デカルト Descartes 言語の説明

1　はじめに

> 我思う、ゆえに我あり
>
> Cogito ergo sum
>
> ルネ・デカルト

デカルト言語は、論理的な推論を中心とした人工知能プログラムのための I/F として設計しました。しかし、文章やデータを処理する、柔軟なスクリプトとしても役立つことでしょう。

名前は有名な数学者・哲学者であるデカルトに因んで名づけています。

デカルト言語は、誰でも簡単に使える言語は目指していません。この言語の利用者が深く理解し習熟し工夫を重ねることにより、素晴らしい結果が得られるような言語にしたいと考えています。

2　デカルト言語とは

デカルト言語は、論理推論をベースとしたロジック言語です。Prolog 言語のように論理的な関係を記述し、それを基に結果を推論することによってプログラムを実行します。

それに加えて、プログラミングの表現力を向上させるために、関数型および手続き的なプログラミングパラダイムを導入しました。また、BNF 記法に準拠した構文解析機能を文法の基本的な要素として導入しています。

決定的なルーチン処理は関数型や手続き型のプロシージャとして記述し、知識集約的な推論が必要な処理には論理型として記述します。

さらに、デカルト言語は、オブジェクト指向の機構を持ちます。このオブジェクト指向機構は、論理的な関係の構造を柔軟に表現し、また、共通に使用するモジュールをライブラリ化するのにも利用しています。

3　実行方法

デカルト言語に引数を指定して起動します。
引数はプログラムを記述したファイルを指定します。

descartes プログラムファイル

引数に指定されたプログラムファイルが読み込まれ実行されます。

たとえば、ファイル hello に、以下のような記述をして実行すると"hello, world"メッセージが出力されます。

```
? <print "hello, world">;
```

実行結果は以下のようになります。

```
$ ./descartes hello
hello, world

result --
(<print hello, world>)
-- true
```

2 行目がメッセージの出力で、最後の行の"true"は実行が成功したことを表します。

注) デカルト言語には、対話的にプログラムを実行する方法もありますが、本ドキュメントでは説明しません。

4　データ型

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。

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

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。

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

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。

5　プログラムの記述

5.1　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。

5.2　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。

関数述語は、func, f, let, letf, rpn, rpnf, compare, comparef, writenl, print の引数の述語に適用されます。

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。

5.3　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

5.4　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>
         ;
```

5.5　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。

5.6　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。

5.7　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。

5.8　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 / 〉;
```

5.9　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))〉;

5.10　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。

5.11　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　 7　 10>;
? <setArray　cell　(10 20)　100>;

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

5.12　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。

5.13　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。

5.14　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)>;
```

5.15　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。

5.16　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 --
(<obj bird <swim>>)
-- unknown知らない

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

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

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

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

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

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

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

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

?::hawk <swim>;　　鷹は泳ぐか？

result --

(<obj hawk <swim>>)

-- unknown知らない。

5.17　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。

5.18　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。

5.19　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。

5.20　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”>;
```

5.21 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。

5.22 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。

5.23　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>;
```

5.24　catch, throw 述語: イベントハンドラ処理

〈catch 変数 述語...〉

引数の述語の実行中に、throw 述語が実行されるとそれ以降の処理を中断して、この catch 述語の全体の処理が成功して終了する。

そのとき throw 述語の引数が catch 述語の変数に設定される。

throwが実行されずに引数の述語がすべて実行されると、catch述語の変数には"()"が設定される。

〈throw 値〉

引数の述語の実行中に、throw 述語が実行されるとそれ以降の処理を中断して、この catch 述語の全体の処理が成功して終了する。

そのとき throw 述語の引数が catch 述語の変数に設定される。

throwが実行されずに引数の述語がすべて実行されると、catch述語の変数には"()"が設定される。

5.25　マルチコア・マルチ CPU・マルチプロセス対応

5.25.1. 並列プロセスモデル

デカルト言語の並列機能では、並列プロセスを同時に多数実行し、結果を並列プロセスを発行した元プロセスに返す処理を実行します。

この並列機能で生成される並列プロセスのモデルは、共有メモリを持たない、個々のプロセスが完全に独立したものです。

デカルト言語では、個々のプロセスが独立しているため、並列処理を実行中のプロセス間の排他処理を行う必要がありません。プロセスは効率よく並列に処理されていきます。

現状ではマルチコアの SMP システムしかサポートしません。

しかし、このプロセスモデルだと、PC クラスタのようなクラスタシステムにも拡張するのは容易に思えます。将来の検討課題ですね。

5.25.2. マルチコア対応の並列機能の構成

さて、デカルト言語のマルチコア対応の並列機能は、以下の4つの述語で実装されます。

1) newproc： 並列プロセス数を指定して並列実行

2) eachproc： 引数のリストの要素に対応して並列実行

3) firstnewproc： 並列プロセス数を指定して、最初に true で終了した並列プロセスの結果を回収

4) firsteachproc： 引数のリストの要素に対応して、最初に true で終了した並列プロセスの結果を回収

上記の 1), 2)の機能により、N コアのシステムであれば、N 個のプロセスを同時に並列に実行することができ、最大で N 倍の実行性能を得ることが可能となります。

また、3), 4)の機能では、大規模な並列処理による探索でも、最も最初に探索を完了した解を回収できるため、Nコアのシステム上でN倍以上(数十倍以上)で実行することさえ可能となるのです。

使い方等詳細は以下に示します。

5.25.3. newproc

newproc 述語は、並列プロセス数を指定して並列実行する機能です。指定された数のプロセスが並列に生成されます。 newproc で生成された全プロセスが終了すると、呼び出した newproc 述語に制御が戻り、全プロセスの結果を回収します。

newproc 述語は以下のような構文です。

〈newproc（プロセス番号変数 プロセス数）並列実行する述語 …〉

〈newproc（プロセス番号変数 プロセス番号初期値 プロセス番号終り値）並列

実行する述語 ...>

〈newproc 結果変数 (プロセス番号変数 プロセス数) 並列実行する述語 ...〉
〈newproc 結果変数 (プロセス番号変数 プロセス番号初期値 プロセス番号終り値) 並列実行する述語 ...〉

プロセス番号変数には、プロセスごとに異なるプロセス番号が設定されます。プロセス番号は、「並列実行する述語」でプロセスごとの処理を切り分けるのに使えます。
結果変数が指定されている場合には、「並列実行する述語」の先頭の述語の第一引数を結果として返します。返すことのできる値は、アトムのみです。それ以外のリスト等の値が返された場合には、返り値は()として設定されることになります。ちょっと不便なのですが、マルチコアのプロセス間では、文字列以外のリストなどのデータ構造が共有できないので、現状はどうしようもありません。
各プロセスから返された結果は、結果変数にプロセス番号順にリストとして設定されます。
結果変数が指定されていない場合には、プロセスの処理結果を回収しません。ただし、その場合でも全プロセスの終了を newproc 述語は待つことになります。
以下にフィボナッチ数を計算する例を示します。

```
〈fib 0 0〉;
〈fib 1 1〉;
〈fib #result #n〉
        〈#n1=#n-1〉
        〈#n2=#n-2〉
        〈#result = 〈fib _ #n1〉+〈fib _ #n2〉〉;

?〈newproc (#i 17) 〈fib #r #i〉 ::sys 〈writenl #i #r〉〉;
?〈newproc #result (#i 17) 〈fib #r #i〉 ::sys 〈writenl #i #r〉〉;
```

フィボナッチ数を 0 から 16 まで、並列に実行します。最後の2行が newproc 述語の実行です。
最後から2行目は、結果の回収を行わない、結果変数を指定しない記述です。
最後の行は、結果の回収を行い、#result に結果のリストが格納されます。
実行結果は以下です。

```
$ descartes newproc.car
```

```
0 0
1 1
2 1
3 2
4 3
5 5
6 8
7 13
8 21
9 34
10 55
11 89
12 144
13 233
14 377
15 610
16 987
result --
        <newproc (Undef6 17) <fib Undef7 Undef6> ::sys <writenl Undef6 Undef7>>
-- true
0 0
1 1
2 1
3 2
4 3
5 5
6 8
7 13
8 21
9 34
10 55
11 89
12 144
13 233
14 377
15 610
```

16 987
result --
〈newproc (0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987) (Undef24987 17)
〈
fib Undef24988 Undef24987〉 ::sys 〈writenl Undef24987 Undef24988〉〉
-- true

2つ目の実行では、result として、結果のリストが格納されているのがわかるでしょうか？
なお、実行性能は、ほぼコア数に比例して向上していました。

5.25.4. eachproc

eachproc 述語は、並列プロセス数を指定して並列実行する機能です。指定されたリストの要素のプロセスが並列に生成されます。 eachproc で生成された全プロセスが終了すると、呼び出した eachproc 述語に制御が戻り、全プロセスの結果を回収します。
eachproc 述語は以下のような構文です。

〈eachproc (要素変数 要素リスト) 並列実行する述語 ...〉

〈eachproc 結果変数 (要素変数 要素リスト) 並列実行する述語 ...〉

要素変数には、要素リストの中の要素ごとに異なる値が各プロセスに設定されます。設定される要素は、「並列実行する述語」でプロセスごとの処理を切り分けるのに使えます。
結果変数が指定されている場合には、「並列実行する述語」の先頭の述語の第一引数を結果として返します。各プロセスから返された結果は、結果変数にプロセス順にリストとして設定されます。
結果変数が指定されていない場合には、プロセスの処理結果を回収しません。ただし、その場合でも全プロセスの終了を eachproc 述語は待つことになります。
以下にフィボナッチ数を計算する例を示します。

〈fib 0 0〉;
〈fib 1 1〉;
〈fib #result #n〉
〈#n1=#n-1〉
〈#n2=#n-2〉
〈#result = 〈fib _ #n1〉+〈fib _ #n2〉〉;

```
?<eachproc (#i (7 8 12 14 11 10 6)) <fib #r #i> ::sys <writenl #i #r>>;
?<eachproc #result (#i (7 8 12 14 11 10 6)) <fib #r #i> ::sys <writenl #i #r>>;
```

フィボナッチ数を 7, 8, 12, 14, 11, 10, 6 のそれぞれに対して、並列に実行します。最後の2行が eachproc 述語の実行です。
最後から2行目は、結果の回収を行わない、結果変数を指定しない記述です。
最後の行は、結果の回収を行い、#result に結果のリストが格納されます。
実行結果は以下です。

```
$ descartes eachproc.car
7 13
8 21
12 144
11 89
10 55
6 8
14 377
result --
        <eachproc (Undef6 (7 8 12 14 11 10 6)) <fib Undef7 Undef6> ::sys <writenl
Undef6 Undef7>>
-- true
7 13
8 21
11 89
12 144
10 55
6 8
14 377
result --
        <eachproc (13 21 144 377 89 55 8) (Undef9 (7 8 12 14 11 10 6)) <fib Undef10
Undef9> ::sys <writenl Undef9 Undef10>>
-- true
```

並列に実行されているため、実行順序が順不同になっていますね。

5.25.5. firstnewproc

firstnewproc 述語は、並列プロセス数を指定して並列実行する機能です。指定された数

のプロセスが並列に生成されます。 firstnewproc で生成されたプロセスのうち一つが終了すると、その結果を回収し、呼び出した firstnewproc 述語に制御が戻ります。最初に終了した以外のプロセスの実行はすべて中断させます。
つまり、最も最初に結果が得られたプロセス以外は中断されて結果は捨てられ、最初に得られた結果だけが返されます。
firstnewproc 述語は以下のような構文です。

〈firstnewproc (プロセス番号変数 プロセス数) 並列実行する述語 …〉
〈firstnewproc (プロセス番号変数 プロセス番号初期値 プロセス番号終り値) 並列実行する述語 …〉

〈firstnewproc 結果変数 (プロセス番号変数 プロセス数) 並列実行する述語 …〉
〈firstnewproc 結果変数 (プロセス番号変数 プロセス番号初期値 プロセス番号終り値) 並列実行する述語 …〉

プロセス番号変数には、プロセスごとに異なるプロセス番号が設定されます。プロセス番号は、「並列実行する述語」でプロセスごとの処理を切り分けるのに使えます。
結果変数が指定されている場合には、「並列実行する述語」の先頭の述語の第一引数を結果として返します。返すことのできる値は、アトムのみです。それ以外のリスト等の値が返された場合には、返り値は()として設定されることになります。
各プロセスから返された結果は、結果変数にプロセス番号順にアトムとして設定されます。newproc 述語とは異なり、リストではないことにご注意ください。
結果変数が指定されていない場合には、プロセスの処理結果を回収しません。

5.25.6. firsteachproc

firsteachproc 述語は、並列プロセス数を指定して並列実行する機能です。指定されたリストの要素のプロセスが並列に生成されます。 Firsteachproc で生成されたプロセスのうち一つが終了すると、その結果を回収し、呼び出した firsteachproc 述語に制御が戻ります。最初に終了した以外のプロセスの実行はすべて中断させます。
つまり、最も最初に結果が得られたプロセス以外は中断されて結果は捨てられ、最初に得られた結果だけが返されます。
Firsteachproc 述語は以下のような構文です。

〈firsteachproc (要素変数 要素リスト) 並列実行する述語 …〉

〈firsteachproc 結果変数 (要素変数 要素リスト) 並列実行する述語 …〉

要素変数には、要素リストの中の要素ごとに異なる値が各プロセスに設定されます。設定される要素は、「並列実行する述語」でプロセスごとの処理を切り分けるのに使えます。

結果変数が指定されている場合には、「並列実行する述語」の先頭の述語の第一引数を結果として返します。返すことのできる値は、アトムのみです。それ以外のリスト等の値が返された場合には、返り値は()として設定されることになります。

各プロセスから返された結果は、結果変数にプロセス番号順にアトムとして設定されます。Eachproc 述語とは異なり、リストではないことにご注意ください。

結果変数が指定されていない場合には、プロセスの処理結果を回収しません。

5.25.7. デカルト言語の並列処理プログラミングのコツ

便利な nop 述語

nop 述語は、引数に何が書かれていても何もしない述語です。文法的にシンタックスエラーになるような場合を除き、引数に文字列、リスト、変数、述語など何が書かれていても、何もしません。

常に実行は true で成功します。

```
〈nop 〜〉
```

このような何もしない述語が、デカルト言語で並列処理をするのにとても便利なのです。どのように使うかというと、上記までの項で説明したマルチコア機能の結果変数に値を返すために使うと便利なのです。

例えば newproc 述語で、並列に述語を実行し結果を集める場合、最初に実行された述語の第一引数の値が返されます。しかし、都合よく最初の述語の第一引数に並列処理を行った結果を設定するのは実際には面倒なことが多いのです。

```
〈newproc #結果変数 (#i 1 10) 〈述語1 #変数 1〉 〈述語 2 #変数 2〉〉
```

上記では、#変数 1 の値が結果として、結果変数に設定されますが、実際には、述語 1 を実行し述語 2 を実行した後に#変数 2 の値を返したい場合には困ってしまいます。

このような場合には、nop 述語を使えば簡単に解決できるでしょう。

```
〈newproc #結果変数 (#i 1 10) 〈nop #変数 2〉 〈述語1 #変数 1〉 〈述語 2 #変数 2〉〉
```

〈nop #変数 2〉を最初の述語とすることで、#変数 2 の値が結果として帰ります。 nop 述語は増やしても何も実行しないのでオーバーヘッドはありません。 #変数 2 には、述語 1 と

述語 2 が実行された後の値が設定されます。
このように、nop 述語を使うと並列プロセスの返り値を任意の値に自在に設定することができるようになります。
for, foreach 述語との関係¶
for 述語は、引数の述語を指定回数繰り返し実行します。構文を見てみましょう。

〈for (変数 実行回数) 述語…〉
〈for (変数 初期値 最終値) 述語…〉
〈for 結果変数 (変数 実行回数) 述語…〉
〈for 結果変数 (変数 初期値 最終値) 述語…〉

newproc 述語の構文とそっくりです。

〈newproc (プロセス番号変数 プロセス数) 並列実行する述語 …〉
〈newproc (プロセス番号変数 プロセス番号初期値 プロセス番号終り値) 述語 …〉
〈newproc 結果変数 (プロセス番号変数 プロセス数) 並列実行する述語 …〉
〈newproc 結果変数 (プロセス番号変数 プロセス番号初期値 プロセス番号終り値) 述語 …〉

実は、for 述語と newproc 述語は、役割もよく似ています。
どちらも、引数の述語を繰り返し実行します。異なるのは、for 述語は、シングルコアで指定された順番に実行されることです。 newproc 述語は、指定された述語はすべて実行されますが、順不同に並列に実行されます。つまり、最も大きな違いは、並列にマルチコアで実行するかしないかが大きな違いであり、論理的には同じ処理を実行するのです。
実行する順番が重要な場合や、マルチコア実行ではオーバーヘッドが大きくなりすぎる場合には、for 述語を用います。
多数のマルチコアを持つシステムでパフォーマンスを求める処理には、newproc 述語を用いたほうがよいです。しかし、newproc 述語では、プロセスを生成しなければならないため、そのオーバーヘッドがあり、あまりに簡単な処理を引数の述語の処理とすると効率が悪いことがあるでしょう。
どちらを使うのが良いかは、プログラマが十分に条件や環境を熟慮して決めるべきでしょう。
foreach 述語と eachproc 述語も同様の関係です。
構文を見てみましょう。

〈eachproc (要素変数 要素リスト) 述語 …〉

〈eachproc 結果変数（要素変数 要素リスト）述語 …〉

〈foreach（変数 リスト）述語…〉
〈foreach 結果変数（変数 リスト）述語…〉

対応していることが分かると思います。 foreach 述語はシリアルに順に実行され、eachproc 述語は並列に順不同に実行されることが違いです。
同様に、firstnewproc 述語には firstfor 述語が対応し、firsteachproc 述語には firstforeach 述語が対応します。

〈firstnewproc（プロセス番号変数 プロセス数）並列実行する述語 …〉
〈firstnewproc（プロセス番号変数 プロセス番号初期値 プロセス番号終り値）述語 …〉
〈firstnewproc 結果変数（プロセス番号変数 プロセス数）並列実行する述語 …〉
〈firstnewproc 結果変数（プロセス番号変数 プロセス番号初期値 プロセス番号終り値）述語 …〉

〈firstfor（変数 実行回数）述語…〉
〈firstfor（変数 初期値 最終値）述語…〉
〈firstfor 結果変数（変数 実行回数）述語…〉
〈firstfor 結果変数（変数 初期値 最終値）述語…〉

〈firsteachproc（要素変数 要素リスト）並列実行する述語 …〉
〈firsteachproc 結果変数（要素変数 要素リスト）並列実行する述語 …〉

〈firstforeach（変数 リスト）述語…〉
〈firstforeach 結果変数（変数 リスト）述語…〉

5.26　GUI

未稿

5.27　RDB データベースの操作

未稿

6　プログラム例

6.1　リストの追加処理

リストの追加処理を行うプログラムを作ります。

<append 答 リスト1 リスト 2>の形式でリスト1にリスト2を追加した結果を答えに設定します。

```
<append #X () #X>;                    // ()に#x を追加すると#x になる
<append (#A : #Z)  (#A : #X) #Y>      // リスト1から#A を除いたリストの追加処理
        <append #Z #X #Y>;

? <append #x (a b c) (d e f)>;        // (a b c) に (d e f) を追加すると
result --
(<append (a b c d e f) (a b c) (d e f)>) // 結果は第一引数に (a b c d e f)と設定
-- true
```

逆に答のリストを得られる入力のリスト1とリスト2を求めることもできる。複数の可能性があるので、結果をすべて調べるために findall 述語を使う。

```
? <findall <append (a b c) #x #y> ::sys<writenl #x #y>>;
() (a b c)
(a) (b c)
(a b) (c)
(a b c) ()
result --
(<findall <append (a b c) Undef48 Undef49> <obj sys <writenl Undef48 Undef49>>>)

-- true
```

6.2　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>
        <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
```

6.3　フィボナッチ数

フィボナッチ数を求めるプログラムです。

〈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
```

6.4　htmlの合成

htmlファイルを合成するプログラムです。
テンプレートのような元のファイルを用意し、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>)
```

6.5 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
        <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
```