THE 2.PAK LANGUAGE: GOALS AND DESCRIPTIONS

Lucio F. Melli
Department of Computer Science
University of Toronto
Toronto, Canada

ABSTRACT

This paper describes a programming language, 2.PAK, whose main aim is to provide a set of primitives suitable for Artificial Intelligence applications. In addition, 2.PAK tries to incorporate principles obtained from research into programming languages in general. The main features of the language include a data base composed of directed labelled graphs, hierarchical and heterarchical control structures, backtracking primitives (applicable to either control structure), and a generalized form of pattern matching.

1. INTRODUCTION

In 1973, a programming language to facilitate AI research at the University of Toronto was designed and implemented. Although it was SNOBOL [1] based, I.PAK [2,3] offered many features found in the more prominent AI languages such as PLANNER [4], CONNIVER [5], QA4 [6] and SAIL [7] since these languages greatly influenced its design. It thus possessed such features as a data base composed of directed labelled graphs, pattern directed information retrieval, pattern-invoked function calls and generators.

I.PAK shared another feature with other AI languages; it was a disappointment. The language, used extensively in a graduate course on AI and for a Master's thesis [9], was heavily criticized due to its lack of good primitives. The primitives it did provide were ill-defined and although they seemed orthogonal, the combination of some primitives yielded unexpected results. In addition, some of the primitives did too much for the user. This resulted in the user losing control of what was happening in the program and having to depend on kludges to constrain such primitives. It should be noted that such criticisms apply not only to I.PAK but to other AI languages as well, e.g. [8],

The problem thus seems one of finding "good" primitives for AI applications. This paper describes a successor to I.PAK which tries to provide some of these primitives. 2.PAK is a successor in the sense that experience gained from I.PAK has been used in the design of the new language. 2.PAK is not an extension of I.PAK.

2. LANGUAGE GOALS

2. PAK's goals center around two major objectives. The first is to provide a good set of primitives suitable for AI applications. The problem here rests with the choosing of the primitives to be offered. One must be careful not to choose primitives that are too low-level; otherwise the convenience of the language user suffers. On the other hand, the primitives must not be too high-level; otherwise the adaptability of the language suffers. As example, one could consider LISP's primitives as being too low-level for AI applications and PLANNER' s primitives as being too high-level. It was the dissatisfaction with LISP that sparked the design of higher level AI languages so that a user could program his algorithms more conveniently. However, PLANNER's automatic backtracking and pattern directed function calls proved to be so high level and powerful that they lacked the finesse the user required and subsequently could not be used. Thus the ideal solution is to find primitives that strike a happy balance between the two extremes and provide facilities within the language for the user to easily define any higher level primitives that may be needed for a specific application. The search for these primitives was centered around some of the previously mentioned AI languages and others including QLISP [10] and SIMULA [11]. Features offered by these languages were examined and a set of primitives was abstracted.

The second objective is one that is overlooked by most AI languages. 2.PAK tries to accomodate principles obtained from research into programming languages in general. Such qualities as efficiency, readability, reliability and understandability are too important to any language to be omitted from the design stage and then be expected to somehow emerge when the language is completed. We believe that AI languages can greatly benefit from the experiences of programming languages in general, and that they should not set themselves apart and have to reinvent the wheel.

In the light of these two major objectives, the general goals of the language are:

1) Efficiency - of program creation and execution.

2) Natural Syntax - to enhance readability of programs.

3) Understandability and Readability - through the use of simple semantics.

4) Minimality - the language should be concise.

5) Involution - consistent use of language features.

6) Orthogonality - independence of language features.

7) Simplicity - to aid in program construction.

8) Implementability - the language should be implementable cheaply and efficiently.

More specific goals in terms of desired features include:

1) A wide variety of data types,

2) Generalized pattern matching without restriction to a set of data types chosen by the language designer.

3) Generalized control structures ti-e. hierarchical and heterarchical control structures).

4) Good abstraction capabilities to aid in defining higher level primitives required by specific applications.

5) Backtracking as it applies to hierarchical and heterarchical control structures.

6) Flexibility to either compile or interpret expressions within the program.

7) Strong typing of variables whenever possible to allow the system to check for "illegal datatype" errors and thereby reduce them.

8) Interactive facilities and tracing facilities to aid in debugging and program execution monitoring.

3. LANGUAGE OVERVIEW

2.PAK is a block structured language whose main features will be discussed according to the divisions: data types, abstraction facilities, basic statements, global control structures, backtracking, pattern matching and miscellaneous. A more detailed description of 2.PAK is available in [12].

3.1 DATA TYPES

2.PAK offers a wide variety of data types so that the user can choose what is best suited for the task at hand. There are the standard data types such as booleans, integers, reals, strings, references (i.e. pointers to user defined records or coroutines), lists and arrays. In addition there exist unevaluated expressions and hash tables as in SNOBOL, patterns which can be matched or combined to form new patterns, and records, which are user defined aggregates of basic types. For example:

```
record BINARY TREE( ref (BINARYJTREE)
     LLINK, RLINK ; string LABEL ) ;
```

defines a record which can be used to construct string labelled binary trees.

2.PAK also possesses a data base in the form of a directed graph with labelled nodes and edges. Such a structure has shown itself to be very convenient as a representational tool, especially for semantic nets, and is very similar to SAIL'S triples, PLANNER'S assertions, CONNIVER's items or QLISP's tuples and vectors. The structure provides a restricted form of associativity applicable to the nodes of the data base, i.e. for a given node, one can determine all the nodes that are related to that node by means of edges either leaving or entering that node. Thus associativity exists for the nodes of the data base, but not for the edges. This form of associativity is clearly less expensive than that provided by SAIL's triples.

The basic units of the data base are the nodes, and operations exist to add or delete nodes to or from the data base, add or delete edges to or from a specific node, generate edge-node pairs whose edges match a specified pattern and enter or leave a specified node and all edges and nodes within a given radius.

All 2.PAK variables must be declared and typed to alleviate the problem caused by illegal datatypes as arguments to operations. One can view type declarations as intentions of what the type of the variable is to be. The system will then provide the necessary checking either at compile or run time. If the type of a variable is not known or if it can be of more than one type, one can declare it to be of type var which allows that variable to take values of any type.

3.2 ABSTRACTION FACILITIES

An abstraction facility is the capability of grouping together entities into a unit that can be assigned a name for reference purposes. For data types, record declaration is an absraction facility since one can construct the required record from primitive data types and then use that record as a primitive data item of the language. For 2.PAK statements, there exist three three types of abstraction facilities: procedures, function procedures and coroutines, Procedures and function procedures are defined as in most programming languages. Coroutines differ from procedures in that they can be used to achieve heterarchical control structures (see section 3.4). In addition, variables of a suspended coroutine instance can be examined or altered.

Lastly, macros provide an abstraction facility for character strings within the text of the source program. For example:

```
macro '?' replaced by '%' ;
```

will replace all occurrences of '?'. following the macro definition, by ;%'.

3.3. BASIC STATEMENTS

The basic statements of 2.PAK are the assignment statement, statements that control local sequencing (i.e. if, while and case) and the I/O statements. Most of these statements are fairly standard as is shown by the examples:

```
if  X - Y
then  SAME := SAME + 1 ;
      end ;

if  X < Y
then  Y :- X ;
else  COUNT := COUNT + 1 ;
      end ;

while  FLAG = true  do
     read X ;
     write X ;
     end ;

case  PRIMARY_COLOUR  of
     'RED'  : ... ;
     'YELLOW'  :  ... ;
     'BLUE'  :  ... ;
     else : MESSAGE :=
       'NOT A PRIMARY COLOUR.' ;
     end ;
```

The example case statement will examine the value of PRIMARY_COLOUR, which must be of string type, and will execute the case containing that value as a case label. If the value is not in the range of specified case labels, the case having else as a case label will be executecT!

3.4 GLOBAL CONTROL STRUCTURES

Global control deals with the primitives that transfer control to or from procedures and coroutines. For procedures there exists the hierarchical control structure provided by the procedure call, possibly recursive, and the return statement. As is the norm for procedures, a procedure's environment is destroyed when a return is executed. On the other hand, coroutines survive transfers of control and are merely suspended until control is returned. Execution then commences at the point where the coroutine was last suspended.

For coroutines, 2.PAK offers two types of control primitives which are in many respects similar to those of SIMULA. The hierarchical type, provided by invoke and detach and similar to the call/ return of procedures, is ideal when it is necessary to have a coroutine execute under the control of some other block, i.e. as is the case for generators. Thus the invoke statement, like a call, carries information to the invoked coroutine as to where a detach should return. However, the heterarchical primitive resume carries no such information. The resumed coroutine has no idea who resumed it and assumes complete control of the computation. Thus one could consider different coroutines as representing different environments and use resume to transfer control among these environments.

3.5 PATTERN MATCHING

2.PAK offers a generalized pattern matching facility not restricted to a set of data types chosen by the language designer, as is often the case. This is achieved by letting the user have the facility for defining the semantics of required pattern matching primitives and their evaluation sequences. Defining the semantics of pattern matching primitives reduces to defining a set of functions that operate on the position of a cursor within the subject structure. Such an approach to pattern matching is applicable to strings, lists, graphs or any user defined structure. The evaluation sequence of a pattern matching primitive specifies when that primitive is to be evaluated. The three possibilities are: evaluate when the primitive is encountered while the matcher is moving in the forward direction (i.e. moving left to right through the pattern), evaluate when encountered whil moving in the backwards direction (i.e. backtracking), and evaluate whenever encountered. Eor example, the string pattern matching primitives FENCE and SUCCEED of SNOBOL exhibit the second type of evaluation sequence, while the edge expressions of l.PAK patterns show the third type. A pattern in 2.PAK is therefore composed of a sequence of boolean expressions with associated evaluation sequences. The 2.PAK pattern matcher executes the pattern in a backtracking mode whereby if an expression evaluates to true the matcher proceeds forward to the next expression; if false the matcher backtracks.

For the convenience of the user, some of the more basic pattern matching primitives for strings, lists, and graphs are provided. However, these primitives should by no means be taken as dogmatic and one is still free to define his own. Sample 2.PAK patterns arc found in section 4.2.

3.6 BACKTRACKING

2.PAK backtracking primitives are completely disjoint from control primitives since such a separation allows one to combine the two in the manner that produces the desired result in the most efficient way. Backtracking is therefore viewed as the means for manipulating state changes made within what is termed a context. Primitives provide facilities for entering a new context, fo'r specifying what changes are to be backtrackable within a context, and when to back up to the previous context and what to do with the backtrackable changes. The context feature is less prominent than CONNIVPR's (where everything is carried out in a backtrackable context) and we consider it more economical since the user can choose when to use the feature.

In addition, there are primitives that can be used for the heterarchical control structure provided by coroutines. Preserve binds a context to a coroutine instance and restore restores the bound context of a specified coroutine instance. Thus restore can be used in conjuction with the resume statement to provide the facility of transferring control within multiple environments, each with it's own context. Another use is the evaluation of expressions outside the current environment. This can be easily achieved by restoring the desired environment, evaluating the expression and then restoring the original environment. Note that for such a task no transfer of control is necessary and that none takes place.

3.7 MISCELLANEOUS

In addition to the described features, 2.PAK provides miscellaneous built-in functions to aid the user. These functions include:

apply - Similar to the apply function of SNOBOL,' this function provides a dynamic function calling facility.

compile - This function accepts as argument a string representation of a 2.PAK expression and returns its equivalent unevaluated expression.

eval - This function evaluates a 2.PAK unevaluated expression and returns the produced result.

trace - This function allows the user to enable tracing of changes made to a specified variable or transfers or control to or from a specific procedure or coroutine.

The language also provides toggles to facilitate extensive tracing. For example, setting the toggle .CTRACE. to 10 will result in the tracing of the next 10 transfers of control made by any coroutine of the program. Other toggles include: trace all variable changes (.TRACE.), trace all function calls and returns (.FTRACE.), and trace the evaluation sequence of pattern matches (.PATTERN.).

4. EXAMPLES

4.1 A ONE-ARMED BANDIT

The concept of a generator is an important one for AI applications. A generator is a function that on successive calls will produce and return elements from a specified range. We wish to demonstrate that 2.PAK's coroutines are more than adequate substitutes for generators. This is done by presenting a coroutine that simulates a one-armed bandit slot machine. The example is to be judged not on efficiency, since much more efficient formalizations can be obtained, but rather on its merits as a generator.

The first thing one needs for the bandit are the rings which display the generated sequence. A bandit usually contains three such rings which are themselves generators of elements from the ordered set composed of the fruits cherry, lemon, orange and apple. Such a generator is simulated by the coroutine:

```
coroutine FRUIT_GEN()
begin
  string FRUIT ;
  FRUIT :- 'CHERRY' ;
  detach ;

  while true do
    case FRUIT of
      'CHERRY': FRUIT := 'LEMON' ;
      'LEMON': FRUIT := 'ORANGE' ;
      'ORANGE': FRUIT :- 'APPLE' ;
      'APPLE': FRUIT :- 'CHERRY' ;
      end ;
    detach ;
    end ;
  end ;
```

The first invocation of FRUIT_GEN generates the string value *CHERRY' in the variable FRUIT and then control is returned to the invoking block by means of a detach. On the second invocation, the coroutine enters the while statement and executes the case statement which uses the old value of FRUIT to generate the next value. Note the cyclic nature of the generator and how easily this is achieved with a case statement. After the new value for FRUIT has been generated the coroutine detaches. Subsequent invocations will commence execution after this detach, thereby remaining within the generating loop.

What is now needed is a mechanism for "spinning" these rings. (We have only defined the prototype ring but we can obtain multiple copies of it). Consider the following procedure:

```
procedure SPIN( ref (FRUIT_GEN) RING ;
                    integer I)
begin
  while I > 0 do
     invoke RING ;
     I := I - 1 ;
     end ;
  end ; /* OF SPIN. */
```

The above procedure takes as arguments a reference to an instance of a FRUIT_GEN and an integer I. It then proceeds to invoke the specified FRUIT_GEN instance I times and thereby achieve the effect of spinning the ring.

Finally we can now define the bandit itself:

```
coroutine BANDITQ
begin

  coroutine FRUIT_GEN() ... ;
  procedure SPIN ... ;

  ref (FRUIT_GEN) FIRST, SECOND,
   THIRD ;
  string RESULT ;
  integer RAND1, RAND2, RAND3 ;

  FIRST :- FRUIT GEN() ;
  SECOND :« FRUIT GENQ ;
  THIRD :- FRUIT GEN() ;
  RESULT := 'CHERRY LEMON ORANGE* ;
  detach ;

  while true do
    /* GENERATE 3 POSITIVE RANDOM
       INTEGERS IN RAND1, RAND2
       AND RAND3.                */

    SPIN( FIRST , RAND1 ) ;
    SPIN( SECOND , RAND2 ) ;
    SPIN( THIRD , RAND3 ) ;

    if (FIRST.FRUIT = SECOND.FRUIT)
      $ (FIRST.FRUIT = THIRD.FRUIT)
    then RESULT :* 'JACKPOT' ;
    else RESULT :* FIRST.FRUIT
        |I * ' It SECOND.FRUIT
        | | ' ' | | THIRD.FRUIT ;
      end ;
    detach ;
  end ; end; /* OF BANDIT */
```

First the coroutine declares the procedures, coroutines and variables it needs. The variables FIRST, SECOND and THIRD are references to instances of FRUIT_GEN and represent the three rings the bandit needs. RESULT will contain the result of playing the bandit. On the first invocation, BANDIT sets up its three ring generators and generates a losing value. (It is assumed that the management will absorb this first loss by initializing the bandit before letting anyone play). The generator then detaches. On subsequent invocations, BANDIT will generate three integer random numbers (unspecified and left to personal preference as to which method to use), and spins its rings by the obtained values. It then generates 'JACKPOT' if all three fruit values are the same or generates the string obtained by concatenating the generated values. The generated values are retrieved by using the dot notation for accessing coroutine variables, i.e. FIRST.FRUIT represents the variable FRUIT in the coroutine instance referenced by FIRST. Note that the generated values depend on the previous states of the rings. Good luck.

4.2 LIST PATTERNS

We now demonstrate how one can achieve some of the list pattern matching facilities offered by CONNIVER. This is accomplished by taking sample patterns from the CONNIVER reference manual [5] and presenting their 2.PAK equivalents. Before proceeding, some list pattern matching primitives have to be defined. (Note that, in this example, list atoms for 2.PAK are assumed to be strings).

```
boolean procedure MATCH_ATOM( string
  SI )
/* THIS FUNCTION MATCHES ATOMS IN A
  LIST AND ADVANCES THE CURSOR. V
begin
    if type (CURSOR)   —  STRING;
    then return false ;
    else if car(CURSOR) - SI
      then CURSOR := cdr(CURSOR) ;
        return true ;
      else return false ;
        end ;
      end;
    end ;     /* OF MATCH_ATOM •/

boolean procedure DOWN( pattern PI )
/* FUNCTION THAT DESCENDS ONE LEVEL IN
  THE LIST AND MATCHES THE PATTERN PI.
  */
begin
    if type(CURSOR) ^= 'LIST'
    then return false ;
      end ;

    if match ( PI , car(CURSOR) )
    then CURSOR :- cdr(CURSOR) ;
      return true ;
    else return false ;
      end ;
    end ;     /* OF DOWN. */

boolean procedure SET( var X )
/* THIS FUNCTION ASSIGNS TO X THE CAR
  OF THE LIST CELL BEING MATCHED. */
begin
    if type(CURSOR) -.« 'LIST'
    then return false ;
      end ;

    X :- car(CURSOR) ;
    CURSOR :- cdr (CURSOR) ;
    return true ;
    end ;     /* OF SET. */

boolean procedure REM( var X )
/* THIS FUNCTION ASSIGNS TO X THE
  REMAINDER OF THE LIST BEING MATCHED.
  */
begin
    if type(CURSOR) -= 'LIST',
    then return false ;
      end ;

    X :- CURSOR ;
    CURSOR :- null ;
    return true ;
    end ;     /* OF REM. */
```

In the following examples atom denotes an arbitrary atom and list denotes an arbitrary list.

1. (MATCH '(FOO !>X) '(FOO BAR))
   - The pattern, denoted by the second list element, matches lists of the form (FOO atom). The subject is the list (FOO EKKT and after the match X will be associated to the atomic value BAR.
   The equivalent 2.PAK pattern match is:
   match(<:MATCH_ATOM('F00'),SET(X):> ,
   <'FOO', 'BAR'>);
After the match X has the value 'BAR*.

2. ((FREDS !>X) . !>REST)
   - This pattern matches lists of the form:
   ((FREDS atom) list)

   The equivalent 2.PAK pattern is:
   <: DOWN(<:MATCH ATOM('FREDS') ,
   SET(X):>) , REMIREST):>

3. (GRANDFATHER !>X /,X)
   - This pattern matches lists of the form:
   (GRANDFATHER atom atom)

where the two atom values are equal. The equivalent 2.PAX pattern is:

<: MATCH ATOM('GRANDFATHER') , SET(X) ,
   MATeH_ATOM(X) :>

4. :>(CREATURE (FEATHERLESS • ,CREATURE)
   (EQ (NUMBER_OF LEGS !,CREATURE) 2))

   - This pattern matches lists of the form:

   (atom (FEATHERLESS atom) (EQ (NUMBER^
   OFTEGS atom) 2))

where the three atom values are equal. The equivalent 2.PAK pattern is:

<: SET(CREATURE) , DOWN(<: MATCH ATOM
  ('FEATHERLESS') , MATCH ATOM(CREATURE)
  :>) , DOWN(<: MATCH ATOM('EQ') ,
  DOWN(<: MATCH ATOMC'NUMBER OF LEGS')
  MATCH_ATOM(CREATURE) :>) , MATCHJVTOM
  ('2') :>) :>

The above 2.PAK patterns are usually much longer than their counter-parts since we have used mnemonic names rather than "!" or "," One could define the primitives with shorter names or use macros to shorten the expressions that need be written. Note also that all of the example 2.PAK patterns happen to have primitives with evaluation sequence "evaluate when encountered while moving forward". Since this seems to be the most common evaluation sequence, it is assumed by default and has no syntactic specification.

The above are by no means all of the available list matching primitives of CONNIVER and are only meant to give a flavour of how one can extend 2.PAK pattern matching to suit a specific structure.

CONCLUSIONS

In this paper, we have introduced and briefly described 2.PAK, a language that provides a set of programming primitives suitable for AI application, and that makes use of experiences gained in the design of programming languages in general Since it is not yet clear what primitives are essential to AI applications, we have chosen the approach of examining existing AI languages and abstracting from them a set of primitives that can be easily combined to produce many of the language features now deemed important. (We are even optimistic that some of our primitives, i.e. co-routines and the heterarchical control primitives, can be used to achieve a form of actors [13].) We hope that the language description and the examples demonstrate that some success was achieved, especially in realizing the specific goals presented in section 2.

The general goals of the language guided the overall design and their influence is evident throughout 2.PAK. For example: orthogonality led to the separation of backtracking primitives from control primitives, and to the realization that pattern-directed procedure invocation is composed of a pattern match and the use of a function such as apply; minimality reduced the set of heterarchical control primitives to three; etc. We would like to stress again that we believe AI languages can greatly benefit from the experiences of programming languages in general and hope that this is evident in 2.PAK.

2.PAK is currently in the process of being implemented as a interactive language. Plans are to implement a kernel of the language and then bootstrap the rest. The kernel will probably be implemented in SPITBOL [14] since it proved to be a convenient and adequate language to use when 1,PAK was implemented. It is expected that an implementation and its subsequent use will answer many of the questions regarding the appropriateness of the provided primitives for AI applications.

ACKNOWLEDGEMENTS

I would like to thank Professor John Mylopoulos for all his help, and the National Research Council of Canada (grant A7905) for supporting this research.

REFERENCES

1. Griswold, R.E., Poage, J.F. and Polonsky, I.P.; The SN0B0L4 Programming Language; Prentice-Hall Inc., Englewood Cliff, New Jersey, 1971 (second edition), 256 pages.

2. Badler, N., Berndl, W., Melli, L. and Mylopoulos, J.,; The 1. PAK Reference Manual; TR 55, University of Toronto, Dept. of Computer Science, August 1973, 145 pages.

3. Mylopoulos, J., Badler, N., Melli, L. and Roussopoulos, N.; "l.PAK: A SNOBOL-based Programming Language for Artificial Intelligence"; IJCAI 3, Stanford University, Stanford, California, August 1973, pp.691-696.

4. Hewitt, C.; Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot; AI TR-258, MIT, Artificial Intelligence Laboratory, April 1972, 408 pages.

5. McDermott, D.V. and Sussman,C.J.; The CONNIVER Reference Manual; MIT AI Lab Memo No. 259a, May 1972 (updated Jan. 1974), 146 pages.

6. Rulifson, J.F., Derksen, J.A. and Waldinger, R.J.; QA4: A Procedural Calculus for In^u^tive ReasoningT" Technical Note 73, SRI AI Center, Nov. 1973.

7. Swinehart, D. and Sproull, B.; SAIL; Operating Note No. 57.2, StanforH AI Project, January 1971.

8. Sussman, G.J. and McDermott, D.V.; Why Conniving is Better than Planning; MIT AI Lab Memo No" 255A, April 1972; 31 pages.

9. Cohen, P.R.; A Prototype Natural Language Understanding System; TR 64, University of Toronto, Dept. of Computer Science, March 1974, 149 pages.

10. Reboh, R. and Sacerdoti, E. ; A Preliminary QLISP Manual; ArtiTicia1 Intelligence Center, Technical Note 81, SRI Project 8721, August 1973, 35 pages.

11. Dahl, O.J. and Hoare, C.A.R.; "Hierarchical Program Structures": in Structured Programming, Academic Press Inc., London, 1972, pp.175-220.

12. Melli, L.F.; The 2.PAK Language: Primitives for AI Applications; TR 73, University of Toronto, Dept. of Computer Science, Dec. 1974, 151 pages.

13. Hewitt, C., Bishop, P. and Steiger, R.; "A Universal ACTOR Formalism for Artificial Intelligence"; IJCAI 3, Stanford University, Stanford, Calif., August 1973, pp.235-245.

14. Dewar, R.B.K.; SPITBOL, Version 2.0; Illinois Institute of Technology, 1971.