DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION

UNIVERSITY OF EDINBURGH

Memorandum: MIP-R-29

Date: 30th November, 1967

Subject: Memo functions: a language feature with "rote-learning" properties

Author: Donald Michie

Memorandum: MIP-R-30

Date: 30th November, 1967

Subject: Memo functions and the POP-2 language

Author: R.J. Popplestone

(Both papers are substantially as submitted to Proc. IFIPS 68 Congress)

DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION

UNIVERSITY OF EDINBURGH

Memorandum: MIP-R-29
Date: 30th November 1967.

Subject: Memo functions: a language feature with "rote-learning" properties.

Author: Donald Michie.

These notes are concerned with ways of handling functions of integer variables within the framework of Algol-type programming languages. Such languages present the user with a choice either to waste space by declaring and initializing arrays, most of the cells of which will never be accessed, or to waste time by repeatedly re-computing the same function.

The proposal here is to provide a single object, which we shall call the "memo function", which can be used to replace both the array and the function procedure. A quick guide to the spirit of the proposal is as follows:

(1) A function is a function. This point of view is inherent in the CPL programming language (Barron, Buxton, Hartley, Nixon and Strachey, 1963) and has been made explicit in POP-2 (Burstall & Popplestone, 1968) where it occupies a central place.

(2) The user should not need to concern himself about the method of evaluation e.g. "procedure call" versus "array access". Nor is there any rule which says that the method of evaluation must be the same on different occasions during computation.

(3) To tabulate values of a function which will not be needed is a waste of space, and to recompute the same values more than once is a waste of time.

(4) A function means what it is defined to mean and nothing

more/

more. Since computation proceeds along a time dimension, there is no reason to prohibit either the method of evaluation or the definition of a function from changing as the computation progresses. Consider the evaluation by a human administrator of some function, such as the Boolean "eligible for post-graduate training award": initially, this is evaluated by rule, but in course of time exceptional instances arise. These are indexed and filed, so that the function can be evaluated in future by a combination of reference to rule ("procedure call") and reference to precedent ("array access") whichever is appropriate. Eventually the rule is re-defined so as to accommodate the exceptions, and the evolutionary cycle begins again. This fundamental inductive process should be explicitly catered for in a programming language.

Let us imagine a program which requires from time to time during its operation the value of an integer function "prime (n)" where n is an integer in the range 1-100 and prime (n) is the nth prime number. If the Algol programmer knows that the number of occasions on which the evaluation will have to be made during the running of the program is small, then he will write a function procedure and be done with it. Otherwise he has the choice either to waste space or to waste time. If he chooses to waste space he will declare "integer array prime 1 : 100;" and initialise it, perhaps by reading in a table of primes. If he chooses to waste time he will write function procedures isprime (a Boolean) and prime. The latter might look like the following:

integer/

```
integer procedure prime (n);
  integer n; value n;
  begin integer i, p;  if n = 1 then
    prime : = 2 else begin p : = 1;
    for i : = 2 step 1 until n do
      begin
  loop: p : = p + 2;
        if not isprime(p)  then goto loop
      end
        prime : = p end
  end of prime;
```

In the first case evaluation on each occasion is done by array access and in the second by procedure call. The preference has in part to be decided by machine-dependent considerations (such as storage limitations; also relative execution-speeds for array access, procedure calls, and evaluation of expressions), and in part unforeseeables, such as the total number of times that the function will need to be evaluated. But whichever method is adopted waste, either of space or of time or of both, is inevitable. Yet, as we shall see, there is no need to suffer this waste at all.

Stepping back from the machine

In the approach adopted here we first step back from questions of machine implementation, until we have obliterated from view the distinction between the "procedure" and "array" representation of functions. From this point of vantage the only meaning of "function" is that of a mapping from sets of arguments to sets of values. The two alternative definitions of the function prime given earlier are now in every way equivalent.

We wish to leave the user undisturbed in this state of simplicity, and to do this we shall re-approach the machine cautiously, an inch at a time. Our aim is to come to a point at which it is again determined, for each evaluation, whether it is done by table look-up or by calculation, but the decisions are now taken where they should be taken - behind the scenes. Further, the user will find that he has been saved both from wasting space and from wasting time: functions will be evaluated by look-up whenever a previously computed value is available and otherwise by calculation. It follows, of course, that as the computation progresses, more and more evaluations will be performed by look-up and less time spent in calculating new values, so that in a sense the program will "learn" to evaluate functions quicker - much as a human calculator gradually extends the range of his memorised multiplication table through practice. This usage of the notion of learning is not altogether trivial, and a detailed example of "rote-learning" in/

in this sense can be found in Samuel's (1959) classic study of machine learning. Samuel devised, and applied to the game of checkers, mechanisms both of rote-learning and of learning by generalization. The present proposal is concerned solely with rote-learning, but in the paper which follows Popplestone shows how standard methods of interpolation can be used to give an effect of "learning by generalization".

Assignment as a "function-update"

In order to destroy the division between the function procedure and the array, we must discard one and build a unified implementation around the other. We shall throw out the array. But before we do so we must note that the two implementations of a function in Algol 60 do not in fact quite coincide in respect of the freedoms given to the user. Whereas it is legal to up-date a function as defined by array, by writing

prime [n]:= <expression>

it is not legal to update the same function as defined by procedure, by writing

prime (n) := <expression>

In jettisoning the array we shall preserve the freedom to update a tabulated value, so that state-:ments like prime (1):= -1; or factorial (3) := 17 become legal. The guiding line is that a function should be at the mercy of the programmer who defines it rather than *vice versa*. We shall see later that this enlarged freedom is not a whim, but is of fundamental application to "learning" programs which acquire skill not only by direct trial and error but also *via* advice imparted by a "tutor". Another application of "function updates" is in the realm of inductive learning, where the object is to describe empirical observations by means of a computed function, i.e. by means of a "theory" or "generalization". As will be seen, a memo function consists of a "rule" part/

part (procedure body) and a "rote" part (dictionary). In inductive learning the rote part is the repository of experience while the rule part is the repository of theory. An observation performed on the outside world (i.e. a "read" operation on some input stream) causes an assignment to the rote part. The process of induction consists precisely in modification of the rule part so as to improve its fit to the observations contained in the rote part and hence increase its accuracy as a predictor of future observations. Inductive prediction is further discussed by Popplestone in the paper which follows.

The dictionary

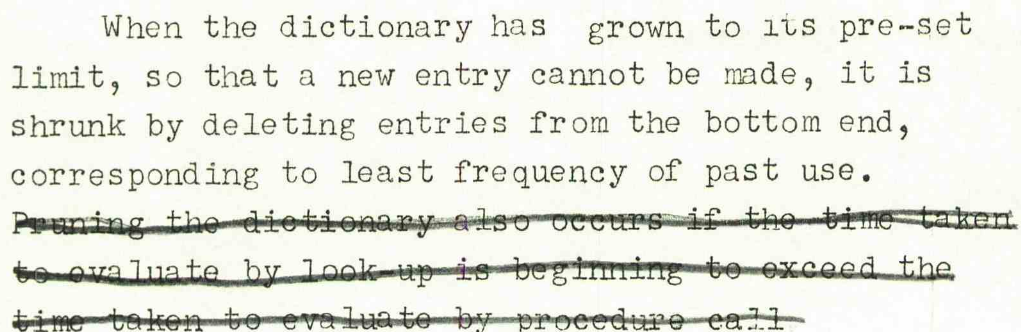
To replace the discarded structure "array", we introduce a new type "dictionary". A dictionary has no independent existence, but is an appendage of a new kind of function procedure, the memo function. A dictionary is an ordered set of pairs. The first member of each pair is an argument list, and the second is a value. Initially the length of a dictionary is zero, and it grows during computation as entries are made in it. An upper bound to the length to which it may grow is specified in the declar-:ation. A full declaration might run as follows

integer procedure prime (n); value (n); integer (n)
dic [20];

The effect of dic [20] is to put an upper limit to the size to which it is to be allowed to grow. Note that dic represents an "own" structure.

The effect of a function call is to initiate a search of the dictionary. If successful the tabulated value is fetched. If the search is unsuccessful the procedure body is entered, the value computed, and, as a side-effect before returning, the argument list and computed value is added to the dictionary. The dictionary entries are maintained in an order corresponding approximately to frequency of use, by the action of a further side-effect which promotes the currently accessed entry to a higher place in the dictionary (compare Samuel's process of "refreshing" for board positions in checkers and Longuet-Higgins & Ortony's (1968) method of adaptive tree look-up).
An/

An up-date of the function has the general form f (<argument>) :=<expression> and causes a dictionary entry direct. An entry of this type may be distinguished by a mark which protects the entry from deletion (see below). Note that a memo function can be used as a pure data-structure, if desired, by defining a dummy procedure. Contrawise, it can be used as an ordinary function procedure by declaring dic [0].



When the dictionary has grown to its pre-set limit, so that a new entry cannot be made, it is shrunk by deleting entries from the bottom end, corresponding to least frequency of past use. ~~Pruning the dictionary also occurs if the time taken to evaluate by look-up is beginning to exceed the time taken to evaluate by procedure call~~

Saving space and saving time

Let us return now to the point with which we started, that current languages force the user to decide, on information which is irrelevant to his interests and often inadequate, whether to waste space or to waste time. The memo function only uses so much space as has actually been found useful, and recomputation of a function for the same arguments is held down to the economic minimum. The savings could be large in each case, the most marked expected gain being the shortening, as the dictionary grows, of the average time used to evaluate a function. In the case of a function which is defined recursively, the gain can be greater as will now be discussed.

Application to recursively defined functions

Suppose that we had defined the prime function used earlier as follows

```
integer procedure prime (n);
  integer n;  value n;
    begin integer p;
      if n = 1 then prime : = 2 else begin
      p : = prime(n-1);
  loop: p := p + 1
      if not isprime (p) then goto loop
      prime := p end
    end of prime;
```

When such a function is first applied to some actual parameter, *m*, it cannot be evaluated without in the process being evaluated not only for *m* but also for *(m-1)*,*(m-2)*... etc. In consequence of the scheme described in the previous section, when the evaluation finally terminates, m dictionary entries have automatically been made, one for each level of the recursion. Suppose now that *prime (n)* is called at some later stage. If $n < m$ the benefits of immediate look-up will be reaped, for the work has already been done, once and for all, for any argument lying in the range from 1 to *m*. Suppose now that *n* is greater than *m*, being equal, say, to $m + i$. Again large savings accrue, for in place of a recursion of depth $m + i$ only *i* calls of the function are made before the recursion hits a dictionary entry and terminates (adding in the process *i* new entries to the dictionary). Provided that/

that the dictionary limit is so small or the time to compute the function so great, that the ratio of look-up time to calculate time is small the economies to be gained are of the order of n-fold, where n is the number of calls of a given function made by a program during its working life. For a program with many recursively defined functions, or with such functions in inner loops, particularly if the program is due for heavy use, the saving of computing time could run to several orders of magnitude. An economy of another type is also involved, namely economy of effort and thought on the user's part, who is freed from the necessity to consider any attribute of the functions he uses beyond how they are defined and what they do. Together with this freedom goes an enhanced power, through the up-date facility, to treat a function as a dynamic entity, capable of evolution and growth, rather than as a set of static argument-value relations fixed for all time.

Pruning the dictionary: "forgetting"

The advantages of the memo function as a conceptual simplification for the user are fairly obvious. Practical usefulness in terms of faster evaluation times can only be assessed by experiment and will certainly vary according to the application. It is worth drawing attention here to the pruning mechanism, whereby the least frequently used dictionary entries are dropped whenever the current estimate of look-up time begins to exceed the corresponding compute time. Herein lies a guarantee that whether or not important savings of time accrue, time need not be *wasted*. When evaluation by procedure call is so fast that there is nothing to gain by having a dictionary, it will simply fail to grow. Lack of growth can be detected by program and a switch invoked to short-circuit the memo mechanism and to revert to evaluation by simple procedure.

In terms of the analogy with human learning, pruning the dictionary can be thought of as a deliberate, or unconscious, forgetting of unwanted knowledge, based on a decision that in such and such a case it is better to "work it out again from scratch" rather than to search one's memory. Skill in problem-solving depends largely on the ability to memorise key facts. What is less widely appreciated, except perhaps among those who solve problems for their living, is that the power and will to *forget* facts of less relevance is equally important.

<u>Thought-processes of a rote-learning automaton</u>

Consider now the problem of programming a simple learning automaton. The automaton's environment might consist of the state signals emitted by some unstable system which the automaton must learn to control, as in the pole and cart balancing problem studied by the author as an exercise in pure rote-learning (Michie & Chambers, 1968). The automaton is endowed with a function which specifies some strategy, and if the automaton is to learn, this function must be modifiable. The question not only of learning but also of <u>teaching</u> then arises (see Donaldson (1960) in the case of pole-balancing). Advice from the tutor can take the form of <u>rule advice</u> (how to compute a strategy) or <u>rote advice</u> (what to do in such and such circumstances). Rote advice is equivalent to updating the automaton's control function by assignment. It is in this context that the need could arise for protecting dictionary entries of this category from the forgetting process, the simplest method being to keep them in a separate list from the main "rote".

One can now sketch the correspondence between learning in control situations on the one hand and the main features of the memo function apparatus on the other.

Memo functions	Learning situation
Arguments	Environmental states
Values	Actions of the automaton
Memo function call (evaluation by rule)	Choosing an action by applying a strategic rule, and memorising this for future use.
Memo function call (evaluation by rote)	Choosing an action from a recollection of past instances. Refreshing memory of state-action pair.
Pruning the dictionary	Forgetting instances which have not recurred in recent experience.
Assignment to memo function	Memorising rote-advice from a tutor.

Concluding remark

These notes belong to the category "Wouldn't it be nice if ...". Wouldn't it be nice, we say, if the programmer were able

(a) to evade the procedure *versus* data-structure choice for the representation of functions, using instead a single all-purpose object,

(b) to update this object by assignment, and

(c) to trust that the system will choose on each occasion the more efficient means of evaluation, will improve its efficiency in an adaptive fashion and will "forget" past experience where desirable.

I have not attempted to say how this attractive fancy should be implemented. Nor have I considered how the basic notion of a memo function might be generalised to arguments other than integers, nor how it might be extended to encompass more ambitious forms of learning than rote-learning. These themes are developed by Popplestone in the succeeding paper. He shows, among other things, that using the POP-2 language the programmer can construct from half a dozen simple POP-2 functions the full memo function apparatus here described.

Acknowledgements

An essential background to these notes is a climate of ideas assimilated in discussions with my colleagues, R.M. Burstall and R.J. Popplestone; in particular the latter's insistence that in programming languages <u>entia non sunt multiplicanda praeter necessitatem.</u>

My thanks are also due to the Science Research Council for financial assistance.

References

Barron, D.W., Buxton, J.N., Hartley, D.F., Nixon, E. and Strachey, C. (1963) The main features of CPL. Computer Journal, 6, 134-143.

Burstall, R.M. and Popplestone, R.J. (1968) The POP-2 reference manual. Machine Intelligence 2, (eds. E. Dale and D. Michie) Edinburgh: Oliver and Boyd, pp. 207-246.

Donaldson, P.E.K.,(1960) Error decorrelation: a technique for matching a class of functions. Proc. III International Conf. on Medical Electronics, 173-178.

Longuet-Higgins, H.C. and Ortony, A. (1968) The adaptive memorization of sequences. Machine Intelligence 3, (ed. D. Michie) Edinburgh University Press (in press).

Michie, D. and Chambers, R.A. (1968) BOXES: an experiment in adaptive control. Machine Intelligence 2, (eds. E. Dale and D. Michie) Edinburgh: Oliver and Boyd, pp. 137-152.

Samuel, A.L. (1959) Some studies in machine learning using the game of checkers. IBM J. Res. Dev. 3, 210-229.

DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION

UNIVERSITY OF EDINBURGH

Memorandum: MIP-R-30

Date: 30th November, 1967

Subject: Memo functions and the POP-2 language

Author: R.J. Popplestone

I am here going to consider how the ideas expressed in the preceding paper by Michie could be implemented in the POP-2 programming language (Burstall and Popplestone 1968), and how these ideas could be generalised, and how they could be used in Machine Intelligence work.

First, let us consider the meaning of updating a POP-2 function. Those POP-2 functions which can be updated are called doublets (cf. the LUPs of CPL - Strachey). A doublet is a function to which is attached another function called the updater of the doublet. This updater is invoked when the function is updated, or, as we say, used in the update sense. Thus suppose that F is a POP-2 doublet and G is its updater. Then the statement

```
2 → F(3);
```

causes G(2, 3) to be executed, where → is the POP-2 assignment symbol, which has its operands in the opposite order from ALGOL.

In order to implement memo functions we will need some functions for handling dictionaries. For the purposes of experiment it was decided to use association lists like those used in LISP (McCarthy et. al) for dictionaries. There is a bound fixed on the length of these association lists, and if a new item is added to a list of this maximum size, then the last item is "forgotten". Every time an item is looked up in an association list, it is promoted one place in that list. Thus the more frequently used items are found faster and are in less danger of being forgotten.

The function for looking up an item x in an association list alist is

```
function assocval (x, alist, equiv);
loop : if alist.tl.null then undef exit
       if /
```

```
        if equiv (alist.tl.hd.front, x) then alist.tl.hd.back,
        if not (alist.hd = 0) then
           alist.tl.hd, alist.hd → alist.tl.hd → alist.hd
        close
        exit
        alist.tl → alist;  goto  loop
    end
```

The argument equiv is a boolean function which is used to decide on the equality of x with any item stored in the association list. It has to be supplied as an argument because it is not possible to define a general purpose equality function in a language with as wide a range of structures as POP-2. Moreover, the user needs the ability to specify what items are to be equivalent. For instance in tic-tac-toe two board positions are equivalent if they can be made to coincide by applying reflections and rotations. The structure of an association list is shown in Fig. 1. The dummy first cell is used when the list is being updated.

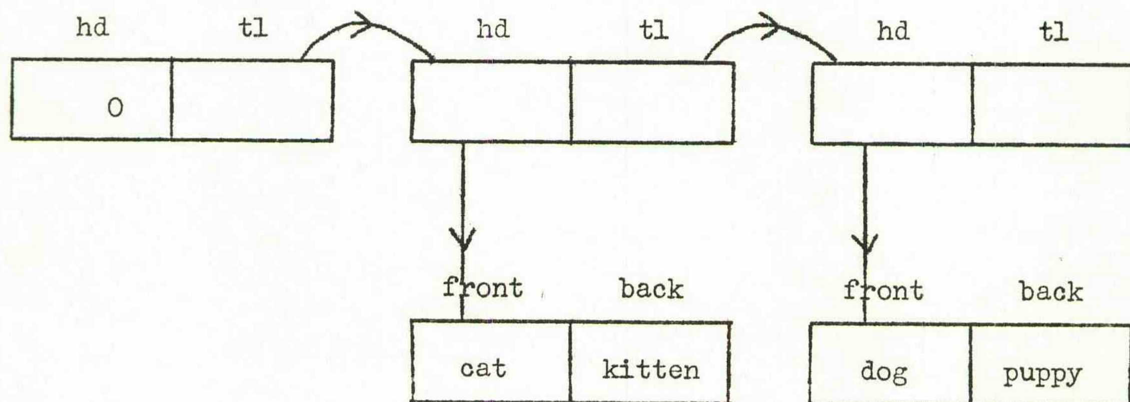


Fig. 1

In a like manner one can define the function assocud to update an association list. This has the arguments y, x, alist, n, equiv where y is the value to be associated with the item x in alist. n specifies the maximum length of the association list.

Let /

Let us now dress these association lists up as functions to give what Burstall (a private communication) calls "thing arrays". A thing array is like an Algol array except that its subscripts are not restricted to being integer. The function for creating these is defined by:

```
function  newassocfn(n, equiv);
vars u v;
        0 :: nil → u;
        assocval (% u, equiv %) → v;
        assocud (% u, n, equiv %) → updater (v);
        v
end
```

The effect of a call of newassocfn is to produce an object very like an array, but whose arguments are not necessarily integer. Thus one might do:

```
newassocfn (100, nonop = ) → youngof:
"puppy" → youngof ("dog");
"kitten" → youngof ("cat");
```

Evaluating youngof ("dog") would now produce "puppy".

How does newassocfn work? The parameters n and equiv specify the maximum size of association list to be used, and the equivalence relation to be used. In the example youngof given above, nonop = specifies that the ordinary POP-2 equality, which deals correctly with words like "cat" and "dog", is to be used (the "nonop" indicates that the equality function is to be passed as an argument rather than be entered).

The first action of newassocfn is to make an empty association list and store it in the local variable u. Next the statement

```
assocval (% u, equiv %) → v;
```

takes the function assocval and forms a new function which is assocval with its last two arguments preset to the values of u and equiv finally placing /

placing the result in v. This operation is called "partial application", and serves the same purpose as the "two bars" function definition in C.P.L. (Barron et. al. 1963) namely to create one function out of another by "freezing" some of its variables. The function now in the variable v is going to be the result newassocfn , but first it must be given an updater. This is accomplished by the statement:

```
assocud (% u, n, equiv %) → updater (v);
```

Finally, v is left as the result of the function.

Now let us consider the control function which will be responsible for organising the "rote or rule" decision in memo functions. It is defined by

```
function control x f a;
vars u;
     a(x) → u;
     if u = undef  then  f(x) → u;  u → a(x);  close
     u
end
```

When this function is used, x will be the argument of a memo function, f will be the "rule" and a will be a thing array, which serves as the "rote", or dictionary. In the first statement of this function, a is consulted to see if the result has been memorised (i.e. is already in the dictionary). If the result, u, is undef (undefined), then f(x) is evaluated and immediately assigned to u, so that in any case, u is the result of control. Note that when u has acquired its value from f(x) (i.e. by "rule"), this value is put into the dictionary by u → a(x).

With this apparatus, we can now define memo functions. The function which actually constructs memo functions is:

```
function newmemo (f, n, equiv);
vars u v;
     newassocfn (n, equiv) → u;
     control /
```

```
        control (% f, u %) → v;
        updater (u) → updater (v);
        v
    end
```

Suppose we have a function f. Then newmemo (f, 100, nonop =) → f; will make f into a memo function with a dictionary of size 100. The first statement of newmemo produces a dictionary of the required size, and holds it in the variable u. Next, the actual memo function is constructed by partially applying control to the "rule" f , and the "rote" u , and finally, the update part of the new memo function is added by taking simply the update part of u . It should be noted that this provides no protection against explicit updates of the memo function being forgotten. The user might wish to provide this for himself, in a case where assignment of a value from the outside world carries more "authority" than where the value has been obtained by calculation. For example, a board state in a game can receive a value

(a) by rule (e.g. a Samuel-type scoring polynomial)
(b) by rote (boardstate encountered and evaluated before)
(c) by umpire (e.g. a ruling that the position is lost, or illegal).

In the last case, absolute protection against forgetting is desirable.

In other circumstances, an assignment might represent an experimental observation in the outside world, subject to error. No need for protection would then arise, and indeed a smoothing process might be required as discussed later.

Experimental results

Memo functions involve time penalties,

(1) in the dictionary lookup procedure,

(2) in the additional function entries involved, and

(3) in taking the rote-versus-rule decision.

Obviously for functions which are quick to calculate by rule no gain would be expected, perhaps even substantial losses. At what stage do memo functions begin to pay off?

The usefulness of memo functions in speeding up calculations was tested using the function prime which finds the n th prime. It was defined as:-

```
function prime n;
vars u;
    if n = 1 then 2 exit
    prime(n-1) → u;
lo: u+1 → u;
    if isprime (u) then u exit
    goto lo
end
```

```
function isprime n;                        true
vars u v; 2 → u;
lo: if u*u > n then [crossed out] exit
    if (n//u → v) = 0 then false exit
    u + 1 → u; goto lo end
```

Two sets of arguments were used, one normally distributed around a mean of 25 with standard deviation 5, and the other uniformly distributed in the range 0 - 50. The results are shown in Figs. 2 and 3. A version of newmemo was used which made it possible to specify where new items were to be placed in the dictionary. The results shown in Fig. 2 indicate that a position one fifth of the way down offers the best performance when the dictionary /

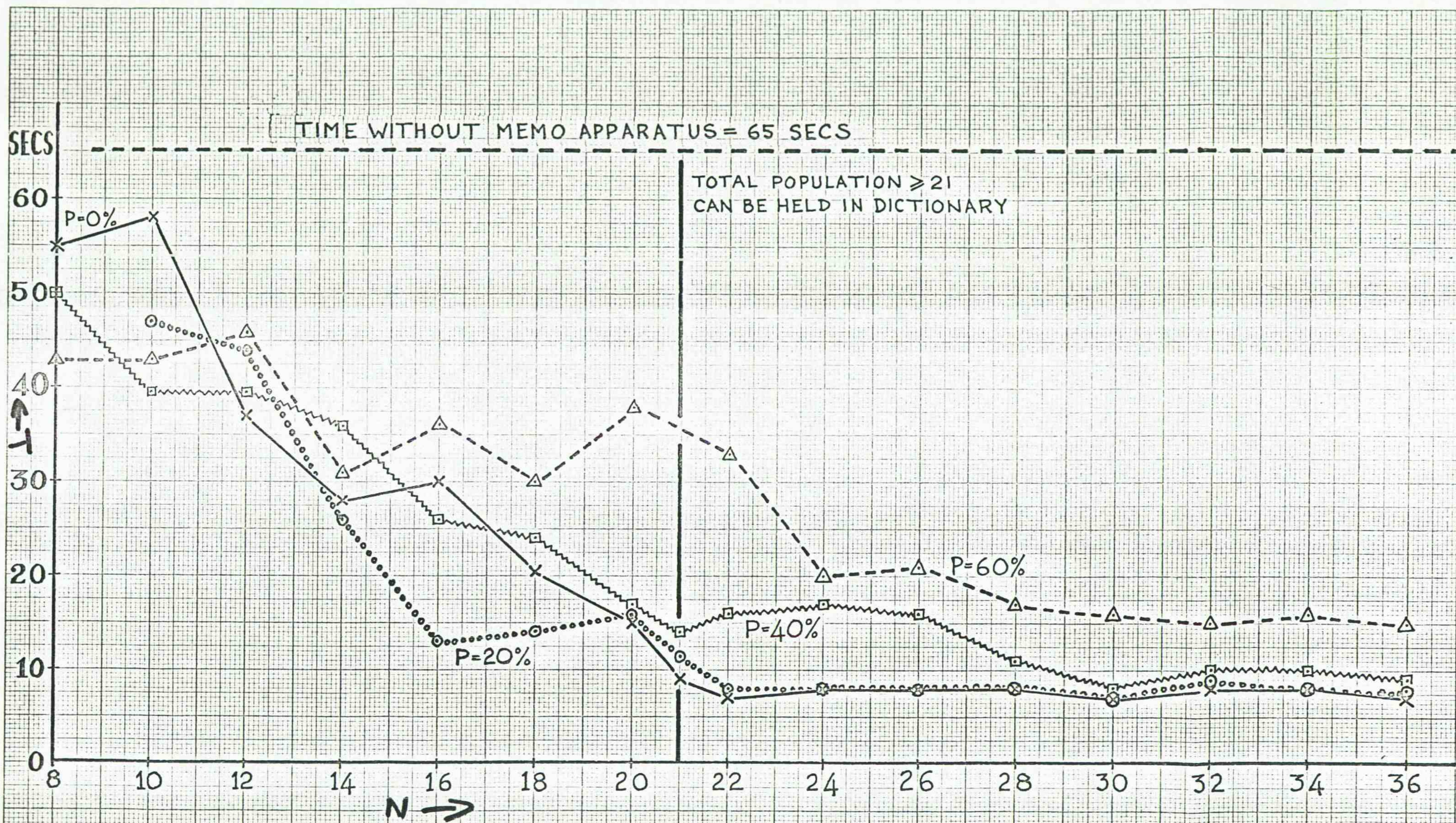
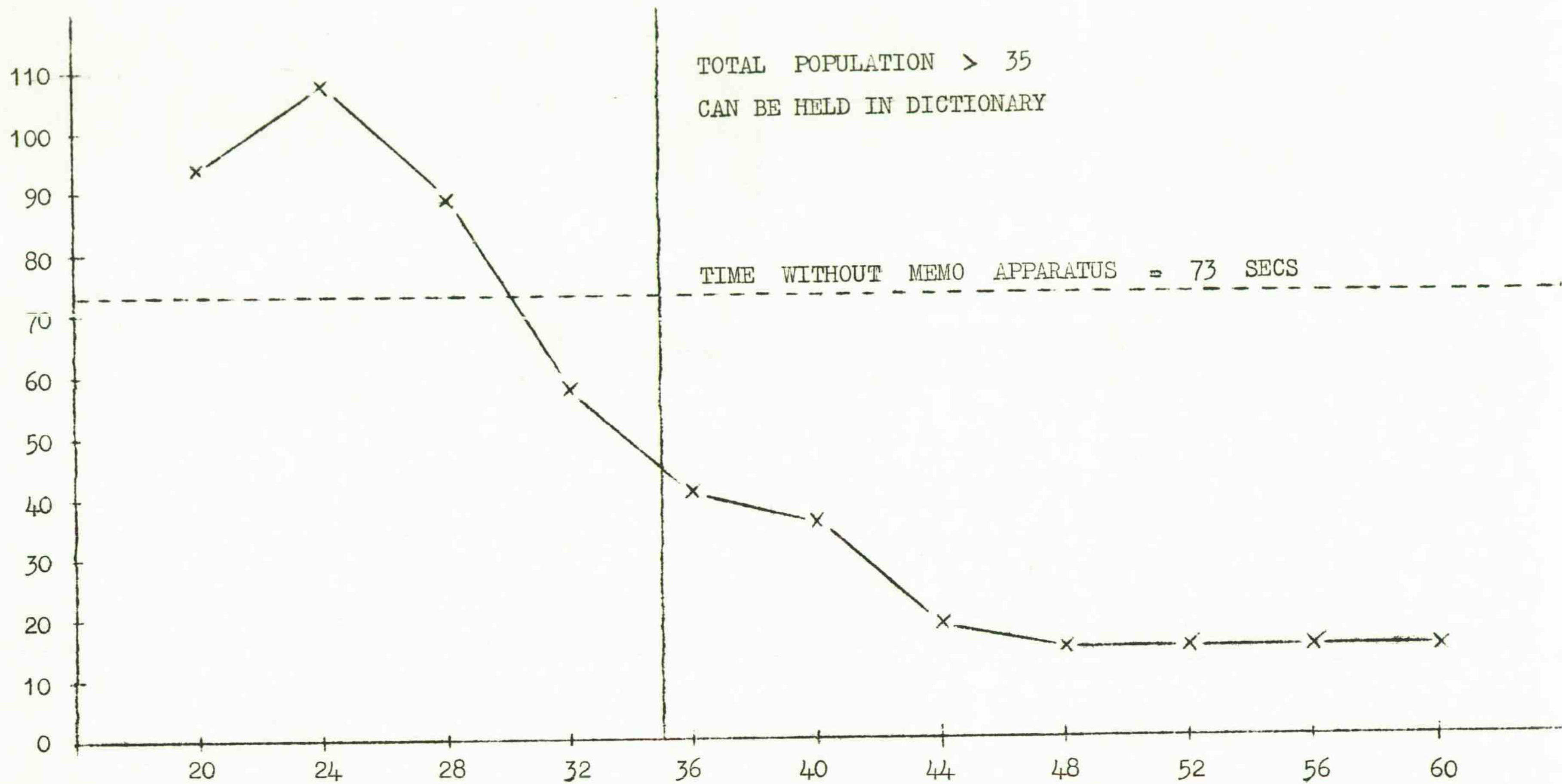


Fig. 2. Time of evaluation (T) plotted against dictionary size (N) for differing values of the position (P) of insertion in the dictionary. Normally distributed population of arguments.



<u>Fig. 3</u> Time of evaluation (T) plotted against dictionary size (N). P = 0. Uniformly distributed population of arguments.

dictionary is too small to hold all the arguments, but that with a dictionary large enough to optimize performance the best place to put new items is at the top. The Figures show the time taken to evaluate the first 100 arguments. Evaluation of the next 100 was consistently done in about one third of these times for dictionary sizes in the neighbourhood of the optimum. No further significant "learning" occurred thereafter.

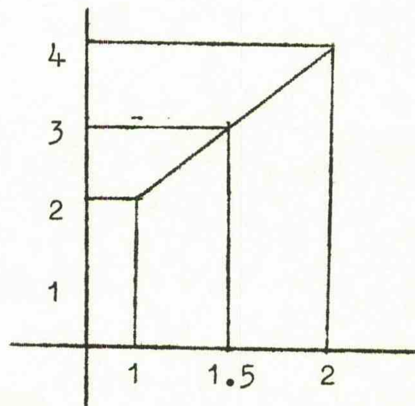
In summary, after an "experience" of 200 evaluations of prime, speeds using memo functions were about 10 times faster than by ordinary function call for the uniformly distributed sample and about 20 times for the normally distributed sample.

Learning with generalisation

An array is a simple example of a doublet. If A is an array, then $4 \to \underline{A}(2)$ causes the system to "remember" that A(2) is 4. One could also say that it has "learnt" that $\underline{A}(2)$ is 4. Such learning is of the form that Samuel calls "rote learning", or learning without generalisation. Suppose times is a 10x10 array initialised to undef in each of its elements. One can "teach" the system its multiplication tables by such statements as $20 \to$ times (4, 5); this would require precisely 100 statements. An intelligent child could cut this by nearly half by observing that times (A,B) = times (B,A); and if it knew about addition, still more by observing that times (A + 1, B) = times (A,B) + B. This is learning with generalisation.

Can programs generalise? In the case of functions of real numbers (or more generally, over real n-space), the process of interpolation provides this ability.

Graph of A



Suppose A is a doublet which we shall call an interpolator. If we make the statements $2 \to \underline{A}(1)$; and $4 \to \underline{A}(2)$; and then ask for $\underline{A}(1.5)$ it will "guess" the answer 3 by linear interpolation. If we now say $3.5 \to \underline{A}(1.5)$, it will take account of this fact, using any of the standard methods of interpolation. Further update commands would be incorporated in the formula, or ignored, depending on whether interpolation produces the assigned value to within a specified tolerance.

Applications of interpolating doublets

Consider a computer simulated automaton like the Doran automaton (Doran 1968). This creature inhabits a universe which at any time is in a particular state. The automaton has sense organs represented by the function perceive which convert the state of the universe into an internal perceived state. In general, the state, or "god's eye view" will contain more information than the perceived state or "automaton's eye view". There is a function sensation : perceived state → real which measures the desirability of the automaton's surroundings on a pain-pleasure axis. (I shall use the mathematical convention f: A → B to mean that the function f has domain A and range B). The automaton has a set of actions which it can take. I shall presume this set to be finite. The automaton aims to choose its actions in order to maximise its pleasure as measured by sense. In order to do so it must have some way of analysing the probable consequences of its actions. Accordingly, it has a function predict : action × perceived state → perceived state list which when given particular perceived state predicts what possible perceived states can occur at the next time instant if a particular action is taken, the predicted perceived states being in decreasing order of probability. predict is in fact a doublet, and far from being infallible, is improved as the automaton gains experience of the world. Suppose for instance that the automaton takes an action a when the perceived state is ps1. The result of this action is that the automaton has a new perceived state ps2 . The automaton memorises this by making the assignment

ps2 → predict(a, ps1);

How does the automaton actually make the choice of an action? It uses a process of lookahead followed by some process of "backing up" calculated outcome values, such as by expectimaxing (Michie and Chambers 1968). It uses predict to estimate the possible perceived states which will stem from the actions it can take, and then it selects that action which will maximise its expected outcome value. The primitive sensation function is not used to measure the desirability of a state. This is the job of the function /

function <u>eval</u> : <u>perceived state</u> → <u>real</u> which uses <u>sensation</u> and the lookahead tree to attach a value to the current perceived state. <u>eval</u> could of course be made a memo function with considerable profit, as has in effect been done by Samuel (1959) in the game-playing context.

Suppose an automaton as described above were given the task of balancing a pole on a cart by unassisted trial and error (the task treated by Michie and Chambers, 1968). <u>perceived states</u> in this problem are quadruples $(x, \dot{x}, \theta, \dot{\theta})$ of real numbers, expressing the position and velocity of the cart, and the angle and angular velocity of the pole. The actions are <u>left</u> and <u>right</u>. The function <u>sensation</u> returns a uniformly high value when the variables are in the permitted region of space, and a uniformly low value when they are outside that region - that is, when the automaton has dropped the pole or crashed the cart. Now the function <u>predict</u> has one discrete argument, namely the <u>action</u>, and one continuous argument, namely the <u>perceived state</u> and so cannot itself be an interpolating doublet. It can however be built out of two such doublets, say <u>predleft</u> and <u>predright</u> which predict the consequences of a left and right action respectively. A memo function for <u>eval</u> should be built with interpolating doublets rather than thing arrays. Such a mechanism would be capable of going beyond the pure rote-learning approach of Michie and Chambers and constructing a cause-and-effect model of its problem environment: at the same time a means is provided of "generalising" about local regions of state space more flexible and less artificial than the "boxes" procedure used by these authors.

Acknowledgements

I wish to acknowledge the help of Professor D. Michie in preparing this paper and also the Science Research Council for financial support.

References

Burstall, R.M. and Popplestone, R.J. (1968) The POP-2 reference manual. Machine Intelligence 2, (eds. E. Dale and D. Michie) Edinburgh: Oliver and Boyd, pp. 207-246.

McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P. and Levin M.I. (1962) LISP 1.5 Programming Manual. M.I.T. Press: Massachusetts Institute of Technology, Cambridge, Massachusetts.

Barron, D.W., Buxton, J.N., Hartley, D.F., Nixon, E. and Strachey, C. (1963) The main features of CPL. Computer Journal, 6, 134-143.

Doran, J.E. (1968) Experiments with a pleasure seeking automaton. Machine Intelligence 3, (ed. D. Michie) Edinburgh University Press (in press).

Michie, D. and Chambers, R.A. (1968) BOXES: an experiment in adaptive control. Machine Intelligence 2, (eds. E. Dale and D. Michie) Edinburgh: Oliver and Boyd, pp. 137-152.

Strachey, C. CPL Reference Manual. Privately circulated.

6693
SGP/S

11th March, 1968.

Professor John McCarthy,
Computer Science Department,
Stanford University,
Stanford, California 94305,
U.S.A.

Dear John,

Many thanks for your letter of January 29th. I have looked up the articles by Fredkin and Pivart and Finklestein in "The Programming Language Lisp ......". As you say, it is concerned with induction on sequences. I found it interesting, and there are some nice things which I would like to loot and tack on to memo functions.

I like your generalization very much. We must certainly incorporate the facility which you suggest: assignment to a function over a class of its arguments defined by a predicate (I hope that I have understood your idea correctly). I quite agree about hash tables. Until POP-2 has hash facilities we will probably be too lazy to do anything about it.

I am favourably struck by your further idea about automatic re-definition of a function after it has been modified several times. Clearly there is a substantial project here. I am also interested by what you say about the use of sub-expressions in simplification; but if the "simplify" function is given an appropriate recursive definition then my "rule and rote" scheme should automatically provide for this. As for equivalence classes, this is handled in memo functions by the function equiv use for rote look-up. I have an article about this sort of thing which is in press for *Nature*.

Again, thank you for your interesting and encouraging letter.

Best wishes,

Yours ever,

DONALD MICHIE.

S.

STANFORD UNIVERSITY
STANFORD, CALIFORNIA 94305

COMPUTER SCIENCE DEPARTMENT

Telephone:
415-321-2300

January 29, 1968

Professor Donald Michie
Department of Machine Intelligence
and Perception
University of Edinburgh
Hope Park Square
Meadow Lane
Edinburgh 8, Scotland

Dear Donald:

Thanks for your paper, "Memo functions...". In the course of getting the stuff off my desk for filing, I picked it up and read it sketchily. I have the following comments:

1. Fredkin or someone else at III did some similar things within LISP, especially induction of a function from examples. What they did is written up in the book, "The Programming Language LISP: Its Operation and Applications". I think it is now available from the Office of Technical Services, U.S. Department of Commerce, but we have a few spare copies if you don't have it.

2. I agree that such a facility is potentially quite useful. I say potentially because it seems to me that it belongs in the general category of useful parts of a large system, and it seems to me that these are useful mainly in a stable time-sharing environment where the system is well debugged and well documented so that the facilities will be easy to learn to use and will stay around for a while. There must also be a large readily usable library of them. I expect that may such facilities will develop in our PDP-6 system when it stablizes (next month?). However, your POP2 machine may be too small.

3. Let me suggest a generalization. One often wants to change a function not by adding individual cases, but by adding special classes of cases. This corresponds to putting on the front of a conditional expression definition a clause of the form, "<u>if</u> p <u>then</u> e". In fact your individual cases could be regarded as putting on the front, "<u>if</u> x = a <u>then</u> b <u>else</u>", although to do it this way would be inefficient in both space and time.

4. POP's implementation in terms of association lists can be quite inefficient compared to an implementation in terms of hash tables. As I believe I told you, I think that languages like LISP and POP2 should have hash facilities added to them in a general way.

See attached notes of Rod.

5. After a function has been modified several times it might be worthwhile to have a program that looks at the new definition and tries to find a more efficient way of expressing it either in space or time. To do a really good job, however, such a function needs to have some usage statistics.

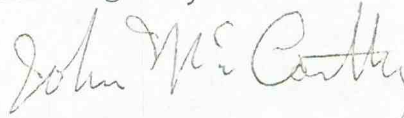
6. Here are two examples of the usefulness of such an approach to functions. First, a simplification program for algebraic expressions even in the course of simplifying a single large expression often encounters subexpressions that it has already simplified. Even a simple list of already simplified expressions helps here and would in your scheme be included in an altered definition of the simplify function. I experimented with this using MAC several years ago, using a simple list, although even at the time I believed that a hash scheme would be better.

More recently people in our group have been working on a program for the game of sprouts. The program probably won't even match the effectiveness of human beings unless it can remember positions it has already evaluated. Unfortunately, it may even be neceesary for it to remember equivalence classes of positions.

7. I mostly take it back about the III work. They were mainly interested in the problem of inducing function definitions. I think you will find it relevant, however.

As you see I found your paper interesting and stimulating.

Best regards,



John McCarthy
Professor of Computer Science

McC/jm