Natsume Sayuri
```

```
Kurahashi Masayo
Sakami Noriko
```

という文字列が格納されているとするとき、そのファイルのパス名を指定して read_file/1 を呼び出すと、

```
?- read_file('/home/arika/friends.txt').
Natsume Sayuri
Kurahashi Masayo
Sakami Noriko
yes
```

というように、そのファイルの内容が出力されます。

Q 8.2.5　読み込んだ文字を、アトムとしてじゃなくて、文字コードをあらわす整数定数として扱いたいんですが、そんなときはどうすればいいんですか。

A　get_code/2 という組み込み述語を使うことによって、文字コードをあらわす整数定数の形で文字をストリームから読み込むことができます。

get_code/2 を呼び出すゴールには、1 個目の引数としてストリームをあらわす項またはエイリアス、2 個目の引数として変数を書きます。そうすると、get_code/2 は、指定されたストリームから 1 個の文字を読み込んで、その文字の文字コードをあらわす整数定数と 2 個目の引数とを単一化します。

get_code/2 は、現在位置がストリームの終わりになっている場合、−1 という整数定数と 2 個目の引数とを単一化します。

それでは、get_code/2 を使ってファイルからすべての文字を読み込む述語を定義してみましょう。

```
read_file(P) :-
    open(P, read, S), repeat_get_write(S), close(S).

repeat_get_write(S) :-
    repeat, get_code(S, C), write_code(C), C == -1.

write_code(-1) :- nl, !.
write_code(C) :- write(C), write(' ').
```

このプログラムの中で定義されている read_file/1 という述語は、引数で指定されたファイルからすべての文字を読み込んで、それらの文字の文字コードを出力します。たとえば、

```
/home/arika/myname.txt
```

というファイルの中に、

```
Togawa Arika
```

という文字列が格納されているとするとき、そのファイルのパス名を指定して read_file/1 を呼び出すと、

```
?- read_file('/home/arika/myname.txt').
84 111 103 97 119 97 32 65 114 105 107 97
yes
```

というように、そのファイルの内容が文字コードの列として出力されます。

Q 8.2.6　ストリームの現在位置を変化させないでストリームから 1 個の文字を読み込みたいんですが、そんなときはどうすればいいんですか。

A　peek_char/2 または peek_code/2 という組み込み述語を使うことによって、ストリームの現在位置を変化させないでストリームから 1 個の文字を読み込むことができます。

peek_char/2 または peek_code/2 を呼び出すゴールには、1 個目の引数としてストリームをあらわす項またはエイリアス、2 個目の引数として変数を書きます。

peek_char/2 は、指定されたストリームの現在位置から 1 個の文字を読み込んで、その文字をあらわすアトムと 2 個目の引数とを単一化します。ストリームの現在位置は変化させません。現

在位置がストリームの終わりになっている場合は、end_of_file というアトムと 2 個目の引数とを単一化します。

peek_code/2 は、指定されたストリームの現在位置から 1 個の文字を読み込んで、その文字の文字コードをあらわす整数定数と 2 個目の引数とを単一化します。やはりストリームの現在位置は変化させません。現在位置がストリームの終わりになっている場合は、−1 という整数定数と 2 個目の引数とを単一化します。

8.3 書き込み

Q 8.3.1 ストリームに項を書き込みたいときって、どうすればいいんですか。

A ストリームに項を書き込みたいときは、write/2 という組み込み述語を使います。

write/2 を呼び出すゴールの中には、1 個目の引数としてストリームをあらわす項またはストリームのエイリアス、2 個目の引数として項を書きます。そうすると、write/2 は、1 個目の引数で指定されたストリームの現在位置に 2 個目の引数を書き込みます。たとえば、S がストリームをあらわす項だとするとき、

```
write(S, mizinko)
```

というゴールで write/2 を呼び出すと、write/2 は、mizinko という項を S の現在位置に書き込みます。

それでは、write/2 を使ってファイルに項を書き込む述語を定義してみましょう。

```
write_file(P, T) :- open(P, write, S), write(S, T), close(S).
```

この述語定義によって定義される write_file/2 という述語は、1 個目の引数で指定されたファイルに、2 個目の引数を書き込みます。たとえば、この述語を、

```
write_file('test.txt', yadokari)
```

というゴールで呼び出すと、カレントディレクトリにある test.txt というファイルに、yadokari という項が書き込まれます。

Q 8.3.2 write/2 を使ってストリームに項を書き込んで、read/2 を使ってそれを読み込むと、もとの項がそのまま得られるんですか。

A かならずしもそのまま得られるとは限りません。

write/2 を使って引用アトムをストリームに書き込んだ場合、実際に書き込まれるのは、その引用アトムがあらわしている文字列です。つまり、その文字列を囲んでいる一重引用符は書き込まれないということです。たとえば、

```
write(S, '83')
```

というゴールを実行したときにストリームに書き込まれるのは、83 という文字列です。

ですから、引用アトムに関しては、write/2 で書き込んで read/2 で読み込んだ場合、もとの引用アトムとは違ったものが得られることになります。

Q 8.3.3 引用アトムをそのままの形でストリームに書き込む組み込み述語って、あるんですか。

A はい、あります。

writeq/2 という組み込み述語は、write/2 と同じように、1 個目の引数で指定されたストリームに 2 個目の引数を書き込むという動作をするのですが、2 個目の引数が引用アトムだった場合、write/2 とは違って、その引用アトムをそのままの形で書き込みます。たとえば、

```
writeq(S, '83')
```

というゴールで writeq/2 を呼び出すと、writeq/2 は、'83' という引用アトムをストリームに書き込みます。

Q 8.3.4 ストリームに改行を書き込みたいときって、どうすればいいんですか。

A ストリームに改行を書き込みたいときは、nl/1 という組み込み述語を使います。

nl/1 を呼び出すゴールの中には、引数としてストリームをあらわす項またはストリームのエイリアスを書きます。そうすると、nl/1 は、引数で指定されたストリームの現在位置に改行を書き込みます。たとえば、S がストリームをあらわす項だとするとき、

```
nl(S)
```

というゴールで nl/1 を呼び出すと、nl/1 は、S の現在位置に改行を書き込みます。

Q 8.3.5　ストリームに文字を書き込みたいときって、どうすればいいんですか。

A　ストリームに文字を書き込みたいときは、put_char/2 という組み込み述語を使います。

ストリームに 1 個の文字を書き込みたいとき、write/2 を使うことによってそれを実行することももちろん可能です。しかし、その動作に要する時間は、put_char/2 を使うほうが write/2 を使うよりもおそらく短くなります。

put_char/2 を呼び出すゴールの中には、1 個目の引数としてストリームをあらわす項またはストリームのエイリアス、2 個目の引数として文字をあらわすアトムを書きます。そうすると、put_char/2 は、1 個目の引数で指定されたストリームの現在位置に、2 個目の引数があらわしている文字を書き込みます。たとえば、S がストリームをあらわす項だとするとき、

```
put_char(S, d)
```

というゴールを実行すると、put_char/2 は、d という文字を S の現在位置に書き込みます。

put_char/2 の 2 個目の引数は、かならずアトムでないといけません。ですから、英字の大文字や数字などを出力したいときは、

```
put_char(S, '3')
```

というように、引用アトムを書く必要があります。

それでは、put_char/2 を使ってファイルに文字を書き込む述語を定義してみましょう。

```
copy_file(PR, PW) :-
    open(PR, read, SR), open(PW, write, SW),
    repeat_read_write(SR, SW),
    close(SR), close(SW).

repeat_read_write(SR, SW) :-
    repeat,
    get_char(SR, C), write_char(SW, C),
    C == end_of_file.

write_char(_, end_of_file) :- !.
write_char(SW, C) :- put_char(SW, C).
```

このプログラムの中で定義されている copy_file/2 という述語は、1 個目の引数で指定されたファイルからすべての文字を読み込んで、2 個目の引数で指定されたファイルにそれらの文字を書き込みます。たとえば、この述語を、

```
copy_file('original.txt', 'copy.txt')
```

というゴールで呼び出すと、original.txt というファイルの内容が、copy.txt というファイルに書き込まれます。

Q 8.3.6　文字をあらわしているアトムじゃなくて、文字コードをあらわしている整数定数を使って、ストリームに文字を書き込みたいんですが、そんなときはどうすればいいんですか。

A　put_code/2 という組み込み述語を使うことによって、文字コードをあらわす整数定数を使ってストリームに文字を書き込むことができます。

put_code/2 を呼び出すゴールには、1 個目の引数としてストリームをあらわす項またはエイリアス、2 個目の引数として、文字コードをあらわしている整数定数を書きます。そうすると、put_code/2 は、1 個目の引数で指定されたストリームの現在位置に、2 個目の引数があらわしている文字コードの文字を書き込みます。

8.4 カレント入出力ストリーム

Q 8.4.1 標準入力って何ですか。

A 「標準入力」(standard input) というのは、どこから読み込むかということを、プログラムを起動するときに指定することのできる仮想的なファイルのことです。

標準入力からデータを読み込むように書かれているプログラムを、シェルを使って起動するときに、それを起動するコマンドの末尾に、

< パス名

というように小なりとパス名を書くと、そのパス名で指定されたファイルが標準入力になります。
デフォルトでは、標準入力はキーボードに割り当てられています。

Q 8.4.2 標準出力って何ですか。

A 「標準出力」(standard output) というのは、どこに書き込むかということを、プログラムを起動するときに指定することのできる仮想的なファイルのことです。

標準出力にデータを書き込むように書かれているプログラムを、シェルを使って起動するときに、それを起動するコマンドの末尾に、

> パス名

というように大なりとパス名を書くと、そのパス名で指定されたファイルが標準出力になります。
デフォルトでは、標準出力はモニターに割り当てられています。
標準入力と標準出力は、総称して「標準入出力」(standard I/O) と呼ばれます。

Q 8.4.3 標準入力ストリームって何ですか。

A 「標準入力ストリーム」(standard input stream) というのは、標準入力に対応しているストリームのことです。

Prolog のプログラムは、起動した時点で暗黙のうちに標準入力ストリームをオープンします。標準入力ストリームをクローズすることはできません。
標準入力ストリームには、`user_input` というエイリアスが与えられています。ですから、

```
read(user_input, X)
```

というゴールで `read/2` を呼び出すと、`read/2` は、標準入力ストリームから 1 個の項を読み込んで、その項と `X` とを単一化します。

Q 8.4.4 標準出力ストリームって何ですか。

A 「標準出力ストリーム」(standard output stream) というのは、標準出力に対応しているストリームのことです。

Prolog のプログラムは、起動した時点で暗黙のうちに標準出力ストリームをオープンします。標準出力ストリームをクローズすることはできません。
標準出力ストリームには、`user_output` というエイリアスが与えられています。ですから、

```
write(user_output, minomushi)
```

というゴールで `write/2` を呼び出すと、`write/2` は、`minomushi` という項を標準出力ストリームに書き込みます。
標準入力ストリームと標準出力ストリームは、総称して「標準入出力ストリーム」(standard I/O stream) と呼ばれます。

Q 8.4.5 カレント入力ストリームって何ですか。

A 「カレント入力ストリーム」(current input stream) というのは、ストリームを指定しないで読み込みを実行したときに、読み込みの対象となるストリームのことです。

Prolog のプログラムの実行が開始された時点では、標準入力ストリームがカレント入力ストリームに設定されています。

カレント入力ストリームから項を読み込みたいときは、read/1 という組み込み述語を使います。この述語は、カレント入力ストリームから 1 個の項を読み込んで、その項と引数とを単一化します。ですから、

```
read(X)
```

というゴールで read/1 を呼び出すと、read/1 は、カレント入力ストリームから 1 個の項を読み込んで、その項と X とを単一化します。

get_char/1、get_code/1、peek_char/1、peek_code/1 という組み込み述語は、カレント入力ストリームから 1 個の文字を読み込んで、その文字をあらわすアトムまたは整数と引数とを単一化します。

Q 8.4.6　カレント出力ストリームって何ですか。

A　「カレント出力ストリーム」(current output stream) というのは、ストリームを指定しないで書き込みを実行したときに、書き込みの対象となるストリームのことです。

Prolog のプログラムの実行が開始された時点では、標準出力ストリームがカレント出力ストリームに設定されています。

Q 4.2.7 で、write/1 という組み込み述語を紹介したとき、これは引数をモニターに出力する組み込み述語だと説明しましたが、その説明は厳密には正しいものではありません。write/1 は引数をカレント出力ストリームに書き込む組み込み述語だというのが、正しい説明です。

write/1 と同じように、nl/0 は、カレント出力ストリームに改行を出力する組み込み述語です。

put_char/1 と put_code/1 という組み込み述語は、カレント出力ストリームに 1 個の文字を書き込みます。

writeq/1 という組み込み述語は、引用アトムをそのままの形で書き込むという点を除いて、write/1 とほとんど同じ動作をする組み込む述語です。

カレント入力ストリームとカレント出力ストリームは、総称して「カレント入出力ストリーム」(current I/O stream) と呼ばれます。

Q 8.4.7　ストリームをカレント入力ストリームに設定したいときって、どうすればいいんですか。

A　set_input/1 という組み込み述語を使うことによって、ストリームをカレント入力ストリームに設定することができます。

set_input/1 を呼び出すゴールには、引数としてストリームをあらわす項またはストリームのエイリアスを書きます。そうすると、その引数があらわしているストリームが、カレント入力ストリームに設定されます。ですから、S がストリームをあらわす項だとするとき、

```
set_input(S), read(X)
```

というゴールを実行すると、read/1 は S から項を読み込みます。

Q 8.4.8　ストリームをカレント出力ストリームに設定したいときって、どうすればいいんですか。

A　set_output/1 という組み込み述語を使うことによって、ストリームをカレント出力ストリームに設定することができます。

set_output/1 を呼び出すゴールには、引数としてストリームをあらわす項またはストリームのエイリアスを書きます。そうすると、その引数があらわしているストリームが、カレント出力ストリームに設定されます。ですから、S がストリームをあらわす項だとするとき、

```
set_output(S), write(kurage)
```

というゴールを実行すると、write/1 は kurage という項を S に書き込みます。

Q 8.4.9　カレント入出力ストリームをクローズすると、どうなるんですか。

A カレント入力ストリームをクローズすると、標準入出力ストリームがカレント入出力ストリームに設定されます。

8.5 項の入出力

Q 8.5.1 read/1 や read/2 は、変数を含む項を読み込むこともできるんですか。

A はい、できます。

read/1 や read/2 は、変数を含む項を読み込むこともできます。

ただし、read/1 や read/2 は、読み込んだ変数を、処理系の内部でそれを扱うために使われる別の形に変換します（それがどのような形なのかというのは処理系に依存します）。

たとえば、変数を含む項を read/1 を使って読み込むと、その項は次のように変換されます（処理系の内部での変数の形は処理系ごとに異なります）。

```
?- read(X).
f(A, B, A).
X = f(_0001, _0002, _0001)
yes
```

Q 8.5.2 変数を含む項を読み込んだときに、その変数の元の形を知りたいときって、どうすればいいんですか。

A read_term/2 または read_term/3 という組み込み述語を使うことによって、読み込んだ変数の元の形を知ることができます。

read_term/2 は、カレント入力ストリームから項を読み込む組み込み述語です。それを呼び出すゴールの中には、1 個目の引数として変数を書いて、2 個目の引数として、「読み込みオプションリスト」(read option list) と呼ばれるリストを書きます。

読み込みオプションリストというのは、項をどのように読み込むのかということに関する細かな指定をあらわす、「読み込みオプション」(read option) と呼ばれる項から構成されるリストのことです。

読み込んだ変数の元の形を知りたいときは、読み込みオプションとして、

variable_names(変数)

という複合項を書きます。そうすると、その中の変数は、

アトム = 変数

という形の複合項から構成されるリストと単一化されます。その複合項は、イコールの左側が変数の元の形をあらわす引用アトムで、イコールの右側が処理系の内部での変数です。

たとえば、次のように、read_term/2 を使うことによって、変数の元の形を知ることができます（処理系の内部での変数の形は処理系ごとに異なります）。

```
?- read_term(X, [variable_names(N)]).
f(A, B, A).
X = f(_0001, _0002, _0001)
N = ['A'=_0001, 'B'=_0002]
yes
```

read_term/3 は、引数で指定されたストリームから項を読み込む組み込み述語です。それを呼び出すゴールの中には、引数として、1 個目にストリームをあらわす項またはストリームのエイリアス、2 個目に変数、そして 3 個目に読み込みオプションリストを書きます。

なお、読み込みオプションには、variable_names のほかに次のようなものがあります。

variables(V)	読み込んだ項に含まれている変数を、最初に出現する順番で並べることによってできるリストと V とを単一化する（変数の形は処理系の内部でのもの）。
singletons(S)	読み込んだ項に含まれている変数のうちで、匿名変数ではなくて、かつ、項の中に 1 回だけ出現するものについて、その元の形をあらわす引用アトム (A) と処理系内部での形 (V) を、A = V という形の項にして並べることによってできるリストと S とを単一化する。

これらの読み込みオプションを使って項を読み込むと、たとえば次のような結果が得られます。

```
?- read_term(X,
    [variables(V), variable_names(N), singletons(S)]).
f(A, B, C, D, B, C).
X = f(_0001, _0002, _0003, _0004, _0002, _0003)
V = [_0001, _0002, _0003, _0004]
N = ['A'=_0001, 'B'=_0002, 'C'=_0003, 'D'=_0004]
S = ['A'=_0001, 'D'=_0004]
yes
```

Q 8.5.3 書き込みオプションリストって何ですか。

A 「書き込みオプションリスト」(write option list) というのは、項をどのように書き込むのかということに関する細かな指定をあらわす、「書き込みオプション」(write option) と呼ばれる項から構成されるリストのことです。

書き込みオプションには、次のようなものがあります。

`quoted(`B`)` B が `true` の場合は、`read_term/3` で読み込むことができるように、引用アトムをそのままの形で出力する。B が `false` の場合は、引用アトムから一重引用符を取り除いたものを出力する。デフォルトは `false`。

`ignore_ops(`B`)` B が `true` の場合は、演算子記法で書かれた複合項を普通の形に変換して出力する（たとえば、`a+b` は、`+(a, b)` という形で出力される）。B が `false` の場合は、演算子記法をそのままの形で出力する。デフォルトは `false`。

`numbervars(`B`)` B が `true` の場合は、`'$VAR'(`N`)` という形の複合項（N は整数）を変数に変換して出力する（N が 0 ならば `A`、1 ならば `B`、2 ならば `C`、26 ならば `A1`、27 ならば `B1`）。B が `false` の場合は、そのような変換はしない。デフォルトは `false`。

Q 8.5.4 書き込みオプションを指定して項を書き込みたいときって、どうすればいいんですか。

A `write_term/2` または `write_term/3` という組み込み述語を使うことによって、書き込みオプションを指定して項を書き込むことができます。

`write_term/2` は、カレント出力ストリームに項を書き込む組み込み述語です。それを呼び出すゴールの中には、1 個目の引数として項を書いて、2 個目の引数として書き込みオプションリストを書きます。

たとえば、次のように、`write_term/2` を使うことによって、演算子記法で書かれた複合項を普通の形に変換して出力することができます。

```
?- write_term(a+b, [ignore_ops(true)]).
+(a, b)
yes
```

`write_term/3` は、引数で指定されたストリームに項を書き込む組み込み述語です。それを呼び出すゴールの中には、引数として、1 個目にストリームをあらわす項またはストリームのエイリアス、2 個目に項、そして 3 個目に書き込みオプションリストを書きます。

8.6 演算子の定義

Q 8.6.1 アトムを演算子にしたいときって、どうすればいいんですか。

A アトムを演算子にしたいときは、`op/3` という組み込み述語を使います。

`op/3` を呼び出すゴールの中には、1 個目の引数として優先順位（Q 3.2.1 参照）、2 個目の引数として記述子（Q 3.2.3 参照）、3 個目の引数として演算子にしたいアトムを書きます。たとえば、

```
op(200, xfx, loves)
```

というゴールで `op/3` を呼び出すことによって、`loves` というアトムを、優先順位が 200 で記述子が `xfx` の演算子にすることができます。ですから、それ以降、

```
loves(hisao, mikako)
```

という複合項は、

```
hisao loves mikako
```

と書くこともできるようになります。
なお、演算子の優先順位を 1200 よりも低くすることはできません。ですから、

```
op(1201, yfx, +++)
```

というゴールで op/3 を呼び出すと、 op/3 は例外を投げることになります。

Q 8.6.2 演算子になっているアトムを普通のアトムに戻したいときって、どうすればいいんですか。

A op/3 を使って、アトムの優先順位をゼロにすると、そのアトムは演算子ではなくなります。
たとえば、 loves というアトムが xfx という記述子で定義された演算子だとするとき、

```
op(0, xfx, loves)
```

というゴールで op/3 を呼び出すと、 op/3 は loves を、演算子ではない普通のアトムに戻します。

Q 8.6.3 命令って何ですか。

A 「命令」(directive) というのは、処理系がプログラムを読み込む時点で実行する記述、または、その記述が呼び出すことのできる組み込み述語のことです。

Prolog のプログラムは、基本的には節（事実または規則）から構成されるわけですが、それだけではなくて、「命令」と呼ばれるものをプログラムの中に書くことも可能です。
命令というのは、

:- ゴール . 1 個以上のホワイトスペース

という構文を持つ記述のことです。たとえば、

```
:- op(500, yfx, +++).
```

というのは命令の一例です。
命令は、それを含むプログラムが処理系によって読み込まれる時点で実行されます。ですから、アトムを演算子として定義するゴールをプログラムの中に書いておくことによって、同じプログラムの中で、そのアトムを演算子として使うことができます。
たとえば、次のプログラムは、 is_member_of というアトムを記述子が xfx の演算子として定義したのち、項がリストの要素かどうかを調べる is_member_of/2 という述語を定義しています。

```
:- op(700, xfx, is_member_of).

E is_member_of [E|_].
E is_member_of [_|T] :- E is_member_of T.
```

命令は、任意の述語を呼び出すことができるわけではありません。命令によって呼び出すことができるのは、いくつかの特定の組み込み述語だけです。命令が呼び出すことのできる組み込み述語のことも「命令」と呼ばれます。 op/3 は命令のひとつです。

8.7 練習問題

8.1 A と B がパス名で、A には何個かの式が格納されているとするとき、 eval_file(A, B) というゴールで呼び出すと、A に格納されているすべての式について、それを評価することによって得られた値、ドット (.)、そして改行を B に書き込む、 eval_file/2 という述語の定義を書いてください。
eval_file/2 は、たとえば、 express.txt というファイルに、

```
38+27.
400*200.
100-120/4.
(500+300)*2.
```

という 4 個の式が格納されているとき、

```
eval_file('express.txt', 'value.txt')
```

というゴールで呼び出したとすると、それらの式の値を、

```
65.
80000.
70.
1600.
```

というように value.txt に書き込みます。

8.2　P がパス名だとするとき、file_list(P, L) というゴールで呼び出すと、P に格納されているすべての項から構成されるリストと L とを単一化する、file_list/2 という述語の定義を書いてください。

file_list/2 は、たとえば、terms.txt というファイルに、

```
hitode.
8011.
uni(203, 55, 707).
[a, b, c, d, e].
```

という 4 個の項が格納されているとき、

```
file_list('terms.txt', X)
```

というゴールで呼び出したとすると、

```
[hitode, 8011, uni(203, 55, 707), [a, b, c, d, e]]
```

というリストと X とを単一化します。

8.3　A と B がパス名だとするとき、space_to_dot(A, B) というゴールで呼び出すと、A に格納されているすべての文字について、それが空白ならばドットを B に書き込んで、空白以外の文字ならばそれをそのまま B に書き込む、space_to_dot/2 という述語の定義を書いてください。

space_to_dot/2 は、たとえば、space.txt というファイルに、

```
diddle  plokta     grind
```

という文字列が格納されているとき、

```
space_to_dot('space.txt', 'dot.txt')
```

というゴールで呼び出したとすると、

```
diddle..plokta.....grind
```

という文字列を dot.txt に書き込みます。

8.4　A と B がパス名だとするとき、quote_file(A, B) というゴールで呼び出すと、A に格納されているすべての行について、その先頭に 1 個の大なり (>) と 1 個の空白を追加したものを B に書き込む、quote_file/2 という述語の定義を書いてください。

quote_file/2 は、たとえば、jargon.txt というファイルに、

```
bells and whistles
Infinite-Monkey Theorem
twirling baton
drunk mouse syndrome
```

という文字列が格納されているとき、

```
quote_file('jargon.txt', 'quote.txt')
```

というゴールで呼び出したとすると、

```
> bells and whistles
> Infinite-Monkey Theorem
> twirling baton
> drunk mouse syndrome
```

という文字列を quote.txt に書き込みます。

8.5　A と B がパス名だとするとき、line_number(A, B) というゴールで呼び出すと、A に格納されているすべての行について、その行の先頭に、その行の番号を追加したものを B に書き込む、line_number/2 という述語の定義を書いてください。

line_number/2 は、たとえば、jargon.txt というファイルに、

```
Knights of the Lambda Calculus
quadruple bucky
sorcerer's apprentice mode
Brooks's Law
```

という文字列が格納されているとき、

```
line_number('jargon.txt', 'linenum.txt')
```

というゴールで呼び出したとすると、

```
1 Knights of the Lambda Calculus
2 quadruple bucky
3 sorcerer's apprentice mode
4 Brooks's Law
```

という文字列を linenum.txt に書き込みます。

8.6　eval_shell というゴールで呼び出すと、eval> というプロンプトを出力して、1 個の式を標準入力から読み込んで、その式を評価して、その値を標準出力に出力する、ということを繰り返す、eval_shell/0 という述語の定義を書いてください。

なお、eval_shell/0 は、exit というアトムを入力することによって終了させることができるようにしてください。

実行例

```
?- eval_shell.
eval> 7+8.
15
eval> 200*30.
6000
eval> 5+7*4.
33
eval> 400/(7-3).
100
eval> exit.
yes
?-
```

8.7　interpreter というゴールで呼び出すと、execute> というプロンプトを出力して、1 個のゴールを標準入力から読み込んで、そのゴールを実行して、それが成功したかどうかを yes または no で出力する、ということを繰り返す、interpreter/0 という述語の定義を書いてください。

なお、interpreter は、halt というアトムを入力することによって終了させることができるようにしてください。また、ゴールの中に変数が含まれていた場合は、ゴールを実行したのち、変数の値も出力するようにしてください。

実行例

```
?- interpreter.
execute> write(neophilia), nl.
neophilia
yes
execute> 100 > 200.
no
execute> X is 7*111.
X = 777
yes
execute> halt.
yes
?-
```

付録 A　練習問題の解答例

第 2 章の解答例

2.1
(a) 正しい。
(b) 正しくない。先頭の文字として英字の大文字が使われているから。
(c) 正しい。
(d) 正しくない。先頭の文字として英字の大文字が使われているから。
(e) 正しくない。先頭の文字としてアンダースコア (_) が使われているから。
(f) 正しくない。英字の中にスラッシュ(/) が混ざっているから。
(g) 正しい。
(h) 正しくない。空白を含んでいるから。
(i) 正しい。
(j) 正しい。
(k) 正しくない。先頭の文字として数字が使われているから。
(l) 正しい。
(m) 正しくない。非英数字の中に数字が混ざっているから。
(n) 正しい。
(o) 正しい。
(p) 正しい。
(q) 正しくない。一重引用符で囲まれた文字列の中に一重引用符が単独で書かれているから。
(r) 正しい。
(s) 正しい。

2.2
(a) 正しい。
(b) 正しい。
(c) 正しくない。マイナスが数字列の右側に書かれているから。
(d) 正しい。
(e) 正しくない。 `0s` という接頭辞は存在しないから。
(f) 正しい。
(g) 正しい。
(h) 正しい。
(i) 正しくない。2 個のドット (.) が含まれているから。
(j) 正しくない。ドットの左側に数字がないから。
(k) 正しくない。ドットの右側に数字がないから。
(l) 正しい。
(m) 正しい。
(n) 正しい。
(o) 正しい。
(p) 正しくない。 `e` の右側に、ドットを含む数字列が書かれているから。

第 3 章の解答例

3.1
(a) 正しい。
(b) 正しくない。丸括弧ではなくて角括弧が使われているから。
(c) 正しい。
(d) 正しくない。関数子がアトムではなくて数値定数だから。
(e) 正しくない。引数の個数が 0 だから。
(f) 正しい。
(g) 正しい。

(h) 正しい。
(i) 正しくない。対応する右括弧が存在しない左括弧があるから。
(j) 正しくない。関数子がアトムではなくて複合項だから。
(k) 正しい。
(l) 正しくない。関数子と左丸括弧とのあいだに空白があるから。

3.2 (a) `+(a, b)`
(b) `+(a, *(b, c))`
(c) `-(/(a, b), c)`
(d) `*(/(a, b), c)`
(e) `+(-(a, b), c)`
(f) 文法的なエラーになる。
(g) `-(a)`
(h) `+(-(a), b)`
(i) `*(+(a, b), c)`
(j) `/(a, -(b, c))`
(k) `-(a, +(b, c))`
(l) `-(+(a, b))`

3.3 (a) 正しくない。先頭の文字として英字の小文字が使われているから。
(b) 正しい。
(c) 正しくない。先頭の文字として英字の小文字が使われているから。
(d) 正しい。
(e) 正しい。
(f) 正しくない。スラッシュ(`/`)を含んでいるから。
(g) 正しい。
(h) 正しくない。空白を含んでいるから。
(i) 正しい。
(j) 正しい。
(k) 正しくない。先頭の文字として数字が使われているから。
(l) 正しくない。文字列を一重引用符で囲んだものは変数ではなくてアトムだから。

3.4 (a) `kanpaku(hideyoshi).`
(b) `kazan(fujisan).`
(c) `aisuru(hironori, mihoko).`
(d) `shumi(tomomi, amimono).`
(e) `shuto(australia, canberra).`
(f) `kariru(haruhiko, kanako, note).`
(g) `yobu(kyouko, tomohiro, tomotan).`

3.5 (a) `shiawase(takako) :- genki(masao).`
(b) `shiawase(hiromi) :- aisuru(kazuyuki, hiromi).`
(c) `shiawase(mariko) :- daifugou(shigeo), aisuru(shigeo, mariko).`
(d) `tsurai(X) :- otoko(X).`
(e) `koori(X) :- kotai(X), mizu(X).`
(f) `himago(D, A) :- kodomo(B, A), kodomo(C, B), kodomo(D, C).`
(g) `shiawase(X) :- shumi(X, _).`
(h) `shiawase(X) :- aisuru(X, _), aisuru(_, X).`

第 4 章の解答例

4.1 (a) 成功する。
(b) 失敗する。

(c) 成功する。
(d) 失敗する。
(e) 失敗する。
(f) 成功する。X = kamome
(g) 成功する。X = a(b,c,d)
(h) 成功する。X = tsubame, Y = tsubame, Z = tsubame
(i) 失敗する。
(j) 成功する。X = karasu
(k) 成功する。X = b(c,d,e)
(l) 失敗する。
(m) 成功する。X = b, Y = c, Z = d
(n) 失敗する。
(o) 成功する。X = b, Y = c
(p) 失敗する。
(q) 成功する。X = b, Y = c
(r) 成功する。

4.2
```
masao
yoshihiko
```

4.3
```
mago(hiroko, haruka)
mago(nobuo, haruka)
mago(midori, takayuki)
mago(kunihiko, takayuki)
mago(naomi, hiroko)
```

4.4
```
couple(shigeo, kumiko)
couple(yoshihiko, manami)
couple(kumiko, shigeo)
couple(manami, yoshihiko)
```

4.5
```
like :- write('I like the world.\n').
```

4.6
```
like(A) :- write('I like '), write(A), write('.\n').
```

4.7
```
like(A, B) :-
    write('I like '), write(A),
    write(' rather than '), write(B), write('.\n').
```

4.8
```
type(A, atom) :- atom(A).
type(A, number) :- number(A).
type(A, compound) :- compound(A).
```

4.9
```
twice(A, T) :- atom_concat(A, A, T).
```

4.10
```
hmtom(H, M1, M2) :- M2 is H * 60 + M1.
```

4.11
```
mtohm(M1, H, M2) :- H is M1 // 60, M2 is M1 mod 60.
```

4.12
```
sumint(N, M) :- M is N * (N + 1) / 2.
```

4.13
```
measure(N, M) :- M mod N =:= 0.
```

第 5 章の解答例

5.1
```
multi_atom(_, 0, '').
multi_atom(A, N, M) :-
    N1 is N - 1, multi_atom(A, N1, M1), atom_concat(A, M1, M).
```

5.2
```
number_to_binary(0, '0').
number_to_binary(1, '1').
number_to_binary(N, B) :-
    N >= 2,
    L is N // 2, number_to_binary(L, BL),
    R is N mod 2, number_to_binary(R, BR),
    atom_concat(BL, BR, B).
```

5.3
```
even(0).
even(s(s(N))) :- even(N).
```

5.4
```
odd(s(0)).
odd(s(s(N))) :- odd(N).
```

5.5
```
factorial(0, s(0)).
factorial(s(N), F) :- factorial(N, F1), multiply(F1, s(N), F).
```

5.6
```
fibonacci(0, s(0)).
fibonacci(s(0), s(0)).
fibonacci(s(s(N)), F) :-
    fibonacci(s(N), F1), fibonacci(N, F2), add(F1, F2, F).
```

5.7
```
greater(s(N), 0) :- natural(N).
greater(s(N), s(M)) :- greater(N, M).
```

5.8
```
divide(N, M, 0) :- greater(M, N).
divide(N, M, s(D)) :- add(M, N1, N), divide(N1, M, D).
```

5.9
```
mod(N, M, N) :- greater(M, N).
mod(N, M, R) :- add(M, N1, N), mod(N1, M, R).
```

5.10
```
ston(0, 0).
ston(s(S), N) :- ston(S, N1), N is N1 + 1.
```

5.11
```
ntos(0, 0).
ntos(N, s(S)) :- N > 0, N1 is N - 1, ntos(N1, S).
```

5.12
```
gcm(N, 0, N) :- natural(N).
gcm(N, M, G) :- mod(N, M, R), gcm(M, R, G).
```

5.13
```
subtree(T, T).
subtree(S, t(_, T, _)) :- subtree(S, T).
subtree(S, t(_, _, T)) :- subtree(S, T).
```

5.14
```
isotree(v, v).
isotree(t(N, LA, RA), t(N, LB, RB)) :-
    isotree(LA, LB), isotree(RA, RB).
isotree(t(N, LA, RA), t(N, LB, RB)) :-
    isotree(LA, RB), isotree(RA, LB).
```

5.15
```
sum_tree(v, 0).
sum_tree(t(N, L, R), S) :-
    sum_tree(L, SL), sum_tree(R, SR), S is N + SL + SR.
```

5.16
```
hanoi(N) :- hanoi(N, a, b, c).

hanoi(0, _, _, _).
hanoi(N, S, W, D) :-
   N >= 1, N1 is N - 1,
   hanoi(N1, S, D, W),
   write(S), write(' -> '), write(D), nl,
   hanoi(N1, W, S, D).
```

第 6 章の解答例

6.1
```
generate_list(_, 0, []).
generate_list(E, N, [E|T]) :-
    N >= 1, N1 is N - 1, generate_list(E, N1, T).
```

6.2
```
member_list(E, [E|_]).
member_list(E, [_|T]) :- member_list(E, T).
```

6.3
```
delete_elem([], _, []).
delete_elem([E|T], E, DT) :- delete_elem(T, E, DT).
delete_elem([H|T], E, [H|DT]) :- delete_elem(T, E, DT).
```

6.4
```
unique([], []).
unique([H|T], UT) :- member_list(H, T), unique(T, UT).
unique([H|T], [H|UT]) :- unique(T, UT).

member_list(E, [E|_]).
member_list(E, [_|T]) :- member_list(E, T).
```

6.5
```
concat_list([], '').
concat_list([H|T], C) :-
    concat_list(T, CT), atom_concat(H, CT, C).
```

6.6
```
sum_list([], 0).
sum_list([H|T], S) :- sum_list(T, ST), S is H + ST.
```

6.7
```
max_list([E], E) :- number(E).
max_list([H|T], M) :- max_list(T, MT), max(H, MT, M).

max(A, B, A) :- A >= B.
max(_, B, B).
```

6.8
```
n_to_one(1, [1]).
n_to_one(N, [N|LT]) :- N >= 2, N1 is N - 1, n_to_one(N1, LT).
```

6.9
```
one_to_n(N, L) :- N >= 1, one_to_n(N, [], L).

one_to_n(0, A, A).
one_to_n(N, A, L) :- N1 is N - 1, one_to_n(N1, [N|A], L).
```

6.10
```
count_atomic([], 0).
count_atomic(A, 1) :- atomic(A).
count_atomic([H|T], N) :-
    count_atomic(H, NH), count_atomic(T, NT), N is NH + NT.
count_atomic(T, N) :- T =.. L, count_atomic(L, N).
```

6.11
```
atomiclist([], []).
atomiclist(A, [A]) :- atomic(A).
atomiclist([H|T], L) :-
    atomiclist(H, LH), atomiclist(T, LT),
    append_list(LH, LT, L).
atomiclist(T, L) :- T =.. LT, atomiclist(LT, L).

append_list([], L, L).
append_list([H|T], L, [H|A]) :- append_list(T, L, A).
```

6.12
```
deepmap(_, [], []).
deepmap(P, A, PA) :- atomic(A), G =.. [P, A, PA], G.
deepmap(P, [H|T], [PH|PT]) :-
    deepmap(P, H, PH), deepmap(P, T, PT).
deepmap(P, T, PT) :-
    T =.. [F|A], deepmap(P, A, PA), PT =.. [F|PA].
```

6.13
```
sort_list([], []).
sort_list([H|T], S) :-
    partition(T, H, L, R),
    sort_list(L, SL),
    sort_list(R, SR),
    append_list(SL, [H|SR], S).

partition([], _, [], []).
partition([X|T], Y, [X|L], R) :-
    X =< Y, partition(T, Y, L, R).
partition([X|T], Y, L, [X|R]) :- partition(T, Y, L, R).

append_list([], L, L).
append_list([H|T], L, [H|A]) :- append_list(T, L, A).
```

6.14
```
is_subset([], _).
is_subset([H|T], L) :- member_list(H, L), is_subset(T, L).

member_list(E, [E|_]).
member_list(E, [_|T]) :- member_list(E, T).
```

6.15
```
inter_sets([], _, []).
inter_sets([H|T], L, [H|IT]) :-
    member_list(H, L), inter_sets(T, L, IT).
inter_sets([_|T], L, IT) :- inter_sets(T, L, IT).

member_list(E, [E|_]).
member_list(E, [_|T]) :- member_list(E, T).
```

6.16
```
power_set(L, P) :- unique(L, UL), power_set1(UL, P).

unique([], []).
unique([H|T], UT) :- member_list(H, T), unique(T, UT).
unique([H|T], [H|UT]) :- unique(T, UT).

member_list(E, [E|_]).
member_list(E, [_|T]) :- member_list(E, T).

power_set1([], [[]]).
power_set1([H|T], P) :-
    power_set1(T, PT),
    map_add_head(H, PT, MPT),
    append_list(PT, MPT, P).

map_add_head(_, [], []).
map_add_head(E, [H|T], [[E|H]|MT]) :- map_add_head(E, T, MT).

append_list([], L, L).
append_list([H|T], L, [H|A]) :- append_list(T, L, A).
```

第 7 章の解答例

7.1
```
not_atom(T) :- \+ atom(T).
```

7.2
```
disjunction(A, _) :- A.
disjunction(_, B) :- B.
```

7.3
```
implication(A, B) :- A, !, B.
implication(_, _).
```

7.4
```
truth_value(G, true) :- G, !.
truth_value(_, false).
```

7.5
```
equivalence(A, B) :-
    truth_value(A, VA), truth_value(B, VB), VA == VB.
```

7.6
```
exdisjunc(A, B) :-
    truth_value(A, VA), truth_value(B, VB), VA \== VB.
```

7.7
```
if(C, T, _) :- C, !, T.
if(_, _, E) :- E.
```

7.8
```
times(N, _) :- N < 1, throw(times(N, 'less than one')).
times(N, G) :- times(N), G, fail.
times(_, _).
```

7.9
```
asterisk(N) :- catch(write_asterisk(N), E, write_error(E)).

write_asterisk(N) :- times(N, write(*)), nl.

write_error(E) :- write('error: '), write(E), nl.
```

第 8 章の解答例

8.1
```
eval_file(PR, PW) :-
    open(PR, read, SR), open(PW, write, SW),
    repeat_read_eval_write(SR, SW),
    close(SR), close(SW).

repeat_read_eval_write(SR, SW) :-
    repeat,
    read(SR, E), eval_write(SW, E),
    E == end_of_file.

eval_write(_, end_of_file) :- !.
eval_write(S, E) :- V is E, write(S, V), write(S, .), nl(S).
```

8.2
```
file_list(P, L) :-
    open(P, read, S), stream_list(S, L), close(S).

stream_list(S, L) :- stream_list(S, dummy, [_|L]).

stream_list(_, end_of_file, []).
stream_list(S, H, [H|L]) :- read(S, T), stream_list(S, T, L).
```

8.3
```
space_to_dot(PR, PW) :-
    open(PR, read, SR), open(PW, write, SW),
    repeat_read_write_stod(SR, SW),
    close(SR), close(SW).

repeat_read_write_stod(SR, SW) :-
    repeat,
    get_char(SR, C), put_char_stod(SW, C),
    C == end_of_file.

put_char_stod(_, end_of_file) :- !.
put_char_stod(S, ' ') :- !, put_char(S, .).
put_char_stod(S, C) :- put_char(S, C).
```

8.4
```
quote_file(PR, PW) :-
    open(PR, read, SR), open(PW, write, SW),
    read_write_quote(SR, SW),
    close(SR), close(SW).

read_write_quote(SR, SW) :-
    repeat, get_line(SR, L), write_quote(SW, L), last_line(L).

get_line(S, L) :- get_line(S, dummy, [_|L]).

get_line(_, -1, [-1]) :- !.
get_line(_, 10, [10]) :- !.
```

```
get_line(S, H, [H|L]) :- get_code(S, C), get_line(S, C, L).

write_quote(_, [-1|_]) :- !.
write_quote(S, L) :-
    write(S, '> '), write_line(S, L).

write_line(_, []) :- !.
write_line(_, [-1|_]) :- !.
write_line(S, [H|T]) :-
    put_code(S, H), write_line(S, T).

last_line([-1|_]).
last_line([_|T]) :- last_line(T).
```

8.5

```
line_number(PR, PW) :-
    open(PR, read, SR), open(PW, write, SW),
    read_write_linenum(SR, SW),
    close(SR), close(SW).

read_write_linenum(SR, SW) :- read_write_linenum(SR, 1, SW).

read_write_linenum(SR, N, SW) :-
    get_line(SR, L), write_linenum(SW, N, L),
    \+ last_line(L), !,
    N1 is N + 1, read_write_linenum(SR, N1, SW).
read_write_linenum(_, _, _).

get_line(S, L) :- get_line(S, dummy, [_|L]).

get_line(_, -1, [-1]) :- !.
get_line(_, 10, [10]) :- !.
get_line(S, H, [H|L]) :- get_code(S, C), get_line(S, C, L).

write_linenum(_, _, [-1|_]) :- !.
write_linenum(S, N, L) :-
    write(S, N), write(S, ' '), write_line(S, L).

write_line(_, []) :- !.
write_line(_, [-1|_]) :- !.
write_line(S, [H|T]) :-
    put_code(S, H), write_line(S, T).

last_line([-1|_]).
last_line([_|T]) :- last_line(T).
```

8.6

```
eval_shell :-
    repeat,
    write('eval> '), read(E), write_eval(E),
    E == exit.

write_eval(exit) :- !.
write_eval(E) :- V is E, write(V), nl.
```

8.7

```
interpreter :-
    repeat,
    write('?- '), read_term(G, [variable_names(N)]),
    execute_goal(G, N),
    G == halt.

execute_goal(halt, _) :- !.
execute_goal(G, N) :-
    G, !, write_variables(N), write(yes), nl.
execute_goal(_, _) :- write(no), nl.

write_variables([]) :- !.
write_variables([A = V|T]) :-
```

```
write(A), write(' = '), write(V), nl, write_variables(T).
```

付録 B 参考文献

[Bowen,1982] D. L. Bowen, L. Byrd, F. C. N. Pereira, L. M. Pereira and D. H. D. Warren, *DECsystem–10 Prolog User's Manual*, 1982.

[Bratko,2001] Ivan Bratko, *Prolog Programming for Artificial Intelligence, Third Edition*, Addison-Wesley, 2001, ISBN 0-201-40375-7.

[Clocksin,1997] William F. Clocksin, *Clause and Effect: Prolog Programming for the Working Programmer*, Springer-Verlag, 1997, ISBN 3-540-62971-8.

[Clocksin,2003] W. F. Clocksin and C. S. Mellish, *Programming in Prolog, Fifth Edition*, Springer-Verlag, 2003, ISBN 3-540-00678-8.

[Deransart,1996] Pierre Deransart, AbdelAli Ed-Dbali and Laurent Cervoni, *Prolog: The Standard: Reference Manual*, Springer-Verlag, 1996, ISBN 3-540-59304-7.

[Nilsson,2000] Ulf Nilsson and Jan Małuszyński, *Logic, Programming and Prolog (2ed)*, 2000.

[Spivey,2002] Michael Spivey, *An Introduction to Logic Programming through Prolog*, 2002.

[Sterling,1994] Leon Sterling and Ehud Shapiro, *The Art of Prolog: Advanced Programming Techniques, Second Edition*, MIT Press, 1994, ISBN 0-262-19338-8.

[Wielemaker,2005] Jan Wielemaker, *SWI–Prolog 5.5 Reference Manual*, 2005.

[安部,1985] 安部憲広、『Prolog プログラミング入門』、共立出版、1985、ISBN 4-320-02237-8。

[有川,1988] 有川節夫、原口誠、『述語論理と論理プログラミング』、「知識工学講座」、第 4 巻、オーム社、1988、ISBN 4-274-07386-6。

[井田,1988] 井田哲雄、『プログラミング言語の新潮流』、「計算機科学/ソフトウェア技術講座」、第 2 巻、共立出版、1988、ISBN 4-320-02377-3。

[黒川,1985] 黒川利明、『Prolog のソフトウェア作法』、「岩波コンピュータサイエンス」、岩波書店、1985、ISBN 4-00-007681-7。

[後藤,1984] 後藤滋樹、『PROLOG 入門——知識情報処理の序曲——』、「ソフトウェアライブラリ」、第 1 巻、サイエンス社、1984、ISBN 4-7819-0352-5。

[斉藤,1987] 斉藤孝、『はじめての人工知能言語 RUN/PROLOG』、CBS 出版、1987。

[柴山悦哉,1986] 柴山悦哉、桜川貴司、萩野達也、『Prolog–KABA 入門』、「岩波コンピュータサイエンス」、岩波書店、1986、ISBN 4-00-007687-6。

[柴山潔,1991] 柴山潔、『並列記号処理』、「並列処理シリーズ」、第 10 巻、コロナ社、1991、ISBN 4-339-02590-9。

[太細,1984] 太細孝、鈴木克志、伊草ひとみ、佐藤裕幸、『Prolog 入門』、啓学出版、1984、ISBN 4-7665-0146-2。

[田村,2003] 田村直之、『Prolog 入門』、2003。

[長尾,1988] 長尾真、『知識と推論』、「岩波講座ソフトウェア科学」、第 14 巻、岩波書店、1988、ISBN 4-00-010354-7。

[中島,1992] 中島秀之、上田和紀、『楽しいプログラミング II——記号の世界——』、「コンピュータ入門」、第 5 巻、岩波書店、1992、ISBN 4-00-007755-4。

[新田,1986] 新田克己、佐藤泰介、『Prolog』、「人工知能用言語シリーズ」、第 1 巻、昭晃堂、1986、ISBN 4-7856-3601-7。

[広井,2005] 広井誠、『お気楽 Prolog プログラミング入門』、2005。

索引