# デカルト言語の文法

## 1. データ型

### 1.1 文字列

```
hello   “こんにちは 世界”    ‘1 2 3’
```

文字列は、文字の並んだシンボルの列です。

“または’で括られたものも文字列です。

空白、タブおよび改行などが文字列に含まれる場合は、”または’で括らなければなりません。

### 1.2 数字

```
100  0.8123  0xa1
```

8byte精度の整数と、long double精度の浮動小数点数が使えます。

16進数は0xから始まる文字で表します

### 1.3 変数

```
#x   #a   #abc
_ 無名変数
```

変数は、#で始まるシンボルの列で表します。

また、無名変数として _ が使えます。

### 1.4 リスト

```
(abc   “こんにちは 世界”  #xy  1 2 3  ( list1 #z))
```

文字列、数字、変数、リスト、関数述語を( )で括ったものです。(関数述語については後述します。)

リストは、Lisp言語のリストと同じ構造を持ちます。空のリストは、()で表します。ドット対記法に相当する表記には、: を使います

### 1.5 関数述語

```
<app  #z  (a b c) (d e f)  <ap #d  d1  d2>>
```

文字列、数字、変数、リスト、関数述語を<>で括ったものです。

実行評価の対象であり、評価された後は、true, false, unknownの3値を取ります。

評価の結果成功した場合は、便宜的に、第一引数は、関数の返り値とみなされます。

引数には、文字列、数字、変数、リスト、関数述語を記述することができます。

# 2. プログラムの記述

## 2.1 述語

述語は実行した結果として、true, false, unknownの3値を取ります。

デカルト言語での述語は、実行された結果trueとなった場合、登録されているプログラムとの単一化が行われ、述語に含まれる変数には値が設定されます。

falseの場合は、述語の単一化は失敗し中断されます。

unknownの場合は、他の可能性が試行され、必要に応じてバックトラックが行われます。

## 2.2 関数述語

述語において、評価の結果成功した場合は、便宜的に、第一引数は、関数の返り値とみなされます。このような述語を関数述語と呼びます。 引数の中に関数述語がある場合には、その関数述語がtrueである場合には、関数述語自身が第一引数の値と置き換えられます。

falseの場合は、呼び出した関数述語もfalseとなり、unknownの場合は同様に呼び出した関数述語もunknownとなります。

関数述語で使えるようにするために、述語を書くときは、第一引数に結果が返るように書いておくと便利で都合がよいです。

## 2.3 注釈コメント

注釈(コメント)には次の3種類があります。

```
・ //から行末まで
・ #から行末まで
・ /* */に囲まれた範囲
```

## 2.4 述語の記述ルール

プログラムは述語の羅列によって記述します。最後は ; (セミコロン)によって区切ります。最初の述語はヘッド(head)、それ以外の述語はボディ(body)と呼びます。

ヘッドとボディを組み合わせたものを節(clause)と呼びます。

```
<述語>          <述語> <述語> ... <述語> ;
---------   ------------------------------------  節
ヘッド              ボディ
```

ボディには、述語だけでなくリストを記述することもできます。

```
<述語> (<述語>   <述語> ( <述語> ) (<述語> <述語>))  ;
```

記述の可視化を向上させるために以下のようにボディをタブで区切って記述すると読みやすくなります。

```
<述語>
        <述語>
```

```
        <述語>
        <述語>
;
```

## 2.5 単一化(Unify)

単一化とは、二つの述語を同じ形（述語名から引数の値まですべて対応する項を等しくすること）にする操作です。

```
<pred1  #x   #y  123  (a b c)>
        対応する項を一致させる。
<pred1  xyz  #z  123  #a>

単一化の結果 #x=xyz,  #y=#z,  #a=(a b c))  となります。
```

## 2.6 プログラムの呼び出し

記述されたプログラムを呼び出すには、述語の先頭に?を付けます。

```
? <述語>;
```

呼び出された述語は、プログラムの中のヘッダと順に比較され、単一化を試みます。単一化に成功すると、次にボディに記述されているプログラムを左から順に呼び出していくことになります。

## 2.7 デバッグ機能

<tron>述語を実行するとトレース機能がオンになり、実行時にトレース情報が出力されます。

トレース機能をオフにする場合は、<troff>述語を実行してください。

## 2.8 数式の計算、逆ポーランド式の計算

**let, letf、letc述語**

```
<let 数式>
<letf 数式>
<letc 数式>
```

数式を計算します。

左辺が変数の場合は、計算結果を代入します。

左辺が数値の場合は、計算結果と等しいか判定します。

letは、整数の計算を行い、letfは浮動小数点数の計算を行い、letcは複素数の計算を行います。 右辺の数式には、関数述語も含めることができます。

```
? <let #x = 1 + 2 +  7 *  5  >;
? <letf #x = ::sys <cos _ 1> + ::sys <sin _ 1>>;
? <letc #x = (1-i)/(1+i)>;
```

letを省略した数式の演算は整数演算として扱われます。 そのため、以下の例はどちらも同じ結果となります。

```
? <let #x = 1 + 2>;
```

```
? <#x = 1 + 2>;
```

**rpn, rpnf述語**

```
<rpn 変数 逆ポーランド式>
<rpnf 変数 逆ポーランド式>
```

逆ポーランド式を計算して、変数に結果を設定します。 rpnは、整数の計算を行い、rpnfは浮動小数点数の計算を行います。 逆ポーランド式には、関数述語を含めることもできます。

```
? <rpn #x 1 2 3 4 5 6 7 8 9 10 + + + + + + + + + >;
? <rpnf #i 10 9 8 * * <rpn #j  10 30 *> / #j / >;
```

## 2.9 数式の比較

**compare, comparef述語**

```
<compare 数式 比較演算子 数式>
<comparef 数式  比較演算子 数式>
```

数式を比較します。 compareは、整数として比較し、comparefは、浮動小数点数として比較します。

比較演算子には以下のものが使えます。

```
=, == 等しい
!=, <> 等しくない
>      大きい
>=     以上
<      小さい
<=     以下
```

また、比較式にはand, or, not演算子を使うこともできます。

```
? <compare  (#x > 1) and (#x < 20)>;
? <comparef  not ((#y > 1.2) or (#z < 3.0))>;
```

## 2.10 グローバル変数

デカルト言語での#変数は、節(clause)の中だけで有効なローカル変数です。 節を超えてデータを保存したい場合は、グローバル変数として設定します。デカルト言語でのグローバル変数は、変数名と値の組を新たな節として定義することにより実装します。

グローバル変数の設定にはsysモジュールのsetVar述語を使います。

値の参照は、<変数名 #値変数>とすると#値変数に値が設定されます。

```
グローバル変数の設定
? <setVar color “red”>;

グローバル変数の参照
? <color #cl>;
#clには、redが設定される。
```

```
```

setvar述語は実行されたとき、すでに同じ名前の変数があればその節を置換え、無ければ新たな節が追加されます。

## 2.11 グローバル配列変数

sysモジュールのsetarray述語によって、グローバル配列変数を設定できます。

配列もグローバル変数と同様に変数名とインデックスに値の組を新たな節(clause)として定義することにより実装します。

インデックスには、数字、文字列、リスト、述語等何でも指定できます。多次元配列の場合はインデックスにリストを設定すると良いでしょう。

```
グローバル配列変数の設定
? <setArray  ary   10 7>;
? <setArray  cell  100 (10 20)>;

グローバル配列変数の参照
?<ary  #v 7>;
#vには、10が設定される。
? <cell    #val (10 20)>;
#valには、100が設定される。
```

## 2.12 ファイルI/O

ファイルの入力や出力の先のファイルを、引数の述語を実行している間だけ切り替えることで実現します。

<openr ファイル名 述語...>

　　ファイル名のファイルを読み取り用にオープンして、述語を実行します。

<openw ファイル名 述語...>

　　ファイル名のファイルを書き込み用にオープンして、述語を実行します。

<openwp ファイル名 述語...>

　　ファイル名のファイルを追記書き込み用にオープンして、述語を実行します。

## 2.13 ライブラリモジュール

ライブラリモジュールの呼び出し方には、以下の３通りがありますがどれも同じ意味を表します。

```
::ライブラリモジュール 述語
例)
::sys <writenl hello>

<unify ライブラリモジュール 述語>
例)
<unify sys <writenl hello>>

<obj ライブラリモジュール 述語>
例)
<obj sys <writenl hello>>
```

通常は、::を使う記述が便利でしょう。

外部のライブラリモジュールを使うためには、ライブラリモジュールをインクルードする必要があります。

```
? <include ライブラリモジュール>

例)
?<include list>;
```

```
?::list <append #list (a b c) (d e)>;
```

ただし、sysモジュールだけは、システムに組み込まれたライブラリモジュールなので、インクルードしなくても使えます。

## 2.14 ライブラリモジュールの作り方

ライブラリモジュールは、ファイル名をライブラリモジュール名で作ります。 中に記述するプログラムは通常のデカルト言語のプログラムを記述すればOKです。 インクルードすることによって、ライブラリモジュールとして使用できます。

例）exampleモジュール

以下をexampleと名付けてファイルに保存します。

```
<append #X () #X>;
<append (#A : #Z)  (#A : #X) #Y>
        <append #Z #X #Y>;
```

以下のようにappendを呼び出します。

```
? <include example>;

? ::example <append #list (abc def) (ghi jkl)>;
```

## 2.15 オブジェクト指向

デカルト言語でのオブジェクトは、ライブラリモジュールをオブジェクトとして見做すことによって実現しています。つまり、ライブラリモジュール名をオブジェクト名として扱います。

オブジェクトの定義は以下のように行います。

```
::<オブジェクト名
        メソッド定義;
        メソッド定義;

        inherit 継承するオブジェクト名;
>;
```

メソッドには、通常のデカルト言語のプログラムである、ヘッドとボディを組み合わせた節(clause)を記述することができます。

## 2.16 オブジェクト指向プログラムの例

ここでは、例として鳥、ペンギンおよび鷹のオブジェクトを定義しましょう。

```
// 鳥のオブジェクト：  飛ぶ、歩く
::<bird
        <fly>;
        <walk>;
>;

// ペンギンのオブジェクト：  飛ばない、泳ぐ、鳥なので歩く
::<penguin
        <fly>   <false>;         // 飛ぶことを否定
        <swim>;          // 新たに泳ぐことを追加
        inherit bird;    // birdを継承
>;
```

```
// 鷹は鳥と同じで、飛ぶ、歩く
::<hawk
        inherit bird;   // birdを継承
>;
```

さて、鳥、ペンギンおよび鷹のオブジェクトに対して質問します。

オブジェクトに対するメソッドの呼び出しにより質問結果を確認できます。

メソッドの呼び出しは、ライブラリモジュールの呼び出し方法とまったく同じです。

```
?::bird <swim>; 鳥は泳ぐか？
result --
(::bird <swim>)
-- unknown      知らない

?::penguin <swim>;  ペンギンは泳ぐか？
result --
(::penguin <swim>)
-- true         泳ぐ。

?::bird <walk>; 鳥は歩くか？
result --
(::bird <walk>)
-- true         歩く。

?::penguin <walk>;  ペンギンは歩くか？
result --
(::enguin <walk>)
-- true         歩く。

?::bird <fly>;  鳥は飛ぶか？
result --
(::bird <fly>)
-- true         飛ぶ。

?::penguin <fly>;       ペンギンは飛ぶか？
result --
(::penguin <fly>)
-- false                飛ばない。

?::penguin <run>;       ペンギンは歩くか？
result --
(::penguin <run>)
unknown         知らない。

?::hawk <fly>;  鷹は飛ぶか？
result --
(::hawk <fly>)
-- true         飛ぶ。

?::hawk <walk>; 鷹は歩くか？
result --
(::hawk <walk>)
-- true         歩く。


?::hawk <swim>; 鷹は泳ぐか？
result --
(::hawk <swim>)
-- unknown      知らない。
```

## 2.17 構文解析

EBNF(拡張バッカス記法)による構文は以下のようなものです。

```
expr            =  expradd
expradd =  exprmul { "+"  exprmul  |  "-"  exprmul }
exprmul =  exprID  { "*"  exprID  |  "/"  exprID }
exprID  =  "+" exprterm | "-"  exprterm  |  exprterm
exprterm        =  "("  expr  ")" | 数字列
```

```
    abcz                = abc [ アルファベット ]
```

EBNF(拡張バッカス記法)の構文を、デカルト言語によって一対一に対応して変換できます。(デカルト言語はLL(*)の文法を受け付けます。)

デカルト言語による構文

```
<expr>  <expradd>;
<expradd>       <exprmul> { "+" <exprmul> | "-" <exprmul> };
<exprmul>       <exprID> { "*" <exprID> | "/" <exprID> };
<exprID>        "+" <exprterm> | "-" <exprterm> | <exprterm> ;
<exprterm>      "(" <expr> ")" | <FNUM #t> ;
    <abcz>      abc [ <A #n> ];
```

〜　：　省略可能

{〜}　：　0回以上の繰り返し。

述語は1ターンの実行後に実行中の変数のバインドがクリアされて最初から実行されます。

|　：　or選択

<>で括られない文字列　：　終端記号

数字列のFNUM以外にも、トークンとして使える述語がsysモジュールには多数定義されています。

入力ファイルは前章で説明したファイルI/Oで指定します。

注) 標準入力からは構文解析の機能は使えません。getline述語かファイルからの入力で使用してください。

## 2.18 構文解析のトークン用述語

```
<TOKEN 変数 述語...>

        入力の構文解析述語実行後に、得られたtokenを変数に
        設定します。

<SKIPSPACE>

        入力のスペースをスキップします。

<C [変数]>

        入力を一文字変数に設定します。

<N [変数]>

        入力が数字であった場合は、変数に設定します。
        違う場合はunknownを返します。

<A [変数]>

        入力がASCII文字であった場合は、変数に設定します。
        違う場合はunknownを返します。

<AN [変数]>

        入力がASCII文字か数字であった場合は、変数に
        設定します。
        違う場合はunknownを返します。

<^>
        行の先頭とマッチする。

<$>
        行の最後とマッチする。

<* [変数]>
        任意の文字列とマッチする。
```

```
<CR>

        入力がCR改行であった場合には、trueを返します。
        違う場合はunknownを返します。

<CNTL [変数]>

        入力がCNTL文字であった場合には、変数に設定します。
        違う場合はunknownを返します。

<EOF>

        入力がEOF(End Of File)である場合はtrueを返します。
        違う場合はunknownを返します。

<SPACE>

        入力がスペースである場合はtrueを返します。
        違う場合はunknownを返します。

<PUNCT>

        アルファベット、数字以外の文字である場合はtrue
        を返します。

<WORD [変数]>

        任意の文字列で、アルファベット、数字、"_"以外
        の文字列の場合はunknownを返します。

<NUM [変数]>

        入力の整数を変換して、変数に設定します。

<FNUM [変数]>

        入力の浮動小数点数を変換して、変数に設定します。

<ID [変数]>

        入力の文字列（先頭はアルファベット、それ以外
        は数字も可)、合致すれば変数に設定します。

<RANGE 変数 文字１ 文字２>
<NONRANGE 変数 文字１ 文字２>

        文字１と文字２の範囲に含まれるならばtrue
        となります。

<GETTOKEN 変数>

        直前の構文解析の結果であるトークンを変数に設定
        します。

<SKIP 文字列>
          文字列までの構文解析を行わずにスキップします。

<SKIPCR>
        改行までの構文解析を行わずにスキップします。
```

## 2.19 TOKEN述語の使い方

TOKEN述語を使用して、任意の文字列に合致するトークンを合成できます。（正規表現の代替となります。）

先頭が英字で２文字目から英数字の文字列

```
<TOKEN #token  ::sys<A _> { ::sys<AN _> }>
```

数字列　(<NUM #token>と同等です)

```
<TOKEN #token  { <N _> }>
```

大文字か数字の文字列

```
<TOKEN #token { <RANGE  _  A  Z> | <N _> } >
```

“DISK”+数字3桁

```
<TOKEN #token  “DISK”  <N _>  <N _>  <N _>>
```

ひらがな

```
<TOKEN #token  { <RANGE  _  “ぁ”  “ん”> } >
```

カタカナ

```
<TOKEN #token  { <RANGE  _  “ァ”  “ヲ”> } >
```

注）　TOKEN述語を使用しないと、中に含まれるトークン間に空白が含まれても許されてしまい、意図した文字列と異なるものとマッチングしてしまいます。

文字列のような字句解析には、TOKEN述語を使い、文字列の連なりである文のような構文の解析には、TOKEN述語を使いません。

### 2.20 timeout述語

timeout述語は、指定時間以内に終了しない述語の処理を打ち切りunknownで返します。指定時間はマイクロ秒単位で指定します。

```
<timeout 指定時間 述語...>
```

実行時間がかかりそうな処理や無限ループに陥る可能性のある処理をとりあえず実行し、指定時間に終わらない場合には他の方法を試す用途に使いましょう。

```
<処理>   <timeout 1000000 <処理A>>;
<処理>   <timeout 1000000 <処理B>>;
<処理>   ::sys<writenl “どれも失敗”>;
```

## 2.21 findall 述語

findall 述語は、引数の述語のすべての解を求めるのに使います。

通常は、述語の実行は、最初に見つかった解によって終了します。しかし、他にも解があることが分かっている場合でも、そのままでは求めることができません。

たとえば経路の探索のような問題の場合は、最初に得られた経路が最適とは限らず、すべての経路に対して、最適かどうかを評価する必要があるかもしれません。

```
<findall 述語...>
```

findall 述語の実行は、無限ループに陥り、処理が永久に終わらない可能性があります。

ほどほどのところで処理を打ち切るためには、timeout述語と組み合わせて利用すると便利でしょう。

## 2.22 for ループ、foreach ループ、map述語

手続き的なループ処理の基本です。

```
<for (変数 実行回数) 述語...>
<for (変数 初期値 最終値) 述語...>
```

指定された回数、引数の述語を実行します。

変数には、順に数が設定されます。実行回数が指定された場合は0からの値が、初期値が指定された場合はその値から実行回数または最終値の値まで、１ずつ増加させます。

述語は１ターンの実行後にすべての変数のバインドがクリアされて最初から実行されます。

```
<foreach (変数 リスト) 述語...>
<map  (変数 リスト) 述語...>
```

リストの要素ごとに引数の述語を実行します。 変数には、リストの値が順に設定されます。 述語は１ターンの実行後に実行中の変数のバインドがクリアされて最初から実行されます。

## 2.23 文字コード

文字コードは、デフォルトでUTF8を使用します。

EUCまたはSJISを指定する場合は、code述語で指定してください。

```
UTF8指定

   ? ::sys <code UTF8>;

EUC指定

   ? ::sys <code EUC>;

SJIS指定

   ? ::sys <code SJIS>;
```