記 号 処 理 11－2
（1980. 3. 21）

The List Processing Language - PSILISP

Kenji UEDA

Department of Mathematics,
Keio University,
Yokohama, 223 Japan

PSILISP is a programming language for list processing. Programs in PSILISP are structural styled and easy to understand. In this paper the syntax and some features of PSILISP is described by using examples. It has notions of construction and recognition and so LISP's primitive function is able to be expressed by means of them. While the construction derives a new list structure, the recognition analyzes a list. PSILISP is a more natural programming language for list manipulation.

1 INTRODUCTION

The programming language LISP was conceived by John McCarthy in 1959 and has dominated for two decades. LISP does not only a list-processing capability to the artificial intelligence (AI) community, but it is a tool whose use within AI goes well beyond the data "list" and whose applicability to non-AI problems is becoming better appreciated. Under such circumstances various LISP-like languages has been developed [1,4,7] for specific requirements and has its own distinctive features.

The PSILISP programming language is also one of such languages, and its aim is the programming according to the representation of list structures. For instance, (a.d) means the following three relations or operations.

```
                car[(a.d)] = a
(a.d)  <=>      cdr[(a.d)] = d
                cons[a;d] = (a.d)
```

In PSILISP, such a structural style, i.e. (a.d), is adopted for these operations, that is, selective, recognizing and constructive operations can be specified by a structural style. Thus it is possible to program the list manipulations naturally. Constructions and recognitions are both presented as the structural styles. Construction composes a new list structure, recognition is a sort of pattern matching facility [2,5,8] including selecting operation.

2 SYNTAX

The syntax of PSILISP includes the syntax of LISP and some structural styles. The syntax is specified in TABLE I. We use the BNF notation and a convenient extension of it. The notation {<syntax-class>}* is used to indicate an arbitrary number (including zero) of occurence of the <syntax-class>. The two syntax-class', <atom> and <identifier>, whose definitions are missing, are usual.

3 LIST NOTATION

Though the data structure manipulated in PSILISP are lists as well as in LISP, the notion of a segment of a list is admitted in the representation level. A segment of a list is successive elements of the list and so regarded as a list whose outer parentheses are removed. Hence we express a segment as follows.

/<list>/

TABLE I. Syntax of PSILISP

```
<form>::=<constant>|<variable>|<conditional-expression>|<application>
        |<construction>|<recognition>|<conditional-recognition>

<constant>::=<S-expression>

<S-expression>::=<atom>|<list>|<dotted-list>

<list>::=({<element>}*)

<dotted-list>::=(<element>{<element>}*.<S-expression>)

<element>::=<S-expression>|<segment>

<segment>::=/<list>/

<variable>::=<identifier>

<conditional-expression>::=[<proposition>-><forms>{;<proposition>-><forms>}*]

<forms>::=<form>{,<form>}*

<proposition>::=<form>

<application>::=<function>[]|<function>[<form>{;<form>}*]

<function>::=<identifier>|<lambda-expression>

<lambda-expression>::=lambda[[];<form>{;<form>}*]
                     |lambda[[<variable>{;<variable>}*];<form>{;<form>}*]

<construction>::=({<construct-element>}*)
                |(<construct-element>{;<construct-element>}*.<form>)

<construct-element>::=<form>|<construct-segment>

<construct-segment>::=/<form>/

<recognition>::=<recognizer>[<form>]

<recognizer>::=<identifier>|<psi-expression>|<form>
              |<structural-recognizer>|<variable>:<recognizer>

<psi-expression>::=psi[[<variable>];<proposition>]

<structural-recognizer>::=({<rec-element>}*)
                         |(<rec-element>{<rec-element>}*.<recognizer>)

<rec-element>::=<recognizer>|<rec-segment>

<rec-segment>::=/<recognizer>/

<conditional-recognition>

             ::=[<recogniition-clause>>{;<recognition-clause>}*][<form>]

<recognition-clause>::=<recognizer>-><forms>
```

Note that <atom> or <dotted-list> is never put between "/"'s. This notation has the following properties.

(al ... ai /()/ aj ... an) = (al ... ai aj ... an)

(al ... ai /(bl ... bm)/ aj ... an) = (al ... ai bl ... bm aj ... an)

(al ... an /x/) = (al ... an . x)

Segment notatin is not necessary to express list structures, but important to describe constructions and recognitions.

4 CONSTRUCTIONS

In LISP a new constructed object is derived by function cons in principle. Thus for the complex symbolic data, complicated expressions formed with many conses and lists are required. But in PSILISP such operation is also described by a structural style called construction. For example ,

list[u;list[v;cons[w;x];list[y;z]]]

is expressed as the following form.

(u (v (w.x) (y z)))

Of course the segment notation is able to be used in construction. The followings are examples of construction.

(A x y) => (A (I J) (K.L))

(/x/ A.z) => (I J A M (N O))

(x /z/ (x)) => ((I J) M (N O) ((I J)))

(((y) /x/ z) => ((((K.L)) I J) (M (N O)))

where x = (I J), y =(K.L) and z = (M (N O)).

5 RECOGNITIONS

Recognition may be regarded as a sort of pattern matching facility, and a recognizer corresponds to a pattern in pattern matching. Recognizers examine the existence of the object by applying themselves to the object.

The value of recognition is defined as follows.

recognizer[object-data]
= if object-data is recognized then object-data
else failure

Here failure is a special value distincted from lists in PSILISP and it is used in the conditional recognition.

5.1 Recognizers

A recognizer is a kind of existance and all data in PSILISP are able to be recognizers. However, they are not recognizers until they are applied to a object. An example of recognition is a following.

A[x]

This means that value of A[x] is x if value of x is A, otherwise failure. Applications of structual-recognizers are performed by applying the leftmost innermost first to the corresponding object and if a recognition whose value failur occurs, the value of whole recognition is also failure.

Moreover in the recognition including recognizers of the following form,

<variable>:<recognizer>

the variables are to be given new bindings with each recognition, if the recognitions are successful. Those new bindings are keeped until the whole recognition is completed. The following examples will clear the recognition process.

(A B C)[(A B C)] => (A B C)

(A x C)[(A (B1.B2) C)] => if x = (B1.B2) then (A (B1.B2) C)
else failure

(A /x/ C)[(A B1 B2 C] => if x = (B1 B2) then (A B1 B2 C)
else failure

(v:A B C v)[(A B C A)] => (A B C A)

(A (w:(B C) D) w)[(A ((B C) D) (B C))] => (A ((B C) D) (B C))

A[A][A] => A[A] => A

5.2 Psi-expressions

With the purpose of powerful recognition we introduce a new recognizer. It is called psi-expression and has the following form.

psi[[<variable>];<proposition>]

This psi-expression represnts an entity which satisfies its own proposition. Therefore psi-expression can be used to recognize a object which satisfies a property. For example,

psi[[x];atom[x]]

is an entity which is an atom. Therefore this is used as a recognizer in the following manner.

psi[[y];atom[y]][ATOM] => ATOM

psi[[y];atom[y]][nil] => NIL

psi[[y];atom[y]][()] => NIL

psi[[y];atom[y]][(A B)] => failure

In the psi-expressions psi[[v];t] is significant. It represents an arbitrary object, because "t" is a constant variable whose value is always true. So we may use "$" in place of this, that is, "$" is defined as the psi-expression. Further the following represents an enttity which is an arbitrary sequence of objects, any segment.

/psi[[v];t]/ or /$/

Several recognitions using psi-expression are shown below.

(psi[[x];member[x;(A B)]] psi[[y];member[y;(X Y)]])[(B X)] => (B X)

(v:psi[[x];eq[cadr[x];D]] S v)[((A D) S (A D))] => ((A D) S (A D))

```
(A /psi[[x];eq[length[x];3]]/ B)[(A X Y Z B)] => (A X Y Z B)

(A x:$ $ x B)[(A (X) (U.V) (X) B)] => (A (X) (U.V) (X) B)

(A /$/ B)[(A B)] => (A B)

(A /$/ B)[(A (J) K L B)] => (A (J) K L B)

(A /y:$/ B y C)[(A A B C B (A B C) C)] => (A A B C B (A B C) C)
```

5.3 Conditional recognition

So far, sevaral sorts of recognitions whose value is either its object or failure are shown. The conditional recognition described below is different from such recognitions, its value depends on results of recognitions which are in the recognition. The form of it is followings.

<conditional-recognition>

::=⊥<recognition-clause>{;<recognition-clause>}*⊥[<form>]

<recognition-clause>::=<recognizer>-><forms>

A conditional recognition has the following meaning.

```
⊥rec1->form1;rec2->form2; ... ;recn->formn⊥[form]
    = if the value of rec1[form] is not failure then form1
 else if the value of rec2[form] is not failure then form2
                    ..........
                    ..........
 else if the value of recn[form] is not failure then formn
```

Here the recognizer of the following form,

<variable>:<recognizer>

plays the significant part in the conditional recognition. If it is used in the recognition part of recognition-clause and its recognition has a value which is not failure, a new binding of the variable and the value is performed and the variable is keeped within the recognition clause. For example, LISP function copy is defined as follows using them.

```
copy[x] = ⊥(car:$.cdr:$)->(copy[car].copy[cdr]);atom:$->atom⊥[x]
```

Another example shows a recursive function using a conditional recognition.

```
palindrome[x] = ⊥()->nil;($)->t;(c:$ /m:$/ c)->palindrome[m];$->nil⊥[x]
```

This is a predicate palindrome which examines whether its argument is a mirror image of itself or not.

6 EXAMPLES

By means of these PSILISP's features we are able to describe various list processing function more intuitively in structural style. The following examples are the five elementary functions and convenient functions of LISP [3] expressed in PSILISP language.

```
car[x] = ⊥(a:$.$)->a⊥[x]

cdr[x] = ⊥($.d:$)->d⊥[x]
```

```
cons[x;y] = (x.y)

eq[x;y] = ⊥x->t;$->nil⊥[y]

atom[x] = ⊥($.$)->nil;$->t⊥[x]

append[x;y] = (/x/ /y/)

list[x1;x2; ... ;xn] = (x1 x2 ... xn)

reverse[x] = ⊥()->();(a:$ /d:$/)->(/reverse[d]/ a)⊥[x]

member[x;y] = ⊥(/$/ x /$/)->t;$->nil⊥[y]

mapcar[l;fn] = ⊥()->();(a:$ /d:$/)->(fn[a] /mapcar[d;fn]/)⊥[l]
```

From these examples LISP's elementary functions are not primitive in PSILISP. Further it is clear that the structural style program is easy to understand.

7 CONCLUSION

PSILISP is designe for natural programming based on the structure of symbolic data. In this paper, the notions of construction and recognition are presented mainly. Especially recognition is the main feature of this language, it is supported by the segment notation and the psi-expression. By means of segment notation, a segment can be handled as list, and so we can use a segment as a argument of ordinary LISP functions. On the other hand, the psi-expression enriches the ability of recognizers. Thus we can program the algorithms of the symbolic data manipulation with structural style. We believe that the programming in PSILISP is natural and intuitive, the programs are easy to understand.

REFERENCES

[1] Bobrow, D.G. and Raphael, B., New Programming Languages for Artificial Intelligence Research. Computing Surveys, Volume 6, number 3, pp. 153-174, September 1974.

[2] Griswold, R.E. et al. The SNOBOL4 Programming Language (second edition). Prentice-Hall, 1971.

[3] McCarthy, J., et al., LISP 1.5 Programmer's Manual. MIT Press, Cambridge, Mass., 1962.

[4] Moon, D.A., MACLISP Reference Manual (Revision 0), Project MAC, M.I.T., Cambridge, Mass., April 1974.

[5] Rulifison, J.F., et al. QA4, A language for Writing Problem-solving Programs. Proceedings IFIP Congress, 1968.

[6] Sussman, G.J. and Winograd, T. Micro-planner Reference Manual. AI Memo No. 203, MIT Project MAC, July 1970.

[7] Teitelman, W., INTERLISP Reference Manual. Xerox Palo Alto Reserarch Center, Palo Alto, California, 1974.

[8] Tesler, L.G. et al. The Lisp70 Pattern Matching System. Proceedings IJCAI3, Stanford, California, August 1973.