Explanation of the Descartes language

1 Introduction

> Cogito ergo sum
>
> Rene Descartes

The Descartes language was designed as I/F for the artificial intelligence program. However, it will serve also as a flexible script which processes texts and datas. The name was connected with Descartes who is famous mathematician and philosopher.

The language with which anyone can use the Descartes language easily is not aimed at. When the user of this language understands deeply, you make it a language with which a wonderful result is obtained.

2 Descartes Language

The Descartes language is a logic programming language. A logical relation is described like a Prolog language and a program is executed by reasoning a result based on it. In order to raise the power of expression of programming in addition to it, the function type and procedural programming paradigm was introduced.

Moreover, the syntax-analysis function based on the method of BNF is introduced as a fundamental element of grammar.

Decisive routine processing is described as a function type or a procedural procedure, and knowledge-intensive reasoning describes it as a logic type for required processing.

Furthermore, the Descartes language has an object-oriented mechanism. This object-oriented mechanism is used also for library.

3 The method for performing

- An argument is specified and started in the Descartes language. An argument specifies the file which described the program.

```
descartes PROGRAM-FILE
```

The program file specified as the argument is read and performed.

For example, if it performs by carrying out the following description to File hello, the "hello,world" message will be outputted.

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

An execution result is as follows.

```
$ ./descartes hello
hello, world

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

The 2nd line is an output of a message.
"true" of the last line means that execution was successful.

*) Although there is also a method of executing a program interactively in a notes Descartes language, this document does not explain.

4 Data type

Character string

```
hello    “Hello, world!!”    ‘1 2 3’
```

A character string is a sequence of the symbol with which the character was located in a line.

It is the character string which was bundled with " or ' .

When a blank, a tab, a new-line, etc. are included in a character string, you have to bundle with " or ' .

Numerical value

```
100   0.8123   0xa1
```

The integer of 8-byte accuracy and the floating point number of 8-byte accuracy can be used.

A hexadecimal number is expressed with the character which begins from 0x.

Variable

```
#x   #a   #abc
_ A variable without a name
```

A variable is expressed with the sequence of the symbol which starts in #.

Moreover, it is an variable without a name. _ can be used.

List

```
(abc  “this is a pen”  #xy  1 2 3  ( list1 #z))
```

Some character string, some number, some variable, some list, and some function predicate are bundled with ().

(The function predicate is mentioned later.)

List has the same structure as List of a Lisp language.

Empty List is ().

The dot pair is ": " .

Function predicate

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

A character string, a number, a variable, List, and a function predicate are bundled with < and >.

Evaluation results are true, false, and unknown.

When it succeeds as a result of evaluation, it is considered for convenience that the first argument is a return value of a function.

5 Description of Program

5.1 Predicate

As a performed result, a predicate takes three values of true, false, and unknown.

As a result of performing the predicate in the Descartes language, when it is set to true, unification with the program registered is performed and a value is set to the variable contained in a predicate.

In false, simplification of a predicate goes wrong and is interrupted. In unknown, other possibilities are tried and backtracking is performed if needed.

5.2 Function Predicate

In a predicate, when it succeeds as a result of evaluation, it is considered for convenience that the first argument is the return value of a function. Such a predicate is called a function predicate. When a function predicate is in an argument and the function predicate is true, the function predicate itself is replaced with the value of the first argument.

It is set to false and, in unknown, the function predicate which called in false, and the function predicate called similarly serve as unknown.

A function predicate is applied to the predicate of the argument of func, f, let, letf, rpn, rpnf, compare, comparef, writenl, and print.

In order to enable it to use in a function predicate, when writing a predicate, if it writes that a result returns to the first argument, it is convenient and convenient.

5.3 Notes Comment

There are the following three kinds of comment.

- Range surrounded by /* */
- from # to the end of the sentence
- from //to the end of the sentence

5.4 Description Rule of Predicate

Enumeration of a predicate describes a program. The last ; (semicolon) It divides. The first predicate calls it a head and the other predicate calls it a body. What combined the head and the body is called a clause.

<predicate> <predicate> <predicate> … <predicate> ;
---------- ---------- clause
head body

Not only a predicate but a list can also be described on a body.

```
<predicate> (<predicate> <predicate> ( <predicate> )
(<predicate> < predicate>)) ;
```

It will become easy to read if a tab divides and describes a body as follows.

```
<predicate>
        <predicate>
        <predicate>
        <predicate>
        ;
```

5.5 Unification

Unification is the operation which makes two predicates the same form (make equal the clause which corresponds altogether from a predicate name to the value of an argument).

```
<pred1  #x   #y  123  (a b c)>
        corresponding clause is coincided.
<pred1  xyz  #z  123  #a>

Result of unification; #x=xyz #y=#z #a= (a b c) It becomes.
```

5.6 Call of Program

? is attached to the head of a predicate in order to call the described program.

```
? <predicate>;
```

The called predicate is compared with the header and order in a program, and tries unification. If it succeeds in unification, the program described by the body below will be called sequentially from the left.

5.7 Debugging Function

If the <tron> predicate is performed, a tracing facility will be set to ON and trace information will be outputted at the time of execution.

Please perform the <troff> predicate, when you turn OFF a tracing facility.

5.8 Calculation of Expression, Reverse Poland Style Calculation

let, letf predicate

```
<let expression>
<letf expression>
```

- Expression is calculated.
- A calculation result is substituted when the left side is a variable.
- When the left side is a numerical value, it is judged whether it is equal to a calculation result.
- let calculates an integer and letf calculates a floating point number.

A function predicate can also be included in the expression of the right-hand side.

```
? <let #x = 1 + 2 +  7 *  5  >;
? <letf #x = ::sys <cos _ 1> + ::sys <sin _ 1>>;
```

The operation of the expression which omitted let is treated as integer operation.

Therefore, both following examples bring the same result.

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

rpn, rpnf predecate

```
<rpn VAR RPN-EXPRESSION>
<rpnf VAR RPN-EXPRESSION>
```

- A reverse Poland style is calculated and a result is set as a variable.
- rpn calculates an integer and rpnf calculates a floating point number.

A function predicate can also be included in a reverse Poland style.

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

5.9 Comparison of Expression

compare, comparef predicate

〈compare Expression Comparison−operator Expression 〉
〈comparef Expression Comparison−operator Expression〉

- Expression is compared.
- compare is compared as an integer and comparef is compared as a floating point number.

The following can be used for a comparison operator.

=, ==	equal
!=, <>	not equal
>	large
>=	large or equal
<	small
<=	small or equal

and, or, not operator can be used for a comparison type.

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

5.10 Global Variable

variable in the Descartes language is an effective local variable only in a clause.

It sets up as a global variable to save data over a paragraph. The global variable in the Descartes language is mounted by defining the group of a variable identifier and a value as a new clause.

The setvar predicate of a sys module is used for a setup of a global variable.

Reference of a value is <variable-identifier #variable>. A value will be set as #variable.

```
Substitution of the value of a global variable
? ::sys <setvar color “red”>;

Reference of the value of a global variable
? <color #cl>;
red is set to #cl.
```

When a setvar predicate is performed, the clause will be replaced if there is already a variable of the same name.

A new clause will be added if there is nothing.

Global array variable

A global array variable can be set up by the setarray predicate of a sys module.

Array defines the group of a value as a variable identifier and an index as new clause.

A number, a character string, a list, a predicate, etc. can be specified as an index anything. Probably, in the case of a multidimensional array, it is good to set a list as an index.

```
set global array valiables.
? ::sys<setvar  ary   7   10>;
? ::sys <setvar  cell  (10 20)  100>;

refference of global array valiables
?<ary 7 #v>;
10 is set as #v.
? <cell  (10 20)  #val>;
100 is set as #val.
```

5.11 FILE I/O

The file of an input or an output is changed only while performing the predicate of an argument.

<openr FILE PREDICATE...>

Open file for reading. and a predicate is performed.

<openw FILE PREDICATE...>

Open file for writing. and a predicate is performed.

<openwp FILE PREDICATE...>

Open file for adding a postscript . and a predicate is performed.

5.12 Library module

Although there are the following three kinds, each expresses the same meaning with how to call a library module.

```
:: library-module predicate
example)  ::sys <writenl hello>

<unify library-module predicate>
example)  <unify sys <writenl hello>>

<obj library-module predicate>
example)  <obj sys <writenl hello>>
```

Usually, probably, the description using :: will be convenient.

In order to use an external library module, it is necessary to include a library module.

```
? <include  library-module>

example)
?<include list>;

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

However, since it is the library module included in the system, only a "sys" module can be used even if it does not include.

5.13 How to Make Library Module

A library module makes a file name from a library module name.

The program described to inside describes the program of the usual Descartes language.

By including, it can be used as a library module.

Example) example module

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

call append

```
? <include example>;

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

Execution result

```
$ ./descartes append
result --
(<include example>)
-- true

result --
(<obj example <append (abc def ghi jkl) (abc def) (ghi jkl)>>)
-- true
```

5.14 Object-orientation

The object in the Descartes language is realized by regarding a library module as an object. That is, a library module name is treated as an object name.

The definition of an object is performed as follows.

```
::<Object-name
    Define-method;
    Define-method;

    inherit Inherit-object;
>;
```

In a method, the clause which is a program of the usual Descartes language and which combined the head and the body can be described.

5.15 Example of Object-oriented Program

The object of a bird, a penguin, and a hawk is defined as an example.

```
// Object of a bird:  flies and walks.
::<bird
      <fly>;
      <walk>;
>;

// Object of a penguin:  does not fly, it swims and walks.
::<penguin
      <fly>        <false>;     // cannot fly.
      <swim>;             // It adds newly swimming.
      inherit bird;       // bird is inherited.
>;

// A hawk is the same as a bird, flies, and walks.
::<hawk
      inherit bird;       // bird is inherited
>;
```

Now, a question is asked to the object of a bird, a penguin, and a hawk. Although a question can be asked by the call of the method to an object, the call of a method is the same as how to call a library module.

```
?::bird <swim>;     // Does it swim in a bird?
result --
(<obj bird <swim>>)
-- unknown

?::penguin <swim>;  // Does it swim in a penguin?
result --
(<obj penguin <swim>>)
-- true
```

```
?::bird <walk>;
result --
(<obj bird <walk>>)
-- true

?::penguin <walk>;
result --
(<obj penguin <walk>>)
-- true

?::bird <fly>;
result --
(<obj bird <fly>>)
-- true

?::penguin <fly>;
result --
(<obj penguin <fly>>)
-- false

?::penguin <run>;
result --
(<obj penguin <run>>)
unknown

?::hawk <fly>;
result --
(<obj hawk <fly>>)
-- true

?::hawk <walk>;
result --
(<obj hawk <walk>>)
-- true
```

```
?::hawk <swim>;
result --
(<obj hawk <swim>>)
-- unknown
```

5.16 Syntax Analysis

It is like the syntax by EBNF (extended Backus Naur form) being the following.

```
expr         =  expradd
expradd      =  exprmul { "+"  exprmul  |  "-"  exprmul }
exprmul      =  exprID  { "*"  exprID  |  "/"  exprID }
exprID       =  "+" exprterm | "-"  exprterm  |  exprterm
exprterm     =  "("  expr  ")" | Number-sequence
   abcz            =  abc [Alphabet  ]
```

The syntax of EBNF (extended Backus Naur form) is convertible with the Descartes language corresponding to 1 to 1. (The Descartes language receives the grammar of LL (*).)

Syntax by the Descartes language

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

```
[～]  :  An abbreviation is possible.
{～}  :  A repetition of 0 times or more.
  |   :  or Selection
<>  Character string which is not bundled : Terminal
```

Many predicates which can be used as a token besides FNUM of a number sequence are defined as the sys module.

An input file is specified by file I/O explained in the preceding chapter. The function of syntax analysis cannot be used from notes standard input. Please use it in the input from a getline predicate or a file.

- The predicate for tokens of syntax analysis

<TOKEN VAR PRED...>
After syntax-analysis PRED execution of an input,
obtained token is set as VAR.

<SKIPSPACE>
The space of an input is skipped.

<C [VAR]>
An input is set as one-character VAR.

<N [VAR]>
When an input is a number, it is set as VAR.
unknown is returned when different.

<A [VAR]>
When an input is the ASCII character, it is set as VAR.
unknown is returned when different.

<AN [VAR]>
When an input is the ASCII character or a number,
it is set as VAR. unknown is returned when different.

<^>
The head of a line is matched.

<$>
The last of a line is matched.

<* VAR>
Arbitrary character strings are matched.

<CR [VAR]>
When an input is CR new-line, it is set as VAR.

unknown is returned when different.

<CNTL [VAR]>
When an input is the CNTL character, it is set as VAR. unknown is returned when different.

<EOF [VAR]>
When an input is the EOF character, it is set as VAR. unknown is returned when different.

<SPACE>
true is returned when an input is a space. unknown is returned when different.

<PUNCT>
true is returned when it is characters other than the alphabet and a number.

<STRINGS VAR>
STRINGS of "..." or '...' is matched, and it is set as VAR.

<WORD VAR>
In STRINGS(s) other than the alphabet, a number, and "_", unknown is returned by arbitrary STRINGS.

<NUM VAR>
The integer of an input is is set as VAR.

<FNUM VAR>
The floating point number of an input is set as VAR.

<ID VAR>
If an input STRINGS (a head is the alphabet and a number is also good except it) and it agrees, it

will be set as VAR.

<RANGE VAR CHAR1 CHAR2>

<NONRANGE VAR CHAR1 CHAR2>

It will be set to true if contained in the range of a character 1 and a character 2.

<GETTOKEN VAR>

The token which is a result of the last syntax analysis is set as VAR.

5.17 How to Use TOKEN Predicate

The token of arbitrary character strings is compoundable by using a TOKEN predicate and compounding syntax analysis of an argument. (It becomes substitution of a regular expression.)

The character string whose head is an alphanumeric character from the 2nd character with an alphabetic character

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

Number sequence (it is equivalent to ::sys<;NUM #token>)

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

The character string of a capital letter or a number

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

“DISK”+ A triple figures number

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

A TOKEN predicate is used for lexical analysis like a character string, and a TOKEN predicate is not used for the analysis of syntax like the sentence which is a sequence of a character string.

5.18 timeout predicate

A timeout predicate closes processing of the predicate which is not ended within the appointed time, and returns it by unknown. The appointed time is specified by a micro second bit.

```
<timeout Appointed-time  Predicate...>
```

Processing which is likely to require execution time, and processing which may lapse into an infinite loop are performed for the time being, and in not finishing within the appointed time, it uses for the use which tries other methods.

```
< Processing >   <timeout 1000000 < ProcessingA>>;
< Processing >   <timeout 1000000 < ProcessingB>>;
< Processing >   ::sys<writenl “False”>;
```

5.19 findall predicate

findall A predicate is used to calculate all the solutions of the predicate of an argument. Usually, execution of a predicate is ended by the solution found first.

However, even when it turns out that there are other solutions, if it remains as it is, it cannot ask. For example, in the case of a problem like search of a course, it may not restrict that the course acquired first is the optimal, but it may need to evaluate whether it is the optimal to all the courses.

```
<findall predicate...>
```

findall Execution of a predicate may lapse into an infinite loop and processing may not finish it eternally. In order to close processing in a moderate place, it will be convenient if you use combining a timeout predicate.

5.20 for Loop and Foreach Loop, Map Predicate

It is loop processing of procedure processing.

〈for (VAR TIMES) PRED 述語...〉
〈for (VAR INIT-VAL LAST-VAL) PRED...〉

The predicate of the specified number of times and an argument is performed. A number is set to a variable in order. The value from 0 makes it increase every [1] from the value to the value of the number of times of execution, or the last value, when the number of times of execution is specified, when an initial value is specified. After execution of 1 turn, bind of all the variables is cleared and a predicate is performed from the beginning.

<foreach (VAR LIST) PRED...>
<map (VAR LIST) PRED...>

The predicate of an argument is performed for every element of List.
List value is set to a variable in order.
Bind of the variable under execution after execution of 1 turn is cleared, and a predicate is performed from the beginning.

5.21 Character Code

A character code uses UTF8 by a default.

When you specify EUC or SJIS, please specify in a code predicate.

```
UTF8 specification
  ?<code UTF8>;
EUC specification
  ? <code EUC>;
SJIS specification
  ?<code SJIS>;
```

6 Example of Program

6.1 Additional Processing of List

The program which performs additional processing of List is made.
<append Result List1 List2>
The result of having added List2 to List1 in the form is set as an answer.

```
<append #X () #X>;          // () It will be set to #x if #x is added.
<append (#A : #Z) (#A : #X) #Y> //   Additional   processing   is performed using List except #A from List1.
       <append #Z #X #Y>;

? <append #x (a b c) (d e f)>;   // (a b c) (d e f) It adds.
result --
(<append (a b c d e f) (a b c) (d e f)>)      //  A  result  is  the first argument. (a b c d e f) Setup
-- true
```

Conversely, it can also ask for Liszt 1 and Liszt 2 of an input who can get Liszt of an answer. Since there are two or more possibilities, a findall predicate is used in order to investigate all results.

```
?  <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 Algorithm of Euclid

The program which asks for the greatest common denominator using the algorithm of Euclid is made.
The greatest common denominator of an integer 1 and an integer 2 is set as an answer in the form of <gcd answer integer1 integer2>.
Please refer to it for the algorithm of Euclid on books or WWW.

```
<gcd #x #x 0>;      // # x and the greatest common denominator of 0 are
#x.。

<gcd #x #a #b>
       ::sys <compare #a >= #b>   // When the #a is larger
       <#c = #a % #b>             // let is omitted.
       <gcd #x #b #c>
       ;

<gcd #x #a #b>
       <#c = #b % #a>             // When the #b is larger
       <gcd #x #a #c>             ;

?<gcd #x 511639100 258028360>;
result --
(<gcd 20 511639100 258028360>)
-- true
```

6.3 Number of Fibonacci

It is a program which asks for the number of Fibonacci.

<fib Result Number>

The number of Fibonacci corresponding to the number of inputs is set as an answer.

Please refer to it for the number of Fibonacci on books or WWW.

The feature of this program is adding like cash the result of having performed "the usual calculation processing" to a program.

The result which accumulates in cash can increase, so that it calculates, and a larger number of calculation can be accelerated.

```
<fib 0 0>;                  // In the case of 0
<fib 1 1>;                  // In the case of 01
<fib #result #n>            // The usual calculation processing
       <#n1=#n-1>
       <#n2=#n-2>
       <#result = <fib #nn1 #n1>+<fib #nn2 #n2>>
       ::sys <setarray fib #result #n>       // A result is written in
cash.
       ;

?<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
       ::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
```