

The outline of the Descartes language

1. Similarity with prolog

Since the Descartes language is carried out based on the first floor predicate calculus, it has a common feature of a prolog language and many. Then, the program which List connects should compare the Descartes language and a prolog language.

In the Descartes language, it is as follows.

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

In prolog, it is as follows.

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

In both of the examples, the list of the 2nd argument and the 3rd argument is connected, and a result is returned to the 1st argument.

Is a difference understanding?

- 1) A predicate is bundled with ().
- 2) A pause of an argument uses a blank.
- 3) Although period "." is finally placed in prolog, semicolon ";" is placed in the Descartes language.
- 4) A list uses not [] but ().
- 5) "|" which divides a list is ":" in the Descartes language.
- 6) Although ":" is used for a pause of a head part and a body part by prolog, there is nothing in the Descartes language.
- 7) In the Descartes language, "#" is attached to a variable.

In order to perform the above-mentioned append in the Descartes language, it performs in the predicate which attached ? as follows.

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

2. Difference with prolog

Although reference was made about the points of comparison of the Descartes language and prolog for the preceding clause, difference is described here.

Unlike prolog by which what can be written to a body part is

restricted to enumeration of a predicate, in the Descartes language, what arbitrary lists can be written for differs greatly. Execution sequence in a list is performed sequentially from the described left.

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

In the above-mentioned example, if example is performed, it will perform in order of e1, e2, e3, e4, and e5.

The advantages of this description are meta processing which summarizes a sequence of a predicate and is passed to the argument of a predicate being performed, or being able to use in order to summarize processing for the sign for syntax analysis shown below.

It is {} repetition as a sign for syntax analysis []
It is also big difference with prolog that the abbreviation possibility of, |, or selection can be used.

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

If def, which of (hij lkm), or the processing in which it succeeds is repeated and both do not correspond, it escapes from a loop, if example2 is performed in the above-mentioned example, abc will be performed, even if nop is successful and it goes wrong, it is processed, and finally end is performed.

3. comment

There are the following three kinds of comment.

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

4. Numerical computation

Integrally in a let predicate and a floating point, a letf predicate is used for calculation of a number.

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

The function predicate mentioned later can be used within expression.

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

let is omissible. The two followings are the same.

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

5. Function predicate

The value of a function which the argument of predicates, such as let, letf, f, and func, is evaluated as a function predicate, and a function predicate returns is the 1st argument. The variable of a return value is convenient if you use obscure variable “_.”

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

What can be used by let and letf is only a function which returns a numerical value.

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

f is an alias of func and carries out the completely same work. Moreover, f can take List as an argument, and when a function predicate is contained in List element, after being evaluated, it is returned as a value of a function.

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

(That it is with “::sys” above expresses the call of the library mentioned later, and it means the call of the getline predicate of a sys module.)

6. Library

The call of a library is performed in the following forms.

```
::LIBRARY-NAME <PREDICATE>  
<unify LIBRARY-NAME <PREDICATE>>  
<obj LIBRARY-NAME <PREDICATE>>
```

These three kinds of methods of calling are the same contents of operation.

7. Object-orientation

The following forms define an object.

```
:: < Object-name  
    programs or inheirt object  
>;
```

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

```
::<bird  
    <fly>;  
    <walk>;  
>;
```

```

::<penguin
    <fly>
        <!><false>;
    <swim>;
    inherit bird;
>;

::<hawk
    inherit bird;
>;

```

How to call an object is the same as how to call a library.
Please try the following.

```

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

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

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

```

7. EBNF

EBNF for syntax analysis can be used.

```

<Name> "Mike";
<Name> "Nancy";
<Name> "hniwa";

```

```

<name #x>
    "I" "am"
    <Name> <GETTOKEN #x>
    ["."]
    ;

```

```

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

```

```

result --
(<obj sys <getline I am hniwa <name hniwa>>>>)
-- true

```
