デカルト言語のプログラム例

例題プログラムを実行するには、その内容を適当なファイルに記述してdescartesの引数として実行してください。

1. Hello, world

print述語でメッセージを表示します。

? <print "Hello, world!!">;

2. 数の計算

let述語で整数を計算します。

? <let #x = 1+2+3>;

変数は頭に#を付けます。

letf述語は浮動小数点数を計算します。

? <let #x = 1.23 + 2.34 + 3.456>;

letを省略すると、整数演算式を計算します。

? <#x = 1+2+3>;

3. リストの追加

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、リストを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(DEC-10 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。

4. for文による繰り返し処理

for述語を使うと、繰り返しのループ処理を実行できます。

?<for (#i 1 10) <print #i>>;

1から10までのループを実行します。

2重以上のループ処理も可能です。

```
?<for (#i 1 10)
        <for (#j 1 10)
                <print (#i #j)>>>;
```

5. 関数述語による計算

let, letf述語の引数は関数述語として評価されます。関数述語の返す関数値は第1引数です。

返り値の変数には、無名変数"_"を指定すると便利です。

? <letf #x = 1 + ::sys <cos _ 3.14>>;

::sys <cos _ 3.14>がcos関数の関数述語です。

6. 関数述語による関数

let, letfは、数を返す関数しか使えませんが、func述語を使えば、さまざまなデータを引数とする関数を記述できます。

?<func #x this is a ::sys <geline _> ".">;

::sys <getline _>は、キーボードより１行入力する述語です。

7. htmlの合成

func述語を使い、htmlファイルを合成するプログラムです。

テンプレートを用意し、その中の可変部分を述語や変数で埋め込み、関数述語として実行すると、目的のｈｔｍｌファイルが合成されます。

```
<html #html #title #body> // ヘッド
        <func #html // func関数述語。これより下の引数がテンプレート
                                // #titleと#bodyが置き換えられる。
        (
"<HTML>
<HEAD>
<TITLE>" #title "</TITLE>
</HEAD>
<BODY>
        test html <BR>"
        #body "<BR>"
        ::sys<random _ >
" </BODY>
</HTML>"
                                )>;
?<html #h "Hello World" "This program is test.">   ::sys <writenl #h>;   // 実行
```

以下は結果です。

```
(<HTML>
<HEAD>
<TITLE> Hello World </TITLE>
</HEAD>
<BODY>
        test html <BR> This program is test. <BR> 693663189 </BODY>
</HTML>)
```

引数として入力した、"Hello World"、 "This program is test."　および乱数の部分が置き換えられているのが判るでしょうか。

8. 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> ::sys<GETTOKEN #x>
    "です"
    ["。"]
    ;
    ? ::sys<getline _ <name #name>>;
```

次のように入力すると名前だけが抽出されます。

私は佐藤です。

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

9. 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
        inheirt 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。

以下を試してみましょう。

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

? ::penguin <fly>;

? ::penguin <run>;

? ::hawk <fly>;

? ::hawk <walk>;

? ::hawk <swim>;

結果は以下のようになります。

result --

(<obj bird <swim>>)

-- unknown 鳥が泳ぐのか分からない

result --

(<obj penguin <swim>>)

-- true　ペンギンは泳ぐ

result --

(<obj bird <walk>>)

-- true　鳥は歩く

result --

(<obj penguin <walk>>)

-- true　ペンギンは歩く

result --

(<obj bird <fly>>)

-- true　鳥は飛ぶ

result --

(<obj penguin <fly>>)

-- false　ペンギンは飛ばない

result --

(<obj penguin <run>>)

-- unknown　ペンギンが走るか分からない

result --

(<obj hawk <fly>>)

-- true　鷹は飛ぶ

result --

(<obj hawk <walk>>)

-- true　鷹は歩く

result --

(<obj hawk <swim>>)

-- unknown　鷹が泳ぐのかわからない

10. ユークリッドの互除法

ユークリッドの互除法を使い最大公約数を求めるプログラムを作ります。

<gcd 答 整数１ 整数２>の形式で整数１と整数２の最大公約数を答に設定します。

ユークリッドの互除法については、書籍やWWW上で参照してください。

<gcd #x #x 0>; // #xと0の最大公約数は、#xである。

```
<gcd #x #a #b>
        ::sys <compare #a >= #b> // #aのほうが大きい場合
        <#c = #a % #b> // letが省略されている
        <gcd #x #b #c> // 末尾再帰
        ;
<gcd #x #a #b>
        <#c = #b % #a> // #bのほうが大きい場合
        <gcd #x #a #c> // 末尾再帰
        ;
?<gcd #x 511639100 258028360>;
result --
(<gcd 20 511639100 258028360>)
-- true
```

11. フィボナッチ数

フィボナッチ数を求めるプログラムをつくります。

```
<fib 答 入力数>
```

入力数に対応するフィボナッチ数を答に設定します。

フィボナッチ数については、書籍やWWW上で参照してください。

このプログラムの特徴は、「通常の計算処理」を行った結果をキャッシュのようにプログラムに追加していくことです。

計算するほどキャッシュにたまる結果が増えて、より大きな数の計算を高速化することができます。

```
<fib 0 0>; // 0の場合
<fib 1 1>; // 1 の場合
<fib #result #n> // 通常の計算処理
        <#n1=#n-1>
        <#n2=#n-2>
        <#result = <fib #nn1 #n1>+<fib #nn2 #n2>> // 再帰関数呼び出し
        ::sys <setarray fib #result #n>             // 結果をキャッシュに書き込む
        ;
?<fib #x 30>;
result --
(<fib 832040 30>)
-- true
```

12. quick sort

quick sort　アルゴリズムを使いリストの中の要素をソートするプログラムを作ります。

```
<append #X () #X>;
<append (#A : #Z) (#A : #X) #Y>
        <append #Z #X #Y>;
<qsort () ()>; // ()をソートした結果は()
<qsort #sortedlist (#x : #list) >
        <qsplit #x #list #l1 #l2> // #list を #xより小さいか大きいかで
                                  // #l1,#l2に分ける
        <qsort #s1 #l1 > // #l1をソート
        <qsort #s2 #l2 > // #l2をソート
        <append #sortedlist #s1 (#x : #s2) >   // 結果を結合する
        ;
```

```
<qsplit _ () () ()>;

<qsplit #x (#y : #list) (#y : #l1) #l2> // #y が#xより小さければ#l1に入れる

        ::sys <compare #y <= #x>
        <qsplit #x #list #l1 #l2>; // 末尾再帰

<qsplit #x (#y : #list) #l1 (#y : #l2)> // #yが#xより大きいので#l2に入れる

        <qsplit #x #list #l1 #l2>; // 末尾再帰

?<qsort #s (11 12 3 7 1 2 0 -1 10 9 4 8 5 6) >;

result --

(<qsort (-1 0 1 2 3 4 5 6 7 8 9 10 11 12) (11 12 3 7 1 2 0 -1 10 9 4 8 5 6) >)

-- true
```