SOME PRINCIPLES OF ARTIFICIAL LEARNING THAT HAVE EMERGED FROM EXAMPLES

John Knapman

Department of Artificial Intelligence
University of Edinburgh.

Abstract

We argue that A.I. should not lose sight of the need for general principles when working with problems in specific domains. We also argue the case for studying Artificial Learning. We present a computer program with a very limited sense modality that acquires some facility with the English language and learns some numerical concepts. The principles by which such learning takes place are expressed in terms of a concept of process and they prove to be applicable to learning the decimal numeral system and forming elementary utterances as well as learning to interpret English sentences.

1. Introduction

Being convinced of the central importance of learning in intelligent behaviour and believing that really complex "performance" programs will be exceedingly difficult to write and even harder to extend by any other means, we have been addressing the problem of Artificial Learning (A.L.). The term "artificial" is intended to carry connotations similar to those in "Artificial Intelligence". The investigation is to be informed by what we think we know about natural intelligence and its development but the computer program which is one of the products of the study will depart from the psychological paradigm whenever this is expedient.

The two alternative approaches to A.L. that we have considered are to increment a performing system that already embodies some knowledge specifically relevant to the material being learned or, on the other hand, to set up a program endowed with the minimum of sensori-motor capability and the skills needed for expansion. Since the incremented program must possess such skills in any case, the naive entity is inherently simpler to build. What is more, it becomes in time a performing system that can be incremented, thus subsuming the alternative approach.

The lesson that A.I. has learned since 1960 (1) is that fruitful research is most likely by considering specific domains in detail. This point of view is elaborated by Minsky and Papert (2). That does not mean, however, that we should abandon altogether the search for general principles, only that success is more likely by generalising from some thoroughly worked particular examples. The need for generality is particularly obvious in learning, since a system that learns only a few well-chosen things may be superficially less impressive that a well written performance program. The methods employed by the learning program must, as far as possible, be made non-specific to the examples chosen. Our approach differs from some (e.g. (3)) especially in that our program is not given express goals to which a plan can be related logically. Such would be inappropriate to a naive entity (apart from general goals like "associate symbol with meaning" or "carry out imperatives" which are implicit). Rather it reacts to its environment, including feedback from its own actions, laying down patterns of behaviour for future use and retrieving them according to a suitably abstracted representation of what was perceived. The latter is normally a part of the pattern of behaviour.

The aspects of intelligence which are examined in the present work involve a dialogue conducted in English and also some numerical concepts. The program acquires both semantic and syntactic knowledge of some words, which it subsequently uses and "comprehends". It also acquires the meanings of the numerals in the decimal Arabic system, using the same basic repertoire of abilities even though numerals are not conventionally considered to be part of a natural language.

The domain in which these activities take place has been selected for the utmost technical sinpli-city. The program reads from and writes to a teletypewriter where a person is sitting. The dialogue is concerned with printing characters, including letters that make up words and sentences. So it is also possible to discuss what is being said; no unnatural separation arises between speech and other activities.

Beginning with no vocabulary and only the general abilities described in section 2 below, the program acquires sufficient knowledge to engage in a dialogue of which the following is an annotated sample. The convention is that lines entered by the human tutor are preceded by a colon; the others are produced by the program.

```
: PRINT THREE ASTERISKS AND A DOT
  *** .
: WHAT DID YOU PRINT BEFORE YOU PRINTED A DOT
  THREE ASTERISK                    (cf. (4))
: WHAT DID YOU SAY
  THREE ASTERISK
: SAY TRIPLE AFTER I PRINT THREE CHARACTERS
```

The word "triple" is new to the program. "After" is like "when" in this context but in the domain in use there is no simultaneity or ambiguous temporal proximity and so "after" is more appropriate. A kind of "demon" has been set up to await the specified condition. These demons are not triggered by goals like those of Papert (5) but by events as they can be characterised in the program's representation. If, later, we enter a group of three characters, the desired result occurs.

```
  TRIPLE
```

The program learns the rudiments of the decimal number system so that it can do addition, subtraction and multiplication with two-place numerals,

```
: WHAT ARE 6 12S
  72
```

The matter is discussed briefly in section 3 . Before that we describe the principles on which this work is based and then present a dialogue demonstrating a complete learning sequence. At the

time of writing, the program to carry out this dialogue has been checked by hand and is being computer-tested.

The principles by which the program lays down patterns of behaviour can be expressed in terms of process and some primitive operations on procedures. It is to be hoped that they will prove to be applicable or extendable to domains other than the simple teletypewriter situation.

2. Principles

We begin by distinguishing between a procedure and a process (one of the distinctions drawn by Newell (6)) and use these concepts to define constructs of memory. After that we show how these are used to discern patterns of repetition in events and to associate words and other symbols with their meanings in a constructive way.

2.1 Processes and Procedures

A procedure consists of a sequence of instructions in a text. We allow the text of a procedure to be examined and modified by other procedures and for this reason the text is maintained in the form of a list, an idea due to Newell and Simon (7). Procedures can call other procedures either by referring in a call instruction to a variable that contains a procedure or by referring in such an instruction directly to a procedure (using an address pointer). In the latter case, we term the called procedure a subroutine constant. We apply the same term to any procedure that is directly addressed by another, whether it is called (i,e, executed) or not.

A process is the execution of one or more Procedures. The programming system (8) builds a data structure to control a process. The execution *of a* procedure is governed by an activation record which points to the procedure and to the next instruction to be executed. It also contains space for the contents of its variables and points to the activation record for the process that invoked it. Finally, it contains provision for non-local variables. (Dynamic binding, as in LISP (9) is the default.)

We want to be able to do several things with processes and the most important is to save them for later use. This happens to the caller whenever one procedure calls another but it can also happen on other occasions. If during a process some vital piece of information is missing, that process can be suspended until the information is found. This idea is used for storing the meanings of words: when a new word is encountered a procedure is synthesised and called - that process then suspends itself to wait until the word occurs again.

Something that can be done with both processes and procedures is to activate them. But when it comes to inspecting them they are quite distinct. A process contains procedures as well as other processes; procedures do not contain processes. A process may be accessed and modified; the value of its variables can be obtained and updated using a primitive VALUE. These variables may refer to procedures and processes. The activation record for the process may be accessed to discover which procedure is in execution and to link back along the chain of invocation

From the current process, the entire state of the program can be ascertained. The structure into which all the processes are woven is called the run time structure.

2.2 Short-term Memory

The run time structure is the store for the first of our two memory constructs: short-term memory (S.T.M.). It holds the run time structure for the few most recent events, discarding the earliest by the criteria of storage space and search time. This seems preferable to lifting something arbitrarily out of psychological theory as, for instance, regulating its size according to Miller's celebrated "magic number 7 +_ 2" (10).

The crucial question to ask of a memory is not what is stored or how, but how it is retrieved. Retrieval keys are procedural text; when presented with a procedure, the STM interface will search the run time structure for a match to it. All processes, including those referenced only by variables, are included. Their execution procedures and their variables' contents are included as are all subroutine constants. Since the STM is of limited size, efficient performance is not a fundamental problem although it would obviously be nice to have some sort of content-addressable memory device.

Four types of match are allowed when retrieving from the memory,

I Exact line for line equivalence.
II The argument is found completely within a procedure: the parts preceding and following the common portion are supplied with the result.
III Exact match apart from a differing subroutine constant in corresponding position: this is indicated with the result.
IV Any of the above I-III with a subroutine constant of a procedure.

The result of I is a reference to a process (i.e. a variable name or execution indicator and an activation record). In the other, more complex, cases results are chained together and always conclude with a process reference.

No problem arises with establishing the equivalence of different variable names during comparisons because in this work all the procedures used against STM are synthesised by the program and obey suitable conventions.

The normal event when interpreting a word is for an operation on STM to take place yielding a result which can then be made available for others to use by leaving it in the run time structure in an execution of a special procedure named RESULT. A common noun like "dot" initiates an STM search with argument [.] and passes the result which can be any of the possibilities I-IV. We often write procedures in square parentheses and always omit print instructions for easier reading. More elaborate words and morphemes generally expect results of these various types, as well as yielding further ones in turn; their expectations are conditioned by the circumstances prevailing at the time they were initially acquired or subsequently modified.

The following is an illustration.
: *,AN ASTERISK AND A COMMA
The phrase "an asterisk" locates [*] within t*,l. This is a type II match and the result is a reference to the procedure [*,] showing that [,") was the part following the common portion.

Suppose that the conjunctive particle "and" is a new word to the program at this point. The procedure for handling new words (known as NEWENTITY) is called and it sets up code for "and" in future that will expect a type II result of this sort from the left. Interpretation of "a comma" yields as result a reference to the variable containing [,] within the current process (for "and")itself and again code to verify this in future is incorporated into "and". The verification is constructive, which means that in a similar phrase in future [*,] will be re-assembled by "and". This step is essential to make imperatives possible.

A word can impose restrictions on the processes it will accept from an STM look-up. For instance, the morpheme "-ed", once it has been acquired, insists on a process prior to the present time. "After" demands a similar relationship between the two processes referred to by the two clauses in the sentence. Apart from specifying its relationship to other processes, one can also assert that the process found must be the execution of a given procedure and that the matching procedure must be in a particular variable.

2.3 Grammar and the Result Mechanism

Before enumerating the possible combinations of occurrence and expectation we must discuss the result mechanism and its purpose. The input stream is being scanned linearly (left to right, see section 2.4), just as human speech is also one-dimensional in substance. Grammar imposes a second dimension so that a sentence is segmented into units of lesser rank (11). The embodiment of this phenomenon in our program is the ability of some word-processes to require (by means of a SEEK) to interpret the following words in the sentence before giving their results and of other word-processes to take results from the interpretation just conducted on their left (with the option of closing an outstanding SEEK).

As an example consider the sentence: "A dot is what you printed." The indefinite article seeks the interpretation of its direct object, the following word. "Dot" will search STM and give a result, the details of which depend on whether and how [.] can be found. The verb "to be" in the third person looks for a result on the left and, expecting a procedure, will extract [.1 from whatever form the result may take. "Is" also requires a result from the right, the complement, and seeks accordingly. The complement is the clause "what you printed". Each of these words seeks to the right, specifying certain attributes (or none) of the process to be located in STM. The last word, although it looks for one, has no direct object. So the verifying STM call that it contains becomes, by default, a search for the example of printing most recently referred to which will, if the statement is true, be found first. This is because it will already have been referenced during the processing for "dot".

In an example such as "3 and A", 3 will seek for a direct object in case a noun, e.g. "dots", is following. "And" demands a result from the left and so causes the interpretation of 3 to continue without an object, thus giving the answer "£££". (This is a case of closing an outstanding SEEK.)

During interpretation a word will expect either one or two results, depending on how many it received initially. Words that expect only one are similar whether that one comes from the right or the left. If case I (referring to section 2.2) is expected, process-type conditions will be asserted and verified with no result being passed on if they are not satisfied. In the other cases, specific procedures (e.g. "CALL [3; COPY" for "two") will be expected in the proper textual relationships and if these are not forthcoming and a renewed scan of STM fails to realise them, a resolving routine is called. Another such routine is called if there is no result at all when one is expected. A little more is said about them in section 2.7.

At the present time, processes that expect two results must take one from each side although we hope to lift that restriction later. The left-hand result can be any of the usual cases I-IV. On the right, however, it must be a process (possibly the current process) since only then can any kind of relationship be established between left and right. For example, in : "A dot is a character", the word "is" will extract from the left the procedure [.] and "character" will find it during its STM look-up, returning as result a reference to the process for "is" itself. If a procedure is expected on the left, one will always be extracted from any of the cases I-IV. If no result appears, a resolving routine is called. It is also called if either of the expected processes does not materialise, whether or not a procedure has occurred instead.

2.4 Long-Term Memory

We turn now to the second memory construct which, not surprisingly, is called long-term memory (L.T.M.). We are not concerned here with a historical record of events from which a verbal account might be constructed but with something much more primitive for the purpose of recognising what has previously been seen and for acting out patterns of behaviour that have been laid down by experience.

The basis of matching in LTM is also on procedural text. The store contains a set of suspended processes, each waiting for the occurrence of a particular procedure. The most common example is that of a process for interpreting a word waiting for the occurrence of that word (in procedural form)■

One conceivable convention for retrieving and activating a process in LTM is to make it automatic upon the appearance of a matching procedure in the run time structure. A more convenient method, and the one adopted here, is explicitly to drive LTM retrieval at strategic times by directing it to match exhaustively every part of a procedure referred to in a particular process. The principal application is the analysis of input (i.e. sensory data) which proceeds in this fashion. The more general scheme could be implemented simply by applying this drive-LTM primitive every time a procedure is synthesised.

By observing a left to right, and for subroutine constants top down, discipline in driving, a straightforward indexing scheme in LTM is possible. In this convention, the record retrieved on each occasion is the one whose key has the longest match with the left-hand end of the source procedure, after which the source is truncated and recursion takes place until it is exhausted.

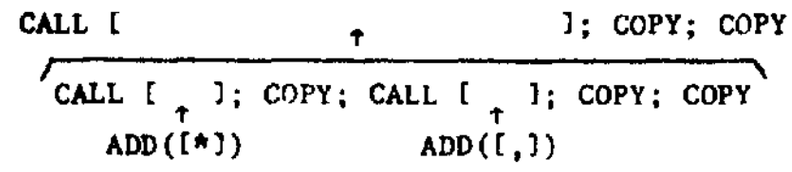
2.5 Perceiving Patterns

The program refers to LTM in order to segment data (presented through the teletypewriter) into familiar units, typically words or other symbols and those characters treated as objects because of past association (e.g. "*"). This is basically the process of driving LTM but for convenience the retrieval and activation stages have been separated, at least for the time being. Before it is possible to refer to LTM the program converts the data into the form of a procedure for printing it.

As the segmentation of data takes place, STM is used in order to detect any repetition that may be present. If the entity currently being scrutinised has occurred earlier in the data stream, the program hypothesises that the items intervening between the present and the most recent occurrence are about to recur and tests the hypothesis on as many subsequent items as it will obtain.

This process of recognition we call procedural abstraction because the end product is a hierarchically structured procedure that will textually recreate the original. For example, the procedural abstraction for "DOT" is "ADD(IDOT))" where ADD is the primitive for appending a procedure to the one that is currently being synthesised and "[DOT}" represents a subroutine constant that would, if it were executed, print the word "DOT". ADD is one of a number of primitives introduced in (12). When a repetition is detected, the unit to be repeated is placed as a subroutine constant in a procedure that will call it and copy the result the correct number of times. For instance, "****"will be represented as "CALL CADD([*])J; COPY; COPY" where COPY is defined in terms of ADD; a few technicalities have been omitted. This procedure ia now in a convenient form to be verbalised as "THREE ASTERISK" by methods shortly to be described.

Repetitions can be detected at any level, not merely at the lowest. For example, the sequence ",,,**,,,**,,," would be represented thus:-**



It might be verbalised: "Two asterisks and three commas three times" although such an utterance would be beyond the capacity of this program to write (as well as being ambiguous).

Once an abstraction has been made from a line of input, it is used to drive LTM. The associated processes are retrieved from LTM and activated. The reason for using the abstracted version rather than the raw input is to allow patterns to be perceived and interpreted as such, so that "..." ia immediately seen as "three dots" rather than as "dot dot dot". In some languages, the plural is formed by repeating the noun; our program readily copes with that form by using the abstraction principle.

2.6 STM-LTM Complementarity

There is an interesting and perhaps surprising complementarity between the two memory constructs. Invoking the STM interface involves searching the near past. A process that calls the LTM interface is suspended and stored until the specified procedural argument occurs. Thus when LTM is invoked it is effectively the future that is being searched.

The two interfaces are uniform so that they can be interchanged in a procedure. This fact is used to great advantage in the present work where a process waiting in LTM for a word and then looking up its meaning in STM can use the same procedure as the one that waits in LTM for the meaning to look up the word in STM. This means that extending the meaning of a word in comprehension will automatically extend its meaning in utterance because the same principles govern the interpretation of objects that are not synbols as govern those that are. The examples of patterns given in section 2.5 can be used to drive LTM just as words can. Indeed, often both are mixed within one line.

We can explain more fully by considering the example of the numeral 2. To begin with, it is associated with " " . That means that a procedure was created and initiated. The process then saved itself in LTM with the key "[2]". Note that the LTM and STM arguments are called key and signature, respectively. When 2 is encountered, that process is restarted and it searches STM with signature "[ff]". The same procedure with its variables reassigned is waiting with key "[££]" and signature "[2] (it is written in a recursive loop to make both processes possible). Now it is taught that 2 can also qualify a noun as in: "**2 ASTERISKS". The program appends a more complex procedure that includes a SEEK and has the signature "CALL[ ]; COPY". This addition necessitates the creating of one more process in LTM with this as key and "[2]" as signature. Then the program could reply: "2 dot" to a question such as: "What did you print?".

For this purpose, the procedure must include code to expect results (e.g. the word "dots") and put "2" in front of it. This code complements the SEEK in the version where "2" is the key. There are, in fact, general rules for deriving such complementary code although they will not be given here. It is arranged so that in each case the complementary code is not executed. (The appended procedure recurses to the original procedure for this purpose.)

2.7 How procedures are extended

In section 2.3 we referred to the necessity for resolving routines. If a word is expecting to be passed a process reference from the adjacent phrase and the result it receives is procedural that means that some information in that procedure has not been explained and a word must be modified to take account of it. Except where the receiving word is two-sided, it is the sending word that must be extended. A procedure known as RESOLVE sets up a conditional call in the word-procedure to some new

code which is composed along similar lines to the initial syntheses performed by NEWENTITY (mentioned in section 2.2).

The second resolving routine, known as XRESOLVE, (X for contradiction) is called when an STM look-up fails. XRESOLVE ends up doing the same as RESOLVE if it is unable to use its own specialities which are as follows. Very often, the procedure that has called XRESOLVE will contain a subroutine constant, known as the signature, that is the basic meaning of the word. E.g. the signature of "dot" is [.]. One course of action is to allow XRESOLVE to create another procedure with its own signature and then extract the common portion of the two signatures and adopt this common portion as the new signature. This is used for class words like "character" and "numeral" (the numerals 1 to 9 all contain "£"; 0 is a special case and is added on by replacing the XRESOLVE call with a new procedure).

Special provision is made in XRESOLVE for imperatives. The procedure for a word like "print" specifies to STM that the matching process must be an execution of the search argument and subordinate to (i.e. invoked directly or indirectly by) the current process. This is clearly the most natural way to say "do it" and XRESOLVE responds by supplying a pending process, i.e. an activation record in an initial (unexecuted) state. This is entirely consistent with other type I results. If it survives the processing of the current line of input it will actually be performed by the program. For example, in : "Print four dots", "four" expects case IV, i.e. [.] as a subroutine constant in "CALL C 1; COPY; COPY; COPY" so it constructs it and since both verification and look-up in STM fail, the result is passed on by XRESOLVE.

There are cases where a process result may have some attributes different from those expected by a word. An interesting case arises when a pending process (imperative) is received by a word expecting the execution of another, specific, procedure. Then it is taken to refer to the future and the receiving process suspends itself by calling LTM with the imperative procedure as key. Thus if the human tutor types in: "I print an asterisk" (e.g. in: "Do something after I ...") then the pronoun "I" dictates that the match must be found in a reading process and that proviso is inconsistent with an imperative interpretation. So it waits until the procedure does turn up during a read.

3. Specimen Dialogue

A convention in some examples is that the program can be made to perform an action by giving it a procedure enclosed in parentheses. Since in every case given here the supplied procedure consists of print instructions, these are omitted. The program works by converting all input to the form of a procedure that would print it (analagous to the proprioceptive theory of speech perception (13)). This results in an uniformity of representation with internally synthesised procedures which is quite advantageous.

We familiarise the program with items in its domain by showing it some. After that it is possible to teach the names, distinguished from the objects by their unfamiliarity.

```
:*ASTERISK
:*AN ASTERISK
```

When the unfamiliar "ASTERISK" is encountered, a procedure is eynthesised that will be activated when the word is encountered again. This procedure is also associated with "*" and there are two corresponding processes waiting in LTM, both of which share this procedure.

In the next set of examples, the verbs "to print" and "to say" will be introduced in the imperative, using the device of indicating the appropriate action in square parentheses. The procedure so supplied is executed in a process that has the attributes of the imperative outlined in section 2.7. We continue with the regular past tense morpheme "-ed", the strong form "said", and the personal pronouns "you" and "I".

```
:PRINT AN ASTERISK [*]
 *
The procedure for "asterisk" locates the execution
of [*] in STM and sets up "print" so that in future
it will either verify or bring about the existence
of a process with similar properties.
:SAY COMMA[ COMMA]
  COMMA
The details are different but the principles are
the same.
:YOU PRINT A DOT [.]
 .
The pronoun's meaning consists of its distinction
from "I" which appears later.
:PRINT A SEMICOLON
 ;
:YOU PRINTED A SEMICOLON
:SAY DOT
 DOT
:YOU SAID DOT
:*
:I PRINTED AN ASTERISK
```

The personal pronoun "I" adapts the expectation set up by "printed" to verify that the asterisk was referred to by a reading process instead of having been executed. An alternative method of indicating the past imperfect tense in English is to use the auxiliary verb "to do".

```
:PRINT AN ASTERISK
 *
:YOU DID PRINT AN ASTERISK
```

It is quite easy now to exhibit the temporal relations "before" and "after". Some of the functions of "when" relating to future time can also be carried out by "after", as was indicated in section 1.

```
:PRINT A COMMA

:PRINT A DOT

:YOU PRINTED A COMMA BEFORE YOU PRINTED A DOT
:YOU PRINTED A DOT AFTER YOU PRINTED A COMMA
```

Imperatives and the present tense frequently relate to future events and "after" can work on procedures to be executed just as it does on those already used in the past. For example:

```
:PRINT A DOT AFTER YOU PRINT AN ASTERISK
```

If the condition in the subordinate clause depends, on an external event then the program must wait for it to happen and this is the case in the section 1

examole: "Say triple after I print three characters', It works by exploiting the complementary nature of STM and LTM as was outlined in section 2.7.

Next we deal with the verb "to be" in the third person singular of the present tense and with the pronoun "what" both as a relative and an interrogative pronoun. If we begin by teaching a class word then some reasonable "is" statements can be made.

```
:*CHARACTER
:,CHARACTER
```

After "*", "character" is taken as synonymous with "asterisk" but subsequently the assumption is violated. The common attribute (i.e. printed or printable) is extracted and associated with the word. Now we introduce "is".

```
:A DOT IS A CHARACTER
```

It can be used to construct a simple relative clause.

```
:PRINT A SEMICOLON

:A SEMICOLON IS WHAT YOU PRINTED
```

Winograd (14) noted the computational similarity between the relative and interrogative pronoun. "What" as acquired above will do everything required in the interrogative role apart from the final stage of verbalising. It would be possible to make verbalising spontaneous at such points but the device of supplying the answer in square parentheses the first time is quite sufficient to achieve the desired effect. It also requires a simpler initial program. It can be argued that it bears analogy with guiding a child's hand when teaching him to write but there is no such equivalent, of course, in speech.

```
:WHAT DID YOU PRINT [SEMICOLON}
 SEMICOLON
:PRINT AN ASTERISK
 *
:WHAT DID YOU SAY BEFORE YOU PRINTED AN ASTERISK
 SEMICOLON
:WHAT DID YOU PRINT BEFORE YOU SAID SEMICOLON
 ASTERISK
```

In the final part of this dialogue, numbers and numerals are taught. To begin with, the program must be acquainted with "two-ness", i.e. it must encounter two of something, before it can associate words with it. It immediately recognises repetition and structures its procedural representation hierarchically to reflect the pattern naturally. After the initial instance of two of something, it can learn the plural morpheme "-s" and some numbers.

```
:**ASTERISKS
:,, COMMAS
:... DOTS
:**** ASTERISKS
:.. TWO DOTS
:,,, THREE COMMAS
```

The higher numbers, up to a reasonable level like ten or twenty, can be taught similarly.

The sequence given for learning the plural morpheme is optimal but the processing (using XRESOLVE) is quite robust and an arbitrary sequence of examples which includes these in various orders will eventually lead to the correct synthesis. At the stage of four objects, the resolving procedure notices that the procedure it is appendirg to is textually equal to the new procedure and so it puts in a recursive call instead. That takes care of all cases, of course.

For large numbers the best approach is to use numerals. A method suggested by the use in infant education of blocks that clip together is to associate the numerals with character strings of appropriate length.

```
:£ 1
:££ 2
```

We continue thus up to and including 10, which is learned as an entity and not understood as a true decimal number. Simple addition and subtraction can now be taught.

The program can also learn to understand the numerals in the 6ame syntactic role as the verbal numbers. Because the repetition in "£££" (the meaning of 3) has been separated in a hierarchically structured procedure from the root "E", the same pattern, in e.g. "***", can be recognised and the meaning of 3 generalised to apply the repetition to "£" only when there is no direct object.

```
:*** 3 ASTERISKS
:PRINT 3 DOTS

:WHAT IS 3 2S
 6
```

This has provided a road to multiplication. (We could easily have taught "are", the plural of "is" as well: e.g. "Dots and commas are characters".) The word "times" could have been used if preferred.

We begin teaching the significance of position in decimal numbers by showing that the presence of zero just after a numeral means that the number is applied to ten instead of to one.

```
:TWO 10S ARE 20
:THREE 10S ARE 30
```

This continues up to and including 100. Another complete set of examples must be given to show that any following numeral and not only zero has this effect. Typical of these examples are the following.

```
:THREE 10S AND 1 ARE 31
:THREE 10S AND 2 ARE 32
```

This is sufficient for any number from 31 to 39 to be understood.

Now two-place arithmetic can be done. The notation itself now carries the program with it for many place numbers. The program is very simple-minded in its interpretation, though. Numbers like 478 are expanded out to a string of that length. Teaching it the algorithms for arithmetic in columns would be the next step. That particular problem has already been tackled by Badre (15) although his approach is not that of learning ab initio that is taken here.

There is, of course, a level of abstraction at which the individual takes account of the procedures for interpreting the meanings of large numbers without actually executing them; the symbolism then becomes the vehicle for higher thought. A possible direction for future research is to find out whether the principles described in the present work would apply also to these more complex procedures.

4. Conclusion

There are a number of possible directions that future research could take within the existing domain. We could explore negation and another

topic would be units of time, time of day and extrapolation into past and future. At some stage too, a language other than English ought to be tried to assess the program's potential for linguistic universality.

It is to be hoped that the methods developed here will extend to other domains and an obvious possibility would be to have the program write turtle procedures in LOGO. Goldstein's (16) approach to this problem is quite different from what we are proposing. His philogophy, widely followed by those engaged in automatic programming, is to try to find a way of expressing what a procedure is to achieve so that a system can write it. This is entirely different from setting a program loose with only general learning objectives and then trying to educate or tame it, as it were.

We have not assumed an indifferent environment for the program that has been described although in section 3 we were at pains to point out that learning the meaning of the plural morpheme "-s" was quite a rugged process and it can be said that the program will cope with examples inappropriate to its stage of development simply by ignoring them. Children, of course, are normally encouraged and trained and get things simplified for them but they do seem able to learn even in unfavourable backgrounds. The assumption behind this program is that a good way to work is to ignore everything you don't understand and wait for the unambiguous situation.

Helen Keller's (17) widely quoted description of learning the word "water" aptly illustrates this idea. It was the unambiguous association of the word spelt in one hand with the sensation of cold water on the other than enabled her, at the age of seven, to begin acquiring language.

Our work may be contrasted with Winston's (18). There is no counterpart here to his "near miss" situation. Moreover our procedural represent-ation seems to offer a better chance of generalisation than his several types and relations.

Acknowledgements

I should like to thank Jim Howe and my colleagues at the A.I. Department for criticism tempered with encouragement. Financial support is gratefully acknowledged from IBM United Kingdom Ltd and the Science Research Council.

References

(1) Newell, A., Shaw, J.C. and Simon, H.A. 'A Variety of Intelligent Learning in a General Problem Solver' in Yovitts, M., and Cameron, S. (eds) 'Self-Organising Systems', New York, Pergamon, 1960.

(2) Minsky, M. and Papert, S. 'Artificial Intelli-gence: Progress Report[1]. A.I. Memo 252, Artificial Intelligence Laboratory, M.I.T., 1972. See section 5.2.

(3) Sussman, G.J. 'A Computational Model of Skill Acquisition' (Ph.D. Thesis), AI-TR-297,A.I. Laboratory, M.I.T., 1973.

(A) Brown, R. and Bellugi-Klima, U. 'Three processes in the child's acquisition of syntax* in Bar-Adon, A. and Leopold, W. (eds) 'Child Language', Prentice-Hall, Englewood Cliffs, N.J., 1971. Page 308 "Two shoe" or "two shoes".

(5) Charniak, E. 'Toward a Model of Children's Story Comprehension' (Ph.D. Thesis), AI-TR-266, A.I. Laboratory, M.T.T., 1972.

(6) Newell, A. 'Process-Structure Distinctions in Developmental Psychology' in Farnham-Diggory, S. 'Information Processing in Children*, Academic Press, 1972. He calls it the "material-activity" distinction.

(7) Newell, A. and Simon, H.A. 'The Logic Theory Machine: A Complex Information Processing System', IRE Transactions on Information Theory, Vol. 1 T-2, No. 3, pp. 61-79 (1956).

(8) Knapman, J.M. 'PROCESS 1.5: Description and User's Cuide', Bionics Research Reports: No. 11, Department of Artificial Intelligence, University of Edinburgh, 1973.

(9) McCarthy, J., Abrahans, P.W., Edwards, D.J., Hart, T.P. and Levin, M.I., 'LISP 1.5 Programmer's Manual', M.I.T. Press, 1962.

(10) Miller, G.A. 'The Magical Number seven, plus or minus two: some limits on our capacity for processing information', Psychological Review 62, No. 2, pp. 81-97, (1956).

(11) Halliday, M.A.K. 'Categories of the Theory of Grammar', WORD, Vol. 17, No. 3 (1961).

(12) Knapman, J.M. 'Programs that write programs and know what they are doing', Proc. of the AISB Summer Conference, University of Sussex, Brighton, Sussex, 1974.

(13) Hormann, H. 'Peycholinguistics' translated by Stern, S.S., Springer-Verlag, Berlin 1971, p. 62 ff.

(14) Winograd, T. 'Understanding Natural Language', Edinburgh University Press, 1972, section 8.1.5.

(15) Badre, N.A. 'Computer Learning from English Text' (Ph.D. Thesis), Memo No. ERL-M372, Electronics Research Laboratory, College of Fngineering, University of California, Berkeley (1972).

(16) Goldstein, I. 'Understanding Simple Picture Programs', Proc. of the AISB Summer Conference, University of Sussex, Brighton, Sussex, 1974.

(17) Keller, H., 'The Story of my Life' Hodder and Stoughton, London, 1958 (original publication 1903), Ch IV .

(18) Winston, P.H. 'Learning Structural Descriptions from Examples', (Ph.D. Thesis), AI-TR-231, A.I. Laboratory, M.I.T., 1970.