

デカルト言語の概要

1. prologとの類似性

デカルト言語は、1階述語論理を基にしているため、prolog言語と多くの共通点を持ちます。そこで、リストの連結するプログラムでデカルト言語とprolog言語を比較してみましょう。

デカルト言語では以下のようになります。

```
<append #Z () #Z>;  
<append (#W : #Z1) (#W : #X1) #Y> <append #Z1 #X1 #Y>;  
  
?<append #x (a b) (c d)>;
```

prologでは以下のようになります。

```
append(Z, [], Z).  
append([W | Z1], [W | X1], Y) :- append(Z1, X1, Y).  
  
? append(X, [a b], [c d]).
```

どちらの例でも、第2引数と第3引数のリストを連結し、第1引数に結果を返します。

違いがお分かりでしょうか。

- 1) 述語は<>で括られる。
- 2) 引数の区切りは空白を使う。
- 3) prologでは最後にピリオド"."を置くが、デカルト言語ではセミコロン";"を置く。
- 4) リストは[]ではなく()を使う。
- 5) リストを分割する"|"が、デカルト言語では":"である。
- 6) ヘッド部とボディ部の区切りにprologでは":-"を使うがデカルト言語では何も無い。
- 7) デカルト言語では変数には"#"が付く。

デカルト言語で、上記appendを実行するには、次のように?を付けた述語で実行します。

```
?<append #x (a b) (c d)>;
```

2. prologとの相違点

前項でデカルト言語とprologとの類似点について言及しましたが、ここでは相違点について述べます。

デカルト言語では、ボディ部に書けるものが述語の羅列に限られるprologとは異なり、任意のリストが書けることが大きく異なります。リストの場合の実行順序は記述された左から順に実行されます。

```
<example #x> (<e1 #x> (<e2 #x #y> <e3 #y>) <e4 #y> ) <e5 #x>;
```

上記の例では、exampleが実行されると、e1, e2, e3, e4, e5の順に実行されます。

この記述の利点は、述語のつらなりをまとめて述語の引数に渡すようなメタな処理を行えることや、次に示す構文解析用の記号のために処理をまとめるために利用できることです。

構文解析のための記号として{}繰り返し、![]省略可能、|オア選択などが使えるのもprologとの大きな相違点です。

```
<example2 #a> <abc> { <def #a> | (<hi j> <lkm>) } ![ <nop> ] <end>;
```

上記の例では、example2が実行されると、abcが実行され、defか(hi j lkm)のどちらか成功する処理が繰り返され、どちらも該当しなければループを抜けて、nopは成功しても失敗しても処理され、最後にendが実行されます。

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

4. 数値計算

数の計算には、整数ではlet述語、浮動小数点ではletf述語を使います。

```
<let #x = 1 + 2>;  
<letf #f = 1.1 + 0.3*(2.3-1.2)>;
```

数式内では、後述する関数述語が使えます。

```
<letf #f = ::sys <sin #x1 3.14>+::sys<cos #x2 3.14>>;
```

letは省略することができます。以下は2つとも同一です。

```
<let #z = #x + #y>;  
<#z = #x + #y>;
```

注) letfは省略できません。

5. 関数述語

let, letf, f, funcなど述語の引数は関数述語として評価され、関数述語の返す関数値は第1引数です。返り値の変数は、無名変数“_”を使うと便利です。

```
<letf #x = ::sys<sin _ ::sys<cos _ 3.14>>>;
```

let, letfの中で使えるのは、数値を返す関数のみです。

```
<f #x ::sys<car #x1 ::sys<cdr _ (a b c)>>>;
```

fはfuncの別名であり、まったく同じ働きをします。
また、fは引数としてリストを取ることが可能であり、リストの要素に関数述語が含まれる場合は、すべて評価された後に関数値として返されます。

```
<f #x (This is a ::sys <getline _)>>;
```

(上記で“::sys”とあるのは後述するライブラリの呼び出しを表し、sysモジュールのgetline述語の呼び出しを意味します。)

6. ライブラリ

ライブラリの呼び出しは以下のような形式で行います。

```
::ライブラリモジュール名 <述語>  
<unify ライブラリモジュール名 <述語>>  
<obj ライブラリモジュール名 <述語>>
```

これら3種類の呼び出し方法は同一の動作内容です。

7. オブジェクト指向

オブジェクトは以下のような形式で定義します。

```
:: < オブジェクト名  
    プログラムまたはinherit 継承オブジェクト  
>;
```

例として鳥、ペンギン、鷹のオブジェクト例を以下に示します。

```
::<bird  
    <fly>;  
    <walk>;  
>;  
  
::<penguin  
    <fly>  
        <!><false>;  
    <swim>;  
    inherit bird;  
>;  
  
::<hawk  
    inherit bird;  
>;
```

オブジェクトの呼び出し方は、ライブラリの呼び出し方法と同じです。以下を試してみてください。

```
?::bird <swim>;  
?::penguin <swim>;
```

```

?::bird <walk>;
?::penguin <walk>;
?::bird <fly>;

?::penguin <fly>;
?::penguin <run>;

?::hawk <fly>;
?::hawk <walk>;
?::hawk <swim>;

```

7. EBNF記法

構文解析のためのEBNF記法が使えます。

```

<名前> "田中";
<名前> "佐藤";

<name #x>
    "私" "は"
    <名前> <GETTOKEN #x>
    "です"
    ["。"]
    ;

```

```

? ::sys<getline _ <name #name>>;

```

```

私は佐藤です。
result --
(<obj sys <getline 私は佐藤です。 <name 佐藤>>>)
-- true

```
