ARTIFICIAL LEARNING

John Knapman

Ph.D. University of Edinburgh

1977

DECLARATION

The following thesis has been composed by myself and the work described herein is my own.

John Knapman

ABSTRACT

Principles of learning are presented and a program that embodies them is described. It works in a simple domain, involving printing on a teletypewriter and conducting a dialogue about what happens, but the principles are apparently much more general. They are related to psychological notions of long and short term memory. These are defined here in strictly computational terms, based on the ideas of control structure and run-time structure. The learning mechanism is intimately bound up with the nature of the memory constructs. The principles are applied to examples of word meaning, grammar and causality and the program learns to use a relative clause, temporal relations and the past tense, as well as doing some work with number. The relevance to Piaget's theories is discussed. The research has a bearing on previous work in natural language processing, cognition and memory. In particular, the doctrine of procedural representation of knowledge is followed. Objects resembling frames emerge naturally from the way memory is organised. Searches are strictly limited and combinatorial explosions do not arise. The work is seen as an alternative to the customary approaches to automatic programming.

ACKNOWLEDGEMENTS

I am indebted to my supervisor, Jim Howe, for allowing me a very large amount of machine time, and for many other things. I would like especially to thank Richard Young, Ben du Boulay and Mark Adler for reading the draft and many other people for discussions, encouragement and ideas. These include Sylvia Weir, Marylin McLennan, Tim Radford, Betty McLeman, Hugh Noble and Nat Goodman. I am grateful to Patrick Winston for his prompt reply to my letter. I wish to thank Janet Savage for beautiful typing and drawing of diagrams, Jean Parker for administrative help, June Johnson for drawing fig. 1.4, and Robert Rae for making POP_2 work under stress. Financial support from IBM United Kingdom Ltd. and from the Science Research Council is gratefully acknowledged. Finally I thank my wife Moira for everything.

FOREWORD

The chapters are grouped into four parts so that the reader may choose a route through them more readily. After reading part I, it should be possible to read the remaining parts in any order. Part II defines the principles and their implementation in detail. Part III describes the tests, introducing principles as they arise in examples. Some readers have found it preferable to read part III before part II. Part IV shows how this work relates to other research in Artificial Intelligence and explores the question of generality, as well as proposing enhancements to overcome known limitations.

TABLE OF CONTENTS

PART I

Chapter 1	Introduction	1
1.1	Objectives	2
1.2	Exterior view of the program	6
1.3	Underlying Principles	12
1.3.1	The Basis of Learning	12
1.3.2	Purpose of Long Term Memory	16
1.3.3	Approach to Grammar	17
1.3.4	STM and LTM complementary	19
1.3.5	Processes and Run-time Structure	21
1.4	Justification	23
Chapter 2	Relation to Child Development	26
2.1	Memory	26
2.2	Development	31
2.3	Number	35
2.4	Child Language	37

PART II

Chapter 3	Procedures and Processes	42
3.1	Procedures	44
3.1.1	List Programming	44
3.1.2	Procedural Representation of Knowledge	45
3.1.3	Lists are used for subroutines	47
3.2	Processes	48
3.2.1	The Run-time Structure	50
3.2.2	Other points of Comparison	50

Chapter 4 Organisation of Memory 53

4.1 Motivation for the memory organisation 53

4.2 Short Term Memory 54

4.2.1 The STM Interface 55

4.2.1.1 Exact and Partial Match 57

4.2.1.2 Results 58

4.2.1.3 Types of Partial Match 59

4.2.1.4 Searching Nested Subroutines 60

4.2.2 Efficiency 60

4.2.3 The "Subordinate" Relations 61

4.2.3.1 Definition 62

4.2.3.2 Implementation of Constraints during STM search 64

4.2.4 The Imperative Option 64

4.3 Long Term Memory 65

4.3.1 Equivalence to Conditional Statements 66

4.3.2 Precedence and Matching 67

4.3.2.1 Restricted Types of Partial Match 68

4.3.2.2 Repeated Calls to the Interface to unravel Nesting 70

4.3.3 Processes in LTM 71

4.3.3.1 Recollections 72

4.3.4 Organisation and Access 72

4.3.4.1 Definition of Key Format and Lookup Algorithm 74

4.3.4.2 Initiating Retrieval 75

4.3.5 Recursion 76

4.4 Contexts 78

4.4.1 Efficiency 79

4.4.2 Complement of a local expectation is the imperative 80

4.5	Summary	81
Chapter 5	Principles of the Program's Operation	82
5.1	Flow of Control	82
5.1.1	Perception and the Unknown	84
5.1.2	A typical process, activated by Perception	84
5.1.3	Communication between Processes	86
5.1.3.1	Interactive Communication	87
5.1.4	The Result Mechanism	89
5.1.5	A framework for Grammars	90
5.1.5.1	Detecting end of clause or group	93
5.1.6	Acts and Utterances	93
5.1.7	The Analysis of Feedback	96
5.2	Learning	97
5.2.1	Operation of the Main Synthesiser	98
5.2.1.1	Synthesis from a Partial Match	99
5.2.1.2	Synthesis from Two Results	100
5.2.1.3	Synthesis when there is no result	101
5.2.2	STM and LTM are complementary	102
5.2.2.1	An example	103
5.2.2.2	The General Transformation for Grammatical Information	108
5.2.3	Generalisation	112
5.2.3.1	A Further Step	116
5.2.3.2	Sundry Facilities	116
5.2.3.3	Conflicts that do not lead to generalisation	117

Chapter 6	Description of the Five Components	119
6.1	Perception	119
6.2	Causality	122
6.2.1	Learning causal relations	122
6.2.2	Two Examples	123
6.2.3	The Details	125
6.3	Meaning	128
6.4	Differentiation	129
6.4.1	Details	130
6.4.2	Complementary version	131
6.4.3	Another kind of generalisation	134
6.5	Grammar	135
6.5.1	The Unit	136
6.5.2	Expectations	138
6.5.2.1	When to generalise	143
6.5.3	Results from the left	147
6.5.4	Structure and Class	148
6.5.5	Complementarity	149
6.5.5.1	On the right-hand side	149
6.5.5.2	On the left-hand side	152
6.6	Conclusion	153

PART III

Chapter 7	A Complete Dialogue	156
7.1	Summary of the Program's Normal Operation	156
7.1.1	Processes invoked by Perception	157
7.1.2	Use of STM in Processes invoked by Perception	160
7.1.3	Conventions	164

7.2	First experiences	164
7.3	Short phrases and sentences	167
7.3.1	The indefinite article	168
7.3.2	First verb	172
7.3.3	Past tense	173
7.3.4	Another verb, and the past tense re-visited	177
7.4	Generalisation over a class	181
7.5	Sentences with subordinate clauses	184
7.5.1	Personal pronoun	184
7.5.2	Temporal relations	186
7.5.2.1	Application to the near future	189
7.5.3	Relative clauses	190
7.5.4	Interrogative	191
7.6	Number and the plural	195
7.6.1	Other sequences	201
7.7	Number	202
7.7.1	Generating a reply involving number	206
7.7.2	Numerals	208
7.7.3	Learning about the blackboard	209
7.7.3.1	Erasure	212
7.7.3.2	Further work	215
7.7.4	Counting	215
7.7.4.1	"Count"	220
7.7.5	Accumulation	223
Chapter 8	A Further Dialogue	225
8.1	Arithmetic	225
8.1.1	Addition	225
8.1.2	Subtraction	227

8.1.3	Multiplication	231
8.1.4	Addition taken further	231
8.1.4.1	Memorising the facts	233
8.1.4.2	Causal Relationships	235
8.2	Two-place numerals	237
8.2.1	Counting	240
8.3	Comment	242
	PART IV	
Chapter 9	Comparison with other Programs	245
9.1	Learning Programs	245
9.1.1	The GPS Tradition	246
9.1.2	Generalisation Learning	250
9.1.3	Program Synthesis (Automatic Programming)	253
9.1.4	Structural Learning	256
9.1.4.1	Difference Descriptions	258
9.1.4.2	Using a Model in a more elaborate situation	263
9.1.4.2.1	Grouping	265
9.1.4.2.2	Learning the ARCADE	266
9.1.4.3	The Question of Generality	268
9.1.4.4	Generalisation Learning	269
9.1.4.5	The Near Miss	270
9.1.4.5.1	The Final Weakness	273
9.1.4.6	The Near Miss Again	274
9.1.4.7	A Proposal by Becker	276
9.2	Natural Language Acquisition	277
9.3	Natural Language Understanding and the Representation of Knowledge	281
9.3.1	The Frame	284

9.3.1.1	Clustering	287
9.3.1.2	Control	288
9.3.2	Procedural Representation of Knowledge	289
Chapter 10	Conclusion	292
10.1	Directions for future research	292
10.1.1	Perception	292
10.1.2	Interracting STM specifications	294
10.1.3	Utterances from Process-like Meanings	296
10.1.4	Technical Improvements	297
10.1.5	Vision	298
10.1.6	Further work in the existing domain	300
10.1.7	Social Behaviour and Practical Application	300
10.2	Eventual Use of a Learning System	300
10.3	Some Principles of Artificial Learning	303
Appendix 1	Description of Process 1.5	305
A1.1	Procedures	305
A1.1.1	Basic Instructions Set	305
A1.1.2	Variables	307
A1.1.3	Compatibility with POP_2	307
A1.1.4	How Procedures are written	308
A1.2	Processes	310
A1.2.1	Control Structures	310
A1.2.2	Variables	313
Appendix 2		315

PART I

INTRODUCTION

Chapter 1.

The view that learning underlies intelligence and is an essential part of it has inspired the research reported herein. Indeed, it is questionable whether an entity that cannot learn can be said to exhibit intelligence at all. Winograd (1971, p.422ff.; p.440) remarks upon the importance of the topic. Minsky and Papert (1972) emphasise the point by writing the word LEARN in capital letters. Minsky (1968, p.14) touches on the subject of learning underlying intelligence when he considers the effect on Bobrow's STUDENT program (Bobrow, 1968) of the sentence: "Distance equals speed times time". A program for understanding a natural language should be able to assimilate such instructive sentences into its cognitive structure and this is clearly an act of learning.

Interest in the subject within Artificial Intelligence has been alive for some time. Newell, Shaw and Simon (1959) viewed learning as a kind of problem solving and defined a learning task to be the synthesis, by the General Problem Solver, of procedures to be used by the GPS in the solution of more specific problems. Later, Waterman (1969), Fikes, Hart and Nilsson (1972), Harris (1972), Sussman (1973) and Hedrick (1976) all address the area of learning more or less directly. Harris analyses the precise capability and limitations of his method where he shows that the grammatical part of his program is capable of generating context-free grammars from examples. Hedrick gives instances of his program generating context-sensitive rules by means of counter-examples. The others all describe programs that generate and amend procedures in some way but do not assess the extent of their power in any formal way. Chapter 9 contains a fuller

review.

The present work goes further in the kind of procedures that can be synthesised. Chapters 4 and 9 (section 9.2) will argue that the learning mechanism expounded here provides a sufficient basis for generating procedures in a universal programming language.

One of the attractions of studying learning is that once a system has been primed with basic abilities like perception and communication it should be able to pull itself up to a higher level by its own boot straps, so to speak, with far less effort than a computer programmer would need to devote re-writing a "performance" system (i.e. a system with no pretentions to learning). Charniak (1972) has made this point in an otherwise sceptical discussion.

1.1 Objectives

To create a system that requires no further programming or internal manipulation is a long term aim of this work. At present, not all the principles are correctly or fully specified. Were it otherwise, a program could be implemented that would be an artificial intelligence in a real sense. One clear objective, therefore, is to complete the links between the principles by which a learning program should operate in a systematic way.

A working computer program named DISCO (Latin: "I learn"), does exist and it learns to conduct a dialogue. No internal manipulation by the tutor is needed during the dialogue and the program is endowed with rather general abilities some of which appear to correspond to natural principles. A little more detail of its operation will be given in section 1.2; a full treatment is the burden of part II of the thesis. The further work that is needed will be defined in Chapter 10.

The search for principles of learning (or of intelligence), as

opposed to ad hoc methods, is not a pursuit undertaken for theoretical reasons alone. There is sound practical merit in setting up a system which is not just a patchwork of special solutions for particular cases. Such a patchwork would only be able to learn what it had been pre-programmed to handle and changing it would be even harder than re-writing a performance system.

Another aim is to have a relatively compact program: relatively, that is, to the structures that it sets up through experience and which determine its future behaviour. These structures have the characteristics of procedures, which are defined precisely in Chapter 3. That is, they are sub-programs or series of instructions to be obeyed by a machine. The learning system, which is itself a procedure, synthesises other procedures and its behaviour can therefore be compared with automatic programming.

Manna and Waldinger (1975) are well-known exponents of automatic programming. Their objective is to make computer programming more satisfactory by enabling the user to specify in a formal language what is to be done and have the system write an error-free program. Normally, the formal specification must be error-free although Smith and Hewitt (1974) are exploring the possibility of creating a Programming Apprentice which will also debug the specification. Interaction with the user during debugging is a pragmatic extension of the idea and Good, London and Bledsoe (1975) describe such a setup.

The present work has more in common with the approaches of Waterman (1969), Winston (1970) and Harris (1972) than with those just mentioned. There is no formal language for the user. Rather one gives the system examples, explanations and orders, as a result of which it writes programs and does things without the user being

concerned with the means.

The systems of Waterman, Winston and Harris do not write programs but are driven by examples as opposed to specifications. Of these three, only Harris deals with natural language. Winston deals with vision and Waterman with game playing. DISCO embodies the opinion expressed by Winograd (1971) that a computer program for processing natural language should integrate functions of syntax, semantics and world knowledge. In fact they are more intimately combined in DISCO than they were in SHRDLU (Winograd's program) and this makes it possible for the same method of learning to be applied in each case. This is something that Harris did not do. DISCO also embodies Winograd's view that such knowledge should be represented as procedures. Chapter 3 includes a discussion.

It has not been possible to build significantly upon the achievements of the three authors cited. The mechanisms of Waterman (1969) and Harris (1972) are too specialised to meet the objectives of DISCO, as chapter 9 shows. Winston's (1971) work on learning is also too specialised. His program builds descriptive networks for use in identifying objects presented to it. He addresses the problem of applying common principles at differing levels of abstraction but finds it necessary to postulate an open-ended set of internal types in order to do so. The network representation favoured by Winston and by Minsky (1975) suffers from other limitations and the questions are dealt with in chapter 9.

Techniques which Winston finds useful for the matching of networks that represent scenes do not seem applicable to other problems. The "near miss" is a case in point. DISCO does not need to be given examples of sentences that are ungrammatical in one respect in order to learn grammar, although Winston's program must be shown a table

with the top removed before it can learn to recognise the structure.

A particular difficulty in building on Winston's work is the lack of detail he provides and the apparent fact that many of the examples work in an _ad hoc_ way.

A general difference between the present work and its predecessors is that previously, scant attention has been paid to psychological questions. By contrast, DISCO is founded on the notions of short and long term memory (see section 1.3). In this research attention has been given to the behaviour of children. It seems easier to imitate the activity of little children than of computer programmers, as would seem to be the objective of automatic programming studies. In a linguistic context, Chomsky (1964, p.35) has expressed the opposite opinion on the grounds that we cannot learn their language and thus have lost a useful tool for investigation. However, this point should be weighed against the advantage of the child's comparative simplicity. His argument is further weakened by the implied reliance on introspection.

The second point above is taken up again in section 1.4. The third must be dealt with now, for the relationship between natural and artificial intelligence (i.e. between a child and the program) is an issue on which a position must be stated. Workers in automatic programming would not claim to be simulating human programmers, and so the implied criticism at the end of the last paragraph is unfair. In the current research, however, the system is aimed at acquiring amongst other things, the ability to use a natural language. This is an ability which is unquestionably the product of natural intelligence and success is most likely by using all we can of what is known about the way people do it or, in this case, the way that children learn to do it.

The position is, then, that we need not choose between the objectives of psychological modelling and the making of an artefact. (The word is used with its proper meaning, viz. a useful object made by a craftsman or artisan.) The working computer program bears some analogy with learning in children and can therefore be viewed as a psychological theory with the benefit of an exacting kind of rigour. Furthermore, this program may prove to be sufficiently powerful when it is complete to have practical application. Chapter 10 contains a few ideas on further steps.

1.2 Exterior view of the program

This section is intended to give a feel for what the program does; the next outlines the underlying ideas. The program exists as a vehicle for verifying the capability and self-consistency of the principles summarised in section 1.3. There are many obvious respects in which its environment and experience differ from that of the child and the first lies in the extreme technical simplicity of the domain in which it acts, which involves printing on a teletypewriter, writing on a simple "blackboard", and conducting a dialogue on what happens. This domain is quite rich enough to exhibit relationships that evoke a number of linguistic and other problems without doing anything more difficult like representing pictures as arrays or connecting a robot arm.

Technical simplicity accords with the declared aim of seeking principles of learning applicable to many domains. It is essential not to get bogged down in unimportant details. Consideration of some examples within the simple domain has led to the construction of novel kinds of long-term and short-term memory and a principle of complementarity exists between the two. These constructs, described in the next section, are defined in purely computational terms,

independently of the examples that led to them.

References to memory through two standard interface functions are the building blocks out of which procedures are synthesised during learning. The basic program that performs the syntheses contains several components, all of which work according to a similar pattern with variations depending on whether they are concerned with perception, causality, meaning, differentiation (i.e. "discrimination") or grammar (using the word in a wide sense).

These components, like the memory constructs, were established by considering examples, trying generalisations, discarding and refining. This process is not complete. The program does not cope with everything that an infant can manage and various extensions are suggested, especially in Chapter 10. Experience has shown, however, that only the detailed consideration of further examples will lead to the correct generalisation. _A priori_ hypotheses have seldom stood the empirical test; they have usually turned out to be in the wrong direction.

This is surely one reason why Minsky and Papert (1972) have argued the case for proceeding from many thoroughly worked examples rather than starting with bold theories. But one must not lose sight of the need for general principles and allow the simple examples to become an end in themselves.

Examples also serve to illustrate the ideas and the mode of operation and a short selection follows. In these samples from the dialogue, a colon precedes the lines entered by the human tutor and the other lines are produced by the program. Basically, the program reads a line, interprets it in the light of experience and performs any appropriate action. It inspects any feedback there may be and prepares to read again. Learning takes place when anything un-

familiar is read or when something is not expected or is not fully accounted for.

The program begins with no vocabulary and only the general abilities to be described later. 'Life' begins by encountering a character and, after that, a word.

: .

: .DOT

It must encounter "." separately in order to be able to read ".DOT" as two entities rather than one. It writes "." into long-term memory (LTM) and on the next line retrieves it, so finding "DOT" to be the new item. "DOT" is written to LTM with a procedure that was synthesised as its meaning. When the word is encountered again, that procedure will be executed. In the first line, a procedure for "." was synthesised but it had a vacuous meaning.

The meaning of DOT is "." and the procedure that was synthesised will look for a dot. So in the following case it will find one on the previous line.

: .

: DOT

The way that the DOT procedure does this is by looking up the short term memory (STM). The STM contains a record of the recent events and the nearest occurrence of "." will be found. As was stated earlier, the building blocks with which procedures are synthesised are references to STM and LTM.

The DOT procedure may be put to use in teaching the indefinite article and the verb "to print".

: .

: A DOT

In order to teach an action there must be a way to make the program do something. It therefore has the convention that anything enclosed in square parentheses is to be interpreted as a program and passed directly to the programming system for execution. When this is done, a record automatically appears in STM.

: PRINT A DOT [PRIN ('.')]

.

The DOT procedure locates the dot in STM and the new word PRINT becomes associated with the situation in which the dot was found. Now it can obey the command

: PRINT A DOT

.

The full dialogue necessary to achieve the above behaviour is longer and appears in Chapter 7. It is generalised for other characters. After that, various words, pronouns and morphemes are taught.

Of particular interest are the temporal relations "before" and "after". They are demonstrated to the program as follows.

: PRINT AN ASTERISK

*

: PRINT A COMMA

,

: YOU PRINTED A COMMA AFTER YOU PRINTED AN ASTERISK

Here the unknown word is "after". Subsequent to more examples it can be used not only to answer questions about the past but to qualify commands which relate to the future. So we may have

: PRINT A DOT AFTER A COMMA

,.

One can go further and set up expectations for the future. These get written to LTM as in the following case.

: SAY TRIPLE AFTER I PRINT THREE CHARACTERS

The word PRINT specifies action but the presence of "I" makes this impossible and the action is converted into an expectation. This conversion involves a simple notational assignment because STM and LTM are complementary, as was mentioned earlier. The effect of the sentence on the program is rather like teaching it the word "triple". If, later, the tutor types in three characters he receives the reply.

: ***

TRIPLE

A simpler application of STM-LTM complementarity is in the transformation from comprehension to utterance. The DOT procedure, for instance, is held in LTM with key "DOT" and looks up STM with key ".". It has a complement, which is the same procedure with a simple notational change (a variable re-assignment). That is held in LTM with key "." and when activated looks up STM with key "DOT". It is obviously necessary for language acquisition to be able to make this transition readily. (Chomsky (1964) mentions the matter, but only in respect of grammar; here it applies to both grammar and meaning, although the number of examples is limited.)

There are other facets to the complementarity which will be taken up in Chapter 4. Particularly interesting is the way in which the LTM primitive is used to establish structure not unlike the frames which Charniak (1975) and Fahlman (Minsky, 1975, pp.264-7) propose. Another consequence of the way LTM is defined is the natural manner in which recursive procedures are synthesised - a matter of theoretical importance which is taken up in section 4.3.

All that has been mentioned so far has stood up to the exacting

requirement of performance on a computer. Had the testing not been carried out, the dialogue in this dissertation would have been much longer but some of the best ideas in the exposition would have been absent; namely the use of the LTM primitive for frames, grammar, differentiation of meaning, recursion and causality.

A considerable amount of untested material is presented in Chapter 8, with an explanation of the missing capability in the program. The work concerns arithmetic. The program has learnt small numbers and numerals and it has been taught to use a simple device called a blackboard where it may write things by printing them preceded by a quotation mark. They stay there until deleted by a stroke (/). Typical actions and their effects are illustrated here.

Input	Blackboard
:"**	**
:	**
:"*	***
:	***
:/*	**

The program learns to predict the consequences of its actions. Not tested is the sequel whereby it is taught addition and subtraction by associating sentences like

: 2 AND 1 IS 3

: 3 TAKE AWAY 1 IS 2

with the above actions. Higher numerals are to be taught thus

: 3 TENS ARE 30

: 30 AND 1 IS 31

and so on. Counting comes next and it should then be able to perform addition, subtraction and multiplication by using its blackboard and counting the result.

1.3 Underlying Principles

The basis of the program's operation and its learning capability is the memory construct. It possesses two kinds of memory, short and long term (STM and LTM). These are inspired by psychological theory but are grounded in the computational ideas of control structure and run-time structure.

The program frequently refers to both kinds of memory by invoking the appropriate interface function for STM or LTM. As it learns, the program builds procedures. At the outset, the program consists of several procedures. As it learns it constructs more. These are in standard forms and they normally consist of a call to the STM or LTM interface with particular parameters (and a certain amount of set up plus routines to deal with the result.) It is argued in chapter 4 that this is sufficient to constitute a universal programming language (in the Turing machine sense).

It is important to establish that point of principle because it implies a kind of generality that is of more than theoretical significance. If the formulation of memory presented here lacked this property it would be too limited to be the basis of a general system. It does not follow, of course, that the program is completely general; this is a necessary, but not sufficient, condition. Again, generality is not sufficient to make it either of psychological interest or of practical importance. These are matters of opinion until the research is taken further.

1.3.1 The Basis of Learning

Fundamentally, the program learns by encapsulating an experience or an action in the form of a procedure which is then written away to long term memory for future use. Recent experiences or actions are in the short term memory and so are located in the course of the program's

operation, which frequently involves references to STM. For example, the program could be interpreting the following.

: PRINT AN ASTERISK

*

: YOU PRINTED AN ASTERISK

Suppose it has performed the printing action and is working on the last sentence, calling down from LTM the procedures it needs to interpret each word. Suppose further that it has already learnt all the words except for the past tense morpheme "-ed". Now it will create a procedure that captures the meaning of "-ed" and it too will be written away to LTM for future use.

The other words in the sentence contain enough information to enable the program to decide that the immediately preceding action is being referred to. During interpretation of a sentence, STM is referenced several times. Because of previous acquisition, each word possesses one or more procedures which the program retrieves from LTM. These procedures each contribute to the meaning of a sentence by specifying parameters to the STM interface function that performs the search of short term memory. The procedure for the word "asterisk" contains an internal code corresponding to the character * while "print" contains specifications of an action procedure which will, if unqualified, cause the asterisk to be printed. Each word-procedure calls the STM interface, passing it its own parameters which it saves from one time to the next in order to form a composite specification. There are ramifications to this process: these are dealt with in later chapters (especially in 5 and 6).

The order in which the words are interpreted is determined by grammatical considerations to be discussed later. Fig. 1.1(a)

shows that first of all "asterisk" contributes a parameter to the STM interface and then "print" contributes another, as in fig. 1.1(b). The result retrieved each time is a reference to the same event. However, that could have been otherwise had the human tutor also just typed an asterisk.

STM parameters	Attributes of result
A) *	A) *
	B) External action
	C) Prior to the present

(a) Position after procedure for "asterisk" has been executed

STM parameters	Attributes of result
A) *	A) *
B) External action	B) External action
	C) Prior to the present

(b) Position after procedures for "asterisk", "an", "you" and "print" have been executed. (Only "print" makes a fresh contribution.)

STM parameters
C) Prior to the present

(c) Parameter to be included in the new procedure for "-ed".

Fig. 1.1 STM interface while interpreting: "you printed an asterisk".

The composite specifications to the STM interface function result in it retrieving a reference to the preceding event. There is a definite set of attributes which may be defined to the STM

function (see below) and which may characterise any result it yields. It is not necessary to express all possible attributes before the STM function will give a result; any combination is permissible. The function will locate the most recent item in short term memory that satisfies the given criteria. It can, of course, fail.

In the example above, the result from STM is a reference to the preceding action of printing an asterisk. This result has the attribute of preceding the current activity (i.e. simple past tense) and this attribute was not defined to the STM function by any of the other word-procedures. The program now associates this unspecified attribute with the new morpheme "-ed", as in fig. 1.1(c). To do this it must construct a procedure and save it in long term memory. Next time the program comes across "-ed" in a sentence, it will be able to call down this procedure from LTM and it will define the past tense attribute to the STM function. Thus the sentence: "You printed an asterisk" will have its STM reference completely specified. This definition of "-ed" can then be the basis for further learning as of the personal pronoun in the next case.

: .

: I PRINTED A DOT

The procedure constructed as the meaning of "-ed" must contain an invocation of the STM function preceded by a definition of the relevant attribute. There are only five attributes that may be specified, and it has been possible to write general procedures that compare an STM result with the attributes that were given and construct a procedure to call the STM interface giving it those undefined attributes. This is the way the program synthesises a procedure. It then places it in long term memory with attributes indicating when it is to be used.

1.3.2 Purpose of Long Term Memory

So far, little has been said about LTM, other than that it is a repository for procedures that the program creates. Neither has anything been said about grammar. That uses the same mechanisms of STM and LTM and relies on the same fundamental learning method of synthesising a procedure to give parameters to STM (or to LTM).

The prime use of LTM in the present work is to recognise words and to retrieve and activate the procedures that have been saved in it during previous experience. Initially LTM is empty and when learning takes place the program writes procedures and puts them in LTM. In order to write something into LTM, the program must call the LTM interface function passing it parameters telling it when the procedure is to be used. In the case of a new word, the attributes include the actual characters of the word and an indication that the procedure is to be invoked when these characters appear while the program is reading from the teletypewriter.

Note the distinction in function between the two interfaces. STM performs retrieval whereas the LTM stores items. Every experience or action is recorded in short term memory unless it is very low level. However, items are only placed in long term memory via the LTM interface. The reasons for making these interfaces perform differently are explained below.

In general, LTM is a means for the program to produce a conditional extension to a procedure. The procedure that reads from the teletypewriter is extended by each procedure that is set up to interpret a new word. Another example is the extension of the meaning of the plural morpheme "-s" from "two" to three and eventually to many. Extending a procedure in order to treat particular cases is known here as Differentiation. Its relationship with discrimination

learning is discussed in chapter 2.

Retrieval from LTM is notionally automatic. In fact it is performed regularly at suitable times. It is done while reading or performing external activity (printing) and at the end of every procedure that the program has synthesised. Retrieval consists of locating a procedure in LTM that has attributes that match the current situation. When one has been found it is executed. Standard actions are performed if retrieval fails.

There are rules to determine which is selected if there is more than one contender for retrieval on any occasion. For example, the words "a", "as" and "asterisk" might all be present in LTM. The shorter ones would not be retrieved if the longer word matched the input. In the event of a tie the most recent entry always wins. Chapter 4 (section 4.3.4) shows how LTM could be implemented efficiently on a large scale.

1.3.3 Approach to Grammar

When interpreting a sentence, the program works from left to right. The longest match to the front of the sentence is found in LTM. The retrieved procedure is executed, followed by any dependents it may have. The remainder of the sentence is then subjected to the same process and this goes on until nothing is left. The direction is, of course, conventional but it corresponds to progression through time, which is not. Speech and writing are one-dimensional but the requirements of interpretation imply that words must interact in the mind of the listener or reader in groupings, the meaning of a word depending on those to its left or right. Often its interpretation must wait until following words have been inspected. The relationship of this active process-directed view of grammar to a linguistic theory based on class and structure is discussed in section 6.5.

Where the meaning of a word involves looking to the left of it in the sentence, the STM function can be used to find the desired reference, since words on the left have already been processed and the records appear in the short term memory. Interpreting words, after all, is an activity of the program like any other. Words that need to obtain results from their predecessors in the sentence will possess procedures that contain the necessary extra call to the STM function. Examples are "before" and "after", "is", "-s", the numerals in counting, etc.

Where the interpretation of a word must await that of following words, there will be associated with it a kind of mini-LTM which acts like the main LTM and is in fact stored in it along with the procedure for interpreting the word. This mini-LTM is known as a stored context or frame. It shares many properties in common with the frames of Minsky (1975) and also with the contexts of CONNIVER (Sussman and McDermott 1972). The frame is instantiated at the same time as the procedure for the word is invoked. While it is in effect, retrievals from LTM automatically come from the frame in preference to the main LTM when they are applicable. There may be more than one frame active at a time. A frame ceases to be active when one of its contents is triggered (and the procedure executed) because its attributes have been met.

In effect, a frame is a set of expectations. Some are general like that associated with the indefinite article "a" which may be followed by many different nouns; others are particular like the case of 1 followed by zero (10). Just as in the main LTM, each item in a frame has a procedure bearing a set of attributes indicating when it is to be invoked as in fig. 1.2. Each lower box contains a procedure specific to the meaning of 1 in each circumstance named.

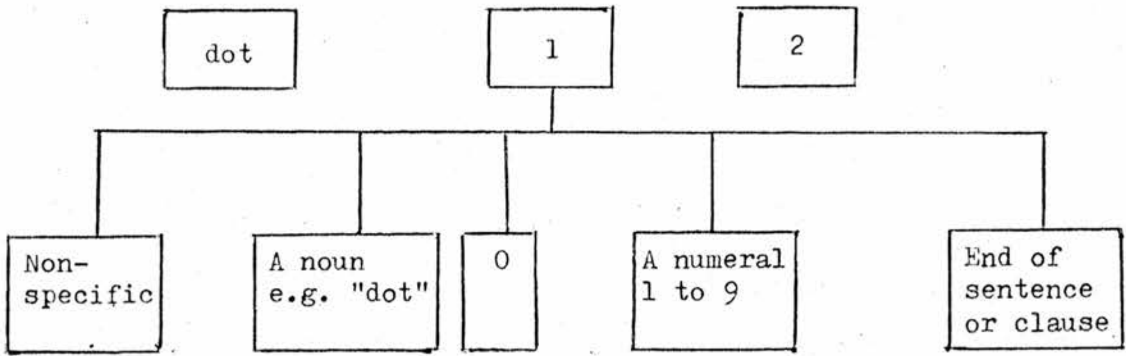


Fig. 1.2 A Frame. The boxes at the top are entries in the main LTM. They contain procedures to be executed when the program reads "dot", 1 or 2. Associated with 1 is the frame, mini-LTM or set of expectations shown on the bottom.

If the program reads 13 on the teletypewriter the procedure in the fourth box will be executed. The program places entries into a frame using the LTM function in exactly the same way as they are written to the main long term memory. An extra parameter is used to indicate to which frame they should go.

1.3.4 STM and LTM complementary

The attributes required by the LTM function characterise the circumstances in which a procedure is to be invoked or triggered. "Circumstances" must mean a class of experiences or actions of the program, such as the perception of a word or whatever. Nothing is defined in terms of the objective real world. Besides the well-known philosophical difficulties of defining what is objective and real, there is the obvious fact that individuals can only perceive reality through their experiences and senses, and the computer program is similar in that respect. Objectivity comes with maturity and, presumably, with much teaching and experience.

The short term memory consists of the recent experiences and

actions of the program and the STM function allows it to search for items possessing given attributes. Whenever long term memory is searched, it is to see whether the current experience or action possesses the attributes associated with any item in LTM, as given to the LTM function at the time the program placed the item there. The current experience or action is at the front of STM and can be characterised by the same set of attributes as the rest.

It follows that the parameters of the STM function are the same as those of the LTM one. Hence these two are complementary. In fact the functions are interchangeable within a procedure. It turns out that this transformation, at least in a number of cases, converts a procedure for interpreting a word to find its meaning into one for uttering the word when its meaning is given.

This can be illustrated with the word asterisk. Fig. 1.3(a) shows the basic form of the procedure that the program synthesises.

LTM ('ASTERISK'); STM ('*');	LTM ('*'); STM ('ASTERISK');
(a) In Comprehension	(b) In Utterance

Fig. 1.3 Basic form of procedure for "asterisk".

Details have been omitted. The (a) procedure consists of two subroutines, such that the first calls the second. Each of them contains a call to one of the memory functions. The program executes the first subroutine as soon as it has created it and this causes the procedure to be saved in long term memory with the appropriate attributes, including the actual letters of the word in coded form.

The (b) procedure consists of exactly the same subroutines as those in (a) but with the second one first. The transformation is

effected merely by re-assigning variables. Again the program executes it (once only) to cause it to be stored in LTM. Subsequently it can be used to answer "what" questions.

Other examples of complementarity at work appear with numbers and the plural. The difficulty of extending the principle to verbs and other parts of speech is discussed in chapter 9.

The same principle can be extended to grammar and the memory functions used there. An interesting application is to counting.

Finally, it applies to causality. This subject has not been taken as far as language, but the program learns to predict the effect of writing on its blackboard device and again the substitution of the LTM function for STM results in a transformation. This time it converts a procedure for verifying the cause of what is now seen into one for predicting what next will be. The predictor is used until something unexpected happens, at which time the complementary procedure takes effect and attempts to account for the discrepancy, performing further learning if possible.

1.3.5 Processes and Run-time Structure

The detailed exposition in part II begins with the distinction between procedure and process and the notion of run-time structure. These are the constituents of short term memory. The STM function performs a search of the run-time structure because it contains a complete record of the program's behaviour apart from low-level details not saved after execution.

Fig. 1.4 illustrates the distinction drawn between process and procedure. The knitting pattern is the list of instructions, with loops and subroutines, that make up the procedure. Granny is performing a process with her needles. A record of that process is found in the unfinished article and that is the run-time structure.

Fig. 1.4

We could search it for the occurrence of particular parts of the pattern. (It is irrelevant to the analogy that this record is also the object of the activity.)

The run-time structure constitutes the program's short term memory. It is an interrelated collection of the records of processes. Each recorded process was the execution of some procedure. One of the STM parameters, therefore, is the execution procedure of the process to be located. In the interests of uniformity, all data are represented procedurally. So characters are represented as the procedures that would print them. The meaning of three includes a procedure to call a subroutine three times.

One consequence of this uniformity is that certain parameters to STM (and hence also to LTM) are interchangeable. A useful application of this capability arises when the program generalises one of the expectations in a frame to the class of numerals. One STM parameter allows partial matches (needed for generalisation over a class) while the other does not.

1.4 Justification

In chapter 9 there will be no more than an indication of how one might be able to develop the program along more practical lines. Obviously, genuine practical application is a long way off as there are many theoretical problems still to be tackled but the eventual solutions could also be useful.

It is a truism that the cost balance between computers and the people required to run them is always shifting in favour of the machines and the trend has been for a long time now to simplify the user's task as much as possible by providing high level programming languages, advanced operating systems, generalised application packages and data base software. It would be a logical extension of

this trend to set up a system that maintains its programs on the basis of discussions with its users.

The cost advantage is fairly clear. More controversial, perhaps, is the question of whether it can be done - not in theory but within the foreseeable future. This question applies to Artificial Intelligence as a whole and not just to learning and it remains unanswered.

The problem of maintaining an increasingly large and complicated system of programs not only provides a reason for introducing more intelligence into computing and data processing installations; it is also one justification for learning research within Artificial Intelligence. For natural language understanding programs that cope with limited topics and a reasonable range of grammatical constructions are large and complicated (e.g. Woods, Kaplan and Nash-Webber, 1972). The prospect of trying to extend such a program to cope with a wider range of topics and then to continue updates and maintenance is appalling, if not actually impossible.

The foregoing argues the potential usefulness of Artificial Intelligence and endeavours to justify particularly the approach through learning. To explain why it is a valid alternative to automatic programming it should be pointed out that today very little human programming effort is devoted to writing something completely new as most projects in automatic programming aim to do: usually an existing system is being extended. So a verbal interaction between the person and a system endowed with world knowledge sufficient to understand the purpose of the existing system would be a profitable means of carrying out this activity.

The final word of justification is for the scientific and philosophical worth of these endeavours. As well as being a scarce

and valuable commodity, intelligence is exceedingly mysterious and we do well to try to understand it. The research described here does have recourse to some psychological ideas of memory and learning. It departs from the natural model when this is expedient. In chapter 2 the relationship will be discussed more fully.

RELATION TO CHILD DEVELOPMENT

Chapter 2.

It is impossible to avoid comparing the behaviour of a program that purports to exhibit intelligence with human behaviour. This is simply because there is no other standard by which to gauge success. Whereas Wilks (1973, p.118), for instance, states his objective as being "to produce a working artifact", the only way to assess whether the artifact is working is by looking at the quality of its translations into French, comparing them with the rendering we ourselves (or a competent translator) would have given.

It therefore seems reasonable to go further and extract the maximum amount of assistance from the body of knowledge in psychology and linguistics. While trying to avoid the constraints of existing theory in those disciplines, I have attempted to make the program consistent with such reliable empirical evidence as there may be - consistent, that is, within the limitations imposed by the technical simplicity of the program's domain.

In fact, the program may be related to more than one theory but it matches none exactly. As far as it goes, it constitutes a fusion of ideas drawn from theories of memory, cognition, language and development. The first topics to be examined are memory and development.

2.1 Memory

DISCO's use of memory is central to the learning process which is itself largely defined by the form of memory reference.

Psychologists have long made the distinction between two kinds of memory: short- and long-term. Short-term memory (STM) is thought to be transient, holding a record of the most recent events; long-term memory (LTM) is permanent, and is evidently much more than a

record of experience. It must contain "active records" (Bartlett, 1932) which come into play at appropriate times; these records constitute the acquired skills or learned behaviour of the individual. Bartlett hypothesises that the recall of events out of memory is a process of reconstruction.

The system embodied in DISCO, when viewed as a model, can plausibly account for the observed phenomena. Writing to LTM is an active undertaking by the program whenever learning takes place. What is written is a record of the learning event in an active form for later use and, given sufficient storage space (as is apparently available in the human brain but not in a computer), this record can be accompanied by those of the neighbouring events as well. Then the recall of events would be a process of working through these records. If they were partially degraded then some reconstruction would be necessary.

There is some considerable doubt as to whether the records of short-term memory (i.e. the most recent events) disappear or remain stacked away and not directly accessible. Bennet (1975) cites evidence for the latter theory by observing the errors that subjects make in experiments on memory and recall (Brown-Peterson situations). When people make incorrect responses to couplets they have learnt, the items almost always come from earlier parts of the experiment and usually bear some relationship of class or semantics to the current item. This evidence does not conclusively distinguish between two possible variations of DISCO. One is that a complete chronological trace of experience is stored indefinitely (with LTM "active records" especially identified) and the other is that STM size is fixed but the current content of STM is copied into LTM at the time of setting up an LTM record. The subjects may be retriev-

ing from LTM during the experiments and, in the second variation of the system, that would cause the associated record of events to reappear in the short-term memory and give rise to the systematic errors that take place. It is in the nature of long-term memory that the retrievals would be relevant to the task in hand thus giving rise to the systematic nature of the erroneous responses reported by Bennet.

For practical reasons, the program's STM is limited by space constraints but if these could be removed, there would remain the overriding limitation of search time. The system has to conduct a search of STM in order to retrieve from it. A succinct method has been devised for a procedure to specify to the STM interface the kind of event to be retrieved but a search is then necessary to find it. The alternative would be for each item in STM to be looking out for a match to itself, as it were, but this proves to be unworkable, as shown in Chapter 4. Since this search constraint, the equivalent of proactive interference, is the crucial one the model would tend to support Bennet's conclusion that the trace of STM is permanent but the search is limited, with the reservation that records are passive with respect to retrieval.

Precisely how a procedure specifies to the STM interface the kind of event required is detailed in Chapter 4. Broadly speaking the form of the specification is procedural, since the short-term memory consists primarily of a record of the procedures most recently active or referenced. Learning involves the synthesis of new procedures which may then also be active or referenced in STM. This generality increases the program's power. Procedures can be linked to form more complex ones, patterns and meanings can be represented and all in an active, dynamic framework.

Although it is unworkable for each item in STM to be on the

look-out for a match, that is effectively what the contents of LTM are doing. Each item is waiting to be fired by an experience such as by the reading of a particular word or by the activation of some procedure. When, for example, a word is read from the teletypewriter the corresponding "active record" comes into play and it performs the necessary actions to interpret the meaning of that word within the sentence. Of course, if the word is unfamiliar to the program, no relevant record will be available in LTM. In that case the program will endeavour to construct one.

As was mentioned above, a succinct format has been developed by which the procedures in the program specify event matches in STM. Exactly the same form is used by items in LTM. The records in LTM are, after all, waiting for events to happen and anything that takes place in the program's experience or behaviour always starts as the first record in short-term memory. (Then it is slowly pushed to the back by later events.) It is therefore quite natural that the format of the match specifications should be the same.

Because they are the same, it is possible to interchange references to STM and LTM within a procedure that the program has created and this makes possible a transition from comprehension to utterance in language, amongst other things. This transition echoes the simple notational change necessary in grammars to convert from comprehension to production and remarked on by Chomsky (1964, p.40) in the context of child language.

The importance of succinctness in characterising events for memory retrieval is crucial in a computer system for it enables high speed random access by means of a key. The key is the tag or identifier associated with an active record. When the program retrieves from long term memory, it sets up a key that describes

the current experience or action. If there is a matching key in LTM, the corresponding record will be activated. The precise technique of retrieval is unimportant. It may be associative like a Fourier holograph as Pribram (1971, p.140ff.) argues to be the case for the brain or it may work by indexing or hash coding. What is essential in order to avoid the combinatorial explosion of searching is the use of a key.

There is a more general arrangement known as the (associative) content-addressable memory in which records may be retrieved because of a match to any part of them - not merely to their keys. The problem is deciding what to do with such records after they have been found. There can be no single "intelligent" component of the program which "knows what to do". If the program is to gain intelligence it will do so by enlarging its repertoire of active records in LTM each of which acts in particular circumstances (of varying specificity). Therefore the content-addressable memory is inappropriate.

References to memory are the building blocks out of which the program synthesises procedures to cope with causality, meaning and grammar. Fundamentally the program learns by connecting the unspecified attributes of a situation. For example, when teaching the plural morpheme "-s" where the word "asterisk" is already known, the unknown "S" is associated with the unspecified attribute which is repetition. More exactly, in the example below, the attribute is duplication, which is represented as a procedure for doing something twice, the something in this case being a procedure for printing an asterisk. These procedures are constructed by the perception part of the program from the two lines below that the human tutor types in.

```
: **
: ASTERISKS
```

After this, LTM will contain a new record with a key indicating that it is to be activated when the program reads "S" again. On those future occasions it will search short term memory for the repetition procedure.

Subsequently, situations may occur in which the attributes are contradicted. Then they are relaxed by means of a process of generalisation. Commonly, over-generalisation will take place and differentiation or discrimination, is needed. This amounts to setting up a record in LTM having a key specifying a more restricted situation. The process is very similar to initial learning and this similarity is one of the most significant achievements of the research.

2.2 Development

It is tempting to compare the active records that the program sets up in LTM to Piaget's assimilatory schemas. Clearly there are differences. Piaget has not studied the process of language comprehension and so does not apply his theory of schemas to it. Indeed, as Berko and Brown have remarked: "Piaget is inclined to see through words as though they were not there and to imagine that he directly studies the child's mind." (Berko and Brown, 1960). He uses the notion of schema most often in relation to infantile sensori-motor activity, writing, for example, of the infant assimilating increasingly varied objects to the schema of sucking (Piaget, 1952b, pp.33-34). He pursues the analogy between cognitive adaptation and the biological processes of assimilation and accommodation.

Despite these differences, the similarities are striking. His three kinds of assimilation: reproductive or functional, generalising and recognitory are very close to the activation, generalisation and differentiation in which DISCO's LTM records engage. The schema is conceived as an organised "totality" which functions and may

perform generalisations while functioning. In setting up special cases (recognition or differentiation) it establishes new schemas, exactly like the computer program.

Undoubtedly, Piaget's conception of the schema is richer than the LTM records described here. It is in generalisation that this discrepancy is most marked. He writes of assimilating new objects to a schema and the program does not exactly do this; rather it relaxes constraints. It can relax them in such a way as to define classes of objects, e.g. characters or numbers, provided that there is some attribute to characterise them, but it does not perform generalisation by assembling a list of apparently unrelated objects. For instance, the program is able to generalise "followed by a numeral" when carrying out a kind of addition (see the end of Chapter 7) because it can detect the common features of use that it has already learnt between 2, 3, 4, 5, 6, 7 and 8. But it cannot generalise "punctuation mark" because it has not derived any unique characteristic for this class. It seems intuitively right that classes named in language should have a defining characteristic; otherwise there would really be no point in naming them. It is easy to think of examples. Many children consider a whale to be a fish for obvious reasons. They are able to learn that it is a mammal, a special case which can be recorded by differentiation or recognitory assimilation.

However, the same effect as Piaget's more flexible form of generalisation is achieved by setting up new LTM records. It is perfectly possible for several words to have the same or similar meaning. Indeed such is the case for the numbers and the individual creation of these records (essentially recognitory assimilation) is a prerequisite for the subsequent generalisation to "followed by a

numeral". It is possible, therefore, for several LTM records to contain similar schemas and this fact appears to be the aspect of Piaget's generalising assimilation that does not closely correspond with DISCO's generalisation capability.

A Piagetian schema can contain other schemas and can be viewed at various levels. So too can the program's LTM records; the process of differentiation, involving as it does the setting up of a new procedure in LTM, is equivalent to appending the new procedure to an old one, the old procedure being present in the key of the new LTM record as part of the succinct specification of the situation in which that new record is to be activated. Furthermore, it is possible for two procedures to be linked by the presence in LTM of a record which is contingent upon the presence of one of them in some situation and which also refers to the other by means of STM. This is quite close to reciprocal assimilation between two schemas.

Piaget writes not only of assimilation but of accommodation as well, pursuing the biological analogy of an organism assimilating an object (as in digestion) or accommodating to it and so changing its own organisation more drastically. To argue with Papert (1973) that a superior analogy is now available for cognitive development, namely that with a computer program, would not do justice to Piaget's thought. For Piaget, the gradual separation between assimilation and accommodation gives rise to the infant's emergence from a state of profound egocentrism to self-awareness and objective knowledge of the world. DISCO cannot be said to have made that transition. However, it does have the capability to acquire knowledge of the world by assembling LTM records that anticipate the results of its actions in the world. Such assemblies of records seem capable in principle (although this has not been proven) of representing knowledge of the world in

increasing complexity, progressively generalising and refining (differentiating) hypotheses in the light of experience. This is possible not because the program has a component that could be labelled "accommodation" but simply because the organisation of LTM is at once efficient for large sizes (because of key retrieval) and general (because the keys are procedures in a universal programming language).

One human quality that Piaget attributes to the antagonism between accommodation and assimilation is curiosity - the motivation to explore. That subject is simply not addressed here. It is not clear how much the program is affected by this limitation. The environment in which it operates at present is so limited that there is very little scope for exploration, although it could conceivably play with the blackboard. As a substitute for an exploratory drive, the human tutor can make the program perform any action he deems useful for it to learn about by typing in a procedure. It is possible that the instincts to probe and play are no more than nature's substitute for this facility. That is not to belittle these instincts; obviously they have profound implications for human destiny. The point is that it is not difficult to see how they could fit in without disturbing the program's existing framework. (There are a few technical problems in task management and interrupt handling but these are well within the state of the art.)

In short, the program has no equivalent of the co-occurrence, articulation, and mutual equilibrium of assimilation and accommodation. Nevertheless, LTM records bear a close resemblance to schemas not only in the manner already pointed out but also in that the program begins with sensori-motor schemas (e.g. for reading, printing and simple feedback) and builds more abstract ones upon

them (e.g. for repetition, word meanings, numbers, etc.) and in terms of them.

As pointed out above, one of the difficulties in making comparisons is that Piaget has not treated the areas considered in the present work at the detailed level which computer programming demands. He has dealt with the sensori-motor schemas of the neonate rather thoroughly but not with language. When we turn to number, a variation of the same difficulty is found.

2.3 Number

Piaget postulates that a child acquires a schema of intuitive qualitative correspondence and his experiments and theory concentrate exclusively on set-theoretic aspects of number. One of his experiments (Piaget, 1952a, p.167, Chapter VII, section 2) could be applied to the computer program as follows.

: PRINT TWO DOTS AND THREE COMMAS

..,,,

: DID YOU PRINT MORE COMMAS THAN CHARACTERS

As far as can be seen from hand working, the program would make the same mistakes as the youngest (stage I) children. That is to say, it would apply the meaning of "character" to the unmatched portion after the meaning of "commas" had been applied through STM. Therefore, the reference of "characters" would be to ".." instead of to "..,,," and the program would answer "yes" when the processing of "more" found there to be more commas than dots.

The error, if one can call it such, would derive in this case from the meaning of "than" which would have been synthesised from prior examples such as

: YOU PRINTED MORE COMMAS THAN DOTS

In consequence an active record in LTM would be set up to wait for

the word "than" to recur. It would contain a procedure that would explicitly expect the following word to match into the remainder from the match found by the previous word.

An example using different words, e.g.

: DID YOU PRINT MORE COMMAS OR MORE CHARACTERS

would still fall into the same error, simply because the manner of STM search is to locate the most recent reference first and the remainder from the matching in "commas" would be the most recent.

Such an achievement would be a step forward from the work of Klahr and Wallace (1972) who describe a program that models the behaviour of stage I children in such experiments. They do not consider any previous learning that might lead up to this developmental level and they do not treat the problem of natural language processing. They have attempted to model the internal cognitive process that gives rise to the child's behaviour and they discuss ways in which it might begin to learn to perform like older children. Their program is written in production rules, a formalism especially suited to modelling learning. Production rules have been used for other applications, and are evaluated in Chapter 9.

Piaget asserts that a grasp of the set-theoretic aspects of number - such as one-one correspondence, class-inclusion comparisons, and the distinction between cardination and ordination - are necessary before one can say that a child had understood number. However, there are many problems on the way. Matters like counting and arithmetic are far from trivial, despite their dismissal by some as mere rote learning, and it is not easy to produce a program that can learn these operations without building in extra facilities beyond those that are already needed to cope with language and causality. Thus Piaget's experiments and theoretical discussion on number are

concerned with a level of development that is beyond the reach of the current research.

2.4 Child Language

The dissimilarities between the computer program and a child are very obvious. The child moves and behaves spontaneously in a rich and loosely structured environment and copes with sensory data from a variety of sources. Some consideration has been given to the manner in which the principles embodied in the program could be adapted to processing visual information and an outline of these ideas is presented in Chapter 10. The questions of spontaneity and simultaneity have already been discussed.

As a research strategy, the choice of such a simple domain of action for the program has the important merit of making the problems tractable although it has the obvious pitfall that the resulting methods may not be widely applicable. This is a potential difficulty with any domain, however, and is a widespread problem in Artificial Intelligence work, especially in learning research as the comparisons in Chapter 9 will attempt to show. While the proof of the pudding is ultimately in the eating, some safeguards against inapplicability are present in this work.

Much has already been made of the fact that the search arguments and retrieval keys against short and long term memory are procedures in a universal programming language. The following additional requirement is to be expected of a general-purpose learning program, namely that it must be capable of generating, from well-defined examples, a set of primitive actions which are in turn capable of generating (by combination) a universal set of actions. That is, it must be capable of arbitrarily combining the primitives of a universal programming language. In Chapter 4 it is shown that the

program is indeed capable of generating essentials like recursion and conditional statements and in Chapter 6 (section 6.4 and section 6.5) it is further shown that procedures can be combined into chains.

Although the requirement of universality is very important, that property is not sufficient to make a program either psychologically interesting or of practical potential. Unfortunately there appears to be no way of formalising this argument. The relationship with one psychological theory has already been discussed and it seems we must be content with a treatment at that level.

Continuing at that level of informality, we turn to the more specifically linguistic matters addressed by the research being reported here. An important theoretical contribution to the study of language acquisition comes from Chomsky (1965) in "Aspects of the Theory of Syntax". It is impossible to divorce language acquisition from the study of learning as a whole (although it is somewhat easier to divorce linguistics from a general study of cognition) as the cornerstone of Chomsky's argument clearly shows. In justifying his view that children possess an innate disposition to learning language in certain ways (a disposition that should express itself in the existence of certain linguistic universals - common attributes of all the diverse human tongues) he quotes Leibniz who was actually writing about number: "The truths of the numbers are within us; nevertheless they are learnt". This only supports Chomsky's own view, of course, of language as a "mirror of mind" (Chomsky, 1972, p.x)

The close relationship between learning of different kinds is reflected in DISCO, which employs a common set of primitives in all its activities. It also possesses a set of innate learning abilities. The requirement for these is a clear implication of the philosophy of

Chomsky, after Leibniz. Although making the "mirror of mind" statement, Chomsky's theory concentrates on the acquisition of grammar. His view of this process as the selection by the child of a grammar from the set of all possible ones has come in for criticism amongst developmental psychologists (cf. Piaget, 1971, section 16) and he appears to modify his position (Chomsky, 1972, p.159; see footnote 41) by admitting that this theoretical idealisation may be only "a first approximation". Nevertheless the basic point stands - there must be learning abilities not possessed by other animals which enable the child to acquire language and other knowledge. The alternative, as Leibniz says in the same context is the "tabula rasa".

Lenneberg (1964) argues the biological case for ascribing linguistic abilities to the human species exclusively. Since he wrote that article, some impressive success has been achieved in teaching language to an ape called Washoe (Gardner and Gardner, 1971). However, the argument is not seriously weakened by the existence of some of the innate learning abilities in a species so closely related to our own. Nobody has succeeded in teaching language to a frog, for example. In any case, Brown (1970) has found that Washoe lacked at least one essential linguistic ability, viz. the disability to make syntactic distinctions in word order. For example, a young child when shown a picture of a cat biting a dog may say "Cat bite", "Bite dog", or "Cat dog" but never "Dog bite", "Bite cat," or "Dog cat". Washoe, it seems, would not make this distinction. So mankind still has the edge on the apes.

The computer program does observe such conventions although the examples that actually work may not be enough to convince the sceptic and none of them are semantically significant like Brown's examples quoted above. (Incidentally, none of Washoe's inversions are either).

This doubt does not, of course, affect the argument for the existence of innate learning abilities; it merely leaves open the question of to what extent the program captures them.

The most obvious superficial difference between the program and a child linguistically is that the program employs only the written form. Much of the literature in child language is on phonetics and is not relevant to the present work at all.

Some problems arise because of ambiguities in distinguishing new words that would not occur in speech because English possesses some sixty phonemes whereas the Roman alphabet contains only twenty-six letters. So "IS" could be the plural of "I", a confusion that could not arise in spoken English. In the dialogue the problem is avoided by a judicious ordering of the introduction of new words but this will clearly be a difficulty for the future. Notice, however, that these ambiguities only occur when a new word is encountered for the first time. Subsequently, the algorithm of longest match for LTM retrieval takes care of these matters.

PART II

PROCEDURES AND PROCESSES

Chapter 3.

John von Neumann is usually credited with having the idea that programs in the electronic computer should be held in the same kind of storage as the data upon which they operate. A consequence is that programs may be viewed either as potentially active agents (capable of being executed by a machine) or as data structures on which other programs may act. The word "procedure" is used here to mean such a data structure and the author has developed a system to exploit the structural aspect to the full. The execution of a procedure is a "process" and the system that has been developed generalises the notion of process beyond previous formulations apart from those that altogether discard contexts and access environments (e.g. Hewitt, Bishop and Steiger, 1973).

Newell (1972) characterises the distinction between structure and process as the "material-activity" distinction. He takes as an illustration the motor car: its parts go to make up its structure and the running of the motor is a process. The parts are relatively static with respect to the running motor but they are not immutable for they are themselves subject to the process of wearing out. In a child, of course, the structures are growing rather than wearing out. Likewise the processes carried out by DISCO act to create, modify and extend the procedures which constitute its acquired knowledge and ability (stored in its long term memory). Any particular process, e.g. the interpretation of a sentence, will involve the execution of many procedures and may involve the modification of one of them or the synthesis of a few more.

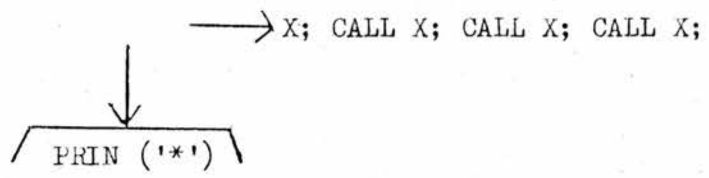
The problem now arises as to the best structural representation for a procedure in order to allow adequate manipulation and still to

retain a natural and efficient correspondence with its meaning or purpose. When the list programming idea was invented by Newell and Simon (1956) it was hoped that both these objectives would be met. That was, at least, the opinion of Miller, Gallanter and Pribram (1960) when they related a computer program to the psychological notion of a plan.

These ideas were incorporated into LISP (McCarthy, Abrahams, Edwards, Hart and Levin, 1962) which has been used by the majority of researchers in Artificial Intelligence. LISP, however, dismembers procedures too finely to capture meaningful patterns efficiently and a compromise is advanced in the present work in which list pointers are used only when two parts of a procedure are to be separated for some purpose connected with the particular problem in hand. For example, when the program reads three asterisks (i.e. ***) from the teletypewriter it first converts it to the procedure

```
PRIN ('*'); PRIN ('*'); PRIN ('*');
```

This is held not as a list structure but simply as a string of characters just as it is written on the above line. In this form it could be matched against other procedures for printing a variable number of asterisks. The program, however, re-writes it as two procedures (section 6.1 will state how it does it) as follows.



This reads: "assign to the variable X the procedure for printing an asterisk and execute it three times." In this form the higher procedure can be matched to any procedure for doing something three times, regardless of the complexity of the subordinate procedure

(subroutine). The only direct address pointer (as used in list processing) is the one represented by the downward arrow. The subroutine is thus structurally embedded as a constant (like a lambda expression) whereas the more usual convention in programming is to refer to subroutines by means of global variables. That method is not suitable for the kind of matching performed by this system.

It will be the burden of Chapter 4 to explain in detail how such structures are matched in memory and manipulated. The rest of this chapter is given over to a precise statement of the constructs.

3.1 Procedures

A procedure is a series of instructions to be executed by a machine. A machine may be another procedure being executed but at the end of the line there is a hardware machine performing actions. An "interpreter" is a software machine, i.e. a procedure that executes other procedures. DISCO uses the programming system PROCESS 1.5 (Knapman, 1973) which includes an interpreter.

3.1.1 List Programming

The interpreter is written in POP_2 (Burstall, Collins and Popplestone, 1971). The procedures (functions or sub-routines) that it executes are encoded in a particular way because of the novel use to which they are put. Like the code that hardware machines perform they are stored as a linear series of instructions, called the text, whereas a procedure in LISP (McCarthy et al., 1962) is held as a nested list structure. When the list-programming idea was being invented with IPL-V (Newell & Tonge, 1960) it was thought that such structures might capture the essence of the program in some purposeful way. However, this does not seem to have been the case and workers like Hewitt (1972) have suggested other ways in which a program could be associated with its purpose or "goal".

The other advantage of a list structured representation for a procedure is that changes may readily be made, even while the pro= cedure is actually being executed. Such amendments may be made by a human user interrupting processing, by another procedure or by the amended procedure itself. This capability would appear to be important to automatic programming.

However, although DISCO does require, on occasion, to modify procedures that have recently been executed, it does not need to return to them after amendment. Hence the text of the procedure may be copied for the purpose of update, provided that we retain a base structure that represents the procedure from the point of view of structure-sharing and is not copied. That is to say, if the procedure is pointed to from more than one place then the amendment will apply to all of them.

The important new use to which procedures are put here is textual matching, primarily in retrieving from short and long term memory. Linear sequences, rather than list structures, are best for this purpose. The details of the instructions are given in Appendix 1. Each instruction is terminated by a semicolon. Matching proceeds effectively character by character (in fact POP_2 words are used).

3.1.2 Procedural Representation of Knowledge

Procedures are represented as sequences of characters (or words) making up series of instructions. Facilities exist for them to be created and modified by other procedures as well as by a human user at a timesharing terminal. These consist of a set of functions to perform compilation (to convert to the internal, interpreted, form from code similar to POP_2) and editing.

Now a few general words are in order about procedures as

representations of knowledge, a matter discussed, among others, by Winograd (1972). He observed that the arguments for augmented transition networks apply equally to procedures because of the equivalence of these two. His strongest argument for procedural representation is that his admirable program works that way; the grammar is a program written in PROGRAMMAR (Winograd, 1969); the meanings of words are procedures to be performed when the words are encountered and knowledge of the world is a system of theorems and assertions in MICRO-PLANNER (Sussman, Winograd and Charniak, 1970).

Charniak (1974) has proffered the view that there are types of high-level knowledge not best represented as procedures. As Winograd (1975) points out, the controversy is not on a factual issue. The distinction between program and data is one of emphasis, since a procedure is a data structure and, conversely, a program may be viewed as a software machine with respect to which its data is a procedure albeit not written in a universal programming language. Moreover, any statement in any language may be represented trivially as a procedure for printing it. Network representations are also subsumed by the procedural formulation (see section 9.3).

The point of the debate is to determine what is the best way to represent, in detail, knowledge of various domains and what is the most helpful way to view it - helpful, that is, to the person trying to construct solutions to the problems. There is, therefore, no general answer. As the state of the art advances, experience should point the direction.

The work with DISCO has profited greatly from the exclusive use of procedures and one other type of object, the process or state. Apart, that is, from the incidental technical use of lists in constructing procedures and dealing with characters when reading and

printing (they are converted to procedures on input). But the work has exploited the dual aspects of potentially active agent on the one hand and compound, possibly recursive, data structure on the other. Hewitt, Bishop and Steiger (1973) have pushed things to the limit by insisting on only one kind, the ACTOR, but one cannot deny that there is an essential difference between a value cell and a procedure even if Hewitt et al. give them both the same name and then make one a special case.

It is interesting that Winograd (1975) finds 'learnability' to be an important attribute of declarative representations. This point of view is not supported by the present work which falls rather into his category of 'modular programming' (P.193), best exemplified by the production systems of Newell and Simon (1972). The representation used in the present work is more extreme than Winograd's in that even a word is represented as a print routine. When long term memory is searched in order to interpret a line of input, the search argument and keys are procedural and a matching process takes place, to be specified in detail in Chapter 4.

3.1.3 Lists are used for subroutines

When a procedure structurally contains subroutines, they are stored in an attached list. The example of three asterisks presented above is actually represented as follows.

```
NOOP CMARKER; ASSIGN X; CALL X; CALL X; CALL X;
```

The marker directs the interpreter to obtain the next subroutine from the list. The attached list of contained subroutines has one item:

```
CALL '*' PRIN;
```

Something like this embedding happens in POP_2 when a lambda expression appears inside a function. In PROCESS 1.5 however,

it is possible to make the structure recursive and this often happens. One case is the meaning of a word: to achieve the STM-LTM complementarity mentioned in Chapter 1 the meaning is coded as two mutually recursive procedures. This should not be confused with the functional recursion that was alluded to in Chapter 1 as arising out of the nature of LTM. That must await a fuller explanation in section 4.3.

3.2 Processes

A process is defined to be the execution of a procedure. So in Newell's (1972) terms the process is the activity while the procedure is the material. The computer program deals only with these two types of data structure (apart from incidental technicalities).

A procedure may be executed many times including the possibility of recursive execution, and on each occasion a new process comes into being. A data structure is created by the interpreter of PROCESS 1.5 to govern and represent the execution and is known as a state descriptor. More loosely, such an object is also called a process. Its purpose is to accommodate the values of the local variables of the procedure, to keep note of which instruction is being executed, to point to the descriptor of the caller so that return may be made to it, and to provide for the proper evaluation of the non-local variables referenced in the procedure.

The programming system was developed from PROCESS 1 (Stansfield, 1972). That system provided a generalised control structure which can be compared to that put forward by Bobrow and Wegbreit (1972) and used in CONNIVER (Sussman and McDermott, 1972; revised 1974). State descriptors are similar to "frames" in their terminology. The most important advantage of PROCESS 1 over the other two is the ability to re-activate a suspended process in a control environment other than that in which it was initiated.

PROCESS 1.5 pushes the generalisation further to allow the full flexibility of alternate access environment as well as control. In consequence it has been necessary to solve the frame problem for processes as well as for procedures which means that the process analogue of the FUNARG problem (Moses, 1970) is dealt with. The extra flexibility achieved thereby makes it possible to construct entities like the frames that Minsky (1975) proposes (see section 9.3). The memory organisation which uses this flexibility to carry out that construction is presented in Chapter 4.

Another unique feature of PROCESS 1 was its system of levels. They are more general than the two levels of SIMULA (Dahl and Nygaard, 1966), a detailed comparison being given by Stansfield (1974). In the present work they have been found too rigid for use and PROCESS 1.5 has a more flexible formulation that retains a helpful property. It is described in Appendix 1. Stansfield tied the binding of variables to the levels in the control structure and so did not encounter (nor solve) the above mentioned variation of the frame problem, which might be termed the PROCARG (process argument) problem.

Hewitt et al., (1973) go so far as to abolish the whole edifice of context and side-effect which gives rise to these frame problems. However, experience with DISCO suggests that context and side-effects are genuine needs in programming for Artificial Intelligence and that it is important to try to solve the problems that arise when these concepts are used. The ACTOR formalism is powerful enough to reconstruct these phenomena (cf. Hewitt, 1975) but that would obviously be pointless by itself. Hewitt hopes that the formal elegance of using a single type, together with other restrictions, will facilitate the implementation, amongst other things, of a

programming apprentice (Smith and Hewitt, 1974). DISCO relies on the simplification to two types, the procedure and the process, while at the same time using a generalisation of existing ideas about context. The final justification for a programming system is successful use and it will be interesting to see how the group at M.I.T. progress in this direction.

3.2.1 The Run-time Structure

At any given time there can only be one current process. All other processes are suspended, which may be because they invoked another, returned or rose to another or because they were copied. Through the current process all its antecedents can be accessed and all other processes must be held in variables belonging to one of these. Hence the entire state of the machine is represented in this structure, known as the run-time structure. Chapter 4 shows how it is used as a memory.

3.2.2 Other Points of Comparison

The régime for levels put forward by Stansfield is here called absolute and in PROCESS 1.5 they are implemented compatibly with relative levels. In that scheme, a procedure may be invoked by CALL or by RUN. Note that the latter must be accompanied by a level number, CALL being equivalent to RUN 1. A procedure may return to its caller by means of the RETURN instruction. Alternatively it may return to an earlier state, bypassing the intervening ones, by issuing RISE n, where the number n must match the level number given in the RUN that caused the suspension of the process to which the rise is now intended to go.

In PROCESS 1, the user declares variables to be associated with a particular level number and that convention has not been carried over into PROCESS 1.5 Instead, dynamic binding is used as in LISP

and POP_2. The implementation of dynamic binding in POP_2 is particularly efficient because of the stack mechanism on which the language is based. Bobrow and Wegbreit (1972 and 1973) succeed in retaining this efficiency for more general control structures by introducing a scheme of numbering the branches on the dendrarchy built by the control primitives. It is then possible to retain global value cells for variables and so avoid a search back along the return chain to find the correct binding for a non-local variable. Instead a shorter search of the numbered value cells held globally for the particular identifier is sufficient. The same technique is used in CONNIVER for implementing contexts in data base manipulation. A superior technique is presented by Wegbreit (1975).

None of these methods is general enough for the ability to run a (suspended) process in an environment other than that in which it commenced because the numbers would need to change. Thus, the ENVEVAL primitive of Bobrow and Wegbreit can only be applied to a form (i.e. a procedure) and not to an environment descriptor. Such an ability is essential for DISCO because its learning consists of collecting suspended processes which are to be run in arbitrary future environments. Furthermore, they may have associated with them stored contexts which must also be instantiated (and augmented) in diverse environments. Stored contexts are a special case of the memory constructs to be presented in the next chapter.

Efficiency measures have been taken, nevertheless. A scheme for direct retrieval via a key has been developed for the contents of a stored context. The idea of keeping a list of the non-local variables referenced in a procedure is an efficiency measure for variable look-up. Every state descriptor points to the value cells of the referenced non-locals so that the search for the bindings

UNIV. EDINBURGH

only takes place once for each instantiation. Variables that the compiler finds to be purely global are located directly without search. Users who modify procedures without using the compiler are responsible for updating these tags and lists.

ORGANISATION OF MEMORY

Chapter 4.

The organisation of memory depends on the concepts of process and run-time structure outlined in Chapter 3.

4.1 Motivation for the memory organisation

The problem is to find a general way to characterise events and situations. Some researchers in Artificial Intelligence, notably Hayes (1970), have used quite abstract formalisms to describe events in the world independently of the perception of those events by someone or something. It is here contended that an essential part of intelligence is the ability to progress to such a relatively objective view of the world by abstracting from experience (i.e. learning). Even leaving aside the question of learning for a moment, the ability of the individual to relate his objective representation to his present experience is still vital. The conclusion is, then, that it is both necessary and sufficient to characterise events and situations from the point of view of the experience of the individual.

In the program, the concept of run-time structure affords this possibility, since it can hold complete information about the experience and behaviour of the active entity. Furthermore its form is promisingly conducive to several goals. First, it is simple to discard records of low-level processing so that unimportant details can be discarded. Second, the run-time structure provides access to those procedures that were involved in an experience or action and are therefore the most likely candidates for amendment. Third, it is both uniform and completely general. Fourth, its relationship with the other basic object in the system, the procedure, is definite.

To be more precise about this relationship the following is true. A process may contain other processes and a procedure other

procedures. A process must contain at least one procedure (of which it is the execution) and may contain more (as the values of variables). A procedure cannot contain a process. Both objects are executable. Both can be amended and copied. A process can only be created by starting a procedure. Because the creation of a procedure is also a process the converse is true as well.

As a representation of experience the run-time structure suffers from two limitations. First, in the course of time it will become indefinitely large and so methods of pruning and abstracting must be found. Second, it is not succinct with respect to the problem of recognising similar or analogous situations. What is more, it is not enough for an intelligent, or a learning, entity to be able to recognise situations; it needs to know what to do about them.

The solution to these problems is to synthesise procedures. An example is given in Chapter 3 of a procedure that characterised repetition; the representation was a procedure for repeating and it serves the dual purpose of doing it and of recognising it. Doing it involves executing the procedure. Recognising it requires that the program be able to note the equivalence between two experiences of repetition by matching the text of the procedures that it is able to construct from the two experiences. The extension of this principle involves establishing new primitives for memory insertion and retrieval. Basically what happens is that synthesised procedures, in a suitable form, are stored in the long term memory and only the run-time structure for the few most recent events is retained to form the short term memory. The earlier records are discarded.

4.2 Short Term Memory

The short term memory (STM) consists of the run-time structure for the few most recent actions and experiences. So insertion to

STM is automatic; anything that happens is recorded there apart from unimportant details.

The regulation of size in order to determine when to discard earlier material could be done in several ways. The most obvious is to measure the total amount of storage space and remove the earliest records when this exceeds a reasonable limit. Another method is to set a limit on the number of procedures in STM, since this is the key factor in determining speed of retrieval. In the present implementation it merely retains the records associated with the five most recent lines of input on the teletypewriter.

As was pointed out in Chapter 2, if a technical means were devised for storing the contents of STM in perpetuity, the requirement for retrieval would be the constraint for limiting STM because retrieval necessitates a search. It seems undesirable to lift a rule, such as Miller's (1956) "Magic Number 7 ± 2", from psychological theory, especially when the reasons for such constraints in human memory are by no means understood. It is much better to choose the criterion that is most suitable for the program, which might also provide a model for psychological theory.

4.2.1 The STM Interface

In order to retrieve from STM it is necessary to invoke the STM interface (more properly the STM interface function) and such invocations constitute the building blocks out of which most procedures are synthesised by the program. The object of invoking the interface is to locate a reference to a procedure somewhere in the run-time structure, either as an executed procedure or as the value of some variable (and so held in a value cell attached to some process descriptor).

For example, the following might happen.

: PRINT AN ASTERISK

 *

: YOU PRINTED AN ASTERISK

The procedure that is run when the word ASTERISK is encountered on the last line (which procedure was itself synthesised earlier on) will invoke the STM interface with search argument equal to the basic meaning of "asterisk", viz.

CALL '*' PRIN;

This procedure will be located in the record of events in the previous line.

Here is another exchange that might carry on from the above.

: WHAT DID YOU PRINT

 ASTERISK

: WHAT DID YOU SAY

 ASTERISK

The word SAY will invoke the STM interface with search argument equal to the procedure that is run when something is said.

Each of the words in the sentences makes its own contribution to the lookup in STM and to provide completeness five parameters can be specified. They are given here.

SEARCHPROC	: The procedure to be located in STM
EXECPROC	: Restricts the search to processes that are the execution of the procedure in EXECPROC
IDENTIFIER	: Names a variable so that only the contents of variables of this name will be searched.
SUBORDINATE	: Specifies a process that must be subordinate to those searched.
SUPERORDINATE	: Restricts the search to those processes that are subordinate to the one specified in SUPERORDINATE

These parameters may be specified in any combination. The result of invoking the STM interface is a truth value and, if "true", a reference to the place where a match was found.

The idea is to match the text of the content of SEARCHPROC with constraints dictated by the other parameters. If SEARCHPROC is undefined there are default meanings for the others, as follows.

1. If EXECPROC is given, an execution of that procedure will be sought.

2. If IDENTIFIER alone is specified, the first use of that variable name, regardless of content, will be returned.

3. If none of the first three parameters is given, the result will always be "false".

Instead of containing a variable name, IDENTIFIER may indicate, by a special token, that only execution procedures are to be searched. In that case only one of SEARCHPROC or EXECPROC can be used, the former taking precedence if both are defined.

4.2.1.1 Exact and Partial Match

The matching of EXECPROC in the search must be exact. That is, the two comparable procedures must either be structurally equal (i.e. they are two references to one and the same procedure) or their texts must match exactly, in a sense defined below.

For SEARCHPROC, partial matching is allowed. In a case like the following,

: *,

: AN ASTERISK AND A COMMA

"*," will be represented as

CALL '*' PRIN; CALL ',' PRIN;

The meaning of "asterisk" will be the SEARCHPROC argument for an STM search and will match into this. Arrangements are then made for

the remainder to be passed on to the word "comma". Partial matching applies to embedded subroutines as well. The meaning of "two" is the procedure for repetition with the subroutine not complete. This is marked by the presence of a null procedure, normally the standard one known as "Empty".

Thus we have:

```
NOOP   ↓    ; ASSIGN X; CALL X; CALL X;
   ⎴‾‾‾‾‾‾‾⎴
   "Empty"
```

If supplied as the SEARCHPROC to STM, this procedure will match with one having the same text apart from the subroutine in the place of EMPTY.

4.2.1.2 Results

If STM retrieval is successful, the value "true" is returned and if the match is exact or if SEARCHPROC is not defined then the result consists of a state descriptor (i.e. a process) and an identifier which is either a variable name or the execution token. If STM finds a partial match there is an expanded representation of the results as five items. The two basic results are known as FOUND and REFERENCE. The three extra parameters for partial match are known as PARM1, PARM2 and PARM3. On any given occasion, no more than one of these three is ever defined. There are four possible kinds of partial match but in the fourth kind all PARMs are undefined. Their purpose is to furnish a pure representation of the type of match in a form in which they themselves can participate in further matching once they have been assigned and take their own place in the run-time structure. The content of REFERENCE in these cases of partial match is a pointer, and is an example of departure from a pure representation for incidental technical reasons as intimated in Chapter 3. Such pointers, by the way, are opaque to STM search.

The precise fashion in which the run-time structure is traversed and the question of efficiency will be discussed after a detailed treatment of the matching conventions.

4.2.1.3 Types of Partial Match

The matching problem is made simple by the fact that all the procedures are constructed by the program or are there to start with. It is not necessary to cope with arbitrary variations such as equivalence apart from the use of different names. Simple word for word, or character for character, equality is all that is required. Matching could be done by a single hardware instruction for comparing strings. Although this is available on many computers today, the present implementation is on a DEC system 10 which does not have that facility.

The four permissible kinds of partial match are these.

I The SEARCHPROC matches the end of a procedure; all the text of the SEARCHPROC appears and is matched. The result in FOUND is the containing procedure; in PARM1 is written a procedure constructed from that portion of the text not contained within SEARCHPROC, i.e. the text in front of the common portion.

II The SEARCHPROC matches the front of a procedure; all the text of the SEARCHPROC appears and is matched. FOUND is the same as in case I and PARM2 contains the text following the common portion.

III The SEARCHPROC exactly matches a (structurally embedded) subroutine of a procedure. FOUND has the procedure that contains the matching subroutine; in PARM3 there is placed a copy of the one in FOUND with the subroutine replaced by the null procedure "Empty" (to be compatible with IV overleaf).

IV The SEARCHPROC matches a procedure apart from a differing subroutine which corresponds to an empty one (i.e. having no text) in SEARCHPROC. The result in FOUND is the differing subroutine; none of the PARM's is defined.

4.2.1.4 Searching Nested Subroutines

In seeking a match within a procedure, all eligible subroutines are searched down to all levels (allowing for recursive loops). Hence, if the result from STM is a partial match one cannot tell at what depth in a nested procedure this might be. The way to find out is to make repeated calls of STM with the FOUND procedure becoming the new SEARCHPROC each time. It must eventually reach the top at which time the result in FOUND will be a process. Notice that in such a loop FOUND must be copied when assigned to SEARCHPROC because, whereas an EXECPROC may match itself, a SEARCHPROC may not. Otherwise, the STM interface function would be in danger of locating the search argument in the process that has just invoked it and from which the argument has just come.

Repeated calls to STM to explore a deeply nested procedure can be made more efficient by maintaining a suspended process for the STM interface which is run several times. That way, it checks that the last result is still applicable before commencing a fresh search. This is done extensively in the program with the consequence that usually the run time structure need be scanned only once or twice in interpreting a sentence even though many invocations take place as the meaning is built up word by word.

4.2.2 Efficiency

The key to efficiency in STM appears to be to keep it small or set a time limit on the search. Conceivably an indexing scheme could be introduced for lookup but its maintenance would almost certainly

be more costly than a search, since it would require updating after every assignment, every edit and every time material is brought down from long term memory. Neither would an associative content addressable memory be all that helpful. It would be advantageous in simple cases where there might be only one match to the SEARCHPROC available but where there were several, a search would still have to be undertaken. This is because the order in which the run time structure is scanned sometimes determines the result. The scan begins with the value cells of the state descriptor that invoked the interface; if one of these contains a process it is searched immediately and so on recursively. Only after the values of local variables have been examined is the return state scanned. (Loops are avoided by using unique markers for each search.) To bring about what seems the most natural sequence for STM scanning the order of declaring variables in certain basic procedures of the program has been determined deliberately.

The only other efficiency measure still open seems to be to re-design the layout of procedural text to be more compact. This can be done in a number of obvious ways like having very short variable names to reduce the time taken by the hardware compare instruction since it may not be inconsiderable for a long string. There is likely to be a trade-off between searching time and interpreter efficiency.

4.2.3 The "subordinate" Relations

A loose end has been left in the STM definition concerning the "subordinate" relation, which is intended to capture the notion of precedence in time. Had the program been implemented in the spirit of Hewitt's Actors this would be a simple matter of saying that Q is subordinate to P if P called Q or there exist $P_1,\ldots,P_n$ all called

by the one before with P_1 called by P and Q called by P_n. Such an implementation is quite easy in PROCESS 1.5 although it involves pruning the return chain which is not strictly part of the language and might lead to curious side-effects if variables are bound into processes that have been pruned. There are other considerations which also render such an implementation undesirable. The program returns to a procedure called the Controller at the end of each line of input. Perhaps that is too rigid when contrasted with human control mechanisms but otherwise the contexts established by local expectations would not terminate at the end of a clause or sentence. The effect of this would not be disastrous. It might impair efficiency a little although, on the other hand, it could have an important bearing on the establishment of more global contexts.

Having made the decision to retain the vestiges of a hierarchical convention through RETURN and RRISE, the "subordinate" relation between processes has to be rather complicated and this fact does suggest that the decision ought to be re-examined. In fact, the alternative (non-return) convention is already in use by the program for the Controller itself. It initiates the interpretation of a line of input. It never returns: it invokes itself recursively at the end of a line and the return chain is pruned to limit the size of STM.

That decision would not affect the arguments for the procedure-process distinction and the importance of context. It would, however, obviate the need for levels.

4.2.3.1 Definition

The "subordinate" relation is best explained with an example,

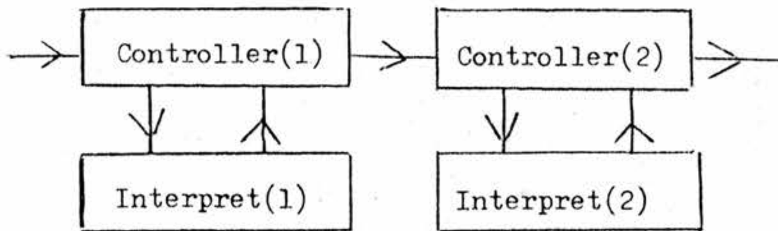


Fig. 4.1 Illustration of "subordinate" problem.

The kind of problem that arises is illustrated in fig.4.1 where two sentences have been processed. Imagine the current process to be within the box marked Interpret(2) interpreting the second sentence. It is necessary for the process of interpreting the first sentence, (indicated by the box Interpret(1)) to appear superordinate (opposite of "subordinate") to the current process, even though it did not itself call the current process but merely returned to the Controller. The Controller recursed which led to Interpret(2).

In order to decide if Interpret(2) is subordinate to Interpret(1) the algorithm is to link back along their respective return chains to find the intersection which in this case is Controller(1). Since there is another execution of the Controller on the chain between Controller(1) and Interpret(2) but not between Controller(1) and Interpret(1) we have Interpret(2) subordinate to Interpret(1).

More formally, define the relation $<$ such that $Q<P$ if P is on the return chain of Q. Then $Q<P$ implies Q subordinate to P. We also have that a process is subordinate to itself. Now if there exists R such that $Q<R$ and $P<R$ then Q is subordinate to P if there also exists S such that $S<R$ and either $Q<S$ or $Q = S$ and R and S are both executions of one and the same procedure. The effect of the option $Q = S$, referring to fig.4.1, is to allow Controller(2) to be subordinate to Interpret(1).

4.2.3.2 Implementation of Constraints during STM Search

When SUBORDINATE and/or SUPERORDINATE are specified in STM retrieval, they and all their return states are marked. Before inspecting a state descriptor, the STM interface must link back along the return chain until a marked state is found - both marked states if both are specified. Then it must apply the algorithm to ascertain whether the current state descriptor satisfies the constraints. If not, its value cells are only searched for further state descriptors on which to repeat the exercise. If both parameters have been specified, it is possible to terminate the search if a state descriptor bears both kinds of back chain markers; it must still be searched but not its return state. Otherwise processing continues until the top level is reached.

4.2.4 The Imperative Option

There is an accompaniment to STM processing which is not implemented as part of the interface but is logically close to it and always follows the STM calls made by synthesised procedures. If STM returns "false" and SUPERORDINATE has been specified then an attempt is made to create an imperative process. The attempt will succeed if there is enough information. If IDENTIFIER contains the execution token and SEARCHPROC is defined, an initial state for that procedure will be created. If EXECPROC is defined, it will become the procedure of the initial state. If SEARCHPROC and IDENTIFIER are also defined, the former will be assigned as the value of the latter in the new state. The state will be frozen into the access environment of the superordinate process and its return pointer will be directed thereto so that not only can it be invoked by means of CALL or RRUN but it also satisfies the subordinate relation even before its invocation and thus will preserve the consistency of STM.

4.3 Long Term Memory

Whereas in short term memory insertion is automatic and retrieval takes place via an interface function, with long term memory it is insertion that uses an interface and retrieval is notionally automatic. In fact retrieval takes place at certain strategic points in the program for reasons of efficiency.

The content of LTM is a set of processes. They are all waiting for events to take place and each is marked by the characterisation of the event or situation in which it is to be run. The characterisation is to be made in terms of procedures, using the parameters SEARCHPROC, EXECPROC and IDENTIFIER as is done for STM.

It is at the time of insertion to LTM that these parameters must be specified and that is when the LTM interface must be invoked. So it is that insertion to LTM is complementary to retrieval from STM and the two interfaces are externally identical. Whereas in invoking STM the near past is being searched, by calling LTM one is undertaking a search of the future. The process that invokes LTM is suspended until the specified situation type arises.

Many of the processes in LTM are for the interpretation of words, having been put there when the word was learnt. For example, the word "dot" has a process in LTM with SEARCHPROC the procedure for printing DOT, with EXECPROC set to the Controller and IDENTIFIER indicating input from the teletypewriter. When a sentence is read the Controller is called and is given the procedure that the program has just synthesised for printing the sentence.

If "DOT" appears there, the process in LTM which was created when the word was learnt will be activated.

A rather different example would be the case of a process waiting for the program to produce output, which is always done by

the Output Phase. If a procedure is passed to it as direct output then it will be performed. In Chapter 1 a device called the "blackboard" was mentioned which the program could use to store characters by printing them with a preceding quotation mark. A process in LTM with SEARCHPROC equal to

CALL '"' PRIN;

and with EXECPROC and IDENTIFIER indicating direct output in the Output Phase would intercept this activity when the Output Phase was started. It could do anything at this point. Something useful it can do is to predict the final state of the blackboard.

A third example demonstrates what happens when SEARCHPROC is undefined. There is a procedure for analysing feedback from the program's external activity such as printing and writing on the blackboard. The simplest rule that the program first learns about feedback is that for straightforward printing the result matches the print procedure. This rule is enshrined in a process in LTM that has EXECPROC set to the Feedback Analyser and IDENTIFIER equal to the appropriate variable name. The setting up of this process in LTM is tantamount to the insertion of an unconditional branch in the Feedback Analyser. Whenever this component of the program is called in the future, that process in LTM will be activated and will verify that the received feedback matches the action of the program (using STM), taking appropriate measures if it does not.

4.3.1 Equivalence to Conditional Statements

If an undefined SEARCHPROC gives rise to an unconditional branch, the presence of a procedure in that parameter amounts to the provision of a conditional branch - conditional, that is, on the presence of a matching procedure in the given variable during an execution of the procedure specified in EXECPROC.

A set of processes in LTM all waiting for a particular variable of a given procedure with various values amounts to the provision of a dynamically constructed COND statement (as in LISP) in that procedure. This is like a sequence in POP_2:

IF...ELSEIF...ELSEIF...ELSE...CLOSE;

With an efficient indexing scheme for LTM this is an extremely fast implementation of such a useful feature as well as being highly flexible. The speed advantage is a matter of principle and not merely technical, because it is the difference between evaluating all the conditions until the right one is found, on the one hand, and a direct access on the other. POP_2 has a feature called "switch" that takes a number i and branches to the i-th label; it provides direct access to the right label but only for elementary numerical problems. Neither is provision made for dynamic modification of the list of labels.

A special case arises when SEARCHPROC is defined to be an empty (or null) procedure. This amounts to an insistence that the variable named in IDENTIFIER must contain a procedure but it need not be any particular one. If, on the other hand, SEARCHPROC is undefined then any data type is allowed.

4.3.2 Precedence and Matching

There has to be a scheme of precedence in LTM to choose between the several stored processes that might be applicable in a situation. If there are two processes with the same specification, the one more recently inserted to LTM takes precedence. Otherwise it is the most specific that is invoked so that, all else being equal, a longer SEARCHPROC that matches will be invoked in preference to a shorter one.

As with STM, the EXECPROC and IDENTIFIER must be equal or match exactly whereas partial matches, on a restricted basis, are allowed with SEARCHPROC. Between these three arguments, EXECPROC has the highest precedence followed by IDENTIFIER and then SEARCHPROC. So a process that merely specified EXECPROC equal to the Feedback Analyser with the other arguments undefined would be invoked in preference to one that had EXECPROC undefined but still matched on the other two arguments. However, if there were two processes with EXECPROC equal to the Controller, IDENTIFIER indicating teletypewriter input and the search arguments set, respectively, to the procedures for printing the words "a" and "asterisk" they would each be invoked at the right time when their words occurred because of the rule that the longer match takes precedence.

4.3.2.1 Restricted Types of Partial Match

It is not necessary or desirable to permit matching of the search procedure as flexibly as in STM. Only two and a half of the four options are allowed. The restrictions are to prevent a search taking place. The idea is to scan the candidate from left to right, activating the relevant processes from LTM just as human speech must be processed, initially, in the temporal sequence in which it occurs. It is the function of grammar (see section 6.5) to impose structure on the sequence as Halliday (1961) clearly states. This idea has been applied to LTM retrieval generally in the present work.

As for STM, the SEARCHPROC must be found completely within the candidate procedure but now the match must be in front with an optional remainder at the end of the candidate. This corresponds to possibility II of STM retrieval as listed in section 4.2. Notice that the candidate must be at least as long as the SEARCHPROC but it is the SEARCHPROC that is in LTM; the candidate is, for example, the

current line of input or whatever is responsible for triggering the retrieval.

A restricted version of type III can occur (this is the half option) whereby a SEARCHPROC in LTM can match a (structurally embedded) subroutine of the candidate. This is only allowed where LTM also contains a match to the higher portions of the candidate.

If a match of type IV occurs (i.e. equivalence apart from a differing subroutine) the system first checks to see if the differing subroutine can itself be matched in LTM. If so, a type III result is returned. The reason for this preference toward type III is illustrated in the example below. Otherwise type IV will be the answer. A type III result is also possible if the candidate is matched in a manner that is deficient both in the type I and the type IV respect; that is to say if the match is with only the front portion of the candidate but even that common portion differs by a subroutine (with an empty subroutine in the corresponding place within the content of SEARCHPROC in an LTM record).

An example of the last kind of match would arise when the program represented the line of input "**," as the procedure:

```
 ↓ → X; CALL X; CALL X; PRIN (',');          (1)
/PRIN ('*');\
```

The LTM would, after suitable learning, contain SEARCHPROC's

```
PRIN('*');                                   (2)
PRIN(',');                                   (3)
and   ↓ → X; CALL X; CALL X;                 (4)
   / "Empty" \
```

The system of LTM retrieval would directly locate procedure (4) as a match to (1). This does not fall into any of the types I to IV and there is no convenient way to represent the two remainders for further

analysis. The system therefore obtains the match to

PRIN('*');

This is a normal type III result and illustrates why this type of result is sought in preference to type IV. The content of PARM3 thereafter would be

```
 ┌──>X; CALL X; CALL; PRIN(',');
 ↓
/ "Empty" \
```

This result can now become the new candidate for retrieval against LTM and it can be seen that it now forms a type II match against the procedure (4), producing in PARM2 the remainder

PRIN(',');

This candidate can trigger an exact match retrieval from LTM and analysis of "**," is complete.

In summary, LTM matches can be of type II, IV, and III with restrictions. Note that there is no restriction on the number and complexity of subroutines, but only one of them may differ in a match. The restrictions are defined in order to avoid searching LTM, which may be indefinitely large. As the example illustrates, these types of match are adequate because once the front of a procedure has been matched from LTM, the remainder becomes the basis for another retrieval. This process can continue until all the text is exhausted and is the normal method by which the program interprets a sentence. This will be elaborated in Chapter 5. It is the purpose of grammar to overcome the limitation of left to right processing.

4.3.2.2 Repeated Calls to the Interface to unravel Nesting

After a partial LTM match, the same kind of repeated calls can be made as for STM in order to find the top level procedure and its place in the run-time structure. However, these calls must be to STM, not LTM, and the necessary parameters may need to be copied from

an LTM invocation to an STM one. This is because a call to LTM suspends the current process. Complementarity is not disturbed by this copying requirement. The first call may be changed from LTM to STM in the complementary version without disturbing the subsequent repeated calls. The program performs such substitutions by reassigning variables dynamically as will be seen in Chapter 5.

4.3.3 Processes in LTM

Recall from Chapter 3 that a process can suspend itself by means of the RRISE primitive. At that time, a process descriptor comes into being. This comes about as the level number of the RRISE is written into the state descriptor for the execution of the procedure that contained the RRISE instruction. The process descriptor thus consists of that state descriptor plus all those on its return chain up to, but not including, the state descriptor to which control was passed by means of the RRISE. The LTM interface issues such a RRISE and so the procedure that invokes LTM must specify a level defining the extent of the process to be saved in the long term memory for subsequent use. It can be written in any of the forms given in section 3.2, e.g.

```
LTM (PFINDP(ANALYSE));
```

will cause all processes below the execution of "Analyse", including the current one, to be saved in LTM as one process for later reactivation and an immediate return to "Analyse" will be effected. For compatibility, a level number is given to the STM interface as well but it is ignored.

When processes in LTM are retrieved and re-activated they are copied; the read-out is non-destructive. Only the state descriptors, including the value cells, are copied, however. If during execution

one of the procedures is modified, that change is permanent in the long term memory.

When a process descriptor is set up, the return states at or above the level number are not properly a part of it but the return pointer, and access pointer if freezing has taken place, are normally not deleted. They are necessary to determine whether the access environment of the variables of the process has changed between suspension and re-activation. When a process descriptor is copied, only the state descriptors below the level number are copied. It is obvious that processes in LTM will be re-activated in different access environments and so in this implementation such pointers are deleted and the process descriptors are stored in a compressed form.

4.3.3.1 Recollections

The question of whether links to the record of events that led to learning should be saved is interesting from the psychological point of view. Empirical evidence is inconclusive although the behaviour of Bartlett's (1932) subjects in slowly uncovering more details of distant recollections might conceivably be accounted for by following such links from LTM and interpreting them. At present the program could not produce verbal interpretations like Bartlett's subjects because it does not possess an adequate command of any language. The representation of events stored in this way is no different in kind however, from the immediate record in its short term memory and so no fundamentally new ability would be needed for the program to indulge in recollection. It deletes these "memories" because it has no room for them but another implementation could save them on some suitable medium.

4.3.4 Organisation and Access

Whereas STM is of limited size, LTM is potentially enormous. In the pilot implementation, LTM retrieval is conducted by exhaustive

search but an indexing scheme is proposed for large-scale applications. Since the operation of the program after it has acquired many abilities through learning will consist mainly of calls to LTM and STM, efficiency is a consideration of much more than technical importance. Psychologists have for some time believed that the organisation of human memory into STM and LTM plays a central role in mental activity and it is appealing to find a computer program in which this is also true.

The unsuitability of content-addressable memories was considered in Chapter 2. The more conventional organisation is to have data associated with keys so that one datum is retrieved when a key is specified. Compound data can then be represented by cross-referencing between items in one or more sets. The arrangements for access depend on the purpose. For LTM, only random retrieval is needed. Unfortunately a hash coding technique cannot be used because of the requirement of partial matching. As it is, a simple index on a key is called for, and occasional re-organisation will be needed as the size increases. An associative memory would not encounter that overhead although its logical behaviour from the point of view of the answers it gave would be the same.

It is important to distinguish the kind of index proposed here from what Sussman and McDermott (1974) call an index. Patterns in CONNIVER may have combinations of variables and fixed items in any sequence. Therefore the kind of direct access key retrieval described in the present work is impossible for CONNIVER. What they call indexing is a technique of collecting into "buckets" those patterns in the data base that contain a given constant in order to reduce the search somewhat. The final selection is then made by a user-written program from a "possibilities list". Such a scheme

places on the learning system the additional burden of constructing procedures to perform the selection (see the discussion in section 9.1).

Although an index has not been implemented here it should be clearly understood that such indexes are commonplace in computer systems and present no difficulty. The hard part of the problem is to find a formulation of pattern that is succinct enough for direct access via a key while being general as well. This has been achieved in DISCO.

Another point from psychology in support of this scheme is the evidence from neurophysiology that the speed of transmission of information in a nerve fibre is slow, about 300 feet per second, when compared with an electronic computer's 60,000 miles a second (about one third of the speed of light). Nevertheless, in order to carry out tasks that we can do, like planning an assembly by a robot using toy bricks, they are many times slower and usually run into a combinatorial explosion on more sophisticated problems. The key to the higher human performance is presumably in mental organisation. One thing that is accepted is that the brain possesses a fast random-access memory. The slow speed of neuronal conduction seems to indicate that only small amounts of activity are possible in between memory accesses and this accords with the DISCO model. The alternative explanation of human high speed performance is that much parallel processing is conducted. This program does not support that view.

4.3.4.1 Definition of Key Format and Lookup Algorithm

Having stated the case, it remains to detail the layout of the keys by which the data in LTM will be known. A key will be a concatenation of the text of the procedure in EXECPROC, followed by the content of IDENTIFIER and concluding with the partial match

component, the text of SEARCHPROC. If one of the first two parameters is undefined it will be denoted by UNDEF. For economy, basic procedures (i.e. not synthesised by the program) can be represented by their names. In order to retrieve, the system must construct a key for the current event and look it up in the index, which will be sequenced to give the longest match first (e.g. "asterisk" before "a"). Failure will prompt the system to replace IDENTIFIER by UNDEF and try again. Three more accesses with EXECPROC undefined would be necessary before concluding that no match could be found. First IDENTIFIER would set equal to the given variable name; next the system would try the execution marker in IDENTIFIER; and finally it would set IDENTIFIER to UNDEF. So five accesses would be the worst case; only one would be required for normal purposes like word look-up.

The above scheme could be implemented efficiently for large-scale use by distributing LTM across at least five direct access devices so that all the hardware movements can be performed in parallel while at the same time the central processor is scanning the local contexts.

The algorithm presented here is the embodiment of the rules of precedence for LTM retrieval given above. They are also embodied in the simpler algorithm employed by the system as implemented at present.

4.3.4.2 Initiating Retrieval

Retrieval from LTM is carried out by the program at strategic points planned in advance. The primitive is called SEARCHLTM and it expects arguments indicating the process and the variable on which to perform a retrieval. It then activates the process from LTM if one is found; otherwise it returns to the caller.

To make LTM retrieval automatic would necessitate calling this primitive once for each local variable in a procedure for every entry and exit. It might be possible with new technology to run these searches in parallel and simultaneously start the procedure, discarding it if a search is successful but this avenue - or rather, this thorny path - has not been explored. So far, SEARCHLTM has been adequate, although there is one embellishment that looks as though it would be useful without costing too much. It involves giving a set of variable names to SEARCHLTM (e.g. FOUND and PARM's 1 to 3) to avoid having to re-enter the index at the top level for each identifier in turn.

4.3.5 Recursion

An important natural consequence of the way LTM is defined is the occurrence of recursion. The capability arises essentially because the EXECPROC argument is allowed to match a candidate textually if the procedures are not structurally one and the same. As has already been implied, if the program needs to extend one of its procedures it sets up a process in LTM with EXECPROC equal to that procedure and with conditions specified by the other two arguments. When this process is reactivated, if it calls an exactly similar procedure, the same LTM process will be activated recursively as long as the conditions apply. This happens when the program builds up the meaning of the plural morpheme "-s" in a training sequence like the following.

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

As a result of the first two lines, a process is written to LTM as the meaning of "-s", so it has EXECPROC set to the Controller, IDENTIFIER indicating input and SEARCHPROC the procedure:

CALL 'S' PRIN;

The process, when activated, will invoke a procedure that shall be labelled PROCa and includes the specification:

```
PROCa: STM(SEARCHPROC = [NOOP     ; ASSIGN X; CALL X; CALL X] )
                              ↓
                          "Empty"
```

where the content of the square parentheses is a procedure (structurally embedded in PROCa with "Empty" embedded in it).
PROCa is the basic meaning of "-s" at this stage, i.e. duplication.
After the next two lines, PROCb and PROCc are synthesised and stored in LTM with the arguments shown in PROCb below which is the actual invoker of the LTM interface and gets suspended; it calls PROCc when the process is re-activated on a subsequent LTM retrieval. The last two procedures are similarly created by the last two lines of the above dialogue.

```
PROCb: LTM (SEARCHPROC = EMPTY
            EXECPROC   = PROCa
            IDENTIFIER = "FOUND")
PROCc: STM (SEARCHPROC = JOIN(RESULT, [CALL X] ))
PROCd: LTM (SEARCHPROC = EMPTY
            EXECPROC   = PROCc
            IDENTIFIER = "FOUND")
PROCe: STM (SEARCHPROC = JOIN(RESULT, [CALL X] ))
```

Since PROCe and PROCc are textually equivalent, the line EXECPROC = PROCc that occurs in PROCd will apply equally to PROCe and recursion will take place for plurals where the number of objects exceeds four. The recursion will terminate when the value of the

variable FOUND (the result of the STM search) in PROCe is a process which will happen when enough copies of the instruction

```
CALL X;
```

have been joined on to the SEARCHPROC to match the event that is being talked about in the sentence.

The component of the program that synthesises PROCa will be defined in section 6.3. Section 6.4 will describe the component that produces the other procedures shown.

4.4 Contexts

There is a connection between a suspended process in LTM and a demon (Charniak, 1972), an invention attributed to Papert. The patterns that invoke demons are in the goal formalism of Micro-Planner rather than the procedural form chosen for DISCO. More recently, Charniak (1975) has renounced demons in favour of frames, an idea recently given a fresh airing by Minsky (1975). The LTM formalism in DISCO allows the construction of objects that have all the advantages of frames put forward by Charniak. Not only does DISCO include an implementation of them, something which Minsky and Charniak have not done, but it is also capable of creating them from its experience. For this purpose we define the local expectation, which is a suspended process exactly like those in long term memory, except that it is attached to another process and has a domain which is limited to those processes that lie on a chain of invocation from the process to which the expectation is attached.

A process with local expectations attached to it constitutes a context, somewhat like the facility of that name in CONNIVER (Sussman and McDermott, 1974). It presents the same kind of problem in matching patterns because appeal must first be made to the contexts, which may be nested, before looking at the global LTM. Unfortunately

their method is not general enough for the present case and the same applies to the improved algorithm of Wegbreit (1975). This is because sets of local expectations can be stored, along with the process to which they are attached, in another context or in the global LTM and can subsequently be reinstated in other environments automatically when the process is re-activated.

A context, or frame, is a miniature version of the LTM. In this implementation, the global LTM is a context attached to the highest level process in the system. A process is written into a context by calling the LTM interface with the SUPERORDINATE argument specifying the process to which the context is attached (or is to be attached if this is the start of a new context). It is contradictory to specify the SUBORDINATE parameter to LTM and so it is ignored.

An example of a stored context, or frame, associated with a suspended process is provided by the numeral 2. When "2" is followed by an object, the duplication procedure is to be applied.

```
: PRINT 2 DOTS
  ..
```

Now if the program is taught to count, 2 should not double 3 and 3 triple 4. Thus the meaning of 2 in counting is different from its meaning in a sentence. So the process for 2 will have a frame containing a general expectation for an object and a specific expectation for an occurrence of the "3" procedure which, by the rules of LTM, will take precedence when it is satisfied.

4.4.1 Efficiency

In order to implement these frames efficiently it would be desirable to have each one indexed like the global LTM. The only way that retrieval can work appears to be by linking back along the return chain looking for contexts. One can dictate a starting point for the

search (using another argument to SEARCHLTM) and this saves time if it is known in advance where to start, as is often the case. The rules for precedence of one LTM argument over another only apply within one frame so that the search can terminate at the first frame to contain a match, be it partial or exact.

As a matter of detail, when SEARCHLTM activates a process from a context, or frame, rather than from the global LTM it does not issue RRUN but instead places the process descriptor into the current return chain as if it had been invoked by the process to which the context is attached. A RRISE to it is then performed.

As pointed out by Charniak, the advantage of a frame implementation over the use of demons is that demons must be set up and their variables bound every time there is a possibility that they will be needed whereas with frames only the local expectation that is actually satisfied needs its variables bound. Therefore no time is wasted on the others - and there may be many of them.

4.4.2 Complement of a Local Expectation is the Imperative

It has already been stated that STM and LTM are complementary. Suppose, for example, that there is a procedure that stores a process in LTM to wait for an occurrence of "dot" and, when re-activated, searches STM for ".". By a re-assignment of variables the same procedure can store a process in LTM waiting for "." and, when re-activated, call STM with the "dot" procedure as argument. In this manner the transition from comprehension to utterance in language is made. When SUPERORDINATE is specified to STM (and SUBORDINATE is not) the imperative is indicated. Hence it is complementary to the local expectation. In Chapter 7 there is an illustration of this principle when the program learns to count.

4.5 Summary

In this chapter a powerful organisation for memory has been introduced which bears some relation to psychological theory and provides a complete programming language affording conditionals, recursion, imperative procedures and stored contexts in a form suitable for use by a learning program. The complementary nature of STM and LTM provides a vital transition from recognising patterns and structure to employing them.

In Chapter 5 the flow of control of the program will be given followed by the basic learning algorithm. In Chapter 6, the five components into which it may logically be divided will be enunciated.

PRINCIPLES OF THE PROGRAM'S OPERATION

Chapter 5.

In this chapter the program's mode of operation is described, first without detailed analysis of the learning processes that take place; the common core of those processes was outlined in Chapter 1 and is treated in section 5.2. Chapter 6 will present the parts of the program that use the core.

5.1 Flow of Control

The program operates in a loop which can be characterised as read-interpret-synthesise-act-examine feedback and is illustrated in fig. 5.1. Data are read from the input device, a teletypewriter, and are converted to the form of the procedure that would print them. This conversion is performed by the Reader which invokes the Controller when finished. The input is matched against LTM by the Perception component and the stored process that is obtained thereby is activated (represented by the empty box in fig. 5.1). Typically that means that the process that is the meaning of the first word in the sentence is run. It will itself cause further input to be interpreted (by Perception) and a return to the Controller is made when the input line has been exhausted. At this time one or more results may have been produced as a result of references to STM. If any of them are imperative, the Controller will act on them (i.e. invoke them) and will act iteratively on any imperative results that they may themselves produce. Usually, this involves invoking the Output Phase. Thereafter, the feedback from these actions is inspected by the Feedback Analyser and finally the loop restarts by a recursive call to the Reader.

As fig. 5.1 implies the mutually recursive calls of the Reader and the Controller are all on one level (as defined in section 3.2)

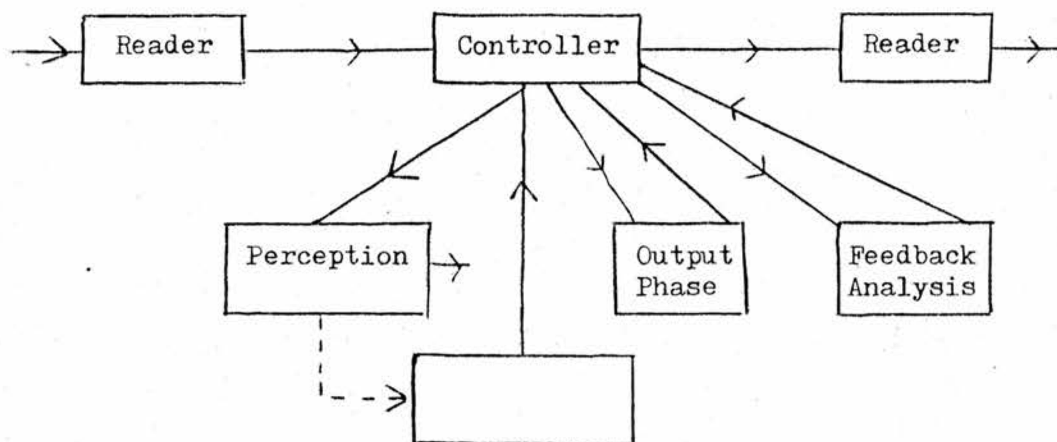


Fig. 5.1 Run-time structure of the program's basic loop

and other processing is at lower levels. The Controller saves the processes below it and they are known as "trails". At the end of each loop it prunes the return chain, cutting off the earliest read-interpret loops if there are more than five. Their dependent processes automatically disappear thereby. This is how the STM size is kept down.

The rest of this section examines the operation in more detail, concentrating on the normal way that the processes operate once Perception has found them. The most important idea here is the necessity for a standard form of communication between acquired processes and a convention for sequencing their activity. The means of communication is closely bound up with the nature of the STM interface which almost all these processes use. A standard form is essential if there is to be a learning capability that has generality.

The sequence in which processes must perform does not always correspond to the order of the words in the sentence. A convention is needed to regulate it and this is based on the left to right sequence of words in the written form, corresponding to the temporal

sequence in speech. Word-processes refer to STM and produce results. These may be passed to the processes for other words on the left or the right-hand side of the current one. The means of doing this has to take account of the fact that words to the right have yet to be interpreted.

The last two subsections deal with the Output Phase and the Feedback Analyser.

5.1.1 Perception and the Unknown

In fig. 5.1, it is Perception that actually retrieves from LTM. It separates the retrieval of a process from its subsequent activation and so uses a slightly modified form of LTM retrieval that is nevertheless consistent with the conventions stated in Chapter 4. The separation allows it to detect repetition and any matches between the input and recent events. These are found by searching STM. A full description of this module will be given in section 6.1.

If the LTM retrieval fails, the New Entity Handler is invoked for the purpose of establishing the meaning of the unknown word or other item. It need not be a word that is unrecognised; it could be an object like a dot, in the early stages, and it might also be a morpheme or the character (a stroke /) that deletes from the blackboard. The Perception component and the New Entity Handler are also used in analysing feedback and so the general names given to these procedures are quite justified. What happens next constitutes learning and will be discussed in section 5.2.

5.1.2 A typical process, activated by Perception

If Perception is successful in finding a match in LTM, the associated process is activated. What follows will obviously depend on the details of the particular word or other entity, as determined by past experience. Unless the meaning is vacuous, as for e.g. the

blank space, one or more invocations of STM will take place.

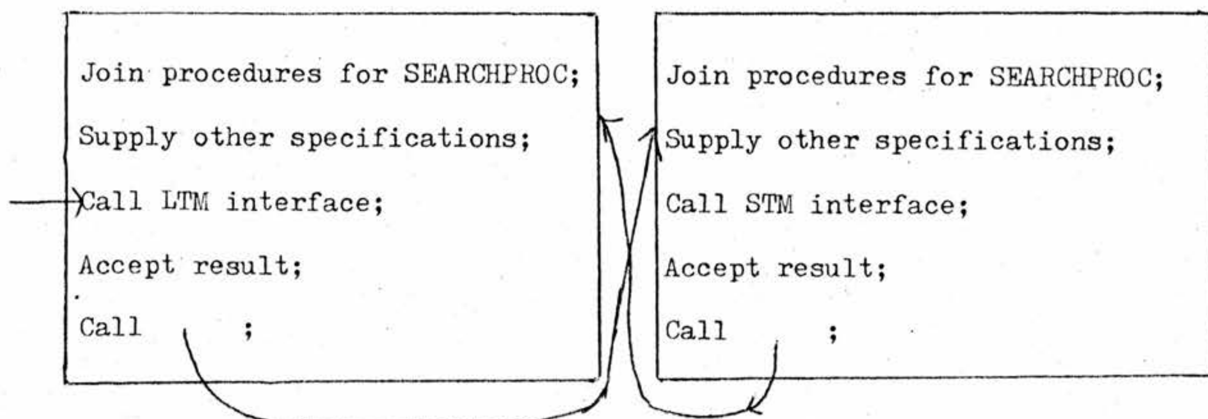


Fig.5.2 Informal representation of a typical stored process in LTM.

A process as Perception would receive it from LTM is illustrated in fig. 5.2. In that diagram, two procedures appear and each has a subroutine pointer to the other so that they are mutually recursive (structurally). The arrow on the left points to the last instruction executed when the process was saved and the process descriptor obtained from LTM contains this pointer.

The process was saved when this word (or other entity) was first learnt. Now that Perception has encountered the word again this process is retrieved from LTM and re-activated. (The read-out is non-destructive so that LTM is not altered in these straightforward cases). If the word in question were "dot" then the SEARCHPROC specification to LTM (left-hand of fig.5.2) would have been

PRIN('D');PRIN('O');PRIN('T');

Upon activation of the process, the right-hand procedure is called (because of the Call instruction with the pointer). In this case the SEARCHPROC specification to STM is

PRIN('.');

The pointer back from right to left is to provide for a call the other way when STM and LTM are reversed in complementary mode.

Section 5.2 contains a description of exactly how this is done. In addition to reassigning the variables that refer to STM and LTM, the effects of the line "Accept result" are also modified (again by re-assigning variables). In fig.5.2, the backward call is never reached because an exit is made to the Result Handler (described below).

For the simple example of "dot", no other arguments are specified to STM and no joining takes place in setting up SEARCHPROC. However in the interpretation of a sentence or other complex pattern several processes can contribute to an STM retrieval specification and some means of communication must be provided between them.

5.1.3 Communication between Processes

Let us begin with another example. In the sentence "Print a dot", the word-process for "dot" will contribute the SEARCHPROC for STM retrieval and the one for "print" (after appropriate learning) will supply the EXECPROC, IDENTIFIER and SUPERORDINATE parameters. Because of the syntax (see section 5.1.6 below), the "dot" process will be constrained to give its argument before the other two word processes. This may by coincidence locate another dot in the recent past if there is one and in some cases, e.g. "Print the character", this might be important. This would be an example of anaphora, a problem not considered in the present work. In the sentence "Print a dot" such a coincidence would not matter. The imperative attribute of the word "print" simply overrules the coincidence (unless it satisfies the superordinate condition, as would be the case if the human tutor supplied a procedure in square parentheses).

It has proved desirable to make the means of communication explicit and this is done by insisting that all STM invocations of this kind be performed through a common interface process which causes execution of the procedure STM when necessary. It serves as

a form of working storage during interpretation of a sentence. It is so designed that repeated calls of it first try to verify that the STM match found during the previous call is still valid and only if it is not is a fresh search of the short-term memory undertaken. Procedures that use the STM interface process can examine the specifications and results already derived, e.g. from other parts of the sentence, before adding in their own. The next subsection can be omitted on a first reading.

5.1.3.1 Interactive Communication

Notice that contributing to the STM interface process need not be restricted to specifying five arguments and therefore limited to five words in a clause. The SEARCHPROC is open ended because of partial matching. Several words can contribute to the final version in the last STM call. For instance the conjunctive particle "and" calls STM twice. In e.g.

: .,

: A DOT AND A COMMA

the word "dot" finds the procedure

PRIN('.');

embedded in

PRIN('.');PRIN(',');

via STM. The processing for "and" receives this result with the PARM2 value (as defined in section 4.2) containing

PRIN(',');

The processing for "comma" matches into PARM2 with "and" when it calls STM and the first STM call made by "and" will join the two together and thus verify constructively the match with the actual event ".,". Such joining also occurs in the meaning of the plural morpheme ("-s") and in many other places. By having the STM interface

in a stored process, such multiple calls are usually verificational and do not require a search.

The constructive nature of verification in words like "and" is essential. Otherwise a command like

: PRINT A COMMA AND AN ASTERISK

,*

would be impossible.

The case of "and" shows up another direction in which the possible range of specifications to STM can be enriched. The result of the first STM call within "and" supplied the argument to be joined to the second. Processing for the word "after" imposes the demand that one result becomes the SUBORDINATE parameter in the first STM call and the result becomes the SUPERORDINATE parameter in the second. Thus one result from STM can contribute to deriving another.

There is a third enhancement to STM specification capability for interpreting a sentence and for any analogous activity there might be. Its implementation is not entirely satisfactory and involves unsolved problems that await further research. The general problem is what to do when a word process is about to specify one or both of the parameters EXECPROC and IDENTIFIER where these are already present in the STM interface process by virtue of the interpretation of another word. In simple cases the arguments are overwritten but sometimes this expedient is inappropriate. There is a class of words that must perform a transformation on the old arguments and, because of the exact match requirement for EXECPROC, it cannot be of the joining and insertion kind used for SEARCHPROC. In the trials so far, no such word has been encountered that is not obvious from the very first example and so provision is made for this possibility in learning meanings. The problem is mentioned again in section 5.2 and once

more in Chapter 10. The first word that falls into this class is the personal pronoun "I".

5.1.4 <u>The Result Mechanism</u>

Whether they contain an STM call or not, all processes obtained from LTM terminate by calling the Result Handler. This is one of the strategic places from which LTM is scanned using the LTM retrieval function and control is given to the process retrieved. (The other places are in Perception and the Output Phase.) The specifications given by the Result Handler to LTM retrieval do not characterise the Handler itself but its caller and the IDENTIFIER named is the variable containing the FOUND portion of the result of the STM search made by the caller.

This is the place, mentioned in section 4.3, where it would be nice to be able to specify efficiently to LTM retrieval the other three variables containing the PARM1, PARM2 and PARM3 portions of the result. The example of the plural morpheme "-s" (in section 4.3.4) would have referred to "PARM2" with SEARCHPROC equal to

CALL X;

had that facility been available, instead of referring to "FOUND" with SEARCHPROC equal to the empty procedure. As that case illustrates, procedures may be extended by having processes in LTM referring to them. One of the functions of the Result Handler is to provide the means (via LTM retrieval) by which such extensions take effect when their conditions are met. It is also a function of the Result Handler to instigate the synthesis of such extensions. This is called differentiation and is described in section 6.4.

Because of the way that frames (or stored contexts, or sets of local expectations) are furnished by the LTM primitives, exactly the same LTM scan that the Result Handler uses to find extensions also

serves to match any appropriate local expectation there may be. Frames (contexts) take precedence over the global LTM (which is the highest frame).

By passing control up to a process located in a frame a result is being passed to the left. This happens, for instance, in "print a dot" when "dot" satisfies an expectation attached to the "a" process and again when that one satisfies an expectation attached to the "print" process. Otherwise, the Result Handler will call Perception to interpret the rest of the sentence. However, its process will remain in the run-time structure (i.e. the short-term memory), and the result that it bears will be usable by a subsequent process. Hence a result may be passed either to the left or to the right and the rudiments of a grammar are taking shape.

5.1.5 A framework for Grammars

In section 6.5 a justification is given for gracing the conventions described here with the appelation "grammar". Although they are reasonably simple they do admit of more richness than might perhaps be apparent at this stage.

Having discussed a vehicle for communication (the STM interface process) and its content we turn to the conventions whereby two processes initiate such communication between themselves. Half the story has already been told. When a process, typically the interpretation of a word, is through, the Result Handler carries out the function of checking whether it satisfies some expectation of an earlier process. If not, the result remains available for use by a subsequent process.

A word like "print" which must look for an object (syntactically) possesses a stored context (frame) attached to its process. The stored process normally resides in LTM and when it is activated its frame is automatically instantiated with it. In order to seek the direct

object (or in general to seek a process that will satisfy the conditions for activating one of the expectations) processing for "print" causes the words following it to be interpreted before it has finished itself. Such an action is termed a seek. Normally, one of the expectations associated with "print" will be satisfied by processing for the following words. Then the interpretation of the meaning of "print" can continue. However, the end of the sentence or clauses could be reached before any process can satisfy any of the expectations. (At most one expectation in a frame can be satisfied for any one instantiation of that frame). A seek with none of its associated expectations satisfied is an outstanding seek.

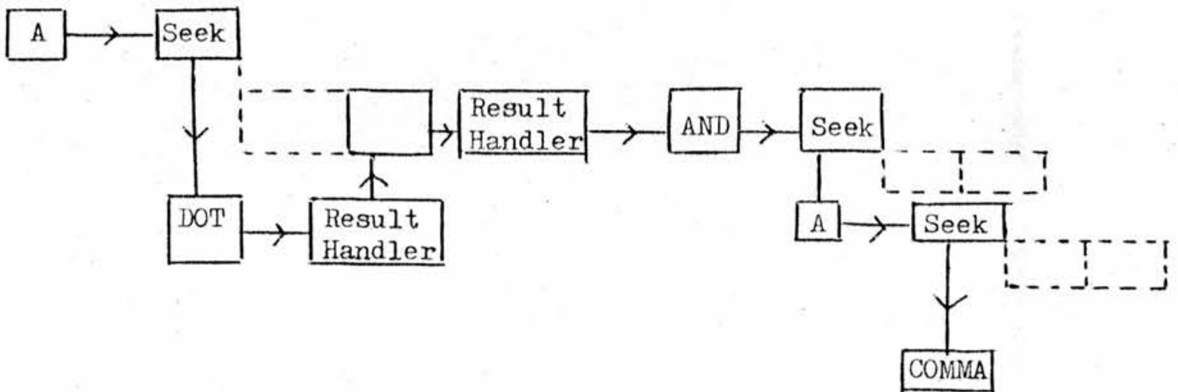


Fig.5.3 Simplified Run-time Structure (Short Term Memory) established by interpreting the phrase "a dot and a comma"

In fig. 5.3 is the run-time structure that is created during the interpretation of the phrase "a dot and a comma". The diagram is a snapshot taken while the process for the word "comma" is active. Notice how the syntactic structure is represented. The idea is that syntax characterises the form of the process of interpretation and is best viewed in that light. Its role in utterance is probably more important.

When a word-process issues a seek, it expects to receive a result from the right. The dotted lines below the seeks in fig.5.3 denote the frames. The DOT process satisfied one of the expectations of A and so returned to it (via RRISE). At the time of the snapshots the seeks from AND and A are still outstanding but COMMA will satisfy them in due course.

The AND process receives a result from the left as well as from the right. An example of a process that receives a result only from the left is that for the plural morpheme "-s". Seeking to the right depends on the LTM primitive. Rather beautifully, seeking to the left uses STM. A result, such as that from "a dot" in fig. 5.3 becomes available in short term memory if no appropriate expectation is outstanding. In this example the AND process seizes it.

Words expect the kind of results that were available at the time of learning and a generalising and refining capability exists for adding new expectations and modifying old ones. The STM and LTM invocations used in these grammatical processes each have complementary modes of their own which have very interesting and useful consequences. An LTM invocation transforms to an imperative and an STM one to a conditional extension of another procedure. Details will appear in section 6.5.

The default local expectation associated with "and", as with many words, is non-specific. Only the IDENTIFIER is defined and this is set equal to "FOUND", the name of the variable used by all the word procedures to accommodate the result of their final STM lookup. In section 4.4 an illustration of a more specific local expectation was given in teaching the program to count. Although no tests have been conducted on the idea, specific local expectations appear to be ideally suited to the problem of idiom where a word takes on a

peculiar meaning because it is followed by a particular word or words.

5.1.5.1 Detecting End of Clause or Group

Certain word processes are empowered to insist on a result from the left by closing an outstanding seek. Only words that need to relate the result to another (i.e. expect a result from both right and left) are permitted to do this and they do it when the STM call that looks for a result on the left does not succeed. An outstanding seek is closed by simulating a null result as if the end of a sentence (i.e. end of input line) had been reached. Effectively this amounts to signalling the end of a clause. It happens for instance at the word "after" in the sentence

: SAY TRIPLE AFTER I PRINT THREE CHARACTERS

thus hiving off the main clause "say triple" from the subordinate clause because "triple" gives no result and the seek from "say" is still outstanding. Similarly with "and" as in

: WHAT IS TWO AND THREE

the word "two" will seek for an object which in this case it must do without.

5.1.6 Acts and Utterances

After the Controller has caused a sentence such as "print a dot" to be interpreted, it will receive a result from its subordinate processes. The result will be a process descriptor for executing the Output Phase of the program: it will be in an initial state (i.e. pointing to the first instruction); it will already have a value assigned to its direct output variable named ACTION; and it will be structured so as to be subordinate to the current execution of the Controller. By virtue of this last attribute, the Controller will execute it. The Output Phase will check the contents of the direct

output variable ACTION and if it is a procedure will call it. Thus the dot will be printed.

When the human tutor types in a procedure in square parentheses, the program automatically assigns it as the direct output in an instantiation of the Output Phase and the Controller calls it before interpreting the sentence. So in the case of learning the word "print", the match found in STM by the word-process for "asterisk" is the value of ACTION in the Output Phase.

```
: PRINT AN ASTERISK [PRIN('*')]
  *
```

These specifications are incorporated into a new procedure that will be the meaning of "print". The imperative attribute is implied because the execution of the Controller is superordinate to that of the Output Phase.

The ability to form utterances and more generally to produce sequences of actions (e.g. counting) that are related to some conceptual structure demands an innate provision within the Output Phase, albeit a fairly simple one. The principle is that no act will be performed without first applying the process of perception and interpretation to it. Default specifications are assigned to the STM interface process so that simple cases will produce simple results.

Hence in the "print a dot" example, the effect of the principle is to produce internally a result consisting essentially of the procedure PRIN('D');PRIN('O');PRIN('T');
This procedure is written to the indirect output (a variable named SOURCE) of the Output Phase. In another example the word "say" is being taught

```
: SAY DOT [PRIN('D');PRIN('O');PRIN('T')]
  DOT
```

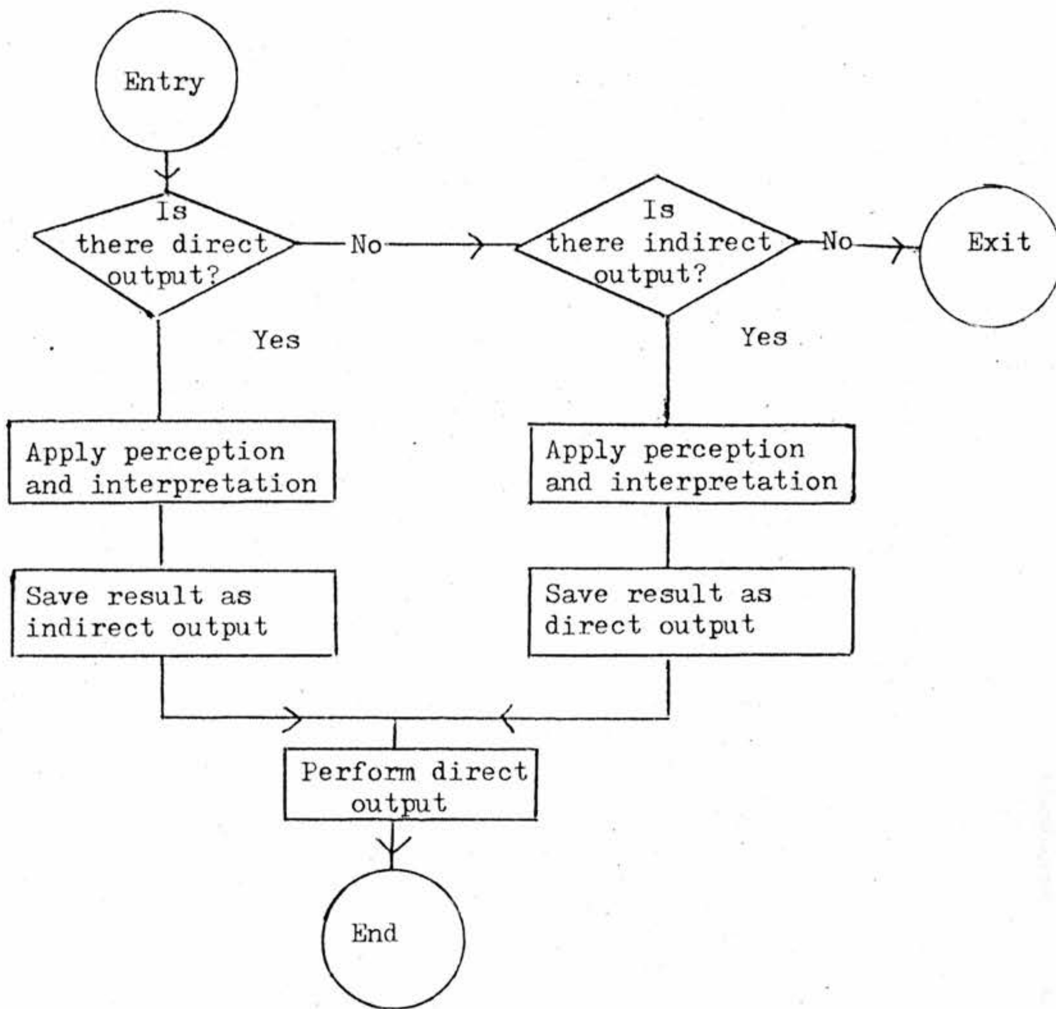


Fig. 5.4 The Output Phase

The word-process for "dot" locates (via STM) the procedure

PRIN('.');

as the indirect output (value of SOURCE) in the Output Phase.

The Output Phase executes the direct output. If it is only given indirect output, it applies perception and interpretation to it, obtaining the result as direct output, which it then executes. This is what happens during saying. The flow is summarised in fig. 5.4.

To provide such an innate ability to form utterances seems a reasonable step, especially as it uses the same methods as the rest of the program. Applying perception and interpretation to its

actions as well as to its affects also provides a satisfying symmetry. What is more this second stage of activity plays an essential role in completing the syntactic form of the raw utterance derived by the first stage. This area has not been explored in depth but one instance is supplying the "s" in "two dot". A somewhat different example of its utility arises in the acquired ability to count (see Chapter 7 for details). Basically, the program learns to start at 1 and remembers what comes next by perceiving and interpreting the action of printing 1 and deleting from the blackboard. Most importantly, it can learn to stop when the blackboard has been wiped clean.

The last thing to mention is the first thing the Output Phase does, namely to retrieve from LTM (including local expectations, of course) to allow intervention by any process with the appropriate conditions. A typical example is a process that predicts the result of an action. Such predicting processes are created by the final stage in the program's cycle, viz. the Feedback Analyser.

5.1.7 The Analysis of Feedback

The kind of learning that derives from feedback will be presented in section 6.2. The ability to relate actions to their consequences is obviously essential for intelligence. In this simple domain, the result of a printing action looks exactly like the action itself when converted to procedural form. However, the blackboard is more interesting and the program learns to predict the effect of the insertion mark (a quote") when something is already there.

The learning component for this kind of knowledge (causality) is a special case of the learning algorithm that appears in section 5.2. Feedback Analysis proceeds by calling Perception exactly as the Controller does when analysing input.

5.2 Learning

After the presentation of the flow of control it is time to see how procedures, processes and expectations are established through experience.

The fundamental principle of learning embodied in the computer program is synthesis, from a situation, of a procedure characterising it in a constructive way. As shown in fig. 5.2 the procedure will be in two mutually recursive subroutines and they each specify the novel elements of two situations. Typically the first subroutine corresponds to the new word in the sentence and the second to those attributes of the situation referenced by the sentence that are not already specified by the other words. Thus in the now familiar example

: PRINT A DOT [PRIN('.')]

.

the new word "print" is associated with the attributes EXECPROC as the Output Phase, IDENTIFIER specifying direct output and SUPERORDINATE equal to the execution of the Controller.

The embodiment of this association is the subroutine pair of fig. 5.2 and the Main Synthesiser constructs such procedures. Immediately thereafter the new procedure is executed so that it creates a process in LTM by calling the LTM interface.

The heart of the matter is the procedural synthesis and this is done in a very similar manner for all the learning situations. Whether it be the meaning of a new word, an extension to the meaning or the establishment of a local expectation, a common routine is employed. Causality is different from meaning but nevertheless its synthesiser is closely related to that for meaning and it participates in remarkably similar activities. For details see section 6.2.

5.2.1 Operation of the Main Synthesiser

The spirit of the (Main) Synthesiser is to specify to STM any attribute of the result it receives which remains undefined in the STM interface process. So when teaching the word "print" in the above example the "dot" process locates the procedure for printing a dot as the direct output of an execution of the Output Phase by calling the STM interface process with just the SEARCHPROC defined. The Synthesiser creates a procedure which will define to the STM interface process the other two arguments just mentioned. It will also define the SUPERORDINATE parameter: the convention is always to do this with respect to the process for the Controller. The 2nd appendix contains an extract of the code that does this.

The above set up applies to the case where there is only one result, the processing being symmetrical for the two sides. If two results are received, the Synthesiser constructs a procedure that specifies what they have in common. It checks for equality of the variable names referenced, structural or textual equality of the execution procedures, exact or partial match of the procedures referenced in the two results (partial match may be either of one into the other or vice versa but not both) and subordinate relations (can have neither, one or both) between the two processes found in the results. For example, the meaning of "what" is a match between the search procedures; the meanings of "before" and "after" are the subordinate relations.

The code which the Synthesiser sets up in such cases contains two STM calls; one to apply the relevant attributes from the left-hand result to the STM interface process as received from the right; the other to apply that result to the saved version of the STM interface process received from the left. Whenever results may be received

from both sides, the STM interface process that contains the attributes and the result from the left is saved (copied) like a working storage area before being passed on to the right for further processing.

There is some useful redundancy built into these synthesised procedures. If there is no result from the left when one is expected, the information from the right can still be used to produce a result. So, for instance, the interrogative form of "what" can work after that word's function as a relative pronoun has been learnt.

5.2.1.1 Synthesis from a Partial Match

The discussion has centred so far on results that refer to processes and are therefore the consequence of an STM retrieval with SEARCHPROC matching exactly or else undefined. Results from partial matches are referred to as procedural and come in the four varieties given in section 4.2. Continuing in the same spirit as above, the Synthesiser constructs a procedure that will specify the unmatched portion of the result. The procedure so synthesised will construct a SEARCHPROC and call the STM interface process. The construction will be appropriate to the kind of match that the result exhibits to the Synthesiser. If the match is with the back or front (types I and II) a JOIN call is synthesised; in the other cases a subroutine insertion in place of the empty procedure is required.

In the procedure set up by the Synthesiser in these circumstances a subroutine constant known as the signature is embedded. The value of this constant is the content of whichever of the result variables PARM1, PARM2, PARM3 or FOUND (see section 4.2) is defined. In other words, the signature is the unmatched portion of the result. In a case like the following

```
: **

: TWO ASTERISKS
```

where "two" is a new word, the procedure

```
PRIN('*');
```

will be found embedded in

```
|—> X;CALL X;CALL X;
v .
```

This is an example of a type III match and PARM3 will contain this last procedure with the empty procedure in the subroutine position. In this form it becomes the signature of the new word "two". The procedure generated by the Synthesiser contains code that will edit a copy of the signature, find the empty subroutine (by using the FP facility with the predicate PEMPTY) and insert the matched part of its result (i.e. the existing value of SEARCHPROC in the STM interface process). Thus "two" will in future apply its meaning to any object (once the local expectation has been generalised, cf. section 6.5).

5.2.1.2 Synthesis from Two results

If there are two results we insist that the right hand one be a process - at least the Synthesiser ignores it if it is not. The synthesis contains two STM calls. The first is exactly like the two-process case defined above. It is followed by a restoration of the saved STM interface process that came from the left. The second call is just like that synthesised for a single procedural result except that the signature is not embedded but is taken to be the procedure resulting from the first STM call (i.e. the value of the named variable in the process reference produced as the result of that STM call).

An important example of this arises for the word "is". The result on the right from the teaching sentence

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

matches the contents of a variable in the execution procedure for "is"

itself, that being the variable that holds the result from the left. In fact, the procedure for "is" does not exist at this early time and the process containing the result variable that is matched by "character" is an execution of the New Entity Handler. However, the Synthesiser detects this condition and substitutes the correct procedure which, of course, it is creating at that very moment. Very similar is the word "and", the only difference being the name of the variable because the match is with the PARM2 component of the result instead of the FOUND portion.

A detail essential to the above example with "is" concerns the result that a word like "dot" supplies when the STM lookup fails as it would here. The word "dot" is of the type that has an embedded signature. In the spirit of being as constructive as possible, the result is a direct reference to the procedure that is the signature and that is what is received by "is".

5.2.1.3 Synthesis when there is No Result

Nouns like "dot" cannot be learnt from the results of interpreting other words. They are learnt from demonstrations like the following.

```
: *
: *ASTERISK
```

The first line is to permit an insertion to LTM that will enable Perception to treat the second line as two entities rather than one. The convention in the Synthesiser is that if there is no result from either left or right the meaning is taken to be the rest of the input. So a procedure is synthesised as the meaning of "asterisk" and it contains an embedded procedure, its signature, equal to

```
PRIN('*');
```

and code to call the STM interface process with this as the SEARCHPROC. There is another possibility connected with Perception finding an STM

match of its own. That will be pursued in section 6.1.

5.2.2 STM and LTM are complementary

It was stated at the beginning of section 5.2 that the synthesised procedure was in two complementary parts. So far only one part has been described. For a simple noun the complementary part is just the same but has as signature the procedure for printing the word rather than the meaning. Fig. 5.5 contains both procedures for "dot". The ciphers WSTM and MSTM stand for variables that take different values in order to produce the complementary versions. In the specimen, the signatures (which are actually embedded subroutines) have been written in square parentheses.

For interpreting the word "dot" in natural language comprehension a process is written to LTM by running the top procedure (labelled the word subroutine in fig. 5.5) with the variable WSTM assigned to the LTM interface. For interpreting the object "." and producing the English word "dot" for possible utterance, a process is written to LTM by running the second procedure (labelled the meaning subroutine), with MSTM assigned to the LTM interface.

This scheme is the model for all the complementary procedures. Words that take results have the necessary calls for seeking, closing an outstanding seek and seeking to the left via STM. They occur within the word subroutine after the conditional exit to the Result Handler but before the passage to the meaning subroutine. There, other parameter assignments and extra calls are inserted as required. The problem now faced is to determine what is the complementary code to be written after the Result Handler exit in the meaning subroutine and what joins or edits are necessary to SEARCHPROC in the word subroutine.

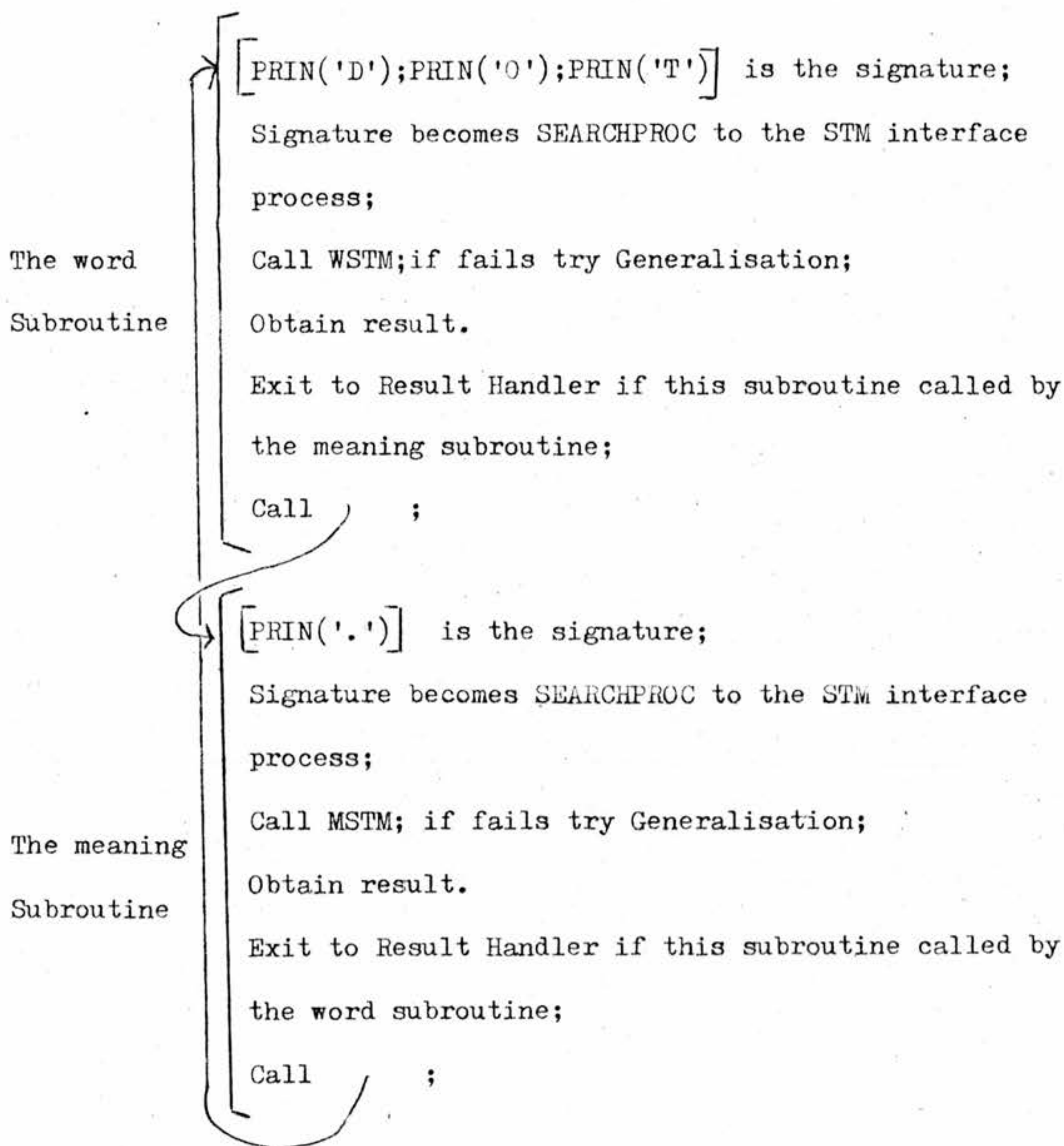


Fig. 5.5 Informal version of the complementary subroutines for "dot".

5.2.2.1 <u>An example</u>

The reasons for being interested in this matter and an indication of the nature of the general solution will become apparent by considering the already familiar case of the word "two". When interpreting the phrase "two dots", the program runs the process from LTM that is the meaning of "two". In its word subroutine there will be a seek that will cause Perception to interpret the following word "dot". The result of processing "dot" will be passed back to "two" and the dot procedure will be inserted into the signature of "two"

giving a procedure that twice prints a dot.

```
 ↓→X;CALL X;CALL X;
/PRIN('.');\
```

The code to carry out this insertion is contained in the meaning subroutine within the process for "two".

Fig. 5.6 illustrates the grammatical considerations in interpreting the phrase "two dot" (the "-s" ignored for the purpose of this discussion). The meaning procedure for "two" cannot perform until it has the result of "dot" upon which to act. So the word procedure for "two" includes a seek, and the meaning procedure includes code to insert the procedure it receives as a result into (a copy of) its own signature.

This pair of subroutines for "two" was created by the Main Synthesiser in this way because, during learning of "two", it received a result from the right (hence the seek) and the result was a partial STM match into a subroutine of the duplication procedure.

It has been claimed that the transition from comprehension to utterance can be achieved by means of the complementary process established by invoking the meaning subroutine. This was demonstrated for a noun like "dot". Now it will be shown for the phrase "two dot". Later, some general rules will be stated.

Utterance involves passing from meanings to words and in this example it means starting with the procedure for twice printing a dot and ending up with a procedure for printing TWODOT. The blank space is omitted by the program. Section 4.3 stated the conventions for partial match in LTM and they imply that when the above procedure for twice printing a dot is presented to Perception for retrieval from LTM the subroutine

```
PRIN ('.');
```

is matched first. After that the repetition portion would be interpreted.

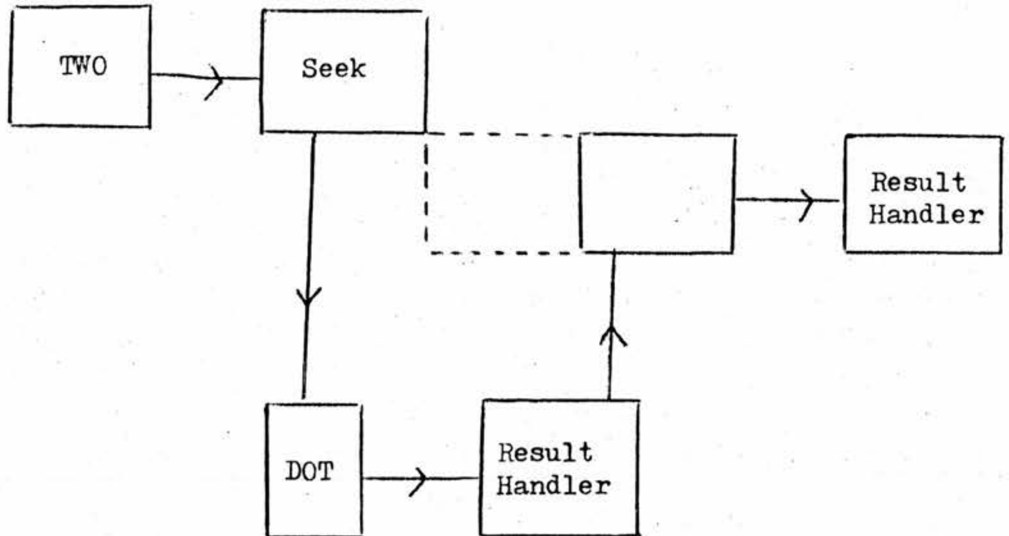


Fig. 5.6 Run-time Structure when interpreting "two dot".

To summarise the explanation so far, there are two processes in LTM, one with SEARCHPROC equal to

CALL 'T' PRIN; CALL 'W' PRIN; CALL 'O' PRIN;

and the other with SEARCHPROC equal to

The word subroutine	[PRIN('T');PRIN('W');PRIN('O')] is the signature; Join the signature onto the front of the previous result (if any); This becomes SEARCHPROC etc. Exit to the Result Handler if this subroutine invoked by the meaning subroutine; Perform a seek (to the right); Invoke ;
The meaning subroutine	[NOOP[ ] ;ASSIGN X;CALL X;CALL X] is the signature; Insert the previous result (if any) into a copy of the signature as a subroutine; This becomes SEARCHPROC etc. Exit to the Result Handler if this subroutine invoked by the word subroutine; Perform a seek to the left; Invoke ;

Fig. 5.7 Informal version highlighting special points of the complementary subroutines for "two". The two subroutines invoke one another although not by direct calls (explained in Chapter 6).

```
NOOP  ↓  ; ASSIGN X; CALL X; CALL X;
  /  "Empty"  \
```

Each of these processes uses the same two mutually recursive subroutines. Both were set up at one time from an example such as

: **

: TWO ASTERISKS

The first process was established by calling the word subroutine and serves the purpose of interpreting the word "two" in comprehension; the second was created by calling the meaning subroutine and is for utterance when the meaning is given. They appear in fig. 5.7.

When interpreting the procedure for twice printing a dot, the program runs the process from LTM that has its SEARCHPROC equal to PRIN('.');

The result of its processing will be an STM reference to a procedure for printing "dot". If the interpretation is being done by the Output Phase of the program (e.g. when answering a "what" question) the necessary STM attributes will have been supplied for actual printing to take place. Before then, the repetition procedure

```
 |——→ X; CALL X; CALL X;
 ↓
/  "Empty"  \
```

must also be matched against LTM. The process thus obtained from LTM will seek to the left to obtain the "dot" result. Next it will call its word subroutine which will contain an appropriate join. The signature is of course the procedure for printing "two" and this must be joined on to the front of the received value of SEARCHPROC in the STM interface process to construct the utterance.

TWODOT

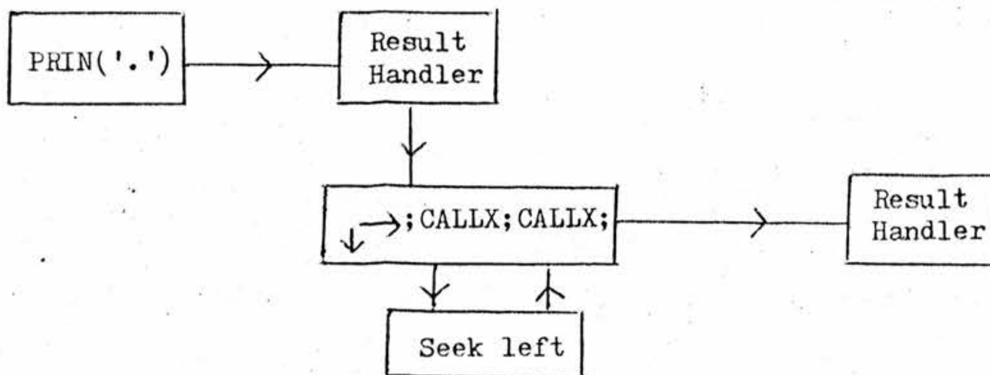


Fig.5.8 Run-time structure when preparing to utter "two dot".

Fig. 5.8 complements 5.6, illustrating the analogy, and the differences, between interpreting the words and interpreting the meaning. In fig. 5.8, the LTM retrieval convention dictates that a subroutine will be matched before the procedure that contains it. It is as if the subroutine were on the left. This convention, by the way, is not arbitrary: the opposite one leads to inconsistencies.

The word procedure for "two" must include code to put its signature (a procedure for printing "two") onto the front of the result procedure (in this case "dot") that it receives. There must also be a seek to the left which as a matter of fact has to go into the meaning procedure for "two".

The Main Synthesiser had to incorporate these provisions into the subroutines at the time they were first created. There must therefore be rules to cater for all cases.

5.2.2.2 The General Transformation for Grammatical Information

The problem to be solved derives from the nature of LTM and the necessity of grammar. In section 5.1.5 it was pointed out that grammatical processing derives its nature from the restrictions in LTM matching when compared to STM. These restrictions are necessary

because of the potentially unlimited size of LTM. They also correspond naturally to the temporal progression of speech which is presumably reflected in speech perception.

When transforming a procedure that is used in language comprehension to use it for utterance, grammatical information must undergo its own transformation. This is because the simple device of interposing the STM and LTM invocations that works for simple nouns could cause a procedure to expect an LTM match of a kind that is only available from STM. In the example of "two" given above, the procedure of fig. 5.6 was created by the Synthesiser. In the original learning situation, the Synthesiser received a result from the right which was a particular kind of partial match. Because it came from the right a seek was provided in the word subroutine. Because of the kind of partial match it was, the meaning subroutine contained a particular kind of insertion instruction. Rules were needed to work out that in complementary mode a seek to the left would be needed in the meaning subroutine and the join in the word subroutine should put the previous result on the right of the signature.

Note, by the way, that these complementary transformations are not the same as those mentioned in section 5.1.5 to be applied to the STM and LTM calls that implement the grammatical constructs. They have other implications and will be discussed in section 6.5. This fact only serves to show further the richness of the idea of complementarity.

In order to write down the general rules for determining complementary code for procedural synthesis by the Synthesiser some further nomenclature will have to be introduced. Results from STM of the kinds (II and III) that correspond to the natural order of

retrieval from LTM are left-handed. Type II is a match with the front of a procedure and type III with a subroutine. The other two kinds are known as right-handed matches. Type I is a match with the back of a procedure and type IV with a procedure apart from a differing subroutine. Types III and IV are known as embedding matches.

The straightforward constructions that the Synthesiser will make are as follows. From an embedding match it will create an insertion, either left-handed or right-handed, respectively. From types I and II it will respectively synthesise a right- or left-handed join. The synthesis is placed in the meaning subroutine. If the result has come from the left, a seek to the left will be inserted in the word subroutine (following the conditional exit to the Result Handler). The code before the WSTM call will be an insertion or a join, depending on whether the LTM match in Perception for the current word or entity was an embedding match or not. The former is rare in English, but occurs in some African languages where repetition of a word may denote the plural.

As LTM matches are conventionally left-handed, rules are necessary to determine whether the join or insertion in the word subroutine should be left- or right-handed and whether, following the Result Handler exit in the meaning procedure, there should be a seek to the right or left. These are now stated.

1. If the result was left-handed, a seek left call is placed in the meaning subroutine and if it was right-handed a seek right is written.
2. If the result came from the left a left-handed join or insertion appears in the word subroutine and if from the right, a right-handed one is put there.

A left-handed join consists of joining the procedure referenced by the result on to the left of the signature and a left-handed insertion involves putting the procedure into the signature. Obversely, to make a right-handed insertion is to embed the signature into the procedure referenced by the result; the join is obvious. In the case of "two" given above, convention 1 is left-handed and convention 2 is right-handed.

When the variable WSTM is assigned to be an LTM insertion process, which happens immediately the synthesis has finished, the presence of code for a join or an insertion does not upset the SEARCHPROC for saving in LTM as the result procedure is defined to be empty at this time.

As the procedures for "dot" show, the question of rules for joining, inserting or seeking does not arise in a case where the Synthesiser receives no result from either side.

When the result is process-like (exact match) instead of procedural (partial match), that is, when the meaning procedure does not define SEARCHPROC but only some of the other arguments to the STM interface process, the result is classified as left-handed for the purpose of applying the above rules. When there are two results the right-hand one is used to determine the complementary code unless the left-hand result is procedural (i.e. a partial match) in which case it is the one used.

The utterance of words with process-like meanings has not been investigated. The conventions for them just given are supplied for completeness and are ascertained from general considerations. The question will be taken up in Chapter 10.

It should be noted that although the above principles have been illustrated with English words and some of the terminology is

suggestive of semantics their application is not confined to natural language as can be seen from the dialogue in Chapter 7.

5.2.3 Generalisation

Once a procedure has been synthesised and saved in LTM its content is not irrevocably fixed. Two further processes, generalisation and differentiation, may be applied to it subsequently as often as necessary. Generalisation involves modifying a procedure in order to relax the specifications it gives to the STM interface process; differentiation leads to the synthesis of new procedures which specify additional STM arguments for restricted cases and is done by putting new entries into LTM, as will be explained in section 6.4.

At the heart of the generalisation process is a generalising matcher. The generalising matcher finds the common portion of two procedures. That can be used as a search argument in STM retrieval. Its results are therefore of the same basic types as for STM retrieval. The difference is that neither argument to the matcher has to be found completely within the other; the restrictions of appearing at front or end, or having the differing subroutine empty, apply not to the two arguments themselves but only to the common portion that is the result of the match. The matcher is symmetrical with respect to its arguments.

Thus there are altogether three related matching algorithms used by the program. The first is for LTM; the matches are of the most restricted type but the number of items available to it is potentially enormous because of key-word retrieval. The second is for STM: more match types are allowed and the search space must therefore be restricted. The third is for generalisation: the most flexible kind of comparison but with searching only within two procedures.

Grammar is needed to bridge the flexibility gap between STM and LTM. Similarly, differentiation to some extent bridges the gap between the generalising matcher and STM. Chapter 6 shows how differentiation is in fact a special case of the provision for grammar.

The significance of generalisation is in the formation of class concepts although this may or may not be its primary function. So the meaning of the word "character" is obtained by a generalising match between the procedures for dot and comma. These are actually represented in the program in the following form which is more elaborate but less comprehensible than the notation used earlier.

```
NOOP 30; ASSIGN X; CALL X PRIN;
NOOP 28; ASSIGN X; CALL X PRIN;
```

The first line will print a dot and the second a comma. The generalising matcher extracts the procedure

```
ASSIGN X; CALL X PRIN;
```

which is not executable but will match into any procedure in STM that contains these instructions in this form.

A more interesting example is the generalisation of numeral that is necessary in adding. It would clearly be unsatisfactory for every combination of ADD from 1 to 9 to be learnt separately. The generalising matcher is capable of finding the commonality between two of the numeral procedures, which only differ in their subroutines below a certain level. The example is interesting because the generalisation is over function and, moreover, is over a function that was acquired by the program and is embodied in a synthesised procedure.

Generalisation is attempted when the STM retrieval in a synthesised procedure fails to give a result. The routines that do it, however, have other options available to them and these are detailed below (sections 5.2.3.2 & 3). When they do generalise, their effect is to

modify the procedure that called them since that must be the one that gave the failing specifications to the STM interface process.

Corresponding to the two result types that the STM interface can give, namely procedural (partial match) and process-like (exact match) there are two types of procedure created by the Main Synthesiser and consequently two kinds of generalisation. Procedures synthesised from partial match contain within them a signature which is the main part of their meaning. For example, when the word "character" has been shown to the program only once, it is taken to mean a comma. Its meaning procedure possesses as its signature the procedure for printing a comma.

The subsequent sentence: "an asterisk is a character" leads to the failure of the STM look-up in the meaning procedure for "character" and so the Generalisation routine removes the signature from the specifications to the STM interface process. The implicit assumption is that the STM look-up must have failed because of the specifications (in this case the signature) most recently supplied to the STM interface process. The assumption is justified because anything specified to the interface process previously must have been verified already since no procedure synthesised by the program will specify STM attributes without also invoking the interface process to verify them.

When the Generalisation routine has stripped the signature from the specifications to the STM interface process, it invokes it again in order to obtain a result from which an alternative signature can be derived. Continuing with the "character" example, the signature is a procedure for printing a comma. When this is removed, the STM interface process still has the specifications supplied by the meaning procedure of "is" and these are sufficient to locate in STM the appropriate result, which is a reference to a procedure for

printing an asterisk. The original and the alternative are compared using the generalising matcher and the common portion becomes the new signature, replacing the old one.

For meanings that involve the process-like parameters EXECPROC, IDENTIFIER, SUBORDINATE and SUPERORDINATE the principle is the same, viz. retrieve from STM without the specifications just supplied by the most recent process and then compare the attributes of the result with the specifications in the current meaning.

An example is the generalisation of the meaning of the indefinite article "a". It is first taught when the referenced item is on the previous line.

: .

: A DOT

Another example could show it being printed.

: [PRIN('.')]

.

: A DOT

In this case, the Generalising routine removes the EXECPROC, IDENTIFIER and SUBORDINATE specifications supplied by the meaning procedure of "a" to the STM interface process. When it invokes the process again, the specifications supplied by the meaning procedure of "dot" are sufficient to locate the preceding print action in STM. The result thus obtained has differing EXECPROC and IDENTIFIER attributes from those given in the meaning procedure for "a" and so this procedure is edited in order to remove those specifications. Since there can be no partial matching of these parameters in STM, the generalising matcher is not used.

The implementation is slightly simpler than just outlined in that in its STM retrieval it removes all the process-like arguments, not

just those most recently supplied. As far as can be seen, this discrepancy has not affected the outcome of any of the examples. It deletes the conflicting specifications by editing the assignments out of the procedure that is being generalised.

5.2.3.1 A Further Step

Although it is a rule of the STM interface that the EXECPROC parameter can only participate in exact matching, it is possible to use the interface to obtain a partial match to an execution procedure by using the SEARCHPROC parameter. Instead of expressing a variable name, IDENTIFIER must now bear a special token, the execution token.

There should be the capability for generalisation to take advantage of this extra degree of freedom afforded by the memory mechanism. In fact it has only been implemented in a particular kind of generalisation concerned with expectations and described in section 6.5.2. It may be that problems of information loss might arise if the principle were applied also to meaning. Further investigation is needed.

Such generalisation is carried out when the comparison between the parameters most recently supplied and the attributes of the result shows that the EXECPROCs do not match exactly and the SEARCHPROCs do not even match partially (or one of them is undefined). In these circumstances, the program applies the generalising matcher to the execution procedures and, if that is successful, changes the procedure that is being generalised so that it passes the appropriate SEARCHPROC and IDENTIFIER arguments to the STM interface process.

5.2.3.2 Sundry Facilities

The generalising routines conduct an exploratory STM invocation. If that fails they return to the caller with a contrived result which

has the search procedure of the STM interface process as the FOUND component of the result and a null value in the REFERENCE part which would normally contain a data structure indicating the type of match. The generalising routines also perform the checking for imperatives mentioned in section 4.2. This is the first thing they do and if it is successful they return immediately to the caller with the appropriate result.

The calls to Generalisation in the word subroutines appearing in the recent examples are only invoked in complementary mode as the LTM interface always returns "true". This is because the suspended process can be re-activated only when a match is found.

5.2.3.3 Conflicts that do not lead to generalisation

A special case arises when a generalising routine detects the presence of the Main Synthesiser on the run-time structure for the current sentence or input. It then returns an indicator rather than modify any procedures. The exploratory STM invocation and subsequent comparison is then performed by the Synthesiser itself. This gives rise to conflicts of the kind mentioned in section 5.1 to which a general solution has not so far been found.

When only the SUBORDINATE and SUPERORDINATE parameters are at variance it appears sufficient to permit the new word to override the specifications of the others. An example is the auxiliary verb "did". It can be taught thus.

: PRINT AN ASTERISK

  *

: YOU DID PRINT AN ASTERISK

In such a case it seems quite reasonable to allow processing for "did" to overrule the present tense provision substituting a past tense (SUBORDINATE containing the value previously held by SUPERORDINATE

with the latter now undefined).

With this meaning the auxiliary "did" works in sentences like

: WHAT DID YOU PRINT

ASTERISK

: WHAT DID YOU SAY

ASTERISK

The code in "did", as generated by the Synthesiser resets these parameters in the STM interface process to the undefined value before giving them their new assignments. The reset is necessary to avoid falling foul of the built-in consistency checks.

However, overriding EXECPROC and IDENTIFIER causes a serious loss of information and the present implementation is unsatisfactory in this regard. An improvement involving the execution of the result will be suggested in Chapter 10.

DESCRIPTION OF THE FIVE COMPONENTS

Chapter 6.

Much of Chapter 5 was devoted to the Main Synthesiser that constructs the meanings of words and morphemes and captures the behaviour of one or two objects that will be described in Chapter 7. There is not much more detail to be given in section 6.3, entitled "Meaning". The version of the Synthesiser used in causality processing (see section 6.2) bears an interesting relation to the one already described. The last two sections on differentiation and grammar show that these themselves are remarkably similar, as well as having much in common with the others because they all use the Synthesiser.

The least well developed component of the program is perception. It is the only part that does not perform significant learning which probably makes it the simplest to explain.

6.1 Perception

Theoretically, retrieval from LTM could be done whenever a procedure is invoked. Once the input from the teletypewriter has been converted to procedural form and handed over to the Controller, the first match from LTM would be located and the associated process activated. In fact, the retrieval from LTM is performed as part of Perception, the main purpose of which is to detect patterns and relations within the input and between the input and other items in the recent past, which means in the short term memory.

So Perception first uses LTM retrieval to decide on an appropriate or meaningful segmentation of the data. Once it has obtained a match from LTM, it invokes STM to see if the matching portion has occurred recently before. In section 5.2 a general method was presented of synthesising a procedure to represent constructively the result of an STM search. However, it is not appropriate to make this method (as

embodied in the Main Synthesiser) standard equipment for Perception. Where the STM match is with the content of the current input line a repetition is implied and it is appropriate to synthesise a procedure that represents it. It takes the simple form of a repeated call that you have already seen in several examples. It is not difficult to see how it could be rewritten as a series of imperative STM calls. Obviously it would then be bulkier but it would fit better into a general pattern.

On the other hand the appearance of a word in two successive sentences, while it is noticeable, does not change the meaning and one would not therefore wish to alter the LTM search argument because of it. For instance, the use of "very" in the second sentence is not affected by the first: "He came very quickly"; "I was very grateful for his help". The insertion of "and" to conjoin the sentences alters the second "very" to an emphatic one, as in immediate repetition: "He is very, very pleased". It seems important to distinguish between a match in the current sentence and in a previous event. The program only changes the representation in the immediate repetition case; more distant repetition is detected but the result is simply saved during Perception. Such a result may be used by a synthesiser to create a procedure if it is run for any of its learning purposes but the detection of such a relation by Perception does not of itself warrant a synthesis. An explanation of such a performance by the Main Synthesiser appears in section 6.3.

If Perception detects a simple repetition, which means that the match in STM is with the next item in the input line, it creates the repetitive procedure and then it loops, cutting off the matching portion each time, until no further matches are found. The detection of nested repetition (e.g. "**,,,**,,,") has not been implemented

although there is no difficult principle involved; it is however an objective to avoid dependence on special cases and it is preferable to wait for a more general solution to the perception problem. If a special purpose solution were necessary, it would involve extending the admissible range of matches from which procedures are synthesised to include the resultants from previous perceptions in the same line of input. For example the sequence

: **,,,

would be represented in the usual way by

```
NOOP ; ASSIGN X; CALL X; CALL X; NOOP ; ASSIGN X; CALL X; CALL X; CALL X;
  ↓                                   ↓
CALL '*' PRIN;                      CALL ',' PRIN;
```

In interpreting the second set in

: **,,,**,,,

the program would recognise the duplicate asterisk first but would match the resultant with the earlier duplicate asterisk procedure via STM. After verifying that the intervening data were also repeated it would construct a procedure that twice called the one exhibited above.

If Perception detects repetition it tries to match in LTM the complete procedure that it has constructed to represent the repetition rather than executing the process it has already retrieved. So if, for instance, the input is "££" it looks to see if there is an LTM record that matches "££". Only if there is not will it run the process that matched "£". Then the duplication procedure will be available for matching later on. This convention is consistent with those for LTM retrieval given in Chapter 4. If a match of the input with LTM fails, the method is to remove and save the instructions one at a time from the procedure until it succeeds. The saved instructions are treated as new entities for which learning is to take place.

6.2 Causality

The "causality" function of the program is concerned with the analysis of feedback from the program's actions and of similar data that reflect the program's effect on its world. Processing was outlined in section 5.1.7. The feedback data are read, converted to procedural form and passed to the Feedback Analyser (see fig. 5.1). It invokes Perception exactly as the Controller does and suitable processes are retrieved from LTM and re-activated.

The purpose is to ascertain the origin of, or "explain", what is seen. The examples to which the program has been applied involve elementary feedback from printing, which matches simply, and the cumulative effects of writing and deleting on a blackboard as mentioned in Chapter 1. These will be treated fully in Chapter 7. The methods described here should be capable of wider application and are presented in a more general way. In particular STM-LTM complementarity will be seen to be a natural transition from explanation _a posteriori_ to prediction of the consequences of an action in a manner analogous to the transition from comprehension to utterance in language.

6.2.1 Learning causal relations

For causality, learning is performed by the Synthesiser for Causality which is very much like the Main Synthesiser. Again, it constructs subroutines. Instead of the word and the meaning they are known as the effect and the cause, respectively. They are in that order because the first analysis is performed on the effect, or feedback, of an action. Therefore the effect is analogous to the word. Although we are not concerned here with the phylogenesis of language the existence of similarities between causality and meaning is very interesting and presumably has a bearing on the evolution of linguistic

capacity in homo sapiens.

There are three main differences. The first is that the memory reference in the effect subroutine is always to the LTM interface: when the memory invocation in the cause subroutine is transformed from STM to the complementary LTM form, that in the former remains as LTM. This has the effect of making a causal prediction, as an example will shortly clarify. The second difference is the manner that the two subroutines communicate with each other. The third is that causality does not have a grammar - or at least it has only a weak one.

The Synthesiser for Causality can still be presented with two results, just as the Main Synthesiser can. In causality, however, no seeking to the right takes place - only to the left. The place of a result from the right is taken by a result derived from the feedback. In fact it is the result of the STM match located by Perception during its analysis of the feedback that is used in place of a result from the right.

A couple of examples should make this clearer.

6.2.2 Two Examples

The result of the STM search conducted by Perception is a reference to the most recent match of the feedback data, and the program hypothesises that to be its cause. The simplest example is to constrain the program to print something.

```
: [PRIN('*')]
  *
```

When analysing the feedback, Perception locates the match. The result is a reference to the preceding action performed by the program's Output Phase. As this is the program's first encounter there is no suitable process in LTM. (Recall that the meaning process for "*" in

LTM includes a specification, using EXECPROC, that it is for matching against input to the program. Therefore it does not apply to the analysis of feedback.) So the New Entity Handler invokes the Synthesiser (for Causality). Details of that will be given in a moment.

The end product is a process in LTM with EXECPROC and IDENTIFIER appropriate to the Feedback Analyser but SEARCHPROC undefined. That is to say that, for reasons that follow later, no mention is made of "*". In future this process would be re-activated any time that feedback is analysed, regardless of the data and it would always expect to find an exact match in STM to the feedback data and the result of the match would have to be direct output in the Output Phase. Complementary to this process is another which is also created and is waiting for future performances of the Output Phase. This process will predict that the feedback from such an action will match that action. Now, if the program prints something it will always establish a local expectation that anticipates that the feedback will be equal to the action procedure.

Now consider an example where this expectation is violated, as happens when the program writes to the blackboard. If the program is made to print a quotation mark before another character, the character is written on the blackboard and the quote does not appear in the feedback.

```
: [PRIN('"'); PRIN('*')]

                *
```

the predictor process sets up an expectation for

PRIN('"'); PRIN('*');

This will not be fulfilled: it simply recedes into the past and eventually disappears as STM is pruned. Fortunately, the comple-

mentary process will still be activated since it applies for any feedback. It will locate the preceding action and note the added quotation mark. It learns from this experience and establishes two more processes in LTM, one of which is a predictor that fires when a quote appears in the direct output of the Output Phase. The other process is again to explain a posteriori and only acts when the prediction is not satisfied.

As a matter of fact, this is also an example of differentiation because the second process is effectively a conditional extension of the process that looked back for an asterisk. The distinction between initial learning and differentiation (discrimination learning) is slight although it is greater in the case of meaning than for causality.

6.2.3 The Details

In both of the above instances, the Synthesiser is at work. Like the Main Synthesiser it creates two subroutines, each containing memory references. The memory retrieval within the effect subroutine is fixed as an invocation of the LTM interface. In normal mode its purpose is to store a process in the global LTM; in complementary mode it makes a prediction in the form of a local expectation. The complementarity transformation is implemented by the re-assignment of variables.

Communication between the subroutines is different from that which is the case in meaning. There is generally no connection between the form of a word and its meaning; onomatopoeia is the exception, not the rule.

In causality, the procedural result from the LTM invocation in the first subroutine (whichever it happens to be) is passed to the other and plays the role of a result there. For instance, a process

in LTM waiting to predict the consequences of an execution of the Output Phase will, when activated, seize the content of the direct output and supply this reference as result when it calls its own effect subroutine. In turn, that will set up a local expectation of feedback, with SEARCHPROC equal to or constructed from the procedure just found. The expectation will die when it has receded into the past and the process to which it is attached has been pruned from the run-time structure. If the feedback is not as expected, the complementary process from the global LTM (which has SEARCHPROC undefined) will be activated and will become involved in a learning situation of the kind exemplified above.

A consequence of passing a result between the subroutine is that they are only able to accept one other result. This is because they follow the same pattern as in meaning and two results (one from each side) is the maximum there. The result passed from the first subroutine takes the place of the result from the right. The other therefore can only come from the left. This greatly limits the possibilities for a grammar of causality. Incidentally, the same principle applies in differentiation, whether it be for causality or for meaning.

These subroutines are generated by the Synthesiser and therefore it must itself be bound by similar conventions. The place of the result from the right is taken by the result of the STM match located by PERCEPTION. For example, when analysing feedback from the action of printing an asterisk, Perception finds a match with the action in short term memory. The attributes of this result lead the Synthesiser to generate the cause subroutine in fig. 6.1. In the figure, CSTM denotes a variable that is assigned as the STM interface process when the effect subroutine is first but, in complementary mode, is assigned as the LTM interface when the Synthesiser generates the LTM process

The effect subroutine	EXECPROC is the Feedback Analyser; IDENTIFIER is the feedback; SUPERORDINATE is set when called by the cause subroutine; SEARCHPROC is derived from the result of the cause subroutine; Call LTM; Extract from the result; Exit if this subroutine called by the cause subroutine; Call ;
The cause subroutine	EXECPROC is the Output Phase; IDENTIFIER indicates direct output; SUBORDINATE indicates the past; SEARCHPROC is derived from result of the effect subroutine; Call CSTM; If fails, call Generalisation; Extract from the result; Exit if this subroutine called by the effect subroutine; Call ;

Fig. 6.1 Specimen product of the Synthesiser for Causality.

that predicts the effects of the Output Phase. (It behaves like MSTM in meaning.)

A case of results coming from the left would have arisen in predicting the effects of combinations of objects on the blackboard

device. Unfortunately, these ideas have not been tested or pursued in enough detail to warrant their presentation here.

The combinations of results possible for the Synthesiser are the same as for the Main Synthesiser with signatures being obtained from partial matches (procedural results) in the same way. Not all combinations have actually been implemented because it has not been necessary but there is no great difficulty in putting them in; the code has to be taken from the Main Synthesiser and modified for communication of results.

A cause subroutine resembles one for meaning. The difference is that instead of extracting details of the result from the right by referring to the STM interface process, the subroutine must derive it from the result produced by the effect subroutine. An effect subroutine sets up parameters to the LTM interface as does one for a word. The details, however, are somewhat different because of the requirement to set up a local expectation when in complementary mode and because there is no signature containing the actual word, that being replaced by the result from the cause subroutine.

Rules for converting grammatical conventions to complementary form are needed by the Main Synthesiser (see section 5.2.2) but are not applicable to Causality as formulated here because it does not possess those degrees of freedom.

6.3 Meaning

As most of section 5.2 was devoted to this topic, little more remains to be said.

When Perception fails to match an input procedure with LTM it strips off instructions one by one until it can. The unmatched portion is presented to the New Entity Handler in a format as if it had been matched successfully. A problem is the accidental embedding

of one word inside an unknown word, e.g. "asterisk" contains the morphemes "a", "s", "er" and "is". It is easily overcome when the accidental morpheme is surrounded by unknown groups of letters, as is the case for "I" within "print". If "I" is already known to the program, "pr" and "nt" will be picked up as two unknown words. The second invocation of the New Entity Handler, however, can readily detect the existence of the first in the run-time structure and can form a composite, deleting all reference to the intervening processing.

After the Main Synthesiser has built the word and meaning subroutines, the New Entity Handler invokes them so that the processes are saved in LTM. At this time, the New Entity Handler supplies the additional specifications of EXECPROC and IDENTIFIER indicating that input from the teletypewriter (in the Controller) is to be matched. These are not required in the causality case as they are already present in the cause and effect subroutines with appropriate values.

6.4 Differentiation

This is the means whereby procedures may be extended to arbitrary lengths, including the insertion of conditionals and recursion. As already explained in Chapter 4, the means of extending a procedure is by storing a process in LTM with EXECPROC referring to that procedure. The conditional is determined by the value of SEARCHPROC. Recursion occurs if LTM has a process with the EXECPROC parameter equal to one of its own execution procedures. It may be equal either structurally or textually and the latter kind of equality arises when two exactly similar procedures are synthesised from two similar experiences, as is the case with the plural morpheme "-s".

Differentiation is the counterpart of generalisation and is much the more significant of the two. It takes place when a result has been obtained from STM of which some of the attributes were not speci-

fied in the invocation of the STM interface. It may be that the result is only a partial match of the search procedure or that the match is exact but not all the process-like attributes of the result appear in the STM interface process which is employed for STM retrieval in synthesised procedures. Thus differentiation is conceived in the same spirit of "explaining" or completing the specification of the attributes of a situation that guides all the program's learning.

Incompleteness is detected when a result is returned to the Controller or the Feedback Analyser or, alternatively, when a procedural result (partial match) reaches a place where a process result (exact match) is expected. Such a place would be the execution of a procedure created by the Synthesiser from a process result. The incomplete specification may be the result of over-generalisation (i.e. the Generaliser relaxing too many parameters) or simply of a change in circumstances such as the appearance of the quotation mark to cause writing on the blackboard.

6.4.1 Details

All procedures observe the convention, when results are received, of storing the process descriptor of the sender so that, should differentiation be found necessary, the details are available. Differentiation creates processes in LTM that effectively extend the procedure that supplied the incomplete result (not the one that received it).

To do this it invokes the Synthesiser with the right hand result already available, it being the one that has just been found to be incompletely specified or "explained". The same applies in the causality case. This provision is similar to the manner in which for causality the place of a result from the right was taken by the

match to the feedback. Differentiation applies both to causality and meaning and in either case it has the incomplete result take the place of the one from the right.

Thereafter, Differentiation assembles its own version of the word subroutine which is known as the expectation subroutine because its purpose is to define to LTM the circumstances in which the meaning subroutine is to be executed; consequently it creates an expectation of those circumstances. Its specifications to the memory interface include EXECPROC equal to the procedure that supplied the incomplete result and IDENTIFIER set to the variable name that contained the FOUND portion of the result. Because of an implementation restriction the variables containing the PARM1, etc., portions of the result are not referenced in this way. It follows that only if the result is left-handed can a useful SEARCHPROC be defined. In the case of "-s", for instance, the extra instruction

CALL X;

was to appear on the end of the procedure which makes it a right-handed join and it is not practicable to allow right-handed matches in LTM retrieval. So on these occasions SEARCHPROC is equal to a null procedure. When the result is a process, SEARCHPROC must be undefined.

6.4.2 Complementary version

The transformation (STM-LTM) to be applied in order to obtain the complementary version of a process is not always immediately obvious from a priori consideration. To be more precise, the awkwardness lies in making the appropriate provision for handling results after the simple re-assignment of variables has produced a complementary process.

In section 5.2.2 the two mutually recursive procedures each contained provision for handling results after the memory invocations. As was explained, variable assignments were such that only after an STM call (on whichever side it might be) would a call to the Result Handler be made. Fig. 6.2 illustrates a pair of procedures synthesised by differentiation and potentially able to establish two complementary processes in LTM by being invoked, each with suitable variable assignments.

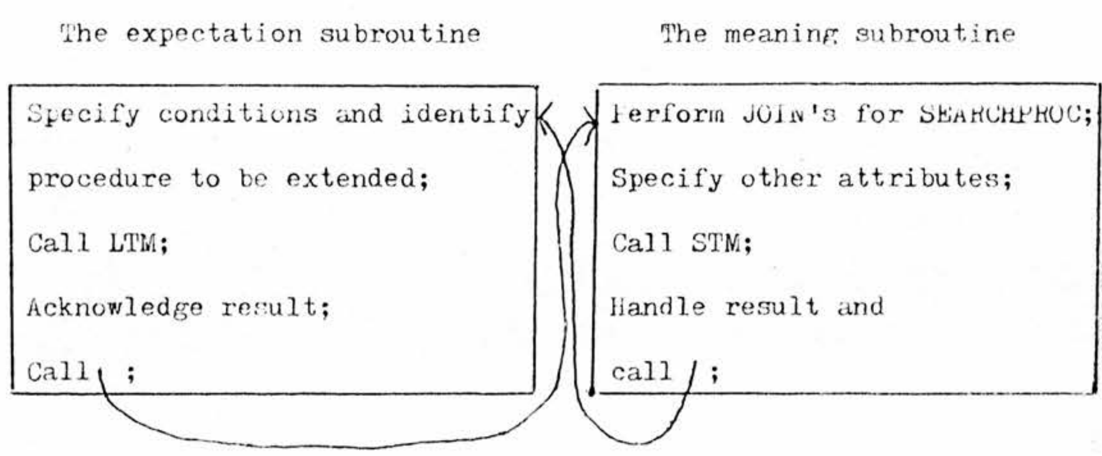


Fig. 6.2 Mutually recursive procedures established by differentiation.

The obvious arrangement has been made whereby to establish the complementary version at learning time. The meaning subroutine is invoked by the program and it invokes LTM, thus creating a process complementary to the one made by a straightforward call of the expectation subroutine.

Hence in the example of section 6.2.2 concerning the quotation mark that causes writing on the blackboard, differentiation establishes meaning and expectation subroutines. The former embodies the knowledge that the quote should appear in front (i.e. it contains an appropriate join instruction) and the latter specifies this process to be an extension of the very simple feedback explainer that only

accounts for the case where the feedback is equal to the action. The complementary process established by calling the meaning subroutine is fired by the occurrence of a quote. (Note that other processes could later be established for other uses of the quote in particular circumstances, as specified by EXECPROC and IDENTIFIER. In this first case the new process would in future be activated by the appearance of a quote in any situation. This is because it was learnt in a situation (direct output) whose attributes were already specified by another process).

What should the process for a quote actually do when activated? It ought to predict the feedback and it could do that by using the specifications in the expectation subroutine to call the procedure that makes predictions in the simple cases. It does not work because that subroutine will set EXECPROC and IDENTIFIER but not SUPERORDINATE. As explained in section 4.2.2 this last parameter is necessary to construct the imperative and so cause the specified procedure to be executed. There it was pointed out that the imperative is the complement of a local expectation and it will be seen how this principle is exploited in grammatical processing. Differentiation does not need to use these constructs: global LTM is sufficient. What is more there are some rather boring technical problems in allowing the use of local expectations (i.e. frames) here: they arise from the danger of muddling these with the grammatical ones. There appears to be no theoretical difficulty in using them but a lot of re-programming would be necessary to re-organise things so that a word-process would not inadvertently answer its own grammatical expectations (intended for other words in the sentence) while still responding to the fruits of differentiation.

Owing to these technicalities, the program does not make causal

predictions as much as it theoretically could. However, learning is not impeded thereby because it takes place when the expectations fail, in any case.

6.4.3 Another kind of generalisation

Section 5.2.3 described a method whereby specifications could be relaxed, resulting in generalisation. There, generalisation was exemplified in the meanings of words but it applies to any procedure that invokes STM. Because of the localised nature of the fruits of differentiation (all the extensions to one procedure may be viewed as a frame even if they are not so implemented) a somewhat different variety of generalisation is possible in addition to the first sort. Indeed it is necessary to prevent many similar extensions being made to one procedure. It involves the attributes in the expectation subroutine.

Provision for generalising the specifications in the expectation subroutine is made for the situation where Differentiation is about to make an extension to a procedure that has enjoyed a similar extension before. That is, the Synthesiser has created a new meaning subroutine but it happens to match one that was synthesised on a previous occasion as an extension to this same caller. For this purpose a record of all earlier extensions is held in such a way as to be inaccessible to ordinary STM lookup (i.e. in the property list of the procedure).

If such a match of the meaning subroutine is found, the implication is that the specifications in the expectation subroutine need to be relaxed. If this check were not made, many similar extensions might be applied to one procedure. In the case of differentiation, shortening the SEARCHPROC as is done during generalisation using a generalised matcher is the only possible generalisation. The same methods are used by grammatical processing and there the possibilities

are more numerous.

If over-generalisation occurs, differentiation will cause special cases to be provided for just as it does in other instances. It avoids generating deeply nested structures of expectations by checking whenever it is called upon to extend an extension. Often the new extension can be placed on the same level as the old one, thus making it more efficient. Actually the program always does this but a better implementation could try both possibilities and would be constrained in its choice by what LTM matching can do.

In this way, a more general expectation is established but the old special case process is still around and by the LTM rules will be activated when it is appropriate in preference to the more general one. Care is therefore taken that the two processes (special and general) have all their procedures shared so that they remain in step during subsequent manipulation.

6.5 Grammar

Just as one may speculate on the evolution from causality to meaning so too it is reasonable to conjecture a phylogenetic path from differentiation to grammar because of the remarkable similarity between the two learning processes and the procedures they synthesise.

As explained above, differentiation is effectively the provision of a conditional extension to a procedure. This is done by synthesising and executing a new procedure containing a condition subroutine which calls the LTM interface with the EXECPROC parameter equal to the old procedure that is to be extended. In grammatical learning a similar subroutine is created in order to establish an expectation but this time the EXECPROC parameter is set equal to the procedure to be expected. Furthermore, an expectation subroutine must set the SUPERORDINATE parameter of LTM in order to include the expectation in

a frame.

The only other difference is that because of the extra degree of flexibility (in EXECPROC) there is more scope for generalisation in the case of grammar. That is taken up in section 6.5.2.

6.5.1 <u>The Unit</u>

According to Halliday (1961), language is in substance a one-dimensional phenomenon (speech through time; writing along a line) upon which grammar imposes a second dimension by segmenting it into units of lessening rank; sentences, clauses, phrases, words and morphemes.

In carrying out the segmentation DISCO must begin with the smallest unit of substance - an instruction in a procedure - and, using long term memory, construct the units in order of ascending rather than descending rank. It does not set out with the objective of segmenting or parsing (see the discussion in Chapter 9). Rather grammar characterises the form of the process of interpreting a sentence.

Consider as an example

: PRINT A COMMA BEFORE AN ASTERISK

: ,*

The run-time structure (saved in a so-called "trail") after the sentence has been interpreted is displayed in simplified form in fig. 6.3. In most cases, the word-processes retrieved from LTM perform a seek (to the right) with accompanying local expectations. The nouns do not and they supply the results for the other processes. The process for "before" also contains a seek to the left in order to relate the events referred to by the respective clauses.

In this example the process for "before" receives from the left a reference to an imperative action because of its own meaning and

imposes the requirement that it be SUPERORDINATE to the reference of the right hand clause (or phrase, or group). Hence it too is imperative (SUPERORDINATE is the means by which imperatives are indicated: see section 4.2.2). Moreover, the other attributes of "print" are carried over to the group "an asterisk" by default. This happens quite naturally because nothing has intervened to alter these specifications and they remain in the STM interface process.

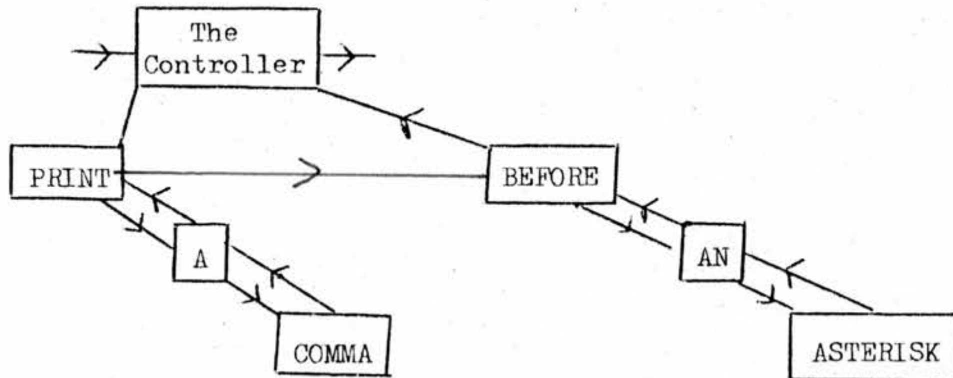


Fig. 6.3 Run-time structure after interpreting: "Print a comma before an asterisk"

Interpretation of the sentence concludes when the process for the word "before" hands back to the Controller.

Fig. 6.3 represents a run-time structure that actually exists. Each box represents one or more process descriptors, one variable (a trail) of which contains the descriptor of the next lower process possessing a return (RRISE) arrow to it.

It is easy to imagine the behaviour recorded by Caplan (1972) in the context of such a two-dimensional record. His subjects heard sentences spoken during which an irrelevant noise would be made. They were subsequently unable to be accurate about the point in the

sentence at which the noise occurred and their errors fell into characteristically grammatical patterns, seldom crossing clause boundaries. Earlier work on this subject was carried out by Ladefoged (1959). It is cited here as evidence for the psychological reality of grammar; evidence that is not inconsistent with the present formulation.

It can be seen that the program being described here has no separable components for syntactic and semantic processing, a point that will be taken up in Chapter 9. However, it is possible to separate the acquisition of grammar even though it works in much the same way as everything else.

6.5.2 Expectations

When a new word is encountered by the program, the New Entity Handler performs a seek which in the special circumstances of a new word constructs a general local expectation attached to its own process (a frame with only one entry). Perception then interprets the rest of the sentence. The form of the local expectation is a suspended process established by invoking the LTM interface with the parameter SUPERORDINATE set to the process descriptor of the seek. It is general because the only other parameter defined is IDENTIFIER equal to a variable name (FOUND) that always holds results in meaning subroutines.

If the end of the sentence is reached without any result, a standard procedure (the End Indicator) returns a null one and the Synthesiser is not given a result from the right. If on the other hand one is produced, DISCO's grammatical component will construct an expectation subroutine. The New Entity Handler will ensure that the subroutine is executed at the right time for it to be attached in a frame to the process for the new word in LTM. The value of EXECPROC

in this subroutine will be equal to the procedure that gave the result back to the seek.

The example of "two" will illustrate this. The program first encounters the word in a situation like the following.

:**

:TWO ASTERISKS

What happens to "-s" is irrelevant; it involves seeking to the left but does not interfere with the program's actions concerning "two" at this stage.

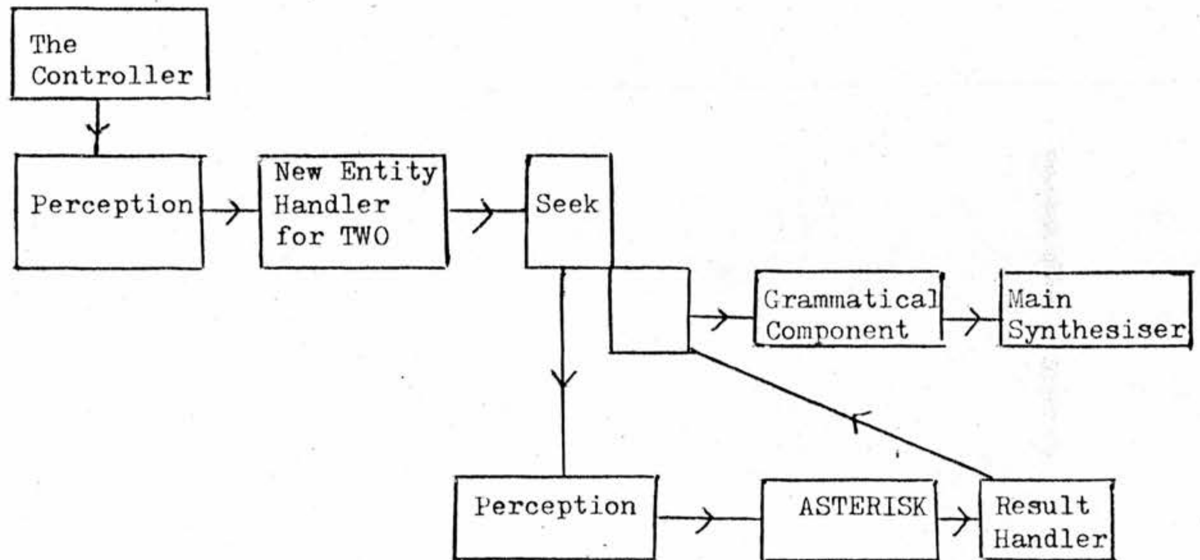


Fig. 6.4 Run-time structure for learning "two" in "two asterisks" (where the program's treatment of "-s" is not shown).

The New Entity Handler always performs seeks both to the right and the left. The one to the left produces no result in this case because "two" is the first word in the input line. The general expectation set up by the seek (to the right) is denoted by the box

joined to the lower right-hand corner of the one marked "seek" in fig. 6.4.

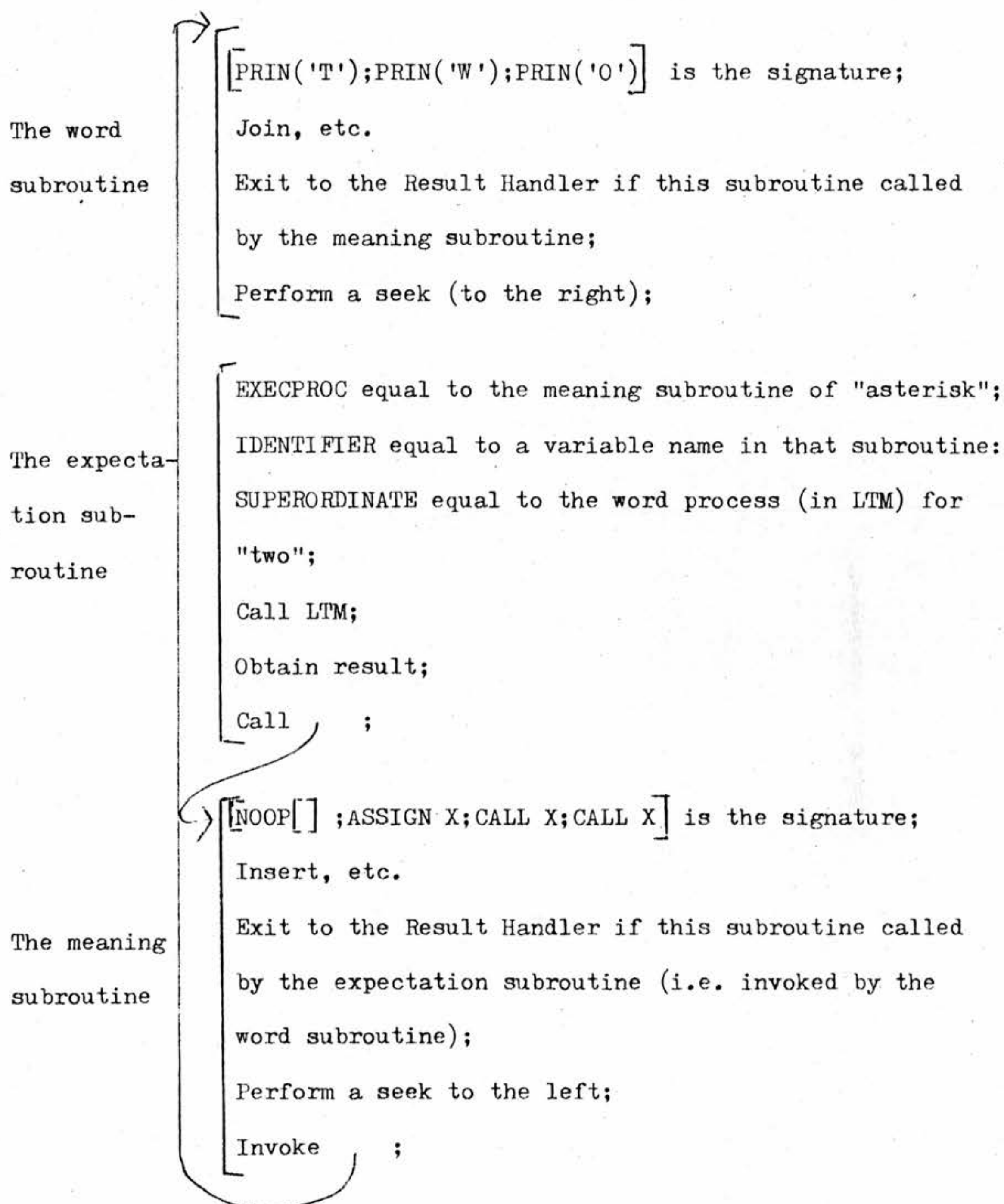


Fig. 6.5 Representation of the grammatical element of "two" built in to the framework of fig. 5.6.

Perception performs LTM retrieval for the next item in the input line, which happens to be a blank space. The process yields no result and is omitted from fig. 6.4. Next, Perception retrieves and activates the process for "asterisk". This locates the record of the preceding line in short term memory and the Result Handler performs the LTM retrieval that causes the general expectation of the seek to be activated. This process receives the result and passes it over to the Main Synthesiser by way of the grammatical component.

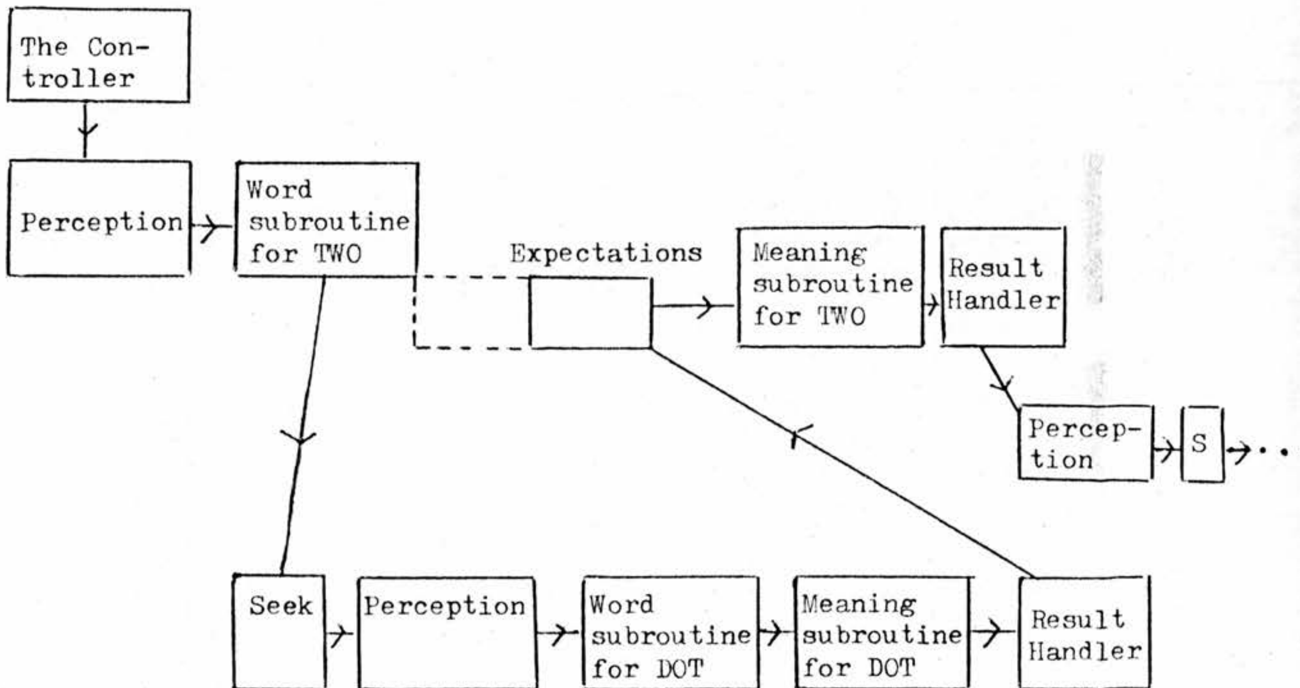


Fig. 6.6 Run-time structure interpreting "two dots".

The Main Synthesiser constructs the word and meaning subroutines shown in fig. 5.6 and represented in fig. 6.5. The invocation of the meaning by the word subroutine is seen here to be indirect (hence "invoke" is written instead of "call" in fig. 5.7). The seek

in the word subroutine causes Perception to proceed with interpreting the following words in the input line. When one of those processes produces a result it causes the appropriate expectation subroutine to be executed, and this calls the meaning subroutine. Fig. 6.6 portrays the flow.

Because of the provision of expectations a word may have several meanings associated with it, each applying in different situations, as illustrated in fig. 6.7. The boxes represent processes suspended

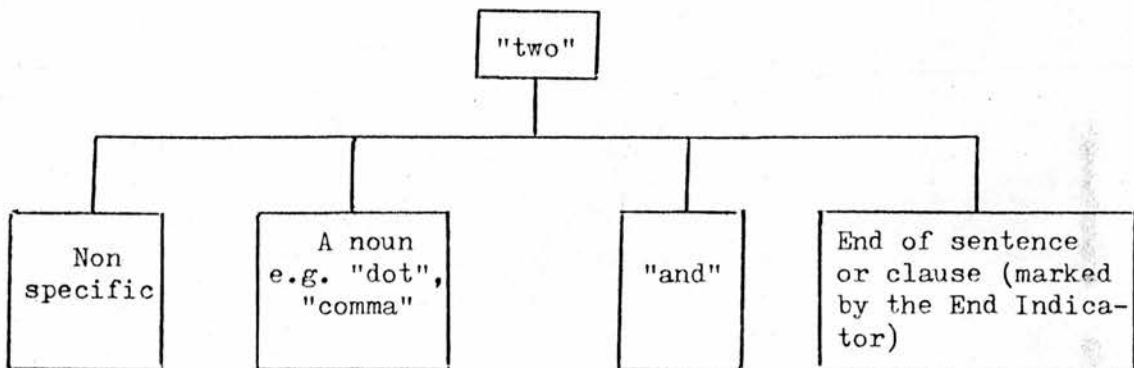


Fig. 6.7 A frame of expectations associated with the word "two"

by a call of the LTM interface. They are each waiting for the occurrence of the situation described inside them. The top process is in the global LTM. When the program reads the word "two", that process is invoked and the frame (stored context) is automatically instantiated.

Each expectation carries a different meaning of "two". For the case of a following noun, the meaning is as given in the example of section 5.2.2.1. When "two" is followed by the End Indicator, it means "££". When followed (directly) by "and" it performs addition.

The non-specific expectation (simply IDENTIFIER = "FOUND") catches anything that the others do not cater for explicitly, because of the LTM precedence rules.

The initial learning of "two" (performed by the New Entity Handler and the grammatical component) can only provide one expectation in addition to the non-specific one. Subsequent construction of a frame takes place through further examples (needing at least one example for each expectation) and can happen in two distinct ways. The first is by activation of the non-specific expectation which is a general catch-all for further learning purposes; the second is by contradiction of the meaning associated with an expectation. In either circumstance, the appropriate action may be to provide a new expectation (i.e. a form of differentiation) or to perform generalisation.

6.5.2.1 <u>When to generalise</u>

Although the expectations established by grammatical learning are not all extensions of one procedure like the fruits of differentiation they are localised in frames and it is possible to exercise the same options of generalising either the meaning or the expectation.

It will be recalled from section 5.2.3 that generalisation of a meaning takes place when its STM retrieval in the subroutine fails. However, when the meaning is associated with a particular expectation DISCO has the alternative of establishing a new expectation i.e. - setting up a special case - rather than generalising the meaning. Conversely, although the normal consequence of a result that satisfies a non-specific expectation established by a seek is the creation of a new expectation, here DISCO's grammatical component has the alternative of generalising one of the existing expectations

to cope with the new case.

The criteria for the choices are complementary with respect to these two situations. The criterion for generalising an expectation is the existence of another expectation in the same frame and with the same meaning. For instance, when the indefinite article "a" is first learnt as in the following

: .

: A DOT

it is set up with a local expectation (value of EXECPROC) specific to the meaning subroutine of the "dot" process. Next time, its general (non-specific) expectation is invoked.

: ,

: A COMMA

The performance of the grammatical component when this invocation takes place is just like Differentiation. It calls the Main Synthesiser to create a new meaning subroutine and searches the procedure of the frame looking for a match to it. (The STM interface is employed). When no match is found it sets up a new expectation but in this example the old and new meanings do match and so the specifications in the expectation subroutine are relaxed and the generalised version is run to establish a more general expectation.

The converse choice arises when the meaning of an expectation is contradicted (by failure of STM retrieval in the meaning subroutine). Here the choice to be made is whether to generalise the meaning or to construct a new expectation in order to make a special case with a particular meaning. The criterion for generalising a meaning is the existence of such an expectation in the frame already. The method is to construct an expectation subroutine particular to the present case

and then see if the frame already contains one that matches it.

The effect of this criterion may be illustrated by continuing with the indefinite article "a". In both the above examples, the STM match was with the character read in the previous line. Therefore, within the meaning subroutine for the word "a" will be the specifications of EXECPROC and IDENTIFIER indicating input from the teletypewriter and SUBORDINATE signifying past tense. So the meaning subroutine constructed in the second example (a comma) matched and the expectation was generalised. The frame of "a" is illustrated in fig. 6.8.

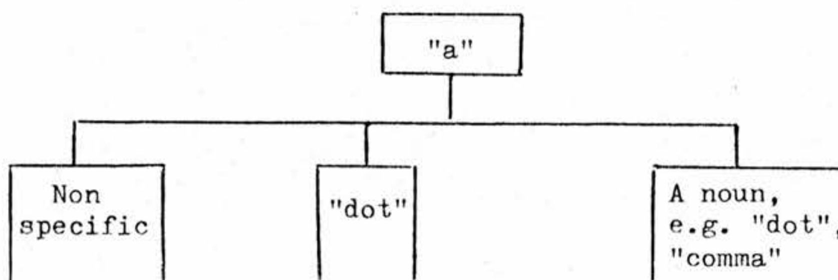


Fig. 6.8 A frame of expectations associated with the indefinite article.

Both the specific expectations have the same meaning subroutine. They are, in fact, structurally the same so that a change to one, e.g. generalisation, will apply to them both. An example which will cause generalisation of the meaning is the following.

:A DOT [PRIN('.')]

.

It causes the specific expectation for "dot" to be activated. When the program constructs an expectation subroutine to represent the present case, it matches with the one just executed and so generalisa-

tion proceeds, relaxing the specifications of all three STM parameters in the meaning subroutine that is common to both expectations.

The example would also have worked with "comma".

:A COMMA [PRIN(',')]

,

Although there is no specific expectation for "comma" in the frame, the program did construct an appropriate expectation subroutine while it was working on the previous example of that word. It does save such subroutines along with the frame. (They are implemented as subroutines of an extensible procedure stored in the property list of the process descriptor for "a" in LTM.)

On the other hand, something different happens if another word is tried.

:A SEMICOLON [PRIN(';')]

;

Now the program creates a new expectation in the frame and it has its own meaning. Whereas to type in "a dot" would have no visible effect, typing in "a semicolon" would cause the program to print one, because of the particular meaning acquired for "a" when followed by "semicolon".

A similar example involving "comma" or "dot" now would cause generalisation covering all cases except "semicolon", which would remain anomalous until a suitable example came along, e.g.

:;

:A SEMICOLON

Alternatively, the expectation for "semicolon" might have been generalised.

:A HYPHEN [PRIN('-')]

-

This would leave the frame of "a" as in fig. 6.9

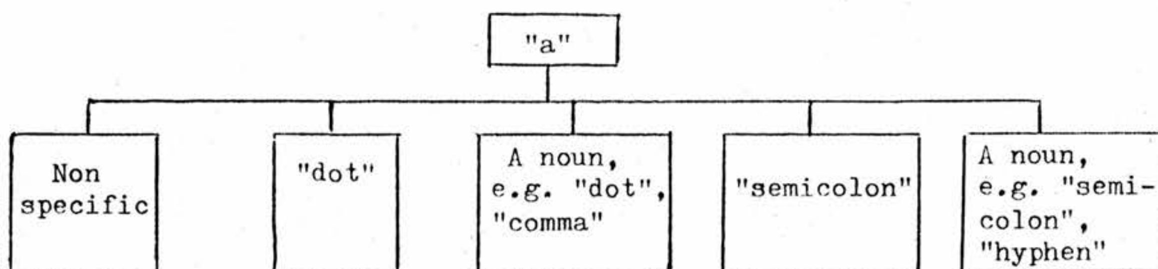


Fig. 6.9 A subsequent frame associated with the indefinite article

Now the right-hand box completely overrides the middle one since it is just as general and is more recent.

6.5.3 <u>Results from the left</u>

A beautiful validation of this formulation of memory and certain cognitive processes comes from the fact that whereas results are received from the right by using the LTM interface, it is STM that supplies them from the left.

The situation is much simpler for results from the left. No choice of meanings has to take place. When a word is learnt, the New Entity Handler furnishes it with a subroutine that will seek to the left (via STM) if a result is present from the left in the learning situation. If not, the possibility of incorporating a seek left at a later time is still open because differentiation affords this capability also.

The attributes of the STM retrieval for a seek left are determined by the sender of the result in the learning example. The same is of course true of expectations that look to the right. There is scope for subsequent generalisation. If such an STM lookup fails, a generalising routine will scan short-term memory for an unused

result (identified as a caller of the Result Handler on the current return chain) and will generalise the STM invocation along the same lines as for meaning.

The significance of using STM for this purpose lies not so much in its direct use but in the transformation deriving from the powerful complementarity principle which appears to be universally applicable to memory calls. This point is taken up in section 6.5.5.

One might think that the extra degree of freedom introduced by allowing textual matching on execution procedures should benefit seeking left as much as it enriched seeking right via expectations. This has not been tried, however, and it seems ripe for further research.

6.5.4 Structure and Class

In addition to the unit, Halliday (1961) declares three other categories necessary to the description of grammar. They are structure, class and system.

The foregoing has shown how the memory matching algorithms realise class by function. Winston (1970, pp.197-8) has stated that this is a most desirable goal. Although a wide variety of classes has not been demonstrated here, a principle has been established with number and the plural. This principle is couched in general terms and it should prove to be more widely applicable.

Halliday enunciates structure as the organisation of classes so that the members of certain classes may appear in certain sequences within units of higher rank. In DISCO, structure is embodied in the expectations of classes. Moreover word-processes with similar sets (frames) of expectations inevitably operate similarly and so make up implicit classes.

A notion that Halliday (1967) developed later, in systemic grammar, is the system. A system is a subclass that is closed. The grammarian can therefore say something about all its members and so can expand his territory in an attempt to come closer to formalising meaning (by an increase of delicacy). The computer program handles semantics and syntax by the same means and the traditional problem of moving outwards from syntax, as it were, (cf. Chomsky, 1965) is no longer relevant. Systems will presumably emerge as classes with a definite number of members (e.g. the interrogative words: where, when, why, who, what, how, etc.). A simple observation is that words like "dot" and "print" refer to sensori-motor procedures while "is" and "and" are defined purely in terms of operations in memory and so are more syntactic in nature.

6.5.5 Complementarity

The procedures synthesised by the grammatical component refer to STM and LTM and the powerful transformation of complementarity can profitably be applied to them also.

6.5.5.1 On the right-hand side

It will be recalled from sections 5.2.2, 6.2.1 and 6.4.2 that, although the elementary interchange of the two kinds of memory reference is quite simple, provision also has to be made in the procedure to support both of the complementary modes in which it can operate. When meaning is transformed to utterance some variables are redefined so that certain segments of the procedure are executed in only one of the two modes (i.e. in only one of the two resultant processes saved in LTM). The loop defined because the subroutines are mutually recursive is never followed and the Result Handler is called only once on each occasion, as intended.

Although the basic substitution (STM-LTM) is quite simple, each new instance of its application presents non-trivial questions as to how the transformation is to be exploited by the procedure to which it is being applied.

An expectation subroutine is synthesised by the grammatical component of the program. Its form has already been outined. When executed, it attaches a new expectation to a frame. Fig. 6.10 contains the essential procedural components of a frame that is established in this way. The meaning subroutine is called by the expectation, rather than the word, subroutine. The question is whether the meaning should call the expectation or the word (subroutine) when in complementary mode. If it calls the expectation subroutine, a close analogy with the simpler cases is preserved but essential information is lost.

In fact it calls first the expectation and then the word. A working example that the program can deal with should make this clearer. It involves counting. Conventionally that is not a grammatical activity but it turns out that the same principles apply. The program is taught to count objects on the blackboard by reciting the numbers while deleting one object at a time. Further details will appear in Chapter 7. By being shown examples such as

: ["***]

: 1/*2/*3/*

it learns to anticipate the delete character (/) following each number. The anticipation is in the form of expectations attached to the frames of the various numerals and possesses the same format as the more conventional grammatical information.

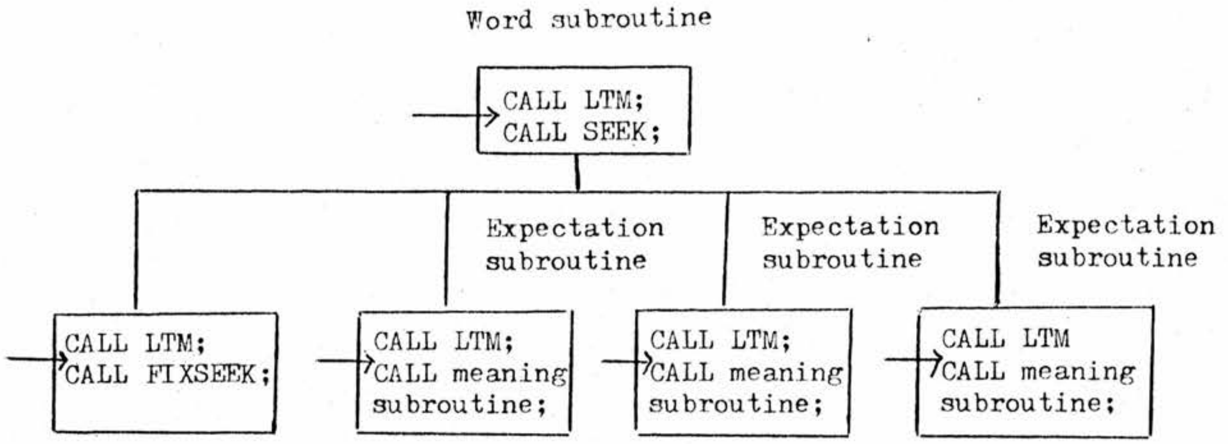


Fig. 6.10 Processes organised into a frame of expections. The arrows indicate the instruction at which each process was suspended.

In complementary mode, the meaning subroutine is executed first. The program, having perceived several instances of counting, is now expected to count by itself.

: ["****]

: COUNT THE ASTERISKS

1/*2/*3/*4/*

It was taught the word "count" and this causes it to start with "1/*". The link to 2 is a function of seek left complementarity and is dealt with below (section 6.5.2).

Here we are concerned with how the program's expectation of the delete character following 2, say, can be converted to the action of printing "2/*" when it does the counting itself. The STM parameters for specifying the print action and the asterisk are obvious. Next, assume that the program is executing the meaning subroutine found within the expectation of "/" attached to "2". The meaning calls the expectation (subroutine) and after that the word.

The expectation subroutine contains the information that established an expectation for "/". But the complement (STM instead of LTM) is the imperative (because SUPERORDINATE is specified) and so the meaning subroutine associated with "/" is actually performed. It in turn calls the word subroutine of "/" thus supplying the "/" in the output line. Finally, the word subroutine for "2" will put "2" in front of "/*". The flow is shown in fig. 6.11.

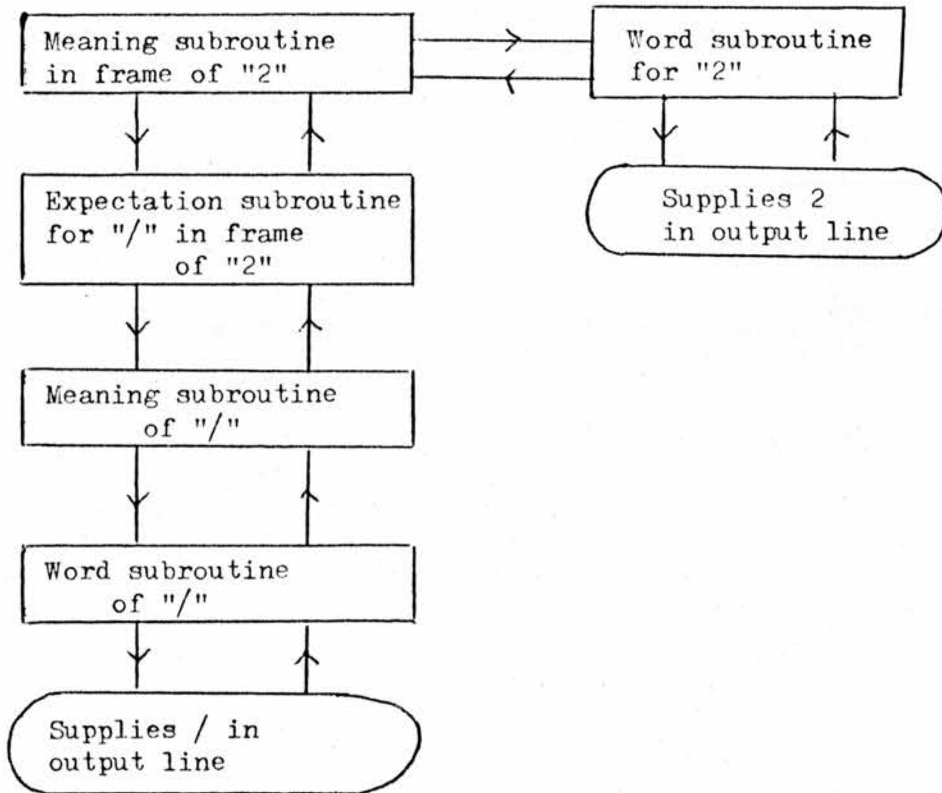


Fig. 6.11 Flow during part of the counting exercise.

6.5.5.2 <u>On the left-hand side</u>

Seeks to the left rely on an STM call, the complement of which is an LTM call that establishes a process whose activation is conditional upon the particular circumstances specified in the

parameters of the call.

In the example of counting given above, the process for "2" receives a result from the left (from "1"). Consequently, a procedure is synthesised (known as a seek-left subroutine) which contains an STM call to access such a result in future. (This procedure is attached to the expectation subroutine for "/" in "2".) This STM call specifies EXECPROC equal to the procedure (meaning subroutine) associated with "1", IDENTIFIER = "FOUND" (a variable in that subroutine which holds a result from an STM call itself) and SEARCHPROC equal to another procedure because the result represented a partial match. (In fact, SEARCHPROC = "Empty" which will not fire in the case of exact match when FOUND refers to a process.)

The complement (LTM for STM) is an LTM record conditional upon a partial match in "1". If there are two asterisks on the blackboard, the program will count to 2 but if there is only one, then "1/*" will produce an exact match and counting will stop.

In complementary mode, the seek-left subroutine effectively invokes the procedures (word and expectation subroutines, if present) which would be invoked by the meaning subroutine with which it is associated. Hence, such behaviour as was described in section 6.5.5.1 is transparent to the use of the seek-left subroutine and the link from 1 (followed by /) to 2 (followed by /) is complete.

6.6 Conclusion

When viewing the program as a psychological model its most primitive components from a phylogenetic, or evolutionary, point of view are causality and differentiation. They rely upon the existence of some similarity between cause and effect or, more generally, between condition and action.

It is essential to language that tokens usually bear a purely formal relationship with the acts and percepts that they connote. This is the first of two essential distinctions between meaning and causality. It may well be that an adequate learning system for causality should also possess the ability to make this separation. That could easily be done in the present program and further research should explore the usefulness of the idea.

The second essential quality of language is grammar. Causality and differentiation only deal with results from the left. However, they already possess the capacity to handle and relate two results in one procedure: the second result is always derived from the input condition. It is a giant leap forward for the second slot to be filled instead by the result of further processing and thus provide the richness afforded by grammar while at the same time eliminating the requirement of a similarity between condition and action, producing language as we know it.

Finally, we note that the vehicle for providing grammar is only a minor adaptation, applied somewhat differently, of differentiation.

Thus the program itself is highly parsimonious and this quality is quite apart from the desirable uniformity achieved by representing knowledge in terms of references to long and short term memory.

PART III

A COMPLETE DIALOGUE

Chapter 7

In this chapter an annotated dialogue on a teletypewriter between the computer program and a human tutor is presented. It is complete in two respects: first, it contains a sequence of exchanges adequate to teach the program what it needs in order to engage in the subsequent activity described; and second, it has been successfully carried out on a computer by a working program that has been described in Part II and is briefly outlined below.

7.1 Summary of the Program's Normal Operation

The diagram of chapter 5 that illustrates the basic loop is reproduced in fig. 7.1

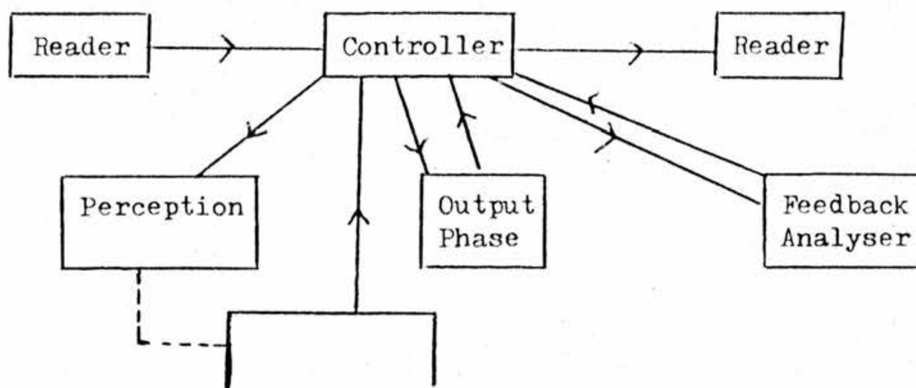


Fig. 7.1 The program's basic loop

Input from the teletypewriter is converted to procedural form by the Reader, which also performs actions on the blackboard prompted by quotes (") and strokes (/) and compiles instructions enclosed in square parentheses ([ ]).

Perception works on the converted input line from left to right looking up long term memory and activating the processes that it retrieves. Typically these processes give results corresponding to the meanings of words and in so doing they interact to produce the interpretation of the sentence. The next section discusses what goes on in more detail.

Sometimes the interpretation of a sentence will lead the Controller to call the Output Phase as in the case of imperatives. The presence of square parentheses in the input line will also cause the execution of that phase. It is not limited to printing. Any procedure (e.g. looking at the time or drawing on a screen) could be performed by it but only the printing of characters and words and use of the blackboard have been dealt with.

The Feedback Analyser performs its own reading and perception process (using the Reader and the Perception routines) on the feedback which the program's printing mechanism supplies to it. The printing mechanism (embodied in the subroutine PRIN which appeared in earlier illustrations) records its actions for this purpose, as well as performing blackboard manipulation in response to a quote or a stroke.

7.1.1 Processes invoked by Perception

The processes activated by Perception are first of all written to LTM as a result of learning through experience. Initially LTM is empty. When Perception fails to locate a record that matches the current input, it invokes the New Entity Handler which attempts to create a process for it. It calls the Main Synthesiser to construct the appropriate procedure and then invokes that procedure in order to create a new LTM record. That consists of the process (e.g. to

perform the interpretation of a word) and a key that expresses when the process is to be used (which would be when that word occurs as input). This is illustrated in fig. 7.2.

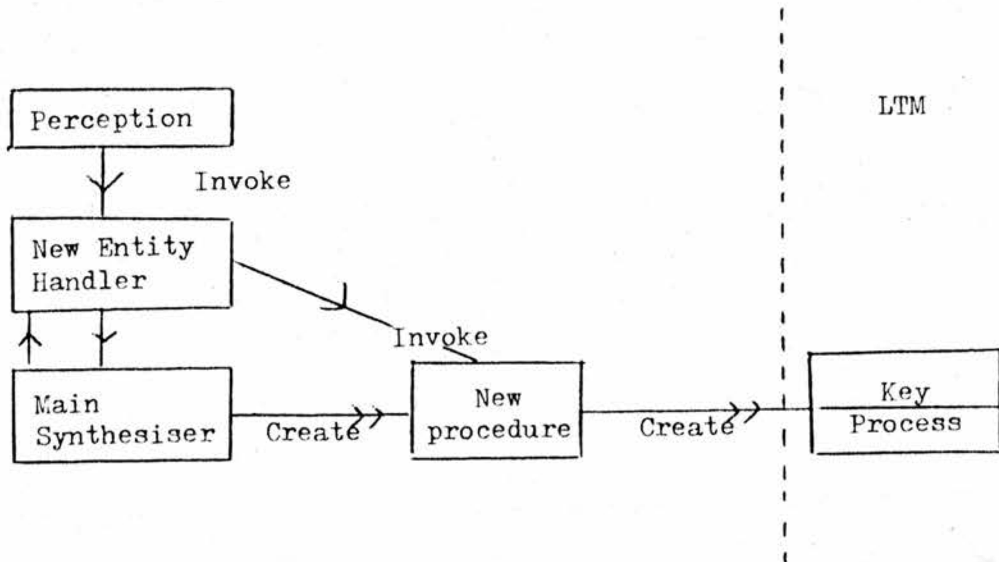


Fig. 7.2 Perception of a new entity

Fig. 7.3 shows how these routines fit into a framework when several entities are found by Perception. In the phrase "a dot", suppose the program has already learned about "dot" and a blank space but is being introduced to "a" for the first time.

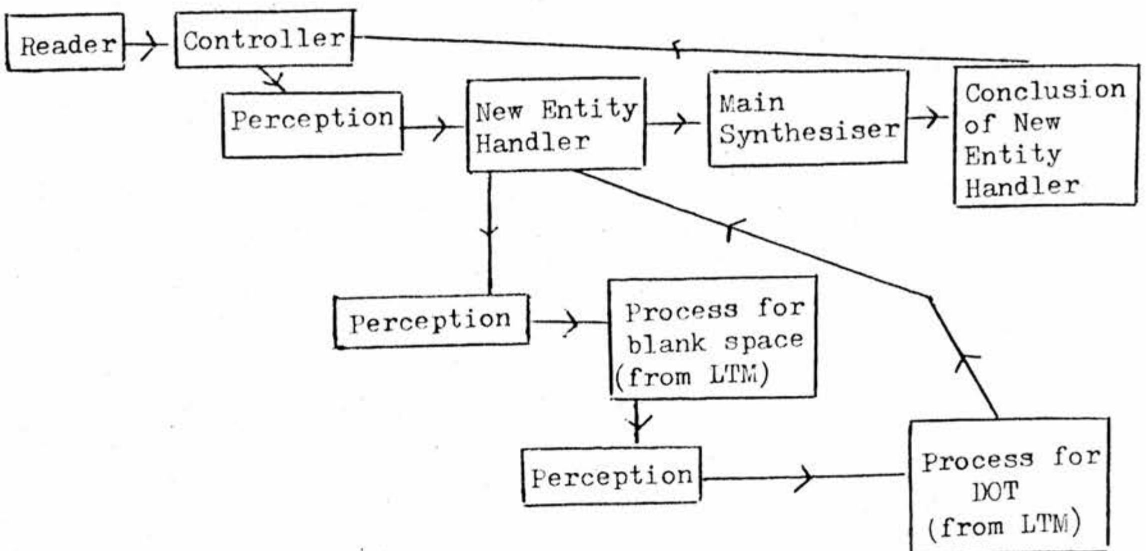


Fig. 7.3 Learning "a" in "a dot"

Many diagrams like fig. 7.3 appear below. In fact they have two aspects. They serve to explicate the main elements of the program's activity in various situations. Since the diagrams contain a record of the program's recent activities and experience they also display the principal contents of the short term memory. Searching and referring to STM is one half of the basis of the program's operation. The dual aspect of these diagrams itself illustrates the power and economy of this formulation of STM: the one primitive can be used to characterise and manipulate all essential facets of the program's behaviour since they are all represented in STM.

The other half of the basis is, of course, LTM. It contains a set of saved processes built by the program. Perception invokes some of them. At the heart of most of these is a reference to short term memory, the record of recent activities and experiences. Some processes, such as the one for a blank space, do not contain such a reference and perform only minor housekeeping functions. Others, such as relational words like "after" and "is" contain two or three (at most). The third is concerned with grammar rather than meaning. (See below.)

```
Supply specifications to STM;
Call STM;
Handle the result;
```

Fig. 7.4 Stereotype process activated by Perception

Fig. 7.4 shows a simple case. For the word "dot", the specifications to STM include the procedure [PRIN('.')] and this en-

capsulates the meaning of the word in a process that will interpret it.

7.1.2 Use of STM in Processes invoked by Perception

Short term memory is searched when the STM interface is invoked. This is done both by the program's built in procedures and by those it synthesises. When interpreting a line of input, typically a sentence, several processes (one or more for each word) will have parameters to specify to the STM interface in order to compose the total meaning. Each will add to the specifications of the others, sometimes using the results of others to complete their specifications. For instance, the process for "after" will take the reference of the clause on the left of the word and put it into the specifications of the clause on the right.

The vehicle for such communication is the process descriptor. This is an active record, so to speak, which is passed from process to process. It contains a set of slots, or variables, for storing the specifications and result of the STM interface function. This record is the implementation of the STM interface process. When it is activated, it passes the specifications it has been given to the STM interface function and saves the result. For efficiency, it avoids searching STM if it has a saved result which already meets the specifications. In the course of interpreting a line it is activated repeatedly as different word processes contribute their portion of the meaning.

The parameters that can be given to the STM interface are shown in table 7.1. They can be specified or undefined in any combination but at least one search argument must be defined before a result can be returned.

	Specifications or Parameters				
Classifications	Search Arguments			Constraints	
Descriptions	Search procedure	Identifier (Variable name)	Execution procedure	Superordinate process	Subordinate process
Names	SEARCHPROC	IDENTIFIER	EXECPROC	SUPERORDINATE	SUBORDINATE
Characteristics	Partial match	Exact match		Relationship includes equality	
Type	Procedural	Process-like			

Table 7.1 Parameters of the STM interface.

The word process for "dot" will contribute the search procedure (SEARCHPROC) [PRIN('.')]. An STM search with this as the sole specification would match any of a range of experiences or activities, as illustrated below.

:.

:DOT

:[PRIN('.')]

.

:DOT

:DOT[PRIN('.')]

.

When the program learns "print", it creates a new process, saves it in LTM, and invokes it subsequently when that word is read. It specifies an execution procedure and an identifier to STM retrieval.

The EXECPROC is the Output Phase of the program and the IDENTIFIER is the name of the variable that contains the direct output. So the meanings of "print" and "dot" together specify that STM is to be searched for an execution of the Output Phase in which [PRIN('.')] was the direct output. Hence, "print a dot" will match only the following case.

:PRINT A DOT [PRIN('.')]

.

The constraints are intended to capture temporal relations. In the above example, the execution of the Controller is superordinate to that of the Output Phase because the former invokes the latter (see fig. 7.5). The Output Phase is executed before Perception so that it is present in STM before the sentence is interpreted.

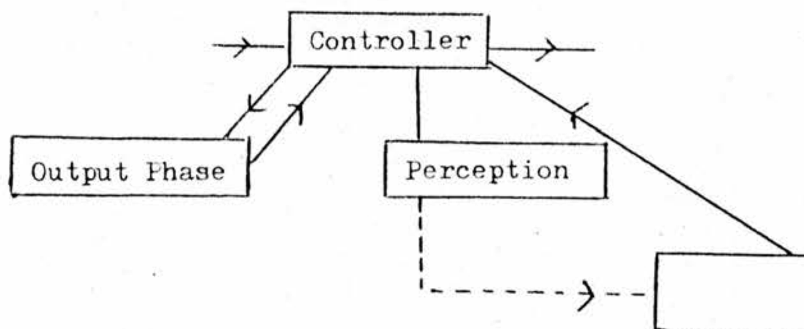


Fig. 7.5 Flow of control when there is a supplied procedure

The word process for "print" includes a specification of the SUPERORDINATE parameter to STM retrieval. It finds the most recent execution of the Controller and supplies this as the value of the parameter.

In a different example, the process for the morpheme "-ed" uses SUBORDINATE.

:PRINT A DOT

.

:YOU PRINTED A DOT

Fig. 7.6 illustrates that although the process marked Controller (1) is superordinate to the execution of the output phase, Controller (2) which corresponds to the line "you printed a dot" is subordinate to it. The rules defining this relationship appear in chapter 4.

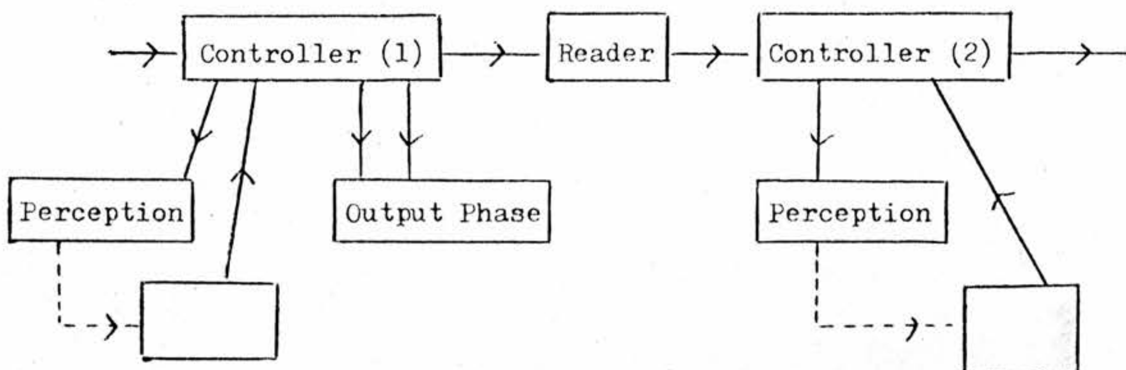


Fig. 7.6 Flow of control when an action is followed by a line of text.

The example illustrates also the imperative. When SUPERORDINATE is specified to STM retrieval and the search fails, the program will create a process having the given execution procedure (in this case the Output Phase) with the implied assignment of the search procedure to the given identifier. Thus, the Output Phase is executed, causing a dot to be printed.

Table 7.1 implies that a search procedure may be involved in a partial match, as in the following.

:[PRIN('*');PRIN('.')]

*.

:YOU PRINTED AN ASTERISK AND A DOT

The processes for "asterisk" and "dot" each specify to the STM interface procedures that only partially match the activity referenced. The "and" process can actually combine them to make an exact match: this is described in a later section. It would not be appropriate to allow a partial match into the execution procedure, the Output Phase. In general, that would give the searching algorithm too many time-consuming options. If a partial match into an execution procedure is needed, it is not permissible to request a match with the value of one of its variables as well.

7.1.3 Conventions

In samples of dialogue in earlier chapters, the conventions were that lines preceded by a colon are the ones typed by the human tutor and the others are produced by the program. Procedures supplied for the program to obey were written in parentheses. In Chapter 7, the only change is that in square parentheses only the characters will be written with the print instructions omitted for the sake of readability. Hence

```
: [*,]
  *,
```

is to be read as

```
:[CALL '*' PRIN; CALL ',' PRIN]
 *,
```

The actual program receives these procedures by way of a macro which expands to the separate print instructions.

7.2 First experiences

The program begins with no vocabulary and only the "knowledge", or capability, built into it that has been defined in Part II. Thus long term memory is empty and in short term memory only the state

descriptor for the Reader is present and it is waiting for input.

It becomes acquainted with a few objects in its domain by being shown them in isolation.

: *

: .

: ,

: ;

Now it has four entries in LTM that will enable it to recognise these characters should they occur again. The entries are in the form of suspended processes that will be re-activated (and copied so that the original is not destroyed) when the program reads these characters again. In these cases the processes will only initiate the retrieval from LTM of a match to what follows them or call the End Indicator if they are last on the line. So if the following is typed in

: *;

the process for "*" will be followed by the retrieval of the ";" process from LTM. The retrieval is initiated by Perception. The End Indicator will thereafter return a null result back to the Controller which is responsible for interpreting a line. Fig. 7.7 contains a diagram of this flow

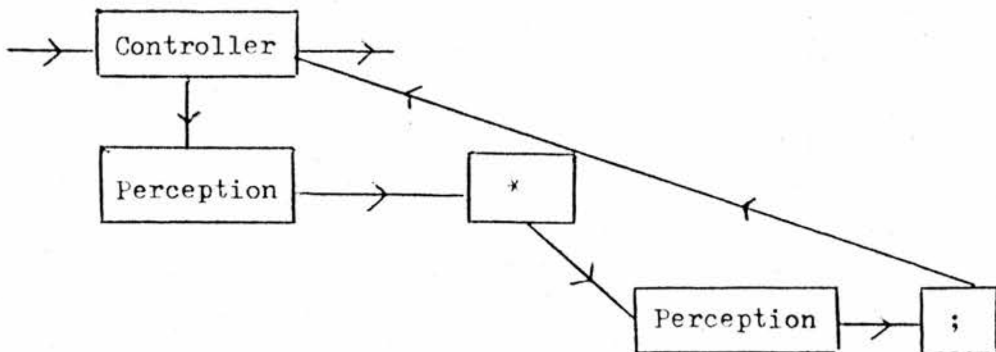


Fig. 7.7 Flow of control for processing the input "*;".

Now that the characters are represented in LTM they can be distinguished from something else on the same line. Their names are taught thus.

: *ASTERISK

: .DOT

: ,COMMA

: ;SEMICOLON

Perception invokes the character processes on each line and then groups the following data as a single entity. Had the training been reversed so that, for instance, the words were the initial percept as in

: ASTERISK

: ASTERISK*

the result would be the same. A reversal in the second line would lead to something quite different. If it had read

: ASTERISK

: *ASTERISK

the symbol "*" would have started to behave like a word, taking on the meaning of "I said" ("I" being the tutor) whereas in the first examples the words take the characters as their meaning and, at the same time, new records are written to LTM for the characters so that they are associated with the words in a complementary fashion.

Notice that if there is sufficient separation between the lines

: ASTERISK

and

: *ASTERISK

or their equally valid counterparts

: *

and

: ASTERISK*

the meaning "I said", or in the latter case, "I printed one of these", does not arise because the earlier line in each case will have disappeared from STM and Perception will not find it. With this caveat, all four examples work equally well and produce the same ultimate state of LTM.

Finally, LTM contains 12 entries, of which the four earliest have been superseded by four of the later ones so that they can never again be retrieved.

Blank spaces occupy a peculiar position in this scheme of things. It does not seem to be possible to avoid making special provision for them although only at the end of a clause has this proved necessary. At such places (marked by words like "is" and "after") the blank space has to be deliberately ignored. In all other cases it is treated like a word or symbol with no memory reference, just like the first characters the program sees. Therefore, it is taught by typing a line that contains one blank space. As was pointed out in Chapter 2, the status of blanks in electronic computing is quite unnatural.

7.3 Short phrases and sentences

After these earliest steps, the program is ready to start using some of its grammatical and generalising capabilities.

In the present treatment of grammar, words are of two kinds: those that require the results of the interpretation of what follows them before their own processing is complete, and those that do not. In the present work, a word, morpheme or other entity that the program perceives must be either of one kind or the other. A proposal that will be presented in Chapter 10 (section 10.1) for perceptual frames

may open the possibility of lifting this restriction.

The nouns turn out not to require results from following words; most other entities do. Fig. 7.3 shows that the New Entity Handler always finishes the interpretation of the input line before synthesising the procedure that will embody the meaning and grammar of the new word. This action of causing interpretation to continue in order to receive a result is known as seeking. If the New Entity Handler obtains a result (an STM reference found by an ensuing process) in this way, it ensures that a seek is also incorporated within the procedure it is synthesising.

Associated with a seek is a set of expectations in the same form as the long term memory but smaller and local in effect. Such a set of expectations is known as a stored context, or frame, and is attached to the process that issues the seek. When the New Entity Handler performs a seek, it establishes a general purpose expectation. If a result is received, it gives to the process that it is creating in LTM a frame containing a specific expectation appropriate to the current example.

7.3.1 The indefinite article

The indefinite article is taught thus

: .

: A DOT

The ultimate meaning of "a" that the program learns is vacuous but several examples are needed. As a result of the above two lines the program establishes a record in LTM with a particular meaning for "a" attached to a specific expectation for the procedure that the program executes when it is interpreting the word dot. Fig. 7.8 shows the LTM record associated with "a" and the frame of expectations below it. A general expectation is always provided by the program in

order to deal with future situations not covered by the specific ones already created from examples.

In effect, the LTM record is a global expectation while those attached to it are local. Both kinds have the same form, consisting of a process to be activated together with a key indicating when this should be done. The difference is that the local expectations are only in force when the LTM record to which they are attached is active and until one of them has its conditions met.

Associated with each expectation under "a" (except the general one) is a meaning, and these can differ. Meanings are represented by procedures that give parameters to the STM interface. In fig. 7.8 the meaning is to look for a past occurrence in short term memory of the object in a reading process. This sets the STM arguments SUBORDINATE, EXECPROC and IDENTIFIER. Subsequently both the meaning and the expectation are generalised.

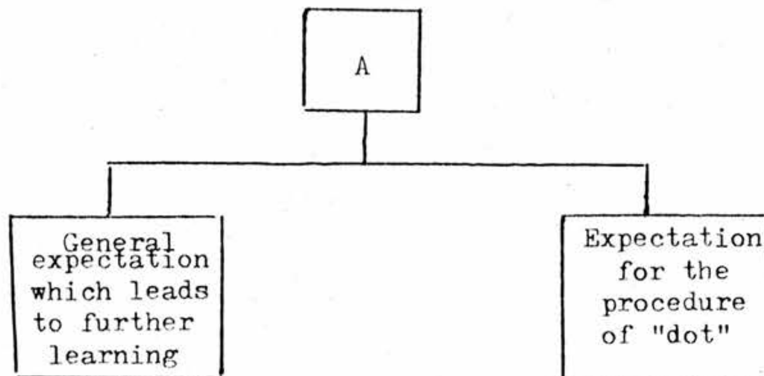


Fig. 7.8 Frame associated with "a" after one example

A sequence which has been successfully tested is the following.

: ,

: A COMMA

As before the STM invocation by "comma" locates the occurrence in the previous line. The meaning generated for the indefinite article this time is exactly the same as last time; only the expectation is different.

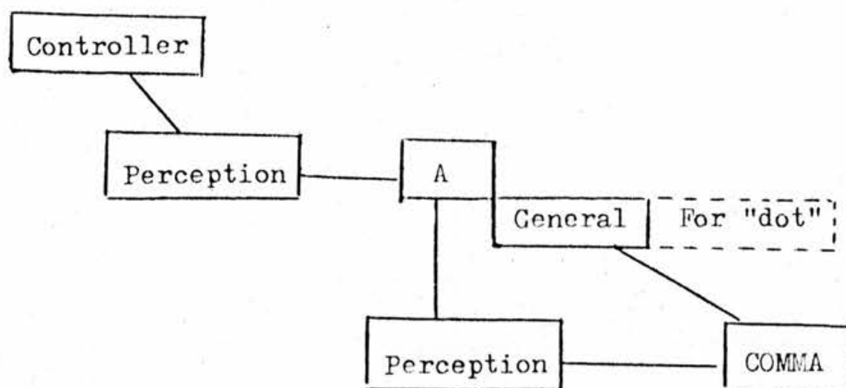


Fig. 7.9 Flow of control interpreting "a comma"

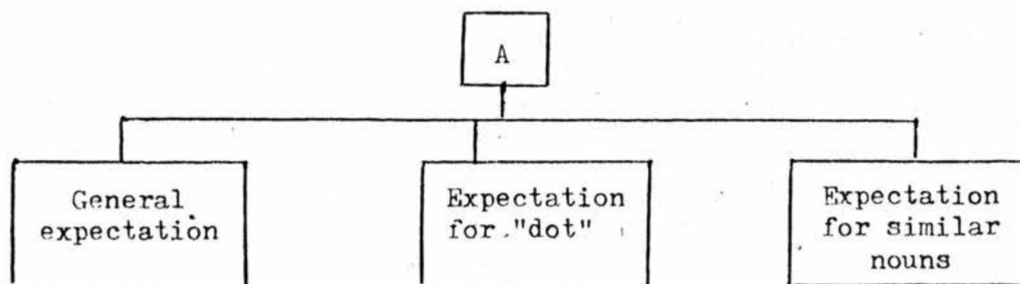


Fig. 7.10 Frame associated with "a" after a second example

Fig. 7.9 shows that the general expectation associated with "a" is triggerred by the process of interpreting "comma". It

leads to the synthesis of a new procedure corresponding to a new meaning of "a" in the context of being followed by this other word. The program detects that the new and old meanings are the same by comparing the procedures. It therefore proceeds to establish a generalised expectation rather than setting up a new special case. The end product is seen in fig. 7.10. The two expectations bear identical meanings.

The program builds one of these by synthesising and executing a procedure that invokes the LTM interface function. This function writes records in LTM and also creates expectations in frames. The parameters passed to the LTM interface function for insertion are the same as those used for searching STM.

In the present example, the new expectation is for the common part of the procedures for "dot" and "comma". As only a part is involved, a partial match must be anticipated. Therefore this common procedure must be the SEARCHPROC argument to the LTM interface, even though it is to match an execution procedure. IDENTIFIER will be set to bear a special token (the execution token) indicating this requirement to the interface. The EXECPROC parameter is then ignored and the key of the new record in the frame is formatted accordingly. This expectation will now apply to any of the nouns mentioned earlier.

The following line causes the meaning to be generalised.

: A DOT [.]

.

This example has caused all the arguments in the meaning of "a" to be nullified at once. It might have been done otherwise. For instance

: [;]

: ;

: A SEMICOLON

This contradicts the EXECPROC and IDENTIFIER parameters previously defined in "a" but not the value of SUBORDINATE. Another example would be needed for that.

This is not claimed to be an adequate treatment of the indefinite article. The description is included here for completeness of exposition. The case of a following vowel where "an" is used instead of "a" arises for the noun "asterisk". Again no claim is made for the adequacy of the way the program handles it. The presence of "-n" before a vowel is a requirement of pronunciation which is not relevant to the present work, and could not be detected by this program. The two following examples teach "-n" as a morpheme of vacuous meaning with an explicit expectation for the word "asterisk".

: *

: AN ASTERISK

: AN ASTERISK [*]

*

7.3.2 First verb

The first verb is "to print".

: PRINT AN ASTERISK [*]

*

The "asterisk" process locates the procedure for printing an asterisk in short term memory as direct output of the Output Phase of the program in a process subordinate to the Controller. These attributes become embodied in the meaning of the verb "print" as IDENTIFIER, EXECPROC and SUPERORDINATE respectively. Fig. 7.11 shows the flow. In fact, it portrays the run-time structure.

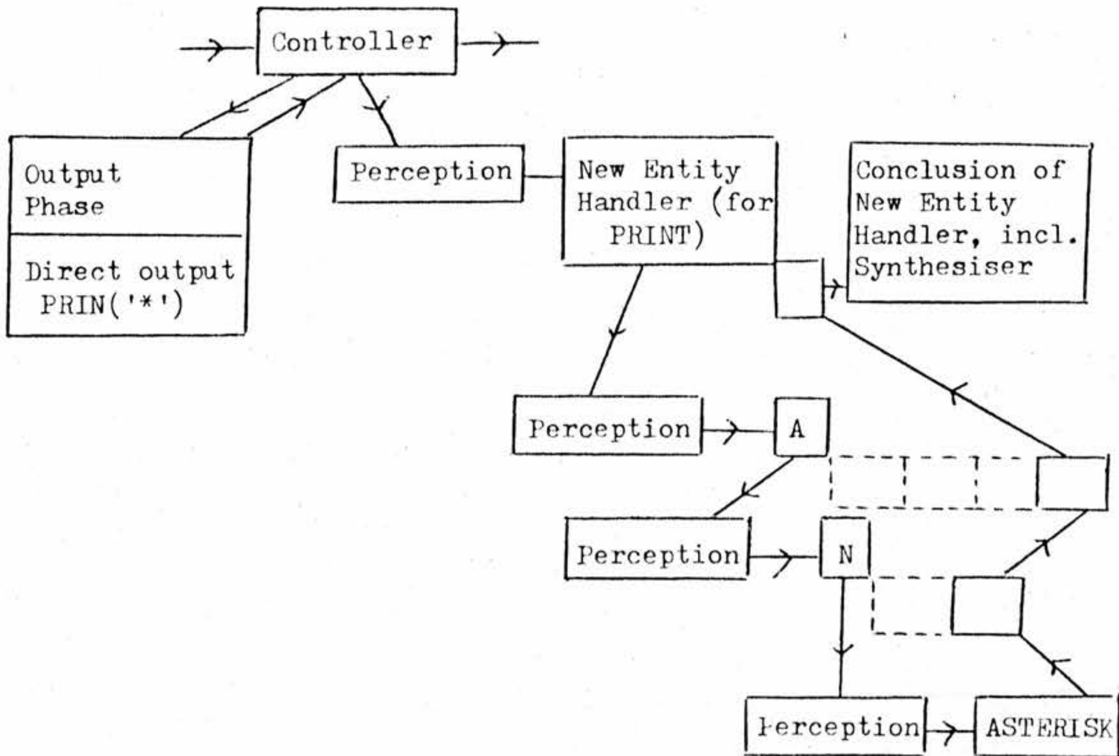


Fig. 7.11 Flow of control learning "print" in "print an asterisk"

Notice that whereas parameters or specifications are given to the STM interface function, it returns results that are said to have attributes and they correspond to the parameters. The attributes of the result presented to the New Entity Handler for "print" are incorporated in the meaning of that word by the Synthesiser. The attributes of the result that were not already specified to the STM interface by the other word-processes will be the parameters that the word process for "print" will specify. The Synthesiser generates the instructions to do this in the procedure for "print".

7.3.3 Past tense

The next step in the dialogue is to teach the past tense morpheme "-ed". In order to use natural sentences, a personal pronoun must first be introduced. The following contains the new word "you" which

takes on a vacuous meaning.

: YOU PRINT A DOT [.]

.

A complication arises because so far the program has always seen the verb "print" followed by "a" and an example of something different must be given. The details are immaterial. The actual dialogue at this stage runs as follows.

: **

: ASTERISKS

: PRINT ASTERISKS [**]

**

: PRINT A COMMA

,

: YOU PRINTED A COMMA

A full discussion of the plural appears below. Here the objective is simply to generalise the local expectation associated with "print". Otherwise, the last line would contain too many new features at once and would be ignored by the program. The line "print a comma" involves no learning but provides a reference for the succeeding sentence. Incidentally, it is now an alternative to the device of using square parentheses.

The resultant meaning established in LTM for "-ed" consists of a procedure that causes the imperative specification (SUPERORDINATE) of the preceding morpheme to be converted to the past tense (SUBORDINATE argument to STM). The processing in detail appears in fig. 7.12. First "you" seeks for a result from one of the subsequent words and then "print" does the same. When the program encounters the unknown "-ed" the New Entity Handler detects the outstanding seek from the word "print" on the left by inspecting the run-time

structure. Now the New Entity Handler itself seeks for a result causing "a comma" to be interpreted. The result received is a reference to the previous event, namely the printing of a comma.

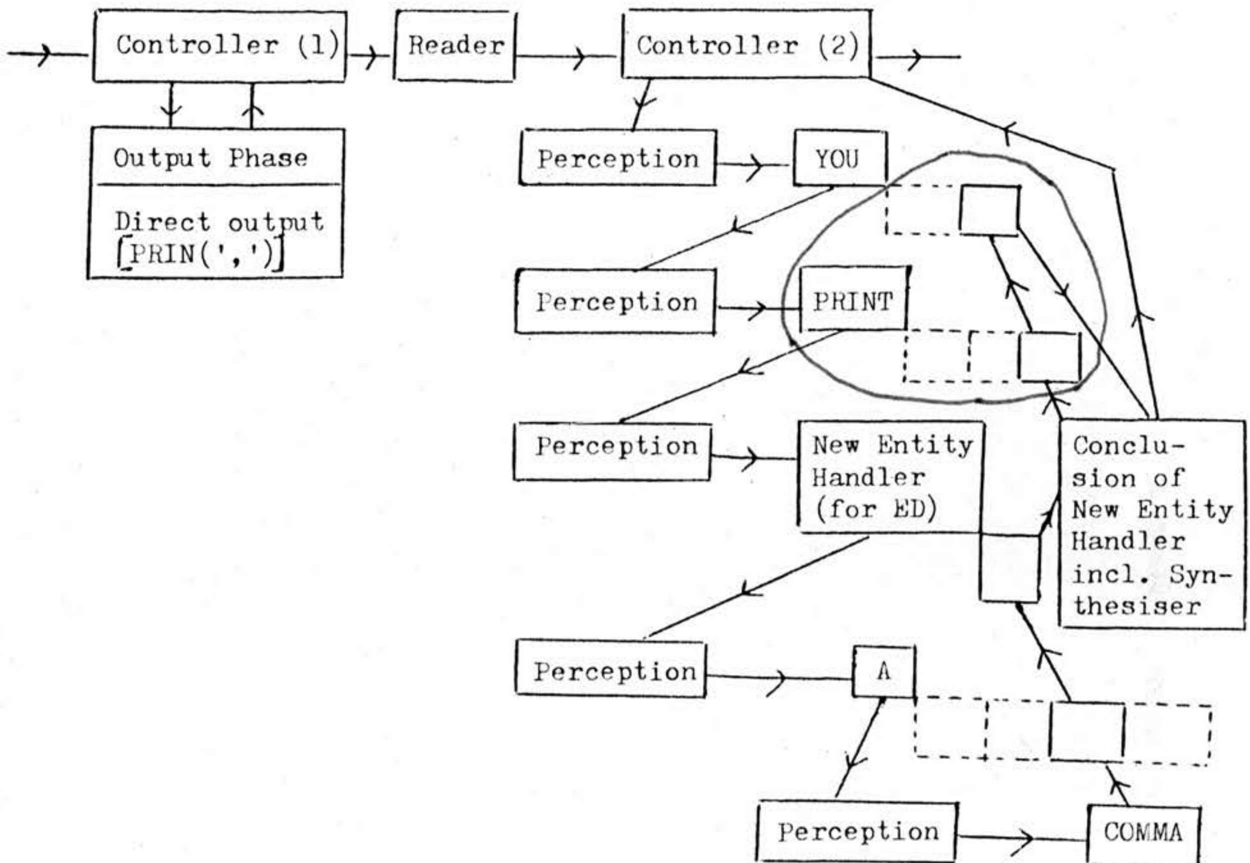


Fig. 7.12 Run-time structure learning "-ed"

Next the program closes the outstanding seek on the left as indicated by the circled processes, causing the "print" process to be run in such a manner that its result is returned. Hence the Synthesiser is given a result from both the left and the right.

Because of the STM attributes specified by "print" and "comma", the result from the left should be an imperative process (i.e. ready for execution) for printing a comma. However, the routine that

establishes imperatives will not take such an initiative when it detects that the New Entity Handler is active. In such circumstances it returns an indicator. The Synthesiser's action on receiving this indicator is analogous to that performed during generalisation. It retries the STM call without the process-like attributes.

The result of the retry is a reference to the preceding event. As this is identical to the result from the right, that is now discarded. The procedure for "-ed" is synthesised at this point. First it contains a call to close an outstanding seek on the left. The next instructions reset the SUPERORDINATE argument of STM to the undefined value. The Synthesiser checks for those attributes of the result that conflict with the STM specification derived from "print" and "comma". Finally, it sets up code to supply the SUBORDINATE parameter to STM, thus completing the specification of past tense. The process for interpreting "-ed" in future is established in LTM in the usual way.

Fig. 7.12 summarises the position. At the top left, the preceding event appears. The STM arguments produced by the interpretation of the sentence match that event in all respects except for sub- and superordinate and "-ed" becomes associated with this difference. Actually, the meaning is attached to an expectation of the indefinite article within the frame belonging to "-ed". Another example is needed to generalise this.

: [**]

**

: YOU PRINTED ASTERISKS

Most of the learning described above is repeated. The procedure now created by the Synthesiser matches that produced earlier and so the specific expectation for "a" is generalised for any word.

7.3.4 Another verb, and the past tense revisited

In English the past tense can also be formed with the auxiliary verb "to do". This is always the case for the interrogative (imperfect tense) in modern English. In the affirmative, "did" is used for emphasis. The program could not learn the use of this auxiliary verb in a question without first encountering it in an affirmative sentence because a sequence such as

: PRINT A COMMA

,

: WHAT DID YOU PRINT

where "did" is the new word, does not contain enough information for the program to work out the meaning. ("Did" might be associated somehow with ","). So it will not understand this verb until it appears in an example like the following.

: PRINT AN ASTERISK

*

: YOU DID PRINT AN ASTERISK

Insofar as this program is a psycholinguistic model, it predicts that the use of "did" in the interrogative would be impossible in English if it were not also used in the affirmative because of the requirement of acquisition by children. The same should apply to similar words in other languages.

The meaning of "did" derived from the above example is, of course, similar to that of "-ed". The syntax is somewhat different because it is the result from the right (instead of the left) that contains the indicator of a failed STM retrieval. The code that the Synthesiser establishes in this case contains the same kind of reassignment of SUPER- and SUBORDINATE for "did" as for "-ed". It is

applied to the result from the right, however, and no attempt is made to close the outstanding seek on the left. Possibly the attempt should be made but this is unclear.

At this stage the word "did" expects to be followed by "print" in a sentence. The verb "to say" can be introduced and that expectation can later be generalised.

: SAY DOT [DOT]

DOT

: SAY COMMA [COMMA]

COMMA

Saying things is a built-in capacity of the program. Briefly, it always performs perception (and hence interpretation) of its actions before doing them so that the meanings of "dot" and "comma" in square parentheses above are derived before the words are printed. This function is performed by the Output Phase which possesses the twin variables direct and indirect output. Interpretation is carried out on whichever one of these is non-empty, the result going to the other. The direct output is then performed.

In the present situation, the direct output is the procedure to print "dot" while the indirect is that for ".". The meaning of "say" thus differs from that of "print" only in that the IDENTIFIER argument to STM indicates indirect instead of direct output. Verbs such as "write" and "count" are of course more complex and will be dealt with later. Further advantages for the present design of the program's uttering ability will be seen when counting and uttering the plural are discussed.

The expectation associated with "did" can be generalised as follows.

: YOU DID SAY COMMA

Although "say" and "print" are so similar their procedures will not match according to the rules of the generalisation matcher and therefore the expectation of "did" becomes non-specific. It is conceivable, however, that future experiments would be able to associate the basic meaning of "did" (as in "he did the job well") with a class-specific expectation of "a", "the", "some", etc. and so exploit the program's theoretical capability for learning grammatical and syntactic relations.

"To say" is a strong verb. The past tense does not use "-ed" and so must be taught separately.

: SAY ASTERISK

ASTERISK

: YOU SAID ASTERISK

: [DOT]

DOT

: YOU SAID DOT

The program can learn an alternative meaning for "say" in the case where the meaning of the word following it is unknown. The unknown word must be introduced to the program beforehand to avoid confusion.

: SPLODGE

: SAY SPLODGE [SPLODGE]

SPLODGE

The program puts "splodge" in LTM, just as it did in the early stages with "*" etc. The meaning is vacuous. Note that the procedure synthesised by the program for "splodge" does not even contain an STM call; it simply contains instructions to pass on to

perceive the next item in the line. Thus it differs from the process for "a" which does contain an STM call although because generalisation has taken place it does not set any parameters.

The consequence of this difference is that the result received by the "say" process does not come from "splodge" but from the procedure (the End Indicator) invoked when an attempt is made to perceive input beyond the end of the line. This provides an alternative expectation associated with "say" to which the second meaning can be attached. Fig. 7.13 illustrates.

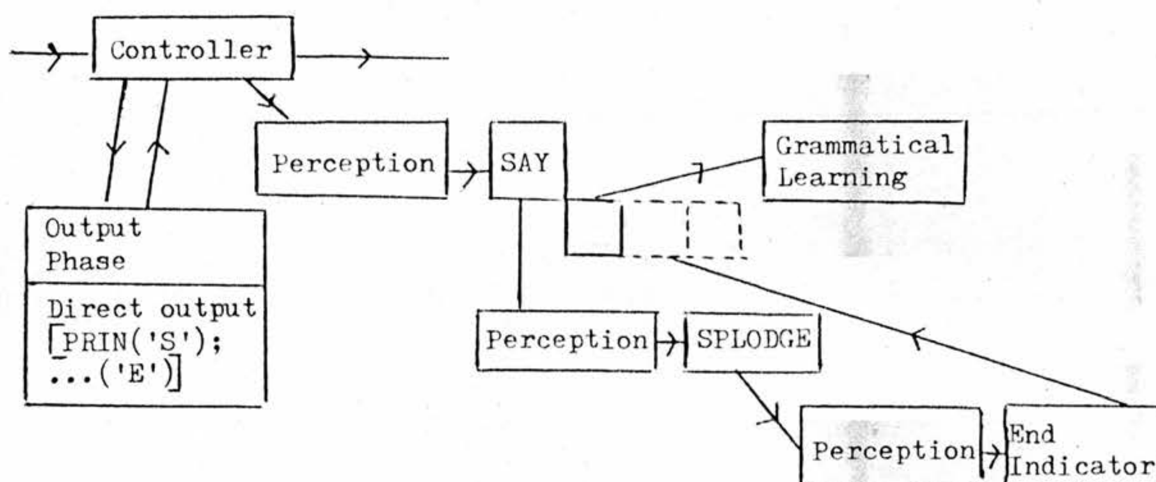


Fig. 7.13 Run-time structure when end of line is reached without a result.

The need for a second meaning is detected when the STM invocation fails in the "say" process. It fails because the indirect output is undefined. The grammatical learning component sets up the new expectation of "say" for the End Indicator. As always, the Synthesiser is called and as it receives only a null result, it works on the STM match to the input text that was located during

perception, as it does on the early noun examples. (Section 6.1 defines the exact conditions.) It is capable of synthesising a procedure that will in future invoke STM with SEARCHPROC equal to the text following "say". The other STM arguments are determined by the details of this match which is to the direct output of the Output Phase in a subordinate process. This is an imperative specification and hence the following will now work.

: WATER

: SAY WATER

WATER

The past tense is quite similar except, of course, for the use of SUBORDINATE instead of SUPERORDINATE which renders the STM call in "said" verificational rather than imperative.

: YOU SAID WATER

7.4 Generalisation over a class

It is convenient at this point to show the third person singular of the present tense of the verb "to be". A rather unnatural sentence has been used for this purpose.

: A DOT IS A DOT

Since it finds no reference to a dot in short term memory the process for "dot" generates a procedure which becomes the left hand result to the new word "is". As usual, the New Entity Handler performs a seek to the right. The second process for "dot" now finds a reference in STM, this being the result just received by the New Entity Handler. Fig. 7.14 shows what happens. So here we see how the meaning of "is" has arisen within the present framework.

The meaning is that the result from the right must match into the process for "is" itself. The only slight difficulty is that at the time this result is obtained in the learning situation, the

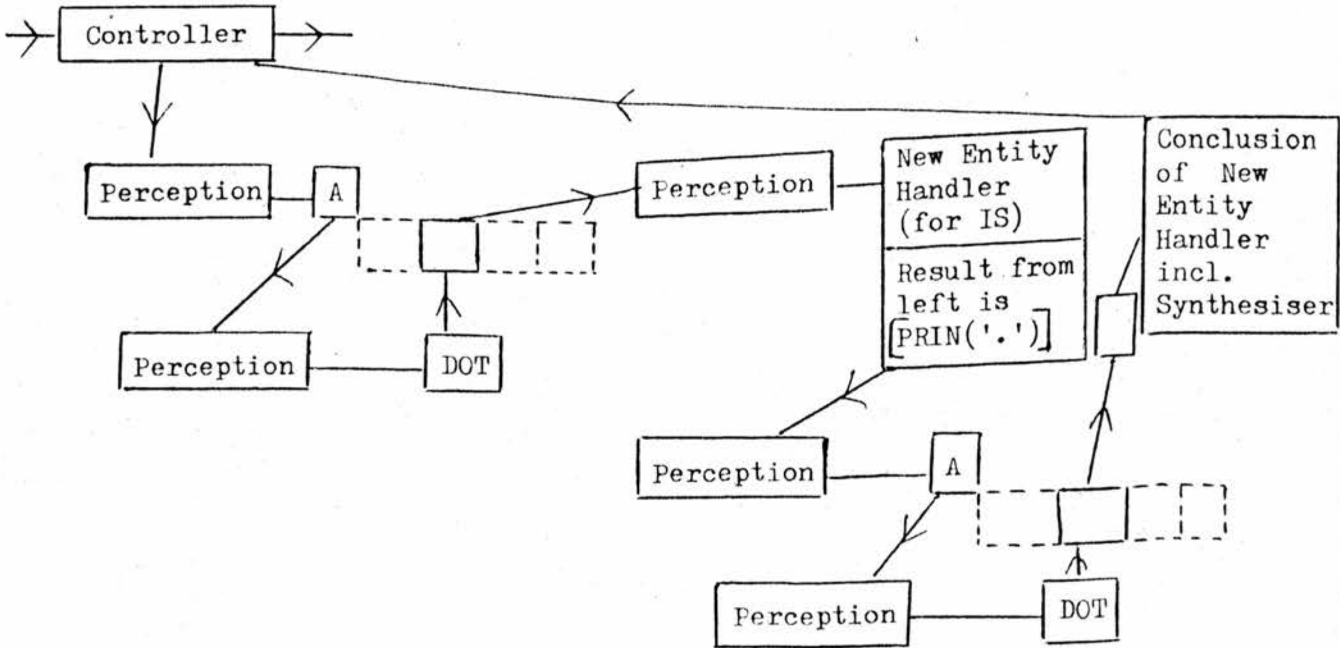


Fig. 7.14 Run-time structure interpreting "a dot is a dot"

procedure for "is" has not yet been synthesised. At this moment the result from the left is held in variables local to the New Entity Handler. This would cause the value of the EXECPROC argument to the STM call within "is" to be set equal to the New Entity Handler instead of being equal to the procedure for "is". However, the Synthesiser detects this condition on EXECPROC and substitutes as its value the meaning procedure for "is" which it is at that moment synthesising.

The use of this word can be demonstrated in connection with the class word "character", which is introduced as follows.

: A COMMA IS A CHARACTER

As has just been explained, "is" seeks to the right and expects a result which matches into itself. Processing for the new word "character" closes the outstanding seek as shown in fig. 7.15, thus re-

ceiving detailed specifications of the STM match within "is". The meaning of "character" is similar to that of "comma" except that it also has the capacity to close an outstanding seek on the left. This capacity does not however preclude its use in situations where there is no outstanding seek. Its syntax then defaults to that of simple nouns.

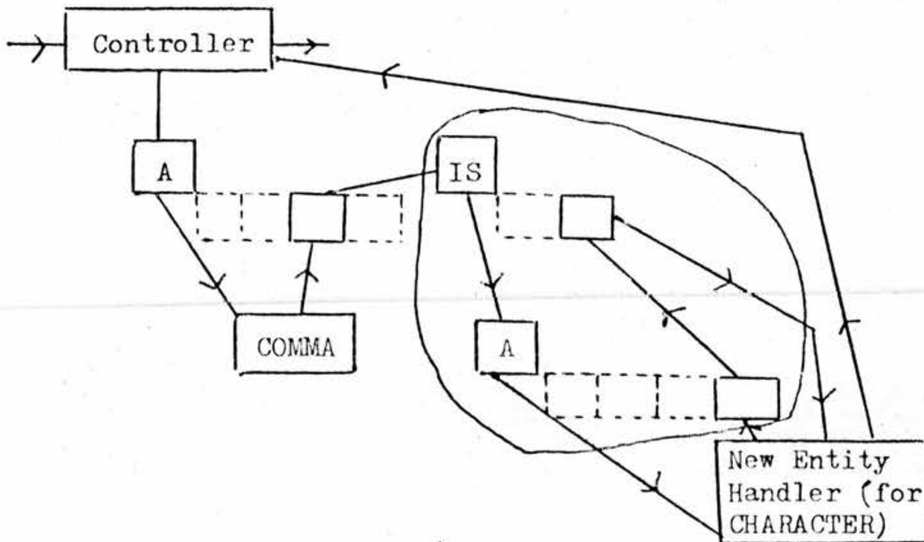


Fig. 7.15 Learning "character": the first example. The references to Perception have been omitted.

The meaning procedure of "character" contributes the SEARCHPROC argument to the STM interface, this being the only parameter left unspecified after the "is" process. The "is" process specifies EXECPROC, IDENTIFIER and SUPERORDINATE parameters that force a match into the result it holds from the left.

The next example will force the program to generalise the meaning of "character".

: AN ASTERISK IS A CHARACTER

When processing for "character" closes the seek on the left it will receive specifications in the STM interface process which oblige it to match into the value of the variable within the execution of "is" that contains the result from the earlier part of the sentence. The value of that variable is a procedure for printing an asterisk and the value of SEARCHPROC given by "character" is a procedure for printing a comma. Consequently, the STM retrieval within processing for "character" will fail. Generalisation is performed using the generalising matcher to obtain the common code from the two printing procedures. This now becomes the value of the SEARCHPROC parameter to STM in the meaning of "character".

The precise value of this parameter is the common portion of the following two procedures, which are the internal representation of PRIN(','); and PRIN('*'); respectively.

```
NOOP 28;  ASSIGN X;  CALL X PRIN;
NOOP 31;  ASSIGN X;  CALL X PRIN;
```

The common portion is

```
ASSIGN X;  CALL X   PRIN;
```

7.5 Sentences with subordinate clauses

Sentences containing the temporal relations and relative clauses are shown. It is necessary to begin with a pronoun.

7.5.1 Personal pronoun

The personal pronoun "I" may be taught. This involves a kind of transformation of STM specifications that gives rise to certain difficulties to be discussed in section 10.1.2. The attributes of the verb need to be converted for the purpose of perception rather than action. That is the function of "I" in the next sentence.

```
:  *
:  I PRINTED AN ASTERISK
```

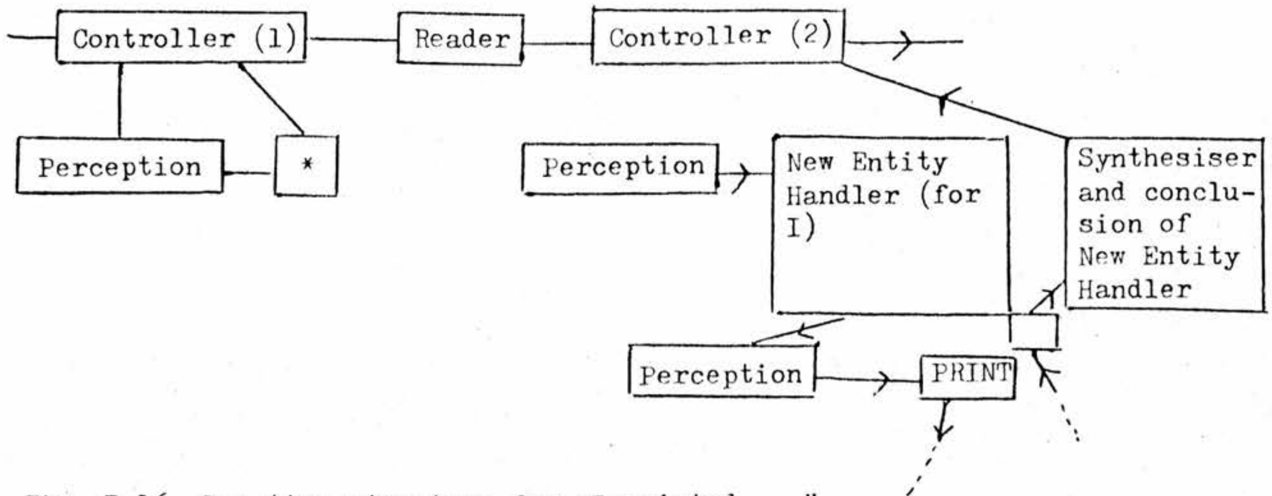


Fig. 7.16 Run-time structure for "I printed ..."

As each word is interpreted, its process contributes specifications to the STM interface process. "Asterisk" supplies the SEARCHPROC, "-ed" gives SUBORDINATE with value the box marked Controller (2) in fig. 7.16, while "print" indicates by means of the attributes EXECPROC and IDENTIFIER that the STM interface is to search for a recent execution of the Output Phase with the value of the SEARCHPROC argument as the direct output. Since such an activity has not recently taken place, the STM interface fails to find a match to these parameters.

This situation is analogous to what happened earlier in the learning of "-ed" and "did". The Synthesiser proceeds by invoking the STM interface function with only SEARCHPROC defined and equal to the value supplied by the process for "asterisk". This is known as relaxing the process-like specifications. Now, of course, a successful match is found in the short term memory, it being a reference to the preceding experience, represented as Controller (1) in fig. 7.16. The STM interface finds the procedure ⌈PRIN('*')⌉ first as input to the Controller.

The Synthesiser now constructs, as the meaning of "I", a procedure that will give the STM interface process the EXECPROC parameter with value the Controller and IDENTIFIER indicating the input line. This means that in a future sentence of this form, the "I" process will override these two specifications given by other word processes in the sentence.

This crude device does not seem to be generally applicable to this kind of problem, involving as it does a loss of information. The point will be taken up in Chapter 10 (section 10.2). An example below (in section 7.5.2.1) illustrates one instance in which something more sophisticated than a simple override takes place.

7.5.2 Temporal relations

An interesting new principle comes into play when one teaches the program the temporal relations "before" and "after".

: PRINT A COMMA

,

: *

: YOU PRINTED A COMMA BEFORE I PRINTED AN ASTERISK

: I PRINTED AN ASTERISK AFTER YOU PRINTED A COMMA

The learning of these two words is very similar. In both cases the New Entity Handler and Synthesiser receive results from both the left and the right.

In the "before" example, the clause to the left of the new word refers to the first activity appearing in fig. 7.17, while the clause to the right refers to the second. These references into short term memory are the results presented to the New Entity Handler in the course of interpreting the sentence. The interpretation process is shown in fig. 7.18. The two clauses, left and right, each present their result.

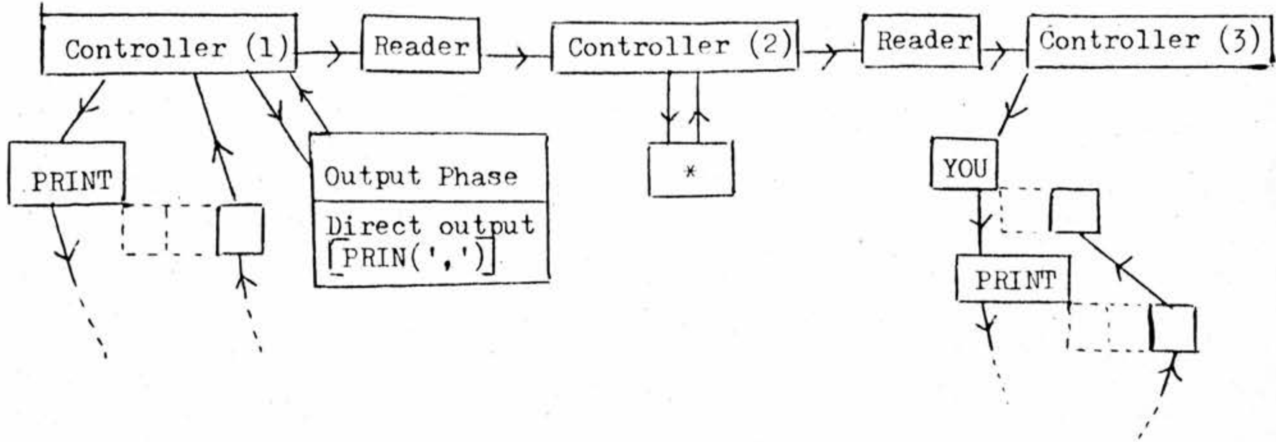


Fig. 7.17 Activities leading up to "You printed a comma before I printed an asterisk"

Controller (3)
YOU
PRINT
ED
A
COMMA
New Entity Handler (for BEFORE)
Synthesiser and conclusion of New Entity Handler
I
PRINT
ED
A
N
ASTERISK

Fig. 7.18 Process of interpreting "You printed a comma before I printed an asterisk".

These results differ with respect to the SEARCHPROC, IDENTIFIER and EXECPROC parameters; they are only comparable with respect to the subordinate relationship. In the case of "before" reference of the result from the right is subordinate to that from the left. For "after" the opposite is true. The meaning of each of these two words, therefore, becomes a procedure that will verify the respective relationship in the future. In accordance with the program's general principles the same procedure will cause actions to be performed when appropriate.

Before illustrating this point, the expectations associated with the words need to be generalised. So the program must see sentences in which they are followed by different words.

: PRINT A DOT

.

: PRINT A SEMICOLON

: ;

: YOU PRINTED A DOT BEFORE A SEMICOLON

: YOU PRINTED A SEMICOLON AFTER A DOT

The omission of the verb in the second clause presents no problem because of the way in which STM invocations are performed by a process which is passed along during interpretation of the sentence. Thus the "print" specifications are carried over from the first clause to the second.

Had these been the first examples instead of the second, the results presented to the New Entity Handler would have had more attributes in common, since both activities now involve printing. Therefore the Synthesiser would have generated meaning procedures for "before" and "after" that verified equal EXECPROC and IDENTIFIER arguments in the two results. Another example would then have been

needed in order to allow Generalisation to remove these over stringent conditions derived from an initial case that happened to be specialised.

7.5.2.1 Application to the near future

In an imperative example the meaning of these words carries over quite naturally.

: PRINT A COMMA AFTER A DOT

.,

: PRINT AN ASTERISK BEFORE A SEMICOLON

*;

In a sense the imperative is the tense of the immediate future. The event referred to in an imperative sentence is usually about to take place. This is already a demonstration of the program's ability to translate past experience into future action. A significantly different example is the following.

: SAY COMMA AFTER I PRINT A COMMA

This sentence sets up an expectation for a comma to be read. If we tell the program to print a comma, it will not satisfy the expectation.

: PRINT A COMMA

,

The correct response can be elicited thus.

: ,

COMMA

The expectation for a comma is established during processing for the pronoun "I". This contains the code that amends the EXECPROC and IDENTIFIER arguments to the STM interface. As mentioned above and explained more fully in section 5.2.3.2, this process will not over-

ride these arguments when SUPERORDINATE has also been specified. Instead, it converts the STM call to (a call of) the LTM interface. This simple and elegant transformation is sufficient to convert a verificational or imperative process into an expectation embodied as a suspended process. When the expectation for three characters is subsequently satisfied, the suspended process is re-activated, completing interpretation of the original sentence. Thus the word "comma" is produced.

7.5.3 Relative clauses

Winograd (1971, pp. 106 and 324) noted the computational similarity between relative and interrogative pronouns. This is reflected in the way the program learns the pronoun "what". The examples rely on the use of the word "is". As a matter of fact the seeks associated with "is" (i.e. the expectations in the frame) were never generalised in the earlier dialogue. The indefinite article always figured in both clauses and within the limitations of this domain some difficulty arises in making a generalisation. However, the definite article may be introduced although without a proper investigation of its true syntactic function.

: PRINT A DOT

.

: YOU PRINTED THE DOT

: PRINT AN ASTERISK

*

: YOU PRINTED THE ASTERISK

: AN ASTERISK IS THE CHARACTER YOU PRINTED

The meaning of "the" is merely to invoke STM without specifying any extra parameters. However, the procedure looks different

from that of "a" because of its different history (i.e. no generalisation of the meaning has taken place) and it serves to generalise the expectation associated with "is".

A sentence containing a relative clause is introduced. (In fact, the last sentence above contained a relative clause with the pronoun understood).

: PRINT A COMMA

,

: A COMMA IS WHAT YOU PRINTED

In synthesising the meaning of "what", the New Entity Handler receives results from both the left and the right. The result from the right refers to the previous event of printing a comma. The result from the left is found within the execution of the process that interprets "is" because of the meaning of "is" as explained earlier. These two results differ in all respects except for equivalence of the procedure referred to (i.e. for printing a comma). Hence the meaning of "what" is to verify that there is a match between the SEARCHPROC arguments of the results from the left and from the right. This is analogous to the meanings of "before" and "after", but here it is SEARCHPROC that is related.

Another example will generalise the expectation associated with "what".

: A CHARACTER IS WHAT AN ASTERISK IS

7.5.4 Interrogative

The interrogative use of the word demands further learning. The following will illustrate the point.

: .

: WHAT DID I PRINT [DOT]

DOT

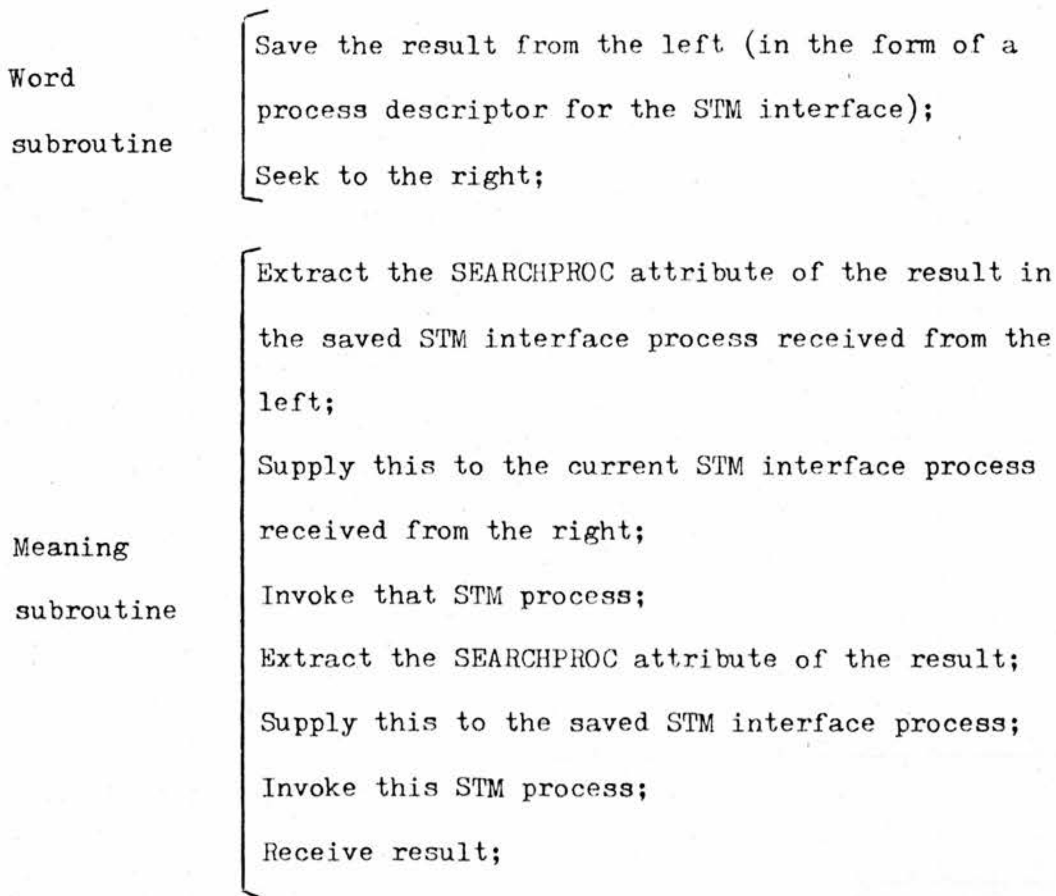


Fig. 7.19 Main points in the procedure for "what"

Although there are no words to the left, processing for "what" will still perform two STM invocations as if there were results coming from a relative clause and a main clause. Fig. 7.19 shows the procedure for "what". The subroutines can be related to earlier flow of control diagrams, e.g. in fig. 7.18 the word subroutine for "print" has its execution shown by a box marked PRINT. The expectations shown attached to the box possess meaning subroutines, one of which is executed.

Fig. 7.19 shows how the relative pronoun "what" saves the result of the main clause to its left before causing interpretation of the relative clause on its right. It then passes the SEARCHPROC attribute of each result in turn to the other, thereby affording the

maximum flexibility in information flow to cater for various situations of partial knowledge.

In the present example, there is no clause to the left of "what" and so no information is present in the saved STM interface process from the left. The clause on the right gives rise to the specification of enough STM parameters to locate the action of the preceding line, referring to the input line to the Controller in a process superordinate to the present. The result of this STM invocation is a reference from which the SEARCHPROC attribute (the actual content of the input line) is extracted.

When supplied to the virgin saved STM interface process, it becomes its only parameter. When invoked the interface therefore locates the most recent activity involving PRIN('.') which is in the Output Phase, as seen in fig. 7.20.

Now the attributes of this result from STM involve indirect output in the Output Phase subordinate to the current execution of the Controller. None of these has been specified to the interface and they must now become part of the meaning of "what" in its interrogative role.

Making a meaning more specific to a particular case is the purpose of Differentiation. It extends the basic procedure for "what" by synthesising a new procedure which is executed conditionally. The condition corresponds to the interrogative use of "what". The new procedure contains instructions to supply the extra parameters to the STM interface. These parameters are derived from the unspecified attributes of the result. As they specify indirect output in the Output Phase, they will lead the program to make a saying action in response to the interrogative "what". The principle of building a procedure based on the unspecified attributes of a result from STM

retrieval is the same in this instance as in all others and the same Synthesiser does the construction.

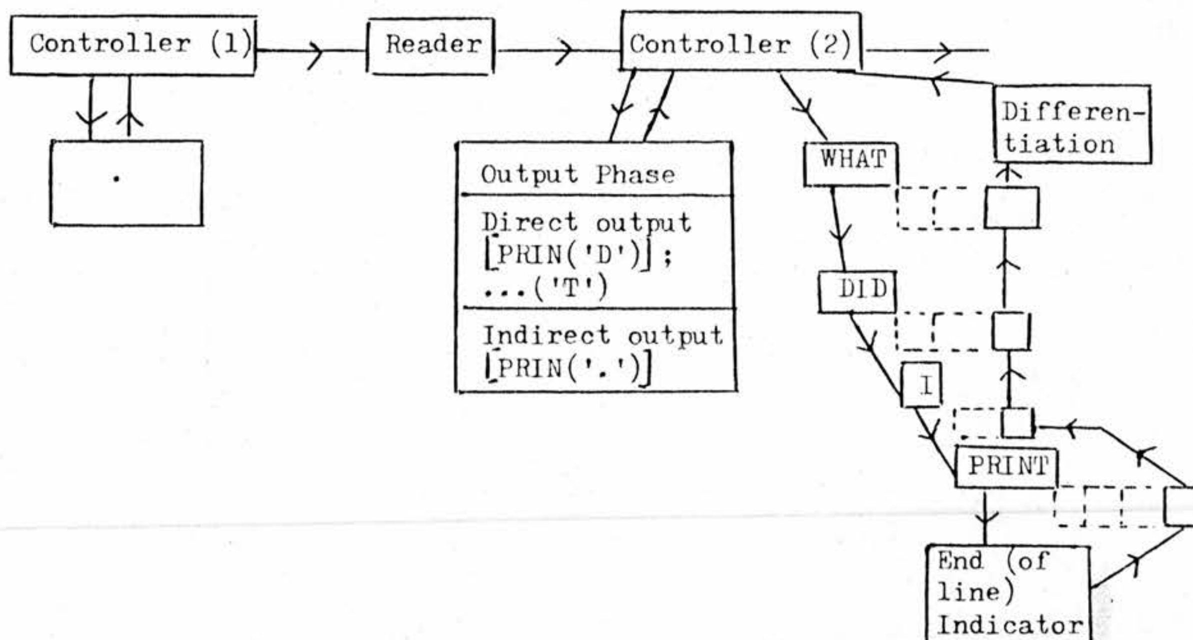


Fig. 7.20 Run-time structure for "What did I print"

The end product is a new record in LTM which possesses this new procedure. The key of this record indicates that it is to be activated when the value of a variable in the "what" procedure is a procedure as opposed to a process reference. The variable is the one that holds the result of the second STM invocation of fig. 7.19. Its characteristics are sufficiently different in the two cases (relative and interrogative) to allow this distinction to be made. Other examples of differentiation, such as the plural morpheme "-s", also depend on the characteristics of the result variable in a similar way. The new procedure, therefore, will only be invoked when "what" is in its interrogative rather than its relative role.

Here are some instances of its use.

: PRINT AN ASTERISK

*

: PRINT A COMMA

,

: WHAT DID YOU PRINT BEFORE A COMMA

ASTERISK

: WHAT DID YOU SAY

ASTERISK

Fig. 7.21 shows where the extension fits in.

7.6. Number and the plural

The plural was introduced earlier and will now be completed. First the seek to the left is generalised.

: ..

: DOTS

The process for interpreting the morpheme "-s" receives a result from the word on its left. The result comes from the STM call made by "dot" which locates the dot procedure embedded in the repetition procedure created by the program on reading. Perception has converted the input line PRIN('.');PRIN(','); to the following form, which is then found by the STM search.

```
 ↓→ X;  CALL X;  CALL X;
PRIN ('.');
```

The STM result is procedural, indicating a partial match of the SEARCHPROC argument PRIN('.'); into the above procedure. The meaning of "-s" therefore contains instructions that will insert an object procedure (in this case PRIN ('.'); into the basic repetition procedure, viz.

$\overset{\rightarrow}{\downarrow}$ X; CALL X; CALL X;

"Empty"

These instructions were generated by the Synthesiser according to the basic constructive principle by which learning takes place. The procedure for "-s" actually contains the repetition procedure in the above form and inserts the SEARCHPROC argument of the received

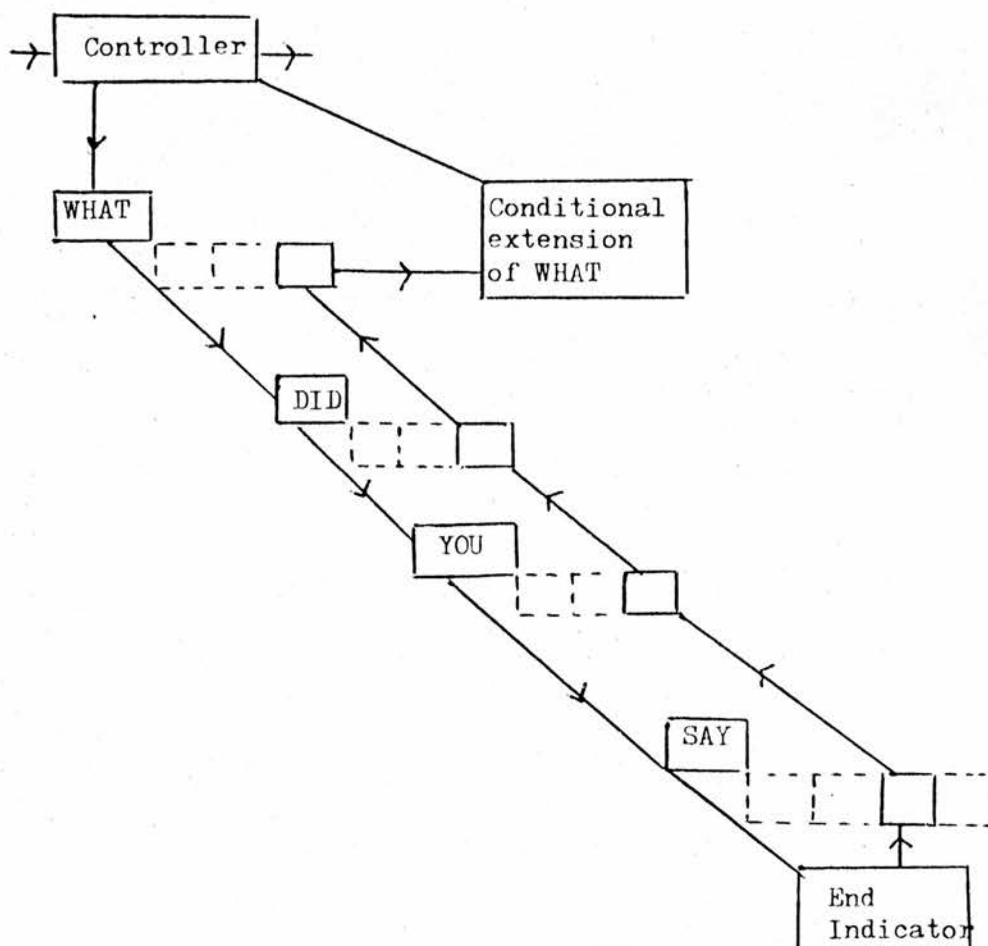


Fig. 7.21 Flow in "What did you say"

result (in this case PRIN('.);) into the empty place. (The repetition procedure is first copied to avoid corrupting it.) The composite procedure becomes the new SEARCHPROC to the next call of the STM interface, giving rise to an exact match with that created by the program on reading. At the same time, the seek to the left is generalised to accept any of the nouns known to the program, since they are all similar in form, differing only by the subroutine they contain for the actual characters.

As a matter of detail, differentiation takes place at this time because of the program's principle of "explaining" or accounting for all the attributes of a result where possible. The result found by the STM invocation within "-s" is process-like (because of an exact match) and refers to the event of the preceding line. So Differentiation sets up an LTM process that will be activated whenever the procedure for "-s" is performed. (The value of SEARCHPROC is undefined, making this an unconditional extension to the meaning of "-s"). This extension procedure will specify to the STM interface that objects referred to by "-s" must be input to the Controller in a previous line, because this happens to be so in this case.

Another example will generalise these specifications away.

: ASTERISKS [**]

**

By a rule of differentiation the program will not synthesise a similar extension again. For the purpose of the following discussion the procedures so far described constitute what shall be termed the original meaning of "-s".

The advantage of the constructive mode of working in the meanings of words and morphemes is that it works for the imperative just as well as in verification. Hence the following works successfully

without any event to match into.

: PRINT COMMAS

,,

The meaning of "-s" is extended by increasing the number of objects in the examples.

: ,,,

: COMMAS

: ****

: ASTERISKS

Differentiation takes place because the STM call in "-s" matches a procedure for duplicating a comma into a similar one for printing it three times. The two procedures are these.

```
    ↓   →X;  CALL X;  CALL X;  CALL X;
 PRIN(',');

    ↓   →X;  CALL X; CALL X;
 PRIN(',');
```

The first is created by the program from the input line by the work of Perception; the second is generated by the meaning procedure for "-s" as the SEARCHPROC argument to STM retrieval. The result of the retrieval is of course a partial match with the remaining instruction CALL X; indicated.

From this result, the Synthesiser composes an extension procedure for "-s" which will join the extra instruction on to the SEARCHPROC argument of the STM interface prior to invoking it. Differentiation ensures that the extension procedure is set up as part of a record in LTM with a key indicating that it will be activated when the original meaning procedure of "-s" gets a procedural result indicating a partial match in STM, that being caused by a match into a repetition

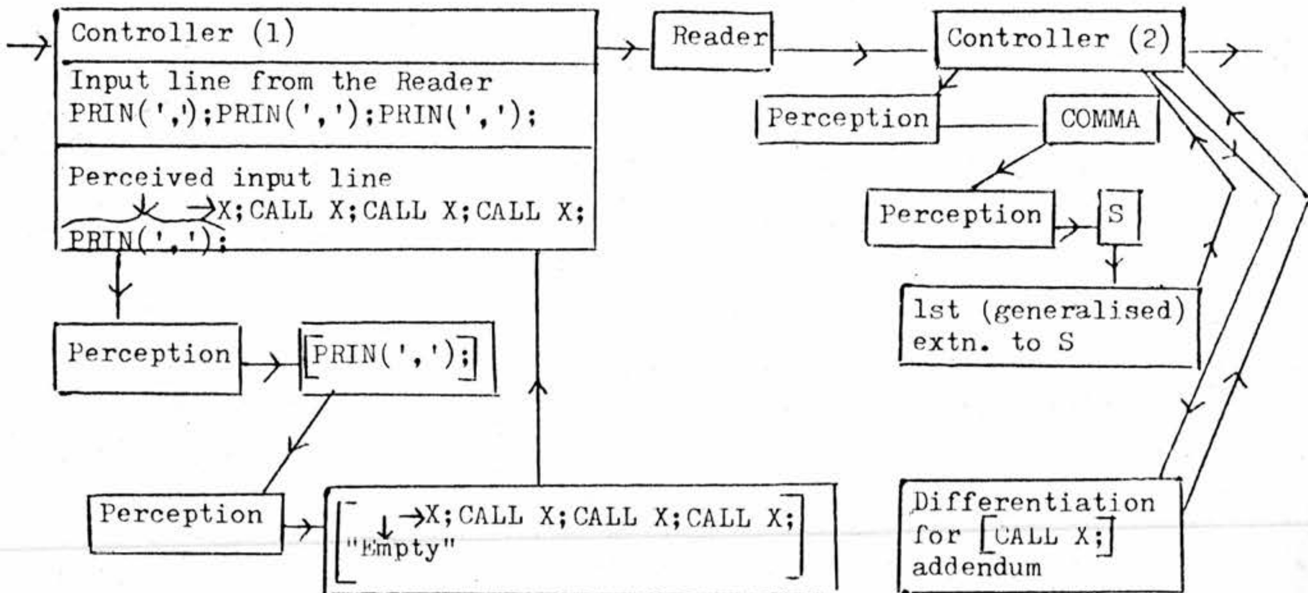


Fig. 7.22 Differentiation extending the meaning of "-s" from two to two or three

procedure greater than two calls in length.

Fig. 7.22 shows the flow of control. Here Perception has been shown explicitly because of the important role it plays in converting the input into its representation as a repetition procedure. The STM searches instigated by the "comma" and "-s" processes locate the perceived input line in preference to that received from the Reader. (Appropriate ordering of variable declarations in the Controller ensures this.)

The same happens again when four objects are encountered but this time Differentiation sets up an LTM record which happens to have a rather unusual characteristic: the procedure within it matches the key itself. This makes it recursive. Referring to fig. 7.23, the

reason is that the procedure that performs the first CALL X; addendum is exactly like that synthesised during Differentiation for the second addendum because the functions are the same. Differentiation now creates the second as a conditional extension to the first but the key of the LTM record so created matches either of them.

The following causes no further processes to be stored in LTM - that is, no further learning takes place. Fig. 7.24 illustrates.

: .....

: DOTS

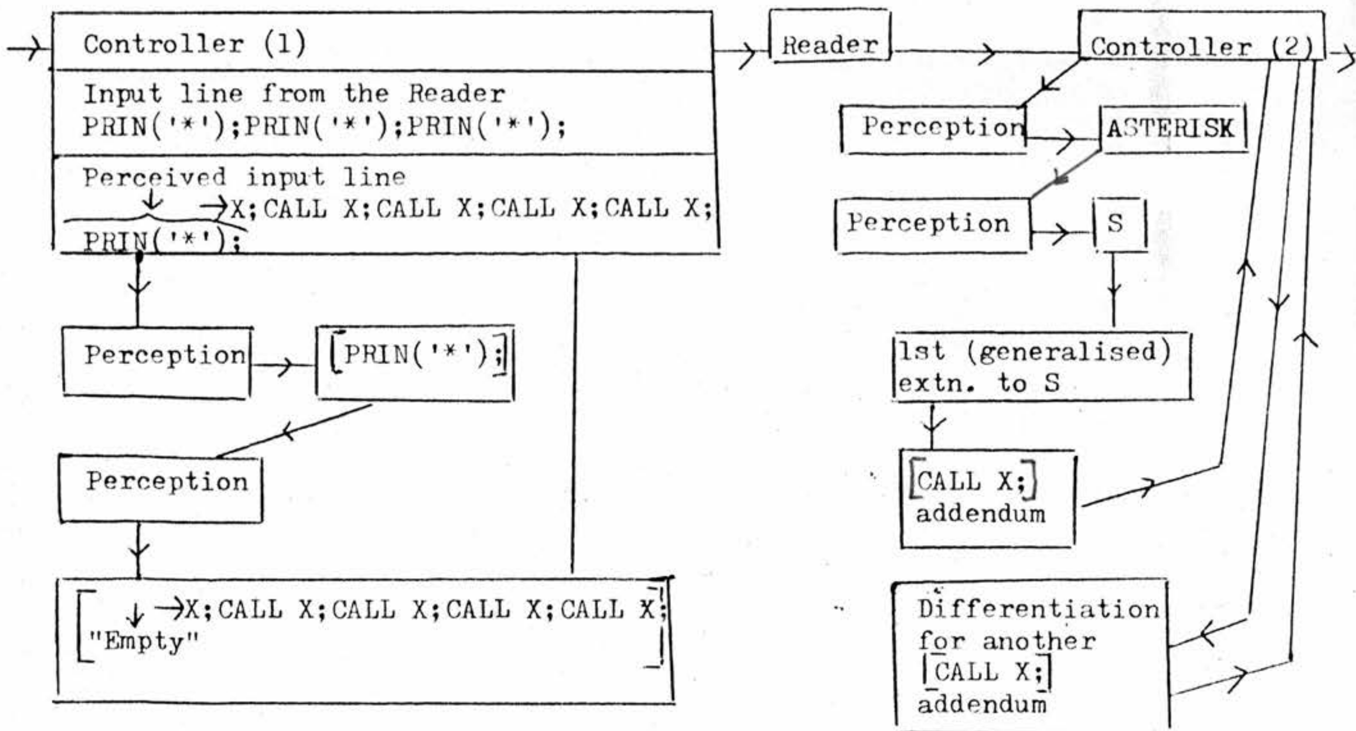


Fig. 7.23 Differentiation extending the meaning of "-s" from two or three to arbitrarily many.

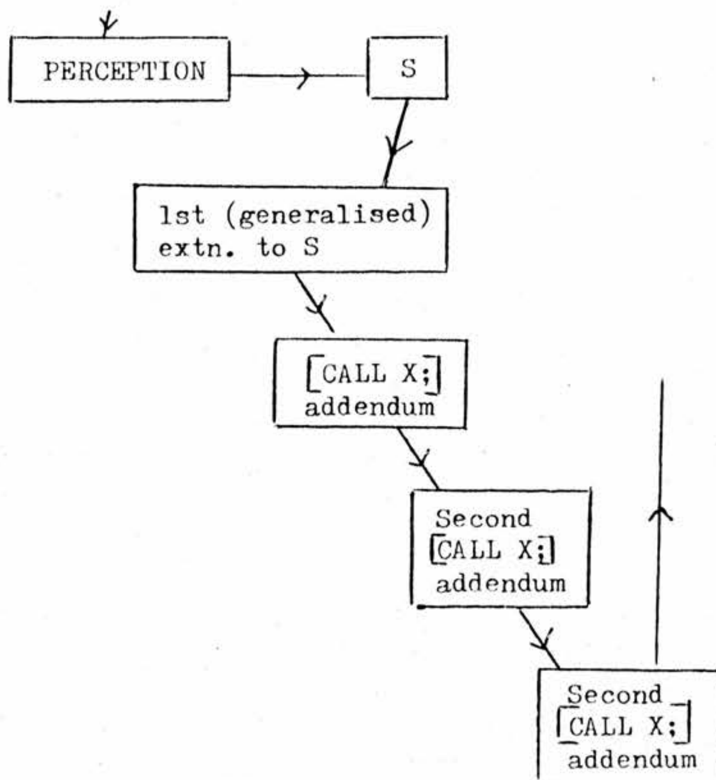


Fig. 7.24 Processing for "-s" when there are five objects

7.6.1 Other sequences

The above sequence for teaching the plural is optimal. The program's design is sufficiently robust to cope with examples in any order. If, for instance, after a case of two objects "-s" was next encountered in connection with a group of five, the result of differentiation would be code to join the following procedure to SEARCHPROC.

CALL X; CALL X; CALL X;

A subsequent example of three objects would cause generalisation to reduce this procedure down to CALL X; Another example of five objects would now give rise, by means of differentiation, to code to join the following procedure.

CALL X; CALL X;

An instance of four objects would cause this to be generalised down

to CALL X; This is now equivalent to the code produced in the optimal sequence and is recursive.

7.7 Number

The numbers can be taught in the obvious way.

: **

: TWO ASTERISKS

Even though "-s" already accounts for the duplication detected in the first line, "two" will account for it again. This is because the New Entity Handler will perform a seek when trying to ascertain the meaning of "two". The process for "asterisk" will answer the seek (merely by providing a result from STM) before "-s" has a chance to operate. This is simply a consequence of the way expectations work. Thus the New Entity Handler receives a procedural result (partial match) and the meaning of "two" is very similar to the original meaning of "-s" before it was extended. In the case of "two", however, it is associated with an expectation of a result from the right. That must be generalised.

: ..

: TWO DOTS

Note that "-s" still receives a result from the noun because of the specific seek to the left that it contains. If it received the result after "two" had operated on it, it would erroneously perform a kind of multiplication - effectively two times two. The program does not do that because of the way its grammatical component works.

Some other numbers can be taught in the same way.

: ***

: THREE ASTERISKS

: ,,,

: THREE COMMAS

and so on.

The program is now clever enough to obey a command mentioned in Chapter 1. It amounts to teaching it a new word "triple" by means of explicit instruction rather than by implication. First the new word must be shown to the program so that it can recognise it in the next sentence without undue complication.

: TRIPLE

: SAY TRIPLE AFTER I PRINT THREE CHARACTERS

This works in much the same way as the earlier and somewhat simpler example "Say comma after I print a comma". The expectation established this time is for three similar characters but not any particular character. If only two are typed in, the program will not respond, but will wait for the correct number.

: **

: ...

TRIPLE

One detail should be mentioned. "After" is one of a class of words having a process that expects to receive a result both from the words on its left and to its right in a sentence. Other such words are "before", "is", "what", etc. They mark syntactic boundaries in a sentence, usually the end of a clause as in this case. Word processes of this kind actually perform the function of delimiting a clause by closing any outstanding seeks (with associated expectations) there may be in the word processes on the left, thus finishing the interpretation of that clause. As illustrated in fig. 7.25, the End Indicator is used to mark the end of a clause in this way. Earlier diagrams (e.g. fig. 7.12) also showed this happening but omitted the detail concerning use of the End Indicator.

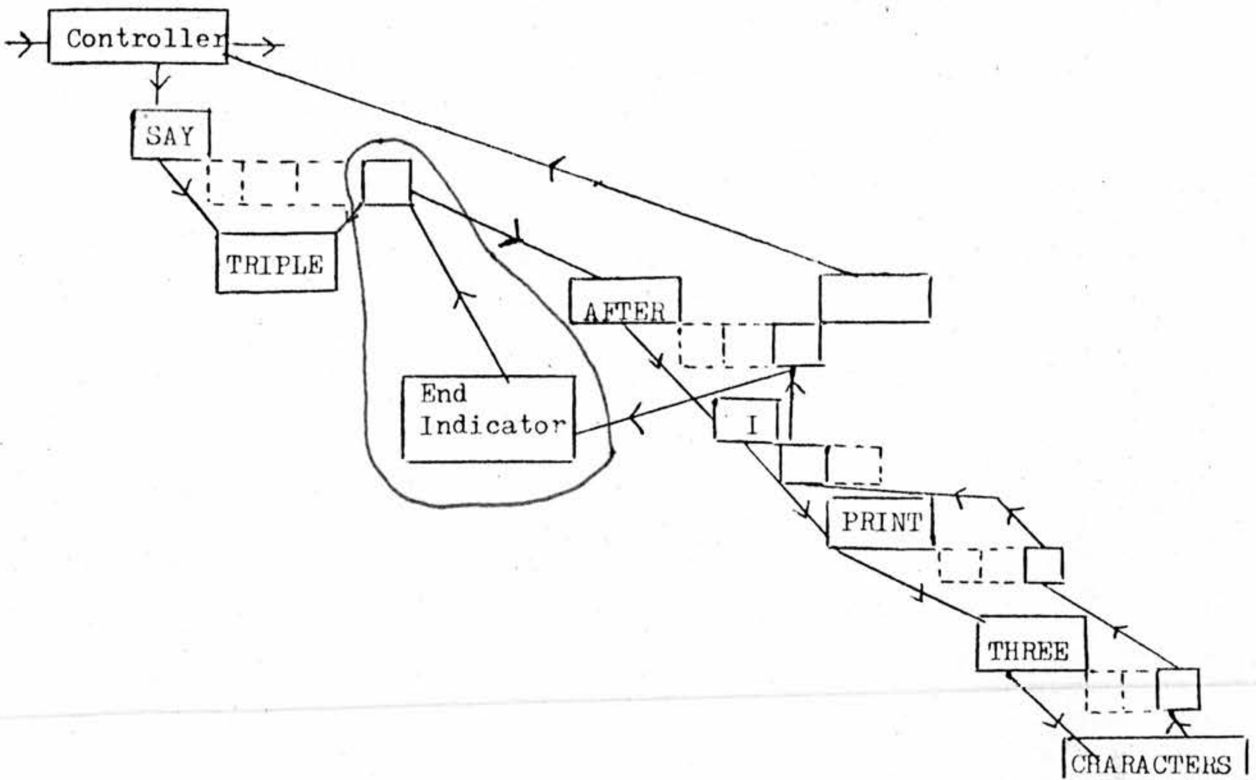


Fig. 7.25 Marking end of clause with the End Indicator closing an outstanding seek.

There is a potential ambiguity in a sentence including "say" for the object of the verb could be the whole of the rest of the sentence. The end of a clause is marked by the End Indicator which is called just as at the end of a sentence. One of the meanings of "say" is associated with an expectation for the End Indicator (as at the end of a sentence) and the theoretical ambiguity does not influence the program's behaviour. This is claimed here to be the most natural way to treat ambiguity. That is to proceed directly to one meaning without even noticing the other possibilities.

A complication arose in testing this sentence. It was that the STM invocation in the process for "character" would locate any

character including letters of the sentence. This meant that an unintended interpretation was being made by the program to await three occurrences of the letter S, that being the character in the sentence that it happened to match. Basically this problem arises because of the extreme simplicity of the domain in which the program operates. Special provision had to be made to prevent this from happening. An _ad hoc_ rule was set up to cope with the problem.

The conjunctive particle will be useful in asking the program questions that effectively involve addition. "And" is introduced here.

: .,

: A DOT AND A COMMA

: **.

: TWO ASTERISKS AND A DOT

: .**

: A DOT AND TWO ASTERISKS

The second and third examples serve merely to generalise the seeks within "and". The initial learning is performed in the first example above. It is similar to "is". When synthesising the meaning of "and", the New Entity Handler receives a procedural result (partial match) from the left. Fig. 7.26 shows this partial match result as received. (Chapter 4 defines the exact form of such a result. The diagram only shows one part of it.) The STM retrieval performed by the "comma" process locates this stored value because it is more recent than the record of the original input line. The only difference between this case and the situation when "is" was learned is the name of the variable (IDENTIFIER) containing the STM match found by the "comma" process. The meaning procedure synthesised for "and"

is therefore similar to that for "is", differing only in that respect.

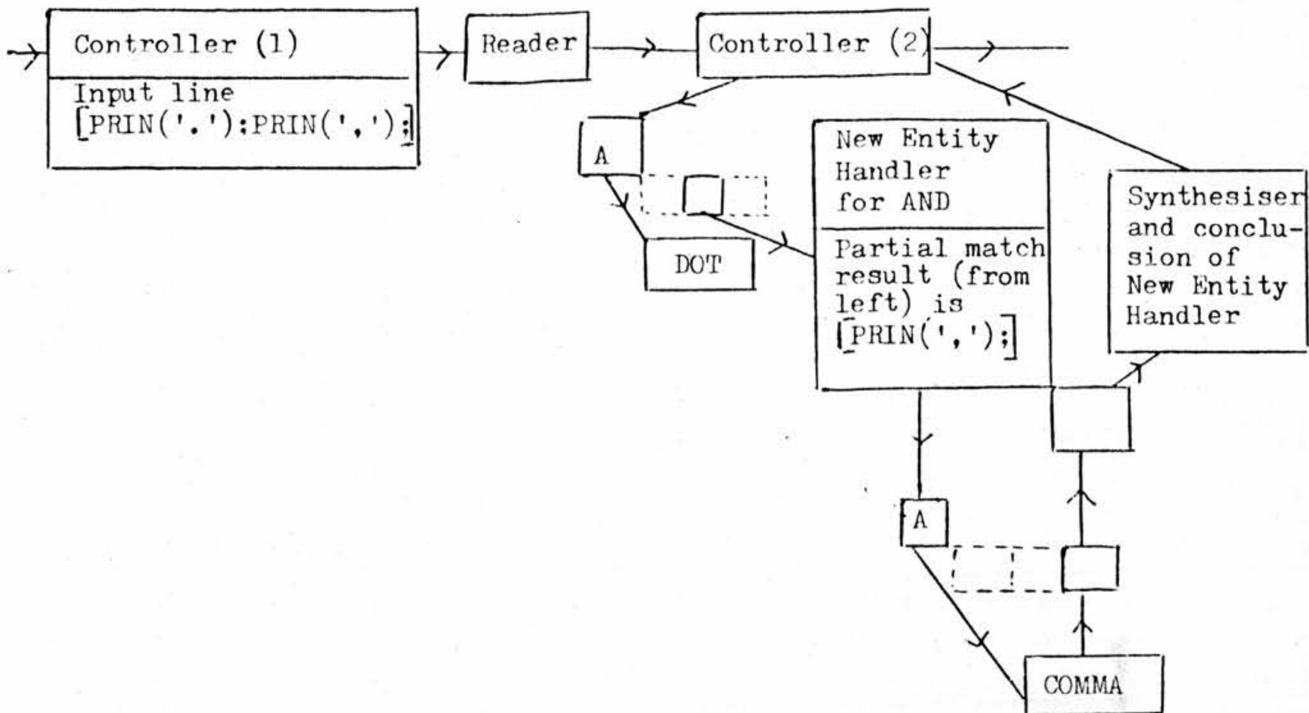


Fig. 7.26 Learning "and". The partial match result is indicated.

The following will now work.

: PRINT TWO ASTERISKS AND THREE COMMAS

**,,,

7.7.1 Generating a reply involving number

If the human tutor enters a certain number of characters, the program can use what it has just learnt to answer a question.

: *****

: WHAT DID I PRINT

FIVEASTERISKS

The acquired process for "what" initiates the Output Phase which is responsible for making utterances. The complementarity rules (defined in section 5.2.2.2) enable the program to produce the above reply by interpreting the line ***** as if it were a sentence.

Perception represents ***** in the input line thus.

```
   ↓→X;  CALL X;  CALL X;  CALL X;  CALL X;  CALL X;
 PRIN ('*');
```

In this form the "what" process assigns it as indirect output to the Output Phase. The rules of LTM matching in interpretation cause this procedure to be recognised naturally as "five asterisk". At this point the Output Phase managed to assemble a procedure for printing these words without the "-s". The procedure is set up as direct output.

The program now goes through a second cycle of re-interpreting the utterance it has just prepared prior to performing it. This is always done before producing output, as mentioned in section 5.1.6. In the simpler cases no effect is visible but now an LTM process for "-s" is fired and the grammatically correct plural is appended before the words are printed. This LTM process is the complement of the seek to the left made by "-s" during interpretation of a sentence; that uses STM and a standard transformation applies. The LTM process is conditional upon the presence of the duplication procedure

```
 ↓→X;  CALL X;  CALL X;
"Empty"
```

in a named variable which occurs within the process for the word "asterisk". This same LTM process would be fired during a normal interpretation of a sentence but, because of the redefinition of local variables within the Output Phase, the utterance behaviour is only exhibited here; normally it is neutralised.

The fact that appending the "-s" involves an extra cycle in the program may explain why young children often omit it in connection with numbers (Anisfield and Tucker, 1968, p. 216).

Semantically, of course, it is redundant. Any relationship between this extra cycle and the transformations of Chomsky's theory (Chomsky, 1965) are not obvious. More examples need to be considered before any useful conclusions could be drawn.

7.7.2. Numerals

Now the numerals are introduced. To begin with they are just like the numbers. 1 is like the indefinite article. There is no equivalent to the numerals in speech, of course, and it might not be surprising if the same principles did not apply to learning them. In fact this point is not proven since the later parts of the dialogue are not tested. They appear in Chapter 8.

Here are the numerals.

: *

: 1 ASTERISK

: ,

: 1 COMMA

: 1 DOT [·]

.

: **

: 2 ASTERISKS

: ..

: 2 DOTS

3 is similar.

Bearing in mind the way a child might be taught numbers in school by the use of bricks or counters, the program is taught to associate the numerals with a given object (the currency sign £) when no object appears in the phrase or sentence.

: £1

: ££

: 2

: £££

: 3

: ££££

: 4

These meanings are attached to different expectations in the frames belonging to the LTM processes for the numerals. They are, in fact, expectations for the End Indicator, the procedure that is invoked at the end of a line or clause. The meanings still contain the basic duplication or triplication instructions as for the regular case but the procedure to print a £ is already embedded. Thus for 2 we have

```
    ──→ X;  CALL X;  CALL X;
   ↓
/‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾\
 PRIN ('£');
```

whereas in the regular meaning of 2 (as in "2 dots") the subordinate procedure would be empty (i.e. having no instructions) and ready to be filled in.

For convenience, numbers to the base five were used. Clearly, no principle is involved in this expedient. Two digit numbers will be discussed in Chapter 8 in the context of counting.

7.7.3. Learning about the blackboard

Before dealing with that topic the program must be given something to count. It is provided with a "blackboard" - a programmed device into which it can write by printing characters preceded by a quotation mark.

: ["*]

*

For readability, the content of the blackboard is shown indented on the page. To the program, however, it appears just like the direct feedback it receives from its printing actions. As a consequence of the above example, it learns that a quotation mark will cause feedback of the characters that follow it. The causality component comes into play.

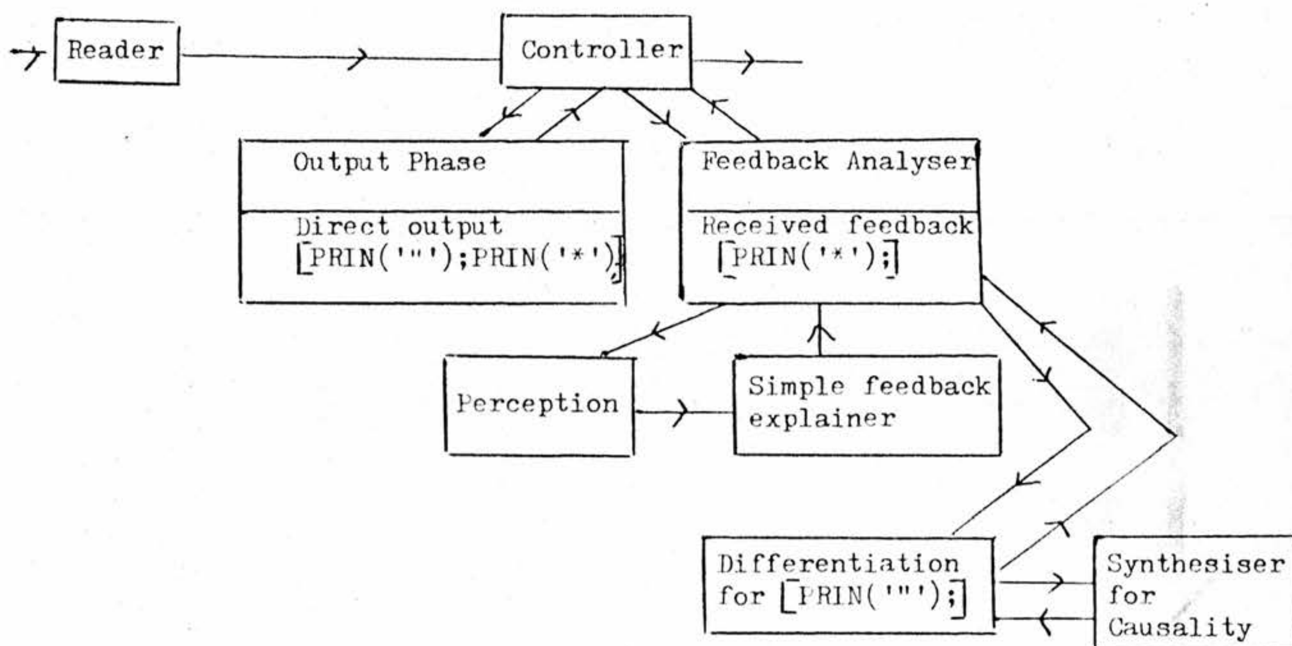


Fig. 7.27 Learning about feedback from the blackboard

The causality component of the program is based on the analysis of feedback from actions. In fig. 7.27, the new learning is performed by Differentiation as a consequence of the unexpected quotation mark discovered by the simple feedback explainer. This explainer is a procedure synthesised by the causality component when it first processed feedback from the Output Phase. It searches STM for an execution of the Output Phase which has direct output matching the

feedback (and which is subordinate to the current execution of the Controller). As can be seen in fig. 7.27, the direct output will partially match the feedback and Differentiation is called to take care of the quotation mark. This it does by creating a conditional extension to the simple explainer in the form of a new LTM record which possesses a procedure composed by the Synthesiser for Causality. (The program has similar, but not identical, synthesisers for meaning and causality.)

The new procedure will invoke the STM interface process, joining PRIN('"'); on to the SEARCHPROC argument already defined by the simple explainer. The LTM record containing this procedure bears a key indicating that it is to be invoked when PRIN('"'); occurs as the value of a named variable in the execution of the simple explainer.

Chapter 6 contained a discussion of the implications of these ideas, particularly the program's ability to convert an explainer into a predictor, a means of anticipating the consequence of an action as opposed to accounting for it after the event.

The object will remain on the blackboard until deliberately erased. Two strokes (//) will wipe it clean. All of the blackboard operations can be performed either by the program (by printing the control characters) or by the human tutor (by typing them in). The following will show an object being retained.

```
: //
: ["*]

          *

:

          *

: //
```

Note that the effect of // is not apparent to the program since it is only just learning about retention.

When a blank line is entered, as above, the content of the blackboard is not altered and so some unexpected feedback is encountered. Thus, the program learns that feedback predicted because of a quotation mark causes similar feedback again. The chain of causal prediction now established begins with a process in LTM waiting for a quotation mark. When activated, it sets up an expectation for the appropriate feedback. When that is found, the associated suspended process is activated.

Now in the next line unexpected feedback occurs. Perception, used by Causality, locates a match in the previous feedback and, to be precise, it finds it in a variable of the re-activated and satisfied expectation there. It therefore establishes a further predictor which will be triggered by the execution procedure of that process.

7.7.3.1 Erasure

In addition to complete erasure, selective erasure is possible. A single stroke followed by one or more other characters removes them from the blackboard.

: [".]

.

:

.

: /.

In these training sequences, the simplest path was adhered to in testing, as illustrated in the above few lines. It is part of the program's design to be able to cope with less favourable sequences, relying on a larger number of examples and using generalisa-

tion to correct inappropriate assumptions. However, the number of variations is large and it has not been practicable to test them.

In the above sample, the program learns simply that the character following the stroke should match into the feedback from the previous line. This knowledge is associated with "/" just as the meaning of a word could be. Indeed this is similar to one case of "said", although certain details differ. The match to "." is located in the feedback, since the action that wrote to the blackboard is further removed in the past.

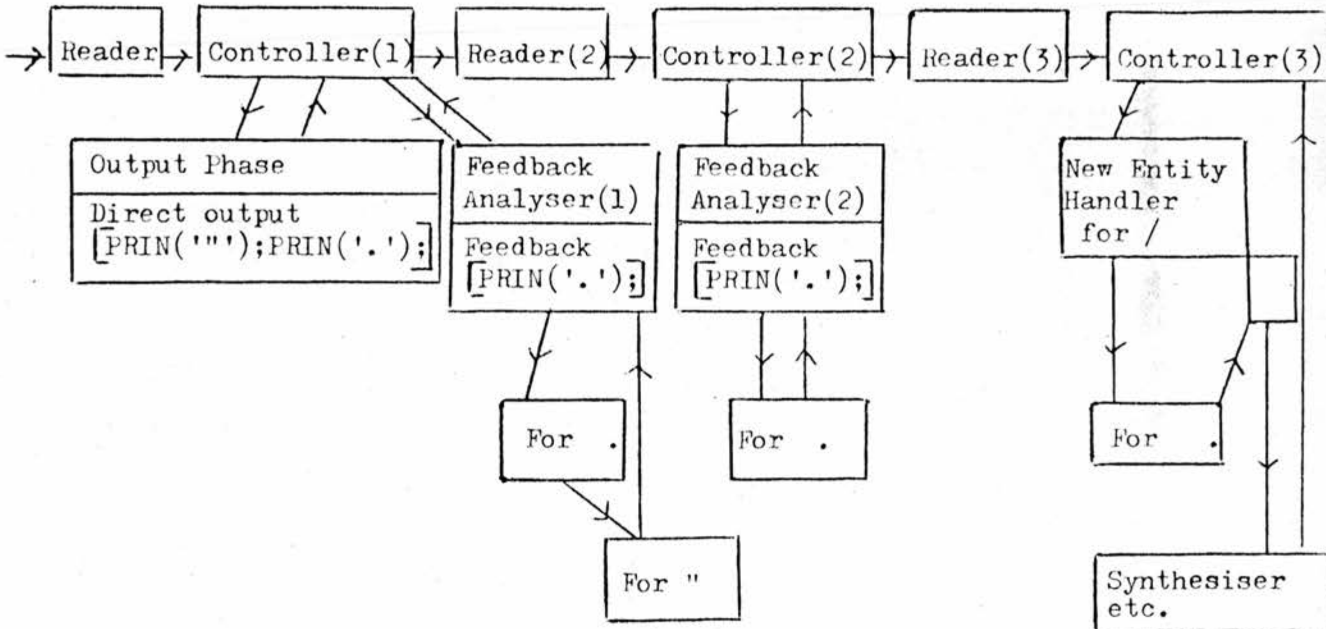


Fig. 7.28 Learning about blackboard erasure

Referring to fig. 7.28, the match located to the dot by Perception and the New Entity Handler is found in the feedback analysis marked (2) which represents the program's experience of the blank line of input while the dot is still on the blackboard.

A brief illustration of robustness can be given here. Had the human tutor not inserted the intervening blank line, the match would have been into the writing action (i.e. a partial match into the direct output of the Output Phase) with the quotation mark as the unmatched portion of the result. This is true even though the box feedback analyser (1) is more recent in fig. 7.28. It is a consequence of the order in which STM is searched, depending as it does on the order in which variables are declared, particularly in the Controller. The sequence would have been as follows.

: [".]

.

: /.

All, however, is not lost. A further, more elaborate, example will correct the program's inappropriate assumption.

: [".,]

.,

: /,

.

: /.

The first stroke does not find its object (a comma) next to a quotation mark because of the presence of a dot. Hence the corresponding instructions within the meaning of the stroke (/) are generalised away. The second stroke finds its object (a dot) within the most recent feedback and so Differentiation sets up the attributes as an extension

to the meaning of the stroke. Once established, that STM specification works in both cases mentioned in connection with fig. 7.28, since the feedback in the shorter example (i.e. no intervening line) still matches, even though it was not the first to be located now because it has acquired the necessary information in the form of initial learning. It matches EXECPROC and IDENTIFIER arguments to the STM interface that explicitly identify that the result must be found as feedback.

7.7.3.2 Further work

There is considerable scope for further experimenting with the causality component of the program. In the next chapter, some well developed ideas will be presented on how the program could observe and predict the effects of combining and selectively erasing various numbers of objects. A particular deficiency that needs investigation is the program's failure to take note of unfulfilled expectations. This is not a problem when some other feedback is encountered: the complementary process (STM - LTM reversal) copes then, as explained in section 6.2.2. When there is no feedback at all, however, the method breaks down. This may be a limitation of the domain rather than of principle; a child would see an empty blackboard after erasure whereas the program sees nothing at all.

7.7.4 Counting

Let us move on to counting. Testing with the blackboard was taken far enough only for this application. The human tutor demonstrates in incremental examples. Once again, only the easiest sequence was actually tested and is presented here.

We are concerned with enumerating objects on the blackboard - not merely with uttering a set of names. The interest lies particularly in the program's ability to convert the relationships it

perceives into actions to be performed and to link these in an appropriate sequence with conditional dependencies. It shows that the general principles of short and long term memory and the synthesis of memory references are adequate to the task.

The program is taught to count 1 by being presented with one character on the blackboard which the human tutor erases, while entering 1.

: ["*]

*

: 1/*

This causes the frame associated with 1 to take on an explicit expectation for a following stroke (/). There is no semantic content. In fact, counting is found here to be essentially syntactic. As will shortly be explained, the relationships between the numbers are established by the grammatical component.

In the next example, the numeral 2 receives a result from the left.

: ["**]

**

: 1/*2/*

It comes from the process for 1, which has simply passed on the result it received from the stroke and asterisk that follow it. In its turn, 2 receives a result from the following stroke as can be seen in fig. 7.29 and its frame is augmented by an expectation similar to that recently acquired by 1. It differs only in that, associated with this new expectation, 2 also has an explicit seek to the left for the 1 process that has just yielded a result to it.

The extra call to the Reader under the box marked Controller(2)

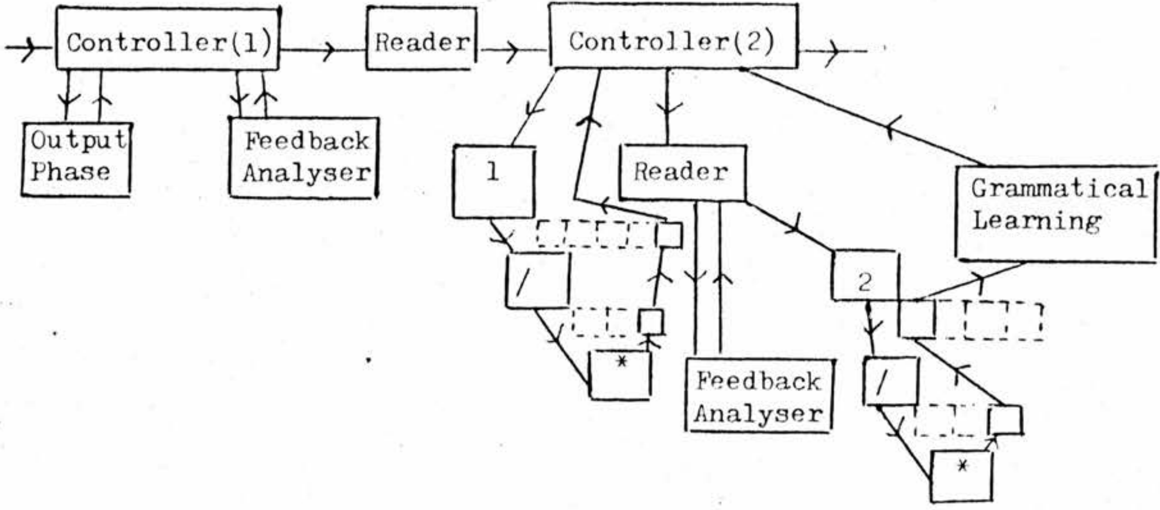


Fig. 7.29 Learning to count up to 2

in fig. 7.29 is a built in function of the program designed to ensure that it notices changes in the status of the blackboard. Execution of the Reader will halt after a delete character and the remaining input line is saved for a subsequent execution as shown. The Feedback Analyser is presented with the blackboard in its revised state. The process for a stroke after 2 in fig. 7.29 will encounter the most recent feedback in its search of STM. This seems the simplest way to model the natural observations a child would make as he removed counters from a pile.

Although control flows through the Reader and Feedback Analyser in this way, the process represented by the box marked 2 receives a result from the 1 process on its left. Therefore, Grammatical Learning generates instructions to perform a seek to the left and includes them in the procedure of the new expectation (for a stroke) that it places in the frame of 2. As always, the seek is as explicit as possible and is specific to the 1 process. In other cases, such seeks were generalised in subsequent examples but that does not happen here since counting always has the same sequence.

Here is the crucial link between the two numbers that is repeated between higher ones. The link has a complementary form which comes into action when the program counts, as distinct from having the tutor count to it. As explained more fully in Chapter 6 (section 6.5.5), seeking to the left involves a search of STM, the state of which is shown in fig. 7.29. The search is explicitly for a process like that associated with 1 which passed a result to the process for 2. At the same time as Grammatical Learning, using the Synthesiser, creates the necessary procedure to perform this search and places it in the frame belonging to 2, it also causes a new record to appear in LTM. The record bears a key indicating that it is to be used when a process like that associated with 1 is executed again.

This is an example of the general capability of constructing an LTM record complementary to an STM search. The parameters to the STM interface function which specify and constrain the search are incorporated in the key of an LTM record, indicating when the procedure contained in the record is to be executed.

Hence there are two consequences of showing the program how to count up to 2. First the frame of local expectations associated with 2 is augmented by an expectation for a following stroke, as denoted by the right-most box in fig. 7.30. Second, a new record is created in LTM which is capable of counting 2 in appropriate circumstances as discussed below.

The record representing the right-most box in fig. 7.30 is just like a record in LTM with a key indicating that the procedure it contains is to be executed when the processes for interpreting a stroke are active. This record is attached to the LTM record for interpreting 2 which is represented by the upper box in the figure. Only when the program reads 2 as input will it begin to look out for the items

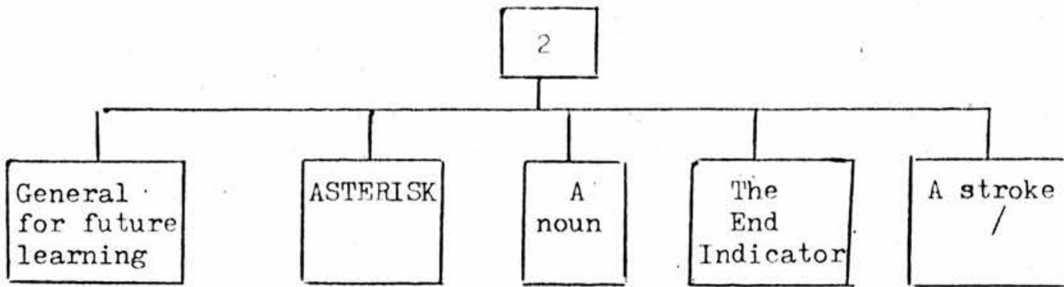


Fig. 7.30 The expectations in the frame associated with 2

named in the lower boxes.

As mentioned earlier, the row of boxes shown below a larger box in diagrams such as fig. 7.29 denote the expectations in the frame in the order in which they were created. The box in completed lines is always the one activated. The five expectations associated with 1 are similar to those in fig. 7.30. In fig. 7.29, the fifth expectation belonging to 2 is just being created and does not appear.

The two expectations contain different procedures: a seek to the left is included in the case of 2. It is implemented as an invocation of the STM interface function with parameters specifying an execution of the procedure for 1 with the appropriate kind of result. That happens to be procedural, i.e. a partial match, because the process for a stroke matches the following asterisk into the feedback of two asterisks in the previous line. The nature of the match is important in the complementary version since it furnishes the means whereby the program can stop counting when all the items on the blackboard have been erased by strokes. In this respect it resembles processing for "-s" although there are significant differences.

Counting was taken as far as 4.

: ["***]

***

: 1/*2/*3/*

: ["****]

****

: 1/*2/*3/*4/*

7.7.4.1 "Count"

The program was taught the verb "to count".

: ["*]

*

: COUNT THE ASTERISK [1/*]

1

After this example, the meaning of "count" is to put the characters 1/ in front of the object. Another illustration is needed before the process-like attributes of the meaning are acquired.

: ["*]

*

: COUNT THE ASTERISK [1/*]

1

This time, the "count" process sets up the SEARCHPROC argument to the STM interface, corresponding to 1/* (i.e. the procedure PRIN ('1'); PRIN ('/'); PRIN ('*');) by joining 1/ on to the argument supplied by the "asterisk" process. STM retrieval finds a match in the Output Phase, illustrated in fig. 7.31. The result of the STM search thus has its process-like attributes unspecified and so Differentiation is invoked to extend the meaning of "count". As usual the extension is implemented as a new record in LTM. The procedure within it includes instructions to specify the extra parameters (EXECPROC, IDENTIFIER and SUPERORDINATE) to the STM interface.

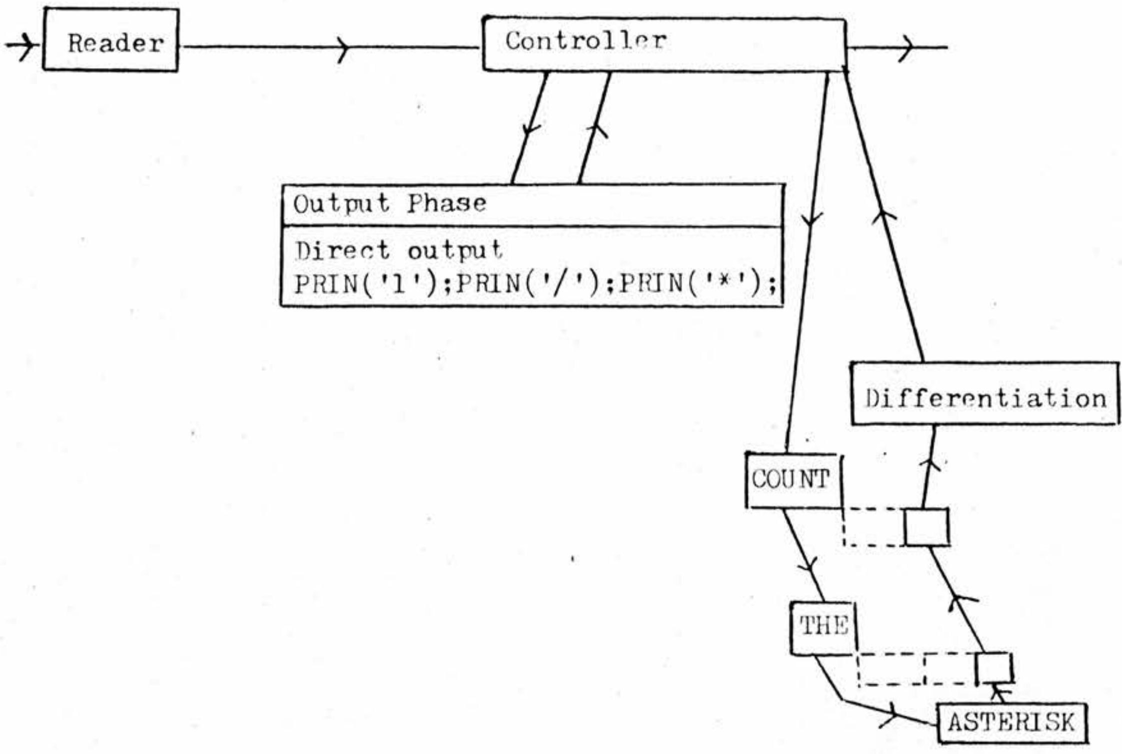


Fig. 7.31 Completing the meaning of "count"

Next time, the verb "count" will cause the procedures for printing 1 and deleting a character actually to be executed, thus triggering further counting because of the established processes in LTM. This triggering depends on re-interpreting each utterance or action that the program is about to make, as mentioned in the context of the plural morpheme "-s". It will perform the following.

: ["..]

..

: COUNT THE DOTS

1 2

: [",,,]

,,,

: COUNT THE COMMAS

1 2 3

: ["."]

.

: COUNT THE DOT

1

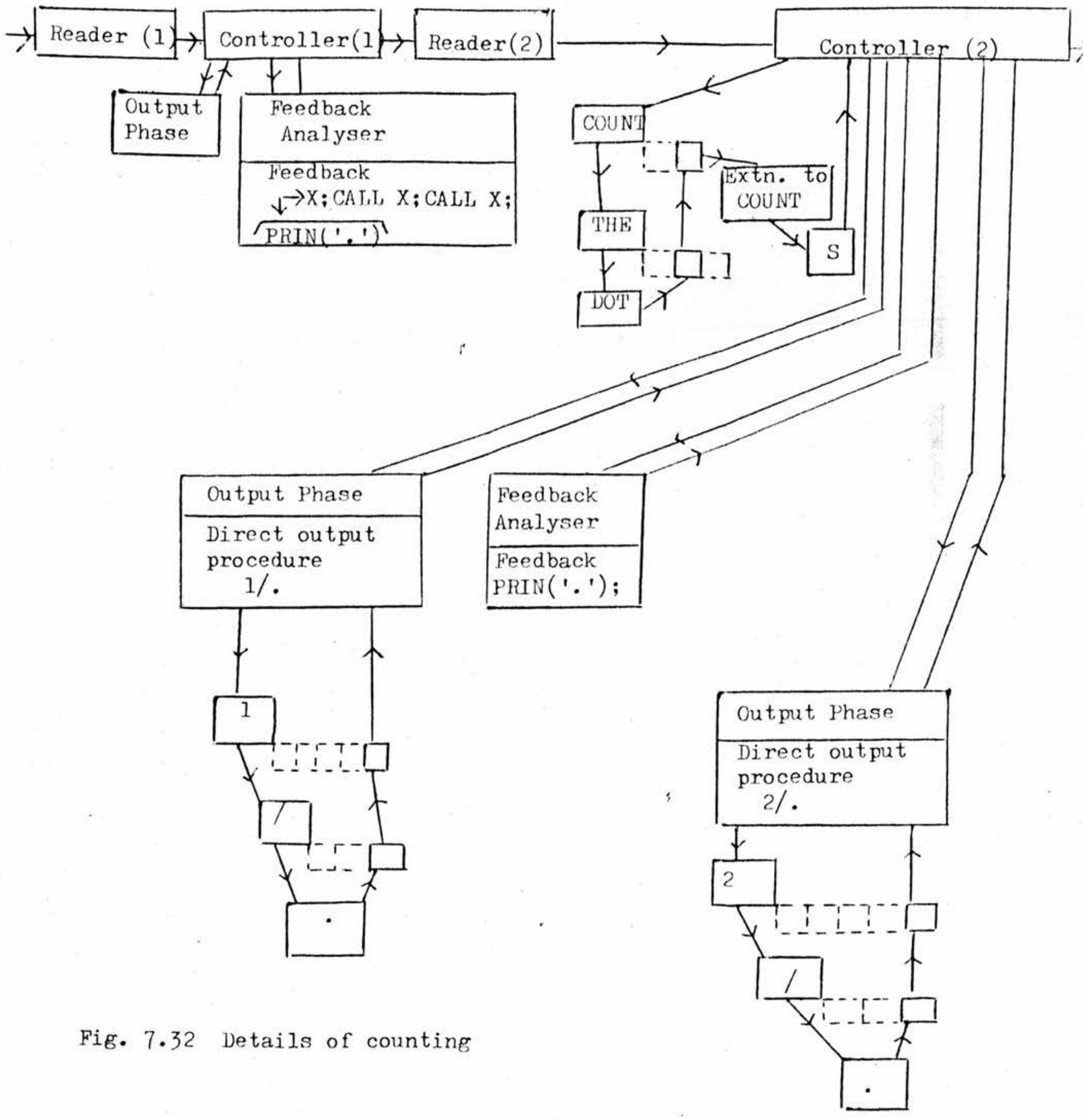


Fig. 7.32 Details of counting

Fig. 7.32 illustrates one of the examples, showing the interpretation of the output by the Output Phase. Earlier diagrams did not show this detail because it was unnecessary but interpretation always takes place. First the processes for "count" and "dot" construct the output 1/. and interpretation of that causes the construction of the further output 2/.

7.7.5 Accumulation

Counting is continued in Chapter 8, which contains the untested material, including work on arithmetic. Here, a kind of implied addition is presented, perhaps more appropriately termed accumulation. It begins with a new verb.

: WRITE A DOT [".]

.

: //

For convenience, the tutor simply erases the dot.

: WRITE TWO ASTERISKS ["**]

**

: //

The second example is needed to generalise the expectation associated with "write".

As in the case of "count", a third example is needed for the acquisition of the imperative (process-like) specifications.

: WRITE A COMMA [",]

,

: //

The operation of adding is introduced in simple stages. First the program is taught to write the named objects on the blackboard.

: WRITE 1

£

: ADD 2 ["££]

££££

: COUNT THE 1S

1 2 3

Here, the verb "add" is introduced as synonymous with "write" but is intended shortly to emerge with a distinct meaning. The above is a primitive form of addition that relies on prompting from the tutor as well as use of the blackboard. The next chapter presents refinements and further developments.

A FURTHER DIALOGUE

Chapter 8

The material in this chapter is not tested, although it has been checked in detail by hand. It is included because it illustrates the wider applicability of the principles of learning that have been expounded, as well as demonstrating a few difficulties. Particular problems are identified towards the end. The general hindrance in testing has been the nature of the implementation of PROCESS 1.5 as an interpreter written in POP_2 and the inclusion in short term memory of many unnecessary items, leading to very slow operation. The implementation has served its purpose as a flexible research vehicle but now requires to be rewritten before much more testing can be carried out.

The work below deals with arithmetic and counting.

8.1 Arithmetic

Continuing from the end of Chapter 7, the tutor proceeds to reduce the amount of prompting he has to do in getting the program to perform arithmetic making use of the blackboard. In section 8.1.4 the question of doing without this aid is discussed.

8.1.1 Addition

The next step is to couple counting onto the meaning of "add". First the expectation should be generalised and the imperative specifications acquired, as for "count" and "write".

```
:  WRITE 1

         £

:  ADD 1 ["£]

         ££

:  ADD 3 ["£££]

         £££££
```

: //

An example of combined adding and counting is now given.

: WRITE 1

£

: ADD 1 ["£ 1/£2/£]

££

1 2

It would have been too difficult for the program to learn all this at once. The object (i.e. 1) of the verb "add" has its meaning (£) inside the procedure for producing "£ 1/£2/£ and STM would not locate it. Now that the writing action (supplying the quotation mark) has been learnt, the STM match performed by the "add" process is adequate and the remaining seven characters (including a space) become part of an extension to the meaning of "add".

Now it can perform unaided.

: WRITE 1

£

: ADD 2

£££

1 2 3

Fig. 8.1 shows the two extensions to the original meaning of "add". The processes combine to produce the output shown, where the procedures are represented in the boxes by the characters that they print. The original meaning of "add" contributes the quotation mark portion of the result; the first extension specifies to the STM interface that the Output Phase in imperative mode is involved; and the second extension supplies the start of counting, which in the present example is continued to 3. (That involves interpretation of the output not illustrated in fig. 8.1 but analogous to

that in fig. 7.32).

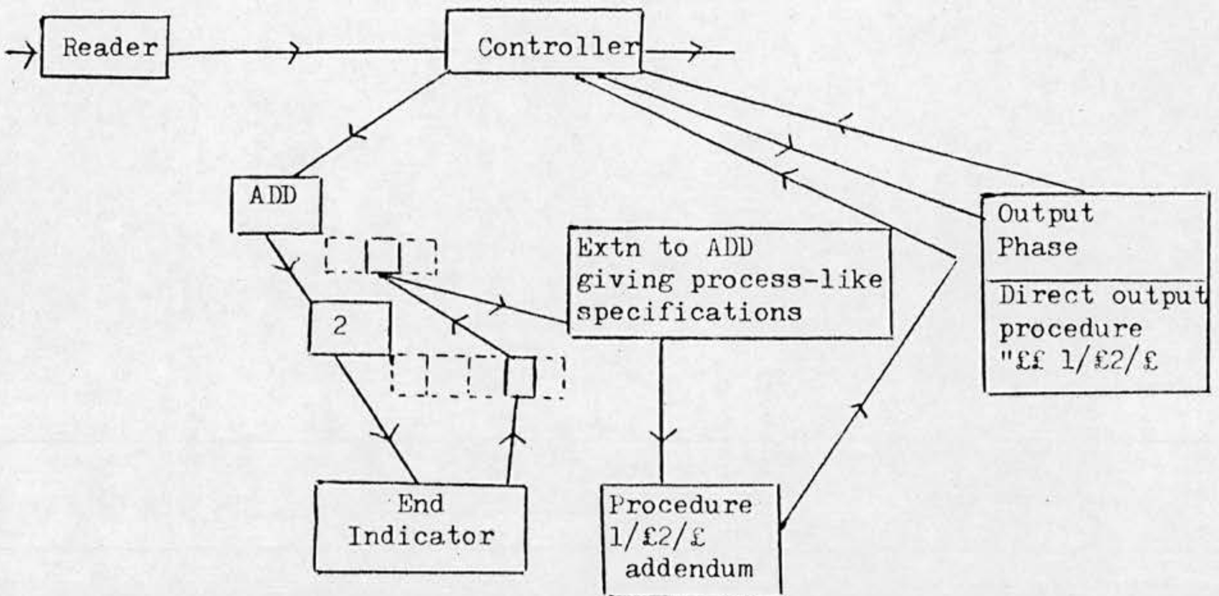


Fig. 8.1 Processing for "add" after the second Differentiation

There is clearly scope for further experimenting here. Obviously this is not a complete treatment of addition. However, many of us will recall from our own schooldays just such an activity as this - counting and combining objects provided by the teacher. Another approach to addition is taken below and it is suggested that various kinds of arithmetic situations are necessary before the individual can be said to have grasped the subject. There are evidently many levels at which it may be understood, from one-two-three to Dedekind sets.

8.1.2 Subtraction

Subtraction turns out to be somewhat more difficult than addition. The approach is to introduce the act of "taking away" by erasing from the blackboard. The complication is that "taking 2 away" is not merely a matter of constructing /££. Rather it is to take 1 away twice by assembling the procedure /£/£. The delete

operator, the stroke, has been constructed to take only one character because this seems to be a natural complication which the program ought to be able to overcome. In order to remove two objects from a group, a child would first remove one and then another.

The actual path of development that the program takes is drastically influenced by word order. The following sequence, for instance, forces the program to alter the meanings of each of the numerals above 1 that are demonstrated. It follows the same pattern as "write" or "count" to begin with.

: WRITE 1

£

: TAKE AWAY 1 [/£]

: WRITE AN ASTERISK

*

: TAKE AWAY AN ASTERISK [/*]

: WRITE A DOT

.

: TAKE AWAY A DOT [/.]

"Take away" now has its expectation generalised, is associated with the stroke (/) and contains imperative specifications, much like other verbs that have been presented. Now it becomes more specific.

: WRITE 2

££

: TAKE AWAY 2 [/£/£]

: WRITE 2

££

: TAKE AWAY 2 [/£/£]

Note that "take away" is seen as one word, regardless of the space in the middle.

In the above, 2 is at the end of the sentence and so, because of previous learning, is taken to mean two currency signs (££) rather than just two itself. As always, the meaning is embodied in a call which the 2-process makes to the STM interface. Here it gives a SEARCHPROC parameter equal to

```
   ↓  →X;   CALL X; CALL X;
PRIN('£');
```

By the rules of STM, this will not match into the procedure for /£/£ which is the direct output. That procedure is like the above but has PRIN('/'); PRIN('£'); in the subroutine position. Consequently, the meaning of 2 when it occurs at the end of a line (i.e. associated with the fourth box in the lower part of fig. 7.30) is generalised to be the same as that of "two", represented by the following SEARCHPROC parameter to STM.

```
   ↓ →X;   CALL X; CALL X;
"Empty"
```

After a further, similar example "take away" acquires a specific expectation for 2 and it includes code to insert a stroke and a currency sign (/£) into the subroutine slot of the procedure supplied by the following process. This can be generalised for higher numerals by means of a similar example with 3.

Afterwards, the program must re-learn (via Differentiation) the meanings of the numerals that are associated with the currency sign.

: ££

: 2

The above is typical. As mentioned in section 6.4, the program contains checks to ensure that a cycle of alternate generalisation

and differentiation does not develop.

Counting must be included within the meaning of "take away" as well.

: WRITE 4

££££

: TAKE AWAY 3 [/£/£/£ 1/£]

1

Then one should be able to get the following.

: WRITE 4

££££

: TAKE AWAY 2

1 2

Fig. 8.2 illustrates.

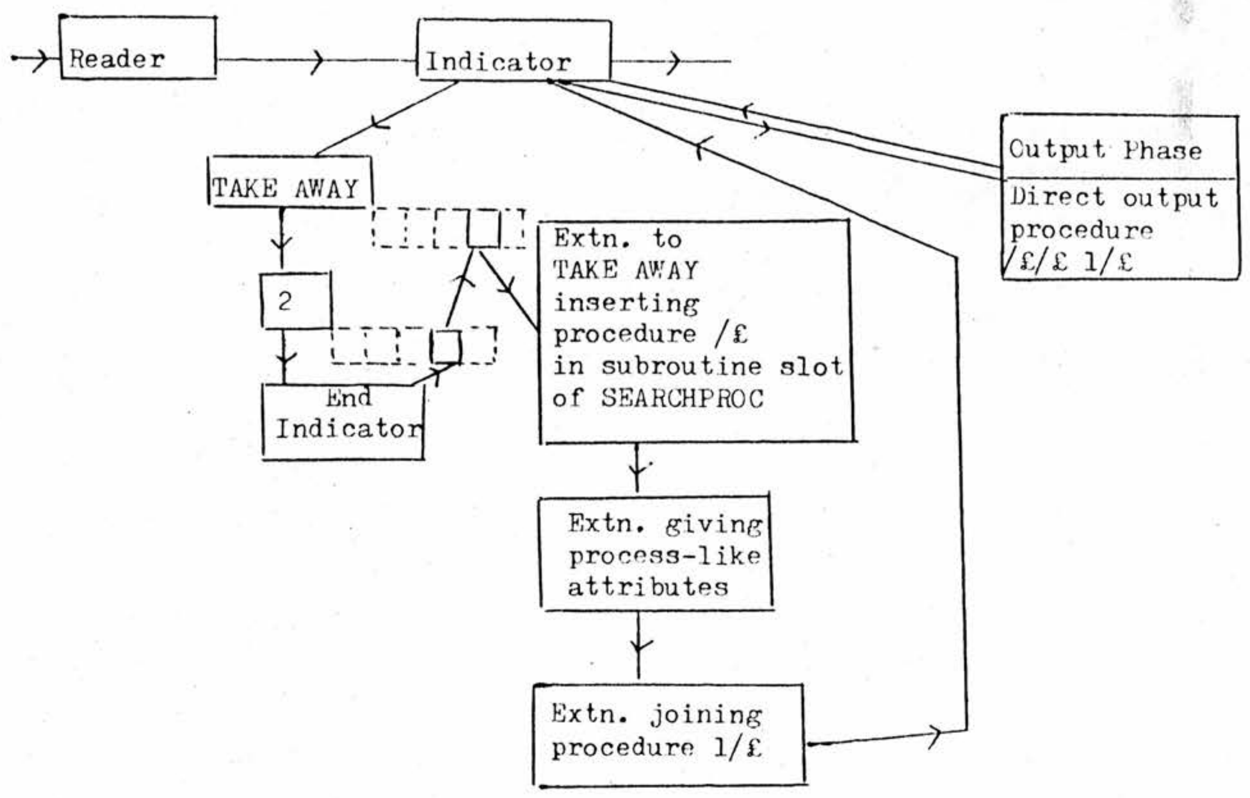


Fig. 8.2 The set of extensions to "take away" that are invoked.

8.1.3 Multiplication

The present approach to number attempts to capture the essential semantics of number: the act of repeating or the perception of repetition. The program can be taught to apply one number procedure to another. The word "times" is used here. It is, of course, commonly employed in teaching multiplication in schools. Thus "2 times 3" causes the meaning of 3 to be duplicated, making 6.

It is introduced in a simple fashion.

```
: **
: TWO TIMES ONE ASTERISK
: ££
: 2 TIMES 1
```

Now the following will work.

```
: WRITE 2 TIMES 2
  ££££
: COUNT THE 1S
  1 2 3 4
```

8.1.4 Addition taken further

Most children develop beyond the stage of counting and commit to memory tables of the 81 digit pairs (1-9, 1-9) in addition, subtraction and multiplication, either through recitation or simply by practice. The program is able to acquire knowledge of each pair in a similarly laborious manner. There seems to be no alternative to memorising the pairs of digits at some stage. Progressive education does not make such memorisation overt in the way that traditional education used to. Nevertheless most of us have grown out of the stage of counting on our fingers and it is difficult to see how this could be achieved without recalling from memory the elementary number facts as acquired from experience, with or without the aid of rote

learning.

After that, children are taught algorithms for handling multi-digit numbers. I have not investigated those. Some work has been done in that area by Badre (1973). His program extended algorithms for adding and subtracting three place numerals to cope with four places. This, however, is the only learning task his program performs and no general principles of learning emerge from his work, which primarily addresses the area of natural language understanding.

A grasp of number involves much more than a mastery of these basics, as acknowledged in the discussion of Chapter 2. Nevertheless, they must still be handled. The following ideas are an attempt to get an understanding of these facts about numbers by having the program observe the effects of combining objects on its blackboard. As far as can be seen, however, without having tested these sequences on a computer, it makes no difference to the form in which the program stores this knowledge whether the example statements are accompanied by suitable actions on the blackboard or not.

A reasonable way to proceed is like this.

```
: WRITE 2

        ££

: WRITE 1

        £££

: 3 IS 2 AND 1
```

There are two distinct threads here. The first is a function of the Causality component to explain (and hence to learn) the relationship between the final procedure for £££ and the two earlier procedures for £ and ££, all in their particular contexts. The second thread is to associate knowledge about the same procedures with the process

(and expectations) of 2, 1 and the conjunctive particle.

8.1.4.1 Memorising the facts

These are both interesting in their own right. To deal with the second thread first, the usual meaning of "and" fails because it simply combines the procedures from the results it receives from left and right. The composite result, which becomes the SEARCHPROC argument of "and" to the STM interface is shown here. It is the join of the arguments supplied by 2 and 1.

```
  →X; CALL X; CALL X; PRIN ('£');
PRIN ('£');
```

Such a procedure cannot be found in the short term memory.

This is a case where the program has the choice either of generalising the meaning of "and" or of creating a new expectation specifically for 1 and adding it to the frame of "and". As explained in section 6.5.2.1, the decision depends on generating the procedure for a new meaning to be associated with such an expectation. If a similar (i.e. matching) procedure were found already to occur in the frame, the meaning embodied therein would be generalised. In this case that does not happen and so a new expectation is appended to the frame of "and", as illustrated in fig. 8.3.

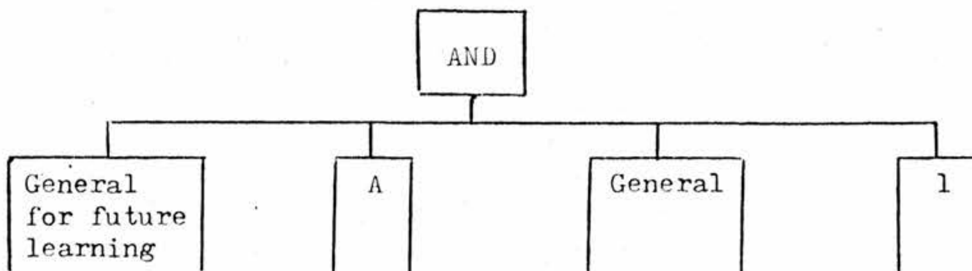


Fig. 8.3 The expectations in the frame associated with "and".

The process for "and" will construct the composite procedure from 2 and 1 shown above regardless of the presence or absence of the related work on the blackboard. If the work is present, it will receive results that refer to the preceding process of analysing feedback but it will extract the procedural content of these references. In a sense, it will 'take note' of the relationship. If the blackboard work is not presented, the meanings of the numerals already contain enough information to make the same internal construction.

A second, similar example is needed to complete the learning of 2 and 1. There is now a unique process associated with "and 1". The process for 2 receives a result from it and establishes a specific expectation for it which becomes attached to the frame of 2. The presence of "3 is" in the sentence causes the particular meaning of 3 to be associated with this expectation. After many examples, a table would effectively be constructed as illustrated in fig. 8.4.

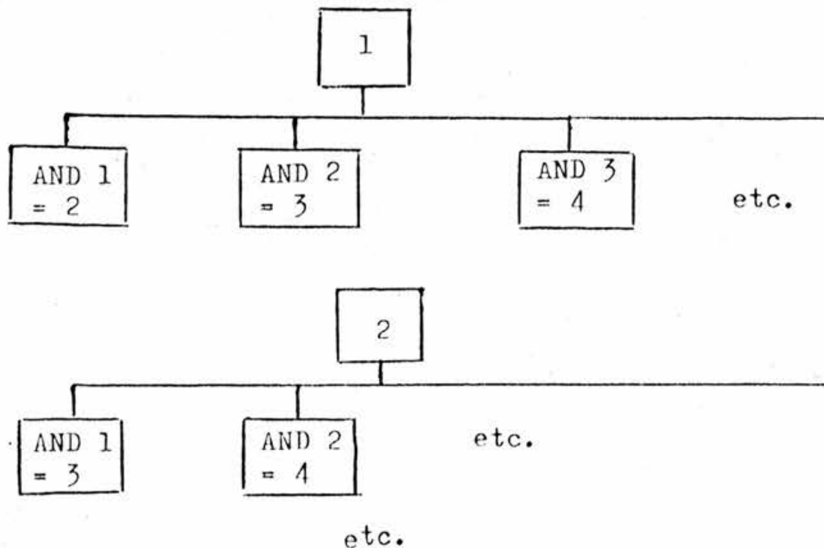


Fig. 8.4 Addition table stored in a set of frames

Similar possibilities should be explored with "times" and "take away".

8.1.4.2 Causal Relationships

It seems possible to go further towards assimilating a more general pattern of combining numbers of objects by representing causal phenomena in terms of predictors and explainers that embody a limited syntactic capability (such as that defined in section 6.2.3). The point of the exercise is to enable the program to explain (and, by means of complementarity, to predict) the effect on two, so to speak, of another one being added in the above example: WRITE 2 WRITE 1. More explicitly, the objective is to express the transformation from

```
  ↓→X;   CALL X;   CALL X;                    (1)
PRIN('£');
```

to

```
  ↓→X;   CALL X;   CALL X;   CALL X;          (2)
PRIN('£');
```

caused by the action

```
PRIN('"'); PRIN('£');                         (3)
```

The transformation should be expressed in a general way, independently of the character £. In other words, a structural relationship must be represented between the procedure PRIN('£'); within the result and the similar code within the writing action, so that the rule applies naturally to any object. This is not generalisation of the kind defined in section 5.2.3. Rather it is the characterisation of structure made apparent by matching and this is a function of the program's constructive learning principles, including the simpler left-hand part of the grammatical component.

When the program interprets the procedure (2) above that constitutes the feedback of £££ the front portion is directly predicted by an LTM process set up from the line: "Write 2". In other words, procedure (1) matches into (2) leaving as remainder

```
        CALL X;                                        (4)
```

Although the line: "Write 1" has predicted the feedback

```
        PRIN('£');                                     (5)
```

that prediction is never satisfied. Instead, a general explainer (actually the complement of this last predictor) is executed and it tries to explain procedure (4) by looking for an action.

```
        PRIN('"');  CALL X;                            (6)
```

Of course, it cannot find anything like (6) so it enters a generalising routine. (As the generalising routines for causality are untried, they are not described in Chapter 6).

Analogously to generalisation of meaning (see section 5.2.3.) it removes the supplied component (4) from the STM search argument and now locates (3) correctly. The simple conclusion that the routine might make is that (3) causes (4). However, it can do better than that.

As explained in section 6.2.3. a causal process is able to receive a result from the left, while the place of a result from the right is taken by the STM reference identified as the cause. In the present case, the process on the left deals with procedure (1) which thus would be the result. Procedure (5), the cause, matches into it and the Synthesiser for Causality would construct code to represent constructively that kind of match.

The outcome is that on performing a write action the program first checks to see if the character it is writing already appears

on the blackboard. Hence, in the following it will now predict four asterisks as represented in (7) below rather than as in (8).

: WRITE 3

£££

: WRITE 1

££££

It predicts "four asterisks"

```
 ↓→ X; CALL X; CALL X; CALL X; CALL X;          (7)
PRIN('*');
```

as opposed to "three and one asterisk"

```
 ↓→X;CALL X;CALL X;CALL X;PRIN('*');          (8)
PRIN('*');
```

Notice that only adding one has been learnt but it applies to an arbitrary initial number (greater than one) on the blackboard as well as to any character. Further examples would be required to teach the combination of arbitrary pairs of numbers.

An important principle has emerged here, namely that the effect need not directly match the cause once a link has been established. More research is needed in the area.

8.2 Two-place numerals

The above operations are greatly enhanced when two-place numerals are learnt. As previously stated, numbers to the base five are used in order to save time. Four is the lowest base that will allow generalisation to be demonstrated. Five was chosen to facilitate the multiplication example (section 8.1.3.). Typically, in numbers 20 to 24, 20 is a special case. The frame associated with 2 acquires an expectation for 0. When the program first encounters 21, an expectation for 1 is added to the frame. Since the numerals

above zero are very similar and the meaning of 2 in 21 to 24 is the same, the example of 22 causes the expectation to be generalised. It thereby covers 23 and 24 automatically. The same is true of the teens, the thirties and the forties.

No experimenting has been carried out with the number words (e.g. twenty-three) although they would probably be easier to deal with than the numerals, various difficulties of which are described below. The numerals are interesting because their denomination is denoted by position. By contrast, special words denote the teens, the multiples of ten (forty, fifty) and hundreds, thousands, etc.

First it is necessary to introduce zero. No attempt is made to teach the concept of nothing. The symbol is simply introduced alone so that it may be used to form higher numbers.

```
: 0
```

Now 10 (to base five).

```
: ££££10
```

An expectation for 0 becomes attached to the frame of the 1 process and bears the meaning:

```
      →X;  CALL X;  CALL X;  CALL X;  CALL X;  CALL X;
     ↓
PRIN('£');
```

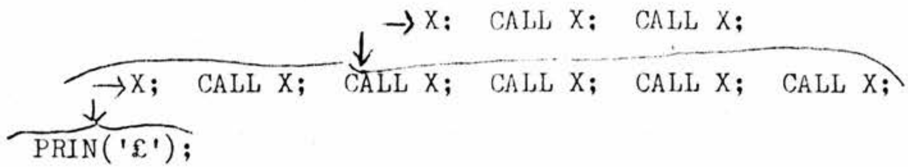
A sentence will teach 20.

```
: 20 IS 2 TIMES 10
```

More simply, the tutor could enter:

```
: 20 IS TWO 10S
```

(Note that 2 10S would cause confusion for obvious reasons). The meaning of 20 becomes the following (literally: two times five times £).



This is used as the basis of the meanings of 21 to 24.

: 21 IS 20 AND 1

: 22 IS 20 AND 2

The result of the processes invoked by the phrase "20 and 1" is the join of the procedure for 20 shown above with that for 1 (which is just PRIN('£');). The "is" process passes this result back to "21". The process for 1 (in 21) finds the appended procedure, strips it off and thus the process for 2 receives the meaning of 20 as result. Hence a new meaning is synthesised and associated with the expectation for 1 in the frame of 2. The procedure synthesised includes code to join the procedure for 20 on to the result of the following process.

The case of 22 is exactly similar, so that the expectation is generalised. The details of the generalisation involve removing the procedure that is specific to each numeral (in the case of 1 it is PRIN('£'); for 2 it is [PRIN('£');] →X; CALL X; CALL X;). The procedure that remains will match any of the numerals 1 to 4. However, it still contains syntactic detail that will distinguish it from other entities such as the nouns.

A demonstration of successful generalisation is that 23 will be interpreted correctly without disturbing other meanings of 2.

: PRINT 23

££££££££££££

: PRINT 2 ASTERISKS

**

The same teaching sequence can be applied to the teens, thirties and forties.

More difficult is something like "23 dots", where both the 20 and the 3 must be applied to the object. Compared with the easier example above, the lowest level procedure should be empty. That in itself presents no particular problem but the process that interprets 2 when followed by a numeral must insert the object in the empty place. However, in "23 dots", it does not receive the object until the process for 3 has operated on it, inserting it in a triplication procedure, resulting in 20 times 3 dots.

This is a distinct difficulty. It would appear to be a problem also when the number words are used as in "twenty-three dots" (with or without the hyphen). Taking a very optimistic view, perhaps it can be considered a strength of the program since no normal human could obey a command like "take 23 paces" without resort to counting.

8.2.1 Counting

Counting with the number words appears to be little harder than the counting already done. Each multiple of 10 has its own word and there is less scope for learning general patterns than is the case with numerals. For this reason counting above 10 with numerals has been investigated and the venture has not met with success. It is concluded that this was not the best approach.

A typical example during the course of the teaching would be as follows.

: WRITE 13

££££££££

: 1/£2/£3/£4/£10/£11/£12/£13/£

The process for 3 within 13 in the last line receives a result

from the stroke process on the right. It contains specific code to obtain a result from the process for 2 on the left via an STM call. Therefore it does so and behaves as if the 1 were not present. The process for 1 has by now acquired a generalised expectation for a following numeral (because of previous examples that stopped at 11 and 12). It also contains code to obtain a result from the preceding process for 1.

The weakness is that no account has been taken of the fact that the result of the process for 2 (in 12) has been used twice, both by the preceding 1 and the following 3. It becomes apparent when one considers the complementary process of counting performed by the program.

Suppose, for the sake of argument, that the program has successfully counted to 12 and one object remains on the blackboard. The complement of a seek to the left is a record in LTM. Hence the process for 1 in 12 will fire a process to generate 1, starting to form 13. However, that process for 1 does not contain code explicitly to start a process for 3, because the complementary expectation has been generalised to cover all numerals. The specific code for 3 is contained in an LTM process triggered by 2, the complement of the seek to the left from 3 to 2. Unfortunately that LTM process gets no chance to become activated because the frame for 1 (in 12) contains a local expectation and they always take precedence over the global LTM.

What is needed is for both processes to be activated, thus complementing the double use of the result from the 2 process that occurs when counting is being taught. However, it is not easy to see how this could be achieved in a reasonably general way.

The resolution of the problem may lie in a quite different direction. Appealing to the principle of being informed by the example of natural intelligence, it is well known that children do not master the Arabic numeral system until their linguistic and other abilities are far more advanced than are those of this program. It may be that the ability to understand extensive verbal explanations is prerequisite.

The conclusion, then, is that research should proceed into enhancing the program's linguistic abilities by devising new dialogues and teaching situations, perhaps in new domains, and to tinker with its principles of operation as little as possible.

8.3 Comment

Underlying the present approach to learning, and hence to intelligence, is the belief that knowledge should be organised as an inter-related set of procedures each designed for use in specified circumstances. Since the only test of an individual's understanding is to observe his actions or hear his words, it is reasonable to suppose that his understanding is built of procedures that can do such things. That is not to say that it consists only of such procedures. It is necessary to allow also procedures that can act on other procedures and are therefore recursive in the general sense - not only in being able to call each other (or themselves) but also able to compare, match, create and break down one another.

To learn is therefore to create a new procedure along with a statement of when it is to be used. This is a discrete theory of learning and I follow Minsky and Papert (1972) in the opinion that although there may be certain kinds of physiological adaptation that form a statistical continuum, higher learning is discrete. The

illusion of continuity may be caused by the individual taking a large number of small steps, each involving some structural change. It may also be the result of misapplying statistical data from a large population.

Popova (1958) describes some interesting experiments on the acquisition of gender agreement in Russian. She showed that in the right conditions with suitable objects for the children to refer to they were able to progress to correct agreement in only one or two examples whereas in other circumstances hundreds of examples sometimes failed to achieve any change in the child's linguistic behaviour.

This evidence clearly supports the discrete view of learning embraced in the present work. However, she does find a statistical element in one kind of learning. Children who always used one gender regardless of agreement were able to respond gradually to training in which only the other gender was present. This led to gender confusion: it was a kind of unlearning after which correct discrimination could be acquired.

Such unlearning is analogous to the generalisation that the program performs. There is definitely a case for some kind of weighting factor here. The program generalises too readily. Words that have been used satisfactorily many times should not be susceptible to having their meanings changed in one instant.

PART IV

COMPARISON WITH OTHER PROGRAMS

Chapter 9.

The history of the study of artificial systems that exhibit learning is almost as old as the young discipline of Artificial Intelligence itself. Most of the research discussed below is not concerned with natural language: that topic has been separated off in section 9.2. After that, comparisons are made with natural language programs that have no claim or pretensions to learning. It is useful, however, to relate the product of an acquisition process to these "performance" models (section 9.3). A theory of knowledge put forward by Minsky (1975) is also discussed.

9.1 Learning Programs

The most striking fact about all the learning programs considered below is that they involve the use of searching techniques of one sort or another that would lead to a combinatorial explosion if applied to a large problem domain, such as the real world of human knowledge and experience. The problem is well known in relation to heuristic search techniques such as were employed in GPS (Newell, Shaw and Simon, 1960), in Samuel's Checkers program (1959 and 1967) and more recently by Harris (1972). Research continues into finding ways of improving heuristic functions with domain specific knowledge in order to limit the search time (e.g. Michie, 1974). Production Systems (Newell and Simon, 1972) as used by Waterman (1970 and 1975) and others suffer from the need to scan all condition-action pairs on every cycle. Winston's structure-matching procedures have to be applied against every model in the memory (Winston, 1970). CONNIVER (Sussman and McDermott, 1974), as used in HACKER (Sussman, 1973), and also STRIPS (Fikes, Hart and

Nilsson, 1972) employ pattern-directed retrieval from a data base which requires substantial searching for matches and subsequent elimination of the contents of possibility lists. All these are discussed more fully below.

The reason that Evan's program (Evans, 1968) avoids the problem is simple and instructive. Because of the particular domain, his program only performs elaborate matching between a limited number of structures. In DISCO, the combinatorial problem is avoided by dividing memory into long-term and short-term components. As well as appealing to psychological theory, this device has the great merit of confining the more elaborate type of matching, which demands a search, to the short-term memory (STM) which is of limited size. This point was made in Chapter 2 and elaborated in Chapter 4. In Chapter 5 an even more flexible kind of matching was produced with the search restricted still further. Retrieval from long-term memory (LTM) is direct; no search is needed. Even Becker (1973), who did propose to structure memory into STM and LTM, failed to capitalise on this benefit; his LTM retrieval algorithm is analogous to that used in CONNIVER.

9.1.1 The GPS Tradition

The principle of heuristic search is to solve problems (e.g. the best move to make in a board game or the next step to take in a mathematical proof) by enumerating, at any given stage, all possible next steps, and perhaps their successors to an arbitrary level, and selecting the most promising by applying criteria more or less specific to the problem. These criteria are embodied in a heuristic function. The expansion of successor steps is a graph with the steps as nodes. In the General Problem Solver (GPS) as applied to proving theorems in logic (Newell, Shaw and Simon, 1960) the heuristic

functions assessed the difference between the "quod erat demonstrandum" of the theorem and the statements deduced from the axioms and previous theorems by GPS at any given time. Similarly, Samuel's Checkers Program possessed a set of criteria for judging the worth of potential board positions. In the latter case, the program chose the move that had the highest numerical result from the heuristic function. In GPS, types of difference were ranked and the step would be chosen which eliminated differences of the highest rank. In a domain like theorem proving, GPS would back up to an earlier choice if a deductive chain reached impasse. In game playing, of course, such backtracking is not allowed.

Such problem solving situations admit of a concise expression of a goal (e.g. winning) and the heuristic functions may be viewed as an attempt to measure the distance from a goal state. An alternative approach to problem solving is then to proceed backward by expanding intermediate goals (or subgoals, e.g. capturing a particular piece). Backtracking then takes the form of abandoning one or more subgoals in favour of another. This philosophy is embodied in PLANNER (Hewitt, 1972) and is carried over in CONNIVER. The contribution of PLANNER in this context is to permit the programmer to incorporate domain specific knowledge about the relationship between goals and subgoals and so limit the search in a way that more general mechanisms (like GPS) do not allow.

The approach to A.I. research of writing domain specific programs carries the risk of losing sight of the need to look for principles of intelligence that are general. Certainly the combinatorial explosions of search time exhibited by programs like GPS that were conceived from general considerations teach the lesson that detailed understanding of particular domains of intelligent

activity may be the best subgoals to pursue as a matter of strategy but one must not forget to look for generalities and nowhere is this need more obvious than in the study of learning. Sussman is a good example of someone who has written a domain-specific program and is fully aware of the pitfalls. In his note to the reader (p.126; end of Chapter IX) he expresses the opinion that his technique of debugging "is at least to some extent independent of the problem domain". The strategy has been encapsulated in the rather wordy title of a paper: "Some principles of Artificial Learning that have emerged from examples" (Knapman, 1975) which summarises some of the results presented here.

The domain-specific approach carries the attendant danger of producing ad hoc methods but that must be balanced against its advantages of allowing progress and providing the discipline of realistic problem areas. The idea of a goal-seeking organisation on the other hand is a thrust in the direction of generality and very many A.I. programs combine the two approaches. But for the kind of goals that arise in learning, a formulation of them is inappropriate, if not impossible. DISCO implicitly embodies general purposes like "learn meanings" or "anticipate consequences" but no search or goal tree is called for in such cases; the program is just made that way. It seems very probable that children are made in such a way too. Halliday (1975) shows how a child constructs a meaning system before the commencement of syntactic behaviour (i.e. before 18 months of age) and this contrasts sharply with Chomsky's (1965) theory of selecting hypotheses, presumably by using some search strategy (see the discussion in Chapter 2 above). The point is probably less controversial now than it used to be and Harris (1972) favours the searching formulation although he does not commit himself about children.

The argument against a goal-directed approach is confirmed by the GPS experience for when GPS was applied to the learning situation proper, a specific goal was not formulated. Rather than measuring the distance of the current state from some ideal target, the heuristic function was used to measure improvement after learning had taken place. Specification of a target state was impossible since the learning task was to produce the best set of heuristics for some problem domain and it would clearly be unacceptable for a best set to be given in advance. Apart from the ambiguity of "best", it would lay the authors open to the charge of cheating if they gave a "right" answer to the program.

The formulation of the learning task in Samuel's Checkers program was not as problem-solving in its own right in the way that Newell, Shaw and Simon attempted to apply GPS recursively to itself. They intended that new differencing functions (for measuring improvement) should be created and they presented a Difference Programming Language (DPL) in which these heuristic functions were represented as manipulable data structures as well as entities capable of activity. Samuel, on the other hand, did not attempt to write a program that could create new procedures. The heuristic function was the linear sum of a set of functions of board positions and the learning task was to adjust the numerical coefficients in the sum by weighting those of the functions that led to winning positions during play or to moves recommended in books. This is an example of a Perceptron, a device which Minsky and Papert (1969) have shown to have profound limitations.

The facility to synthesise functions (procedures) appears to be essential to all but the most trivial kind of learning. Sussman, Becker, Fikes et al., and Waterman certainly use it, and it is also

employed in DISCO. Like GPS and Samuel's Checkers program, Waterman's earlier work (1970) is addressed to the problem of learning heuristics for graph-traversing. Like Newell, Shaw and Simon, he became involved in the automatic manipulation of functions. The synthesis of heuristic functions was not carried out in any significant way; they were derived directly from advice presented from a model program, a human trainer or a decision matrix consisting of recommendations for various situations in the game of draw poker.

9.1.2 Generalisation Learning

Waterman's main contribution was in the subsequent generalisation of advice from one situation to a class of possible situations. The heuristic functions were implemented as production systems. (Although Newell, Shaw and Simon proposed to synthesise fresh heuristic functions, it should in fairness be pointed out that their learning version of GPS was only hand-worked in outline whereas Waterman's and Samuel's were run on a computer).

The principle of generalisation employed by Waterman is analogous to the relaxation of constraints carried out by DISCO during the generalisation process defined in Chapter 5 (section 5.2.3). A production system consists of an ordered series of condition-action pairs (production rules) and a working memory, or short-term memory. Activity proceeds by selecting the first rule with conditions satisfied by the contents of working memory. Actions may handle input and output, modify working memory or create new production rules. The production rules are analogous to the elements of DISCO's long term memory except that, as has already been pointed out, exhaustive searches are necessary to locate PR's and this makes them unsuitable for large-scale application without some further refinement. Both

formalisms offer the attraction of providing effective schemes for procedural synthesis and modification because of the lack of side effects when new entries are made. The working memory is completely different from DISCO's short term memory, which is considerably richer as well as having the important feature of a complementary interface with LTM. A consequence is that DISCO's generalisations can be applied to the actions of rules as well as to their conditions - something which production systems do not facilitate in any way.

Generalisation of a kind similar to Waterman's is employed in STRIPS (Fikes, Hart and Nilsson, 1972) although the entities on which it is performed are plans which were themselves produced by STRIPS during problem solving in a manner akin to GPS. The generalisation method is to replace constants by variables, taking precautions against inconsistencies that can be detected by the resolution theorem prover that is a part of the system. So, for instance, a plan for moving a particular box from room R1 to room R2 is generalised to moving any box from the room it is in to any other room. The generalisation, however, takes place at the time of synthesis rather than during subsequent experience in contrast to Waterman's program (which also generalises by enlarging numerical ranges in its conditions) and to DISCO. Moreover, these plans that STRIPS creates, although not derived quite so directly from domain-specific information as are the production rules of Waterman, are nonetheless logically deduced from information about doors, rooms and boxes programmed in to the problem solver as well-formed formulas in the predicate calculus. Hence, the two systems both perform solely the generalisation component of learning.

Whereas a production system has to evaluate the conditions of each rule until one is found, the position for STRIPS is much worse. There, the operators (including the synthesised "macro" operators and their derivatives constructed by successive removal of steps from the front) must not only be matched for the applicability of their pre-conditions but also for the suitability of their consequences as applied to the current state of the world model.

In HACKER, Sussman (1973) employs exactly the same generalisation technique of substituting variables for constants, although this is done on subsequent examples like Waterman's process and as in DISCO, rather than initially as STRIPS does. Thus the combinatorial search problem is somewhat alleviated because fewer generalised plans are created. DISCO, of course, eliminates the searches altogether. Although DISCO has not been applied to the traditional variety of robot planning situation, the problems it has already solved are certainly no more trivial. Like the other systems, HACKER is supplied with a collection of domain specific knowledge which in this case is contained in "libraries" and catalogued according to the purpose for which each item is used. Procedures, or plans, are synthesised by copying appropriately labelled routines from a library and incorporating other such routines set up for use in various error situations that are detected by the earlier ones and are analysed by a procedure that is also domain-specific. Sussman sometimes calls this process "learning" but his title "A Computational Model of Skill Acquisition" is really more appropriate.

All three systems discussed suffer from the inability to apply generalisation to the action, operator or procedure independently of the condition, set of pre-conditions or pattern (in the respective

terminology of Waterman, Fikes et al., and Sussman). DISCO is able to do so, as was explained in Chapters 5 and 6.

9.1.3 Program Synthesis (Automatic Programming)

More recently, Waterman (1975) presents a scheme for inducing production rules that characterise (i.e. predict) the patterns in certain fixed-period cyclic series (e.g. ABHBCICD). His idea of generalisation on examples subsequent to initial synthesis has, however, been abandoned; the most general hypothesis is made for each new rule when it is created (as happens with STRIPS) and a heuristic is employed to prune the search. The other two schemes presented in that paper consist of an implementation of Memo Functions (Michie, 1968) and a simplified version of the rote learning program EPAM (Feigenbaum, 1963) in which pairs of nonsense syllables are remembered. Inducing algorithms that predict series is a very similar problem to that tackled by Hardy (1974 and 1975) and by Shaw, Swartout and Green (1975) of producing a LISP program to transform a list expression as specified by a given input-output pair, e.g. (ABCD) (ABCDBCDCDD) which is solved by them both. It is not likely that their methods would prove applicable to a large class of problems but in a new field the ad hoc method does have its place.

The search for generality is pursued by Green and Barstow (1975) in another paper at the Fourth International A.I. Conference and by Manna and Waldinger (1975) whose paper also appears in the proceedings of that conference in shortened form. Green and Barstow define a system of rules for generating programs that transfer the elements of an input set to an output set. These rules are a restatement of a program that synthesised a sorting algorithm. They speculate that it might be possible to specify quite a large body of knowledge

about programming in such a form although they have not in fact developed a precise language for these rule statements; they are currently expressed as a commentary.

Manna and Waldinger have gone somewhat further along the road to generality by producing a precise method of forming recursive loops and the means by which DISCO forms them can be compared to it. In their system, a recursive call is formed when a sub-goal is generated that matches the top-level goal. DISCO, of course, does not have goals but it characterises situations procedurally so that recursion appears automatically, as was explained in Chapter 4 (section 4.3.4). Manna and Waldinger then have to include a test against infinite recursion because conditionals are not built in to their recursive calls in the way that they are in DISCO's LTM matching process. In their example of reverse (ℓ), (e.g. reverse (A (B C)D) = (D (B C)A)) the system verifies that tail (ℓ) is shorter than ℓ before approving the recursive call reverse (tail(ℓ)) for inclusion in the synthesised program. All the recursive calls that DISCO generates are of the type that cope with what is left over. That, after all, is the purpost of the component of DISCO - differentiation (see section 6.4) - that is capable of producing potentially recursive LTM records. So in general, whereas conditionals arise in Manna and Waldinger's work from the pre-specified knowledge of the domain (in this case:
is false if empty (S)), conditionals in DISCO arise naturally from learning situations by virtue of the formulation of long-term memory reference.

Burstall and Darlington (1976) present a strategy for program synthesis which is a step towards a general purpose programming assistant. Instead of commencing with an example or a declarative

statement their system improves a program already written but written for clarity and simplicity rather than efficiency. The system interacts with a human programmer but performs three actions automatically, viz. folding, unfolding and abstraction. "Folding" amounts to inserting a recursive call and "unfolding" is the removal of a call, replacing it by the content of the function; "abstraction" simplifies by removing common sub-expressions. This last resembles DISCO's perception process (section 6.1) although their matching is more flexible. Inserting a recursive call in their system (folding) is done after matching expressions have been located in a function body: DISCO therefore comes slightly closer in spirit to this method than to that of Manna and Waldinger. Harris (1972, pp.103-5) presents a similar operator which he attributes to Feldman (1970). Because Harris only considers context-free grammars he does not deal with conditionals. Like Manna and Waldinger, Burstall and Darlington require the conditionals to be given to the system: it does not generate them as DISCO does.

There is obviously a close connection between automatic programming and learning. The fundamental difference is in the task specification. For Manna and Waldinger, the objective is "the construction of a computer program from given specifications". In learning, there are no specifications. Those workers mentioned above who are working on procedural synthesis from examples are between the two positions. Hardy (1975) points out that to specify an algorithm is at least as difficult as writing it. Smith and Hewitt (1974, p.203) propose to debug the specification along with the program. Nevertheless it remains the primary objective of these systems to produce programs for explicit tasks whereas a learning system like

DISCO behaves and expands, interacting with its environment and constructing an organised body of experience and ability; synthesis of procedures is an internal process within such a system.

The goal of research into automatic programming is therefore to emulate, replace or participate with the human programmer. The difficulty of this task is appreciated by those working in the field. Green and Barstow write: "The size of the set of rules suggests the complexity of the process of writing programs and that much work will be required to codify significant amounts of programming knowledge..." (Abstract, p.232). This realisation is echoed by Manna and Waldinger: "Many of the abilities we require of a program synthesizer, such as the ability to represent knowledge or to draw conclusions from facts, we would also expect from a natural language understanding system or a robot problem solver. These general problems have been under study by researchers for many years..." (p.176). The arguments of Chapter 1 in favour of the learning approach therefore apply as much to automatic programming as to natural language research or any other endeavour in A.I. It follows that programming should be taught to an advanced learning system as a task just as counting has been taught to DISCO. It may seem a far-off dream, but the state of the art in automatic programming or most other A.I. specialities is certainly no nearer to it than are the learning programs.

9.1.4 Structural Learning

A learning program that has received considerable attention is that described in the Ph.D. Thesis of Patrick Winston. This has been cited (e.g. by Solomonoff, 1975, p.277) as "perhaps the most competent induction program yet completed" and is examined in detail in the following discussion.

Unfortunately, there is difficulty in evaluating the exact extent of Winston's contribution because of the disturbing lack of detail in his exposition. One of the difficulties lies in ascertaining to what extent the programs have been tested for their capability to carry out the requirements put forward in the thesis. The problem is rather different from that of considering a proposal. The practice of producing proposals (even as the finished work for a Ph.D.) is growing in A.I. (e.g. Charniak, 1972; Goldstein, 1975) but is actually quite old. The GPS learning system (Newell, Shaw and Simon, 1960) was a proposal. It is rather like a mathematician propounding a theorem without the proof. In A.I. it is woollier than that because no-one could check the program even if the listings were presented in full and even a program that exhibited the claimed behaviour still might rely on some trick that did not accord with the principles stated by the author. Such deviations need not go undetected indefinitely: many A.I. programs have been re-implemented in other laboratories and their limitations discovered. Unfortunately this kind of careful evaluation - so important in science - is not very glamorous or popular and one rarely sees the results of such investigations published.

In response to a letter, Winston (1976) states that all the learning aspects of his program have been implemented and run but the testing was not extensive. The point about testing is also made in the thesis. He now expresses a reluctance to resurrect the programs. There seems little more one can say about the matter. What one can talk about is the extent of his contribution to the body of human knowledge because that is contained in the thesis.

He describes a kind of generalisation learning involving pre-defined classes, and he puts forward a notion of "near miss" - essentially a negative example - for the purpose of inducing emphatic and negative relations in a structural model. Unfortunately he only gives a "cursory introduction" (p.254) to the network matcher which carries out these tasks. The described algorithm is not capable of identifying a generalised model with a scene. Neither does Winston show how it could handle any of the near miss examples. Winston describes the matcher as "a hastily programmed, slow and stubborn stumblebum". All those words seem quite reasonable, except for "stubborn".

Generalisation learning is the simplest kind but it is not satisfactory if the result of the process is unusable without <u>ad hoc</u> programming for each example. The near miss is a more important concept but it too appears to be ad hoc. The following sections argue that it is also unsatisfactory from the psychological point of view as well as being unnecessary within a more coherent framework for representing knowledge. The final argument is that his representation with its open-ended proliferation of primitive types of node and arc offers little hope of wider application and is, moreover, inefficient in principle (i.e. leads to a combinatorial explosion) for the perception problem.

9.1.4.1 <u>Difference Descriptions</u>

The basis of Winston's work is to match structural descriptions and produce difference descriptions. These ideas are due to Evans (1968) and it is instructive to make comparisons with his ANALOGY program. It solves problems of the type: "A is to B as C is to 1, 2, 3, 4 or 5" as illustrated in fig. 9.1

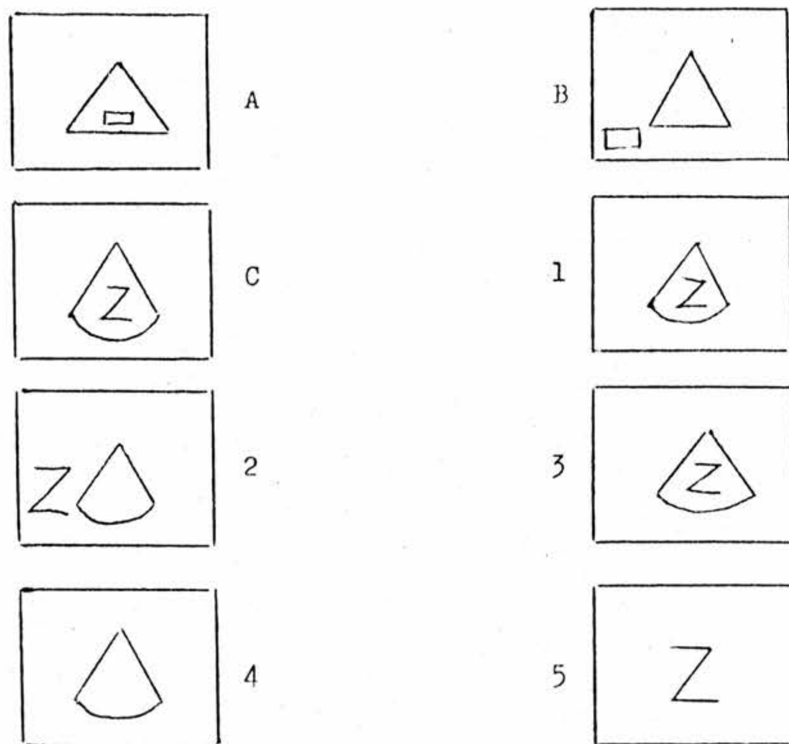


Fig. 9.1 From Evan's ANALOGY program

His program accepted input in the form of the x,y - co-ordinates of start and end-points of lines and the curvature between them. Part 1 would create a list structure representing the figures of each scene and relations between them such as LEFT, INSIDE, ABOVE and SIMilar. Part 2, which is less domain-dependent, produces another structure, which is effectively a net, relating the description of A to that of B in terms of a list of figures added in transforming A into B, a list of those removed and a list of those matched. In fact, the program produces structures representing all possible relations that are compatible with the similarity information computed by Part 1. Next the same process is applied to each of the pairs C-1 to C-5 and those candidate matchings which do not possess the same number of

ADDs, REMOVEs and MATCHes as one of the A-B structures are eliminated. There follows a complex numerical procedure which narrows down the remaining field to one pair of matchings and that gives the answer.

The network matching that Winston describes constructs a difference net from two networks representing scenes. In the appendix to his thesis, he describes an algorithm to ascertain which nodes in two networks should be linked (identified) for the purpose of further comparison. The scenes illustrated in fig. 9.2 would be represented by the structures of fig. 9.3 and the matcher should establish links as shown although none of the examples in the description of the matcher contain a horizontal pointer like the SUPPORTED-BY arc so it may be that the method is ad hoc.

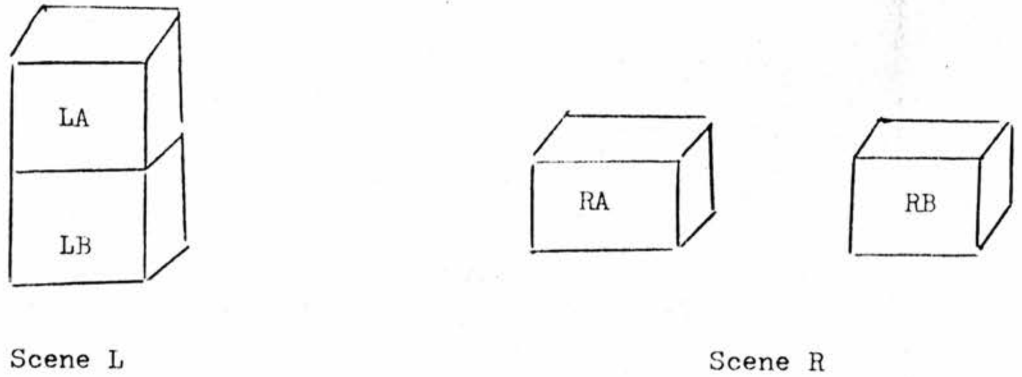


Fig. 9.2 From Winston's Thesis (his fig.4-8)

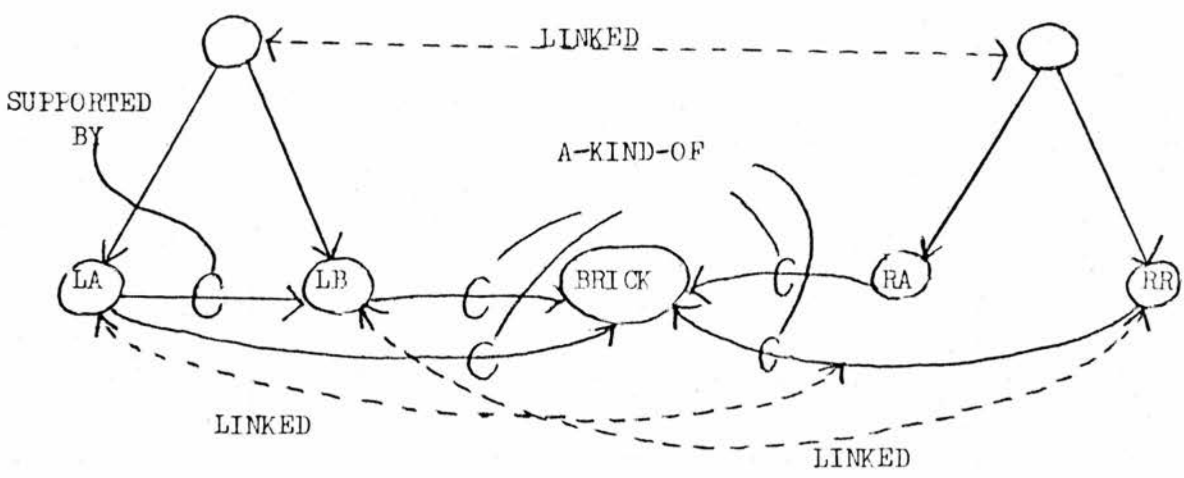


Fig. 9.3 Elaboration of Winston's Fig. 4-9

Now he constructs a "skeleton" which is supposed to be a copy of the common portions of the two nets. However, the only skeleton he ever draws (his fig.4-2) is derived from two identical scenes and so is a trivial case. The next step is to synthesise a structure representing the presence of a pointer in the left diagram not present in the right. This is illustrated in fig. 9.4.

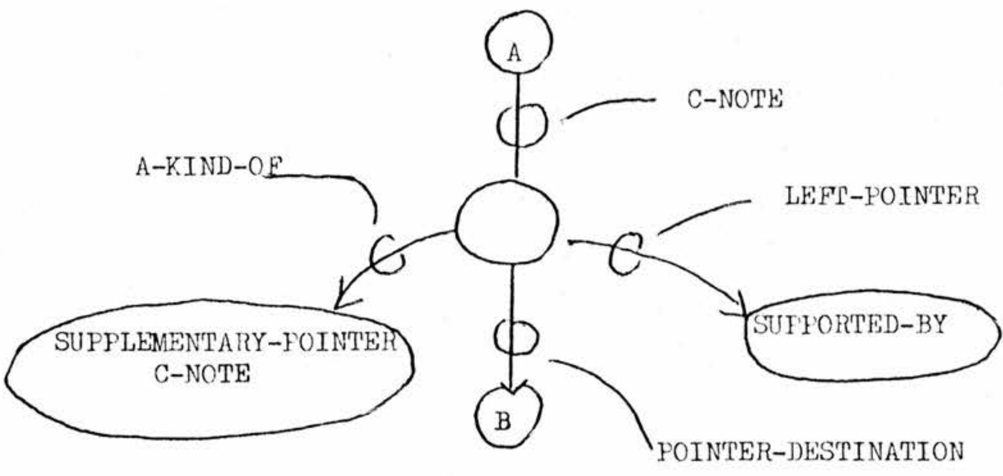


Fig. 9.4 Network representing the difference between scenes L and R

The two are then linked so that the C-notes are "attached to the skeleton like grapes on a grape cluster".

The use of such comparison nets (called C-notes) in the learning process that he outlines in Chapter 5 is fairly clear, at least in the simple cases. They are used to create models of concepts like the arch, the pedestal and the house shown in fig. 9.5 (fig. 9.6 shows the house model). What is less clear is Winston's claim (p.109) that his network matcher can compare C-note structures and so solve ANALOGY problems like Evan's program does. It is not that this claim is particularly staggering; merely that Winston does not explain how it is done but just states that it is done. As a reader one is inclined to stretch a point and give the benefit of the doubt until one encounters the much stronger claim on page 111: "But of course there is no limit, and with time and memory machines could happily think about extended problems involving an arbitrary number of comparison levels". Such a claim requires justification, more especially since Minsky and Papert (1972, section 4.6) consider this ability to apply the same method recursively at a higher level of abstraction to be one of Winston's major contributions.

Winston's solution to ANALOGY-type problems relies on a metric for the details of which one is referred to Chapter 7. In section 7.4 a numerical scheme is given for choosing between different models when trying to identify a scene, which on the face of it would appear to be a different problem from trying to match C-notes. However, Winston (1976) states that this scheme is the one that was used for the purpose and it ran successfully, although he is reluctant to send more details because the programs are off in archives.

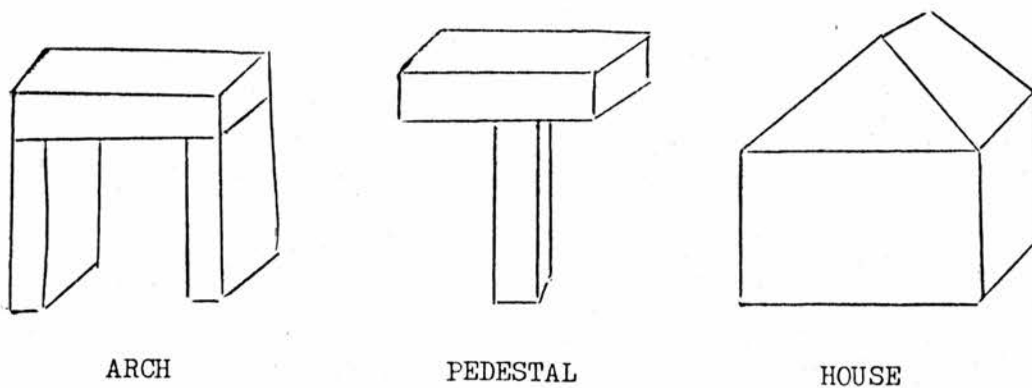


Fig. 9.5 Typical scenes for Winston's program

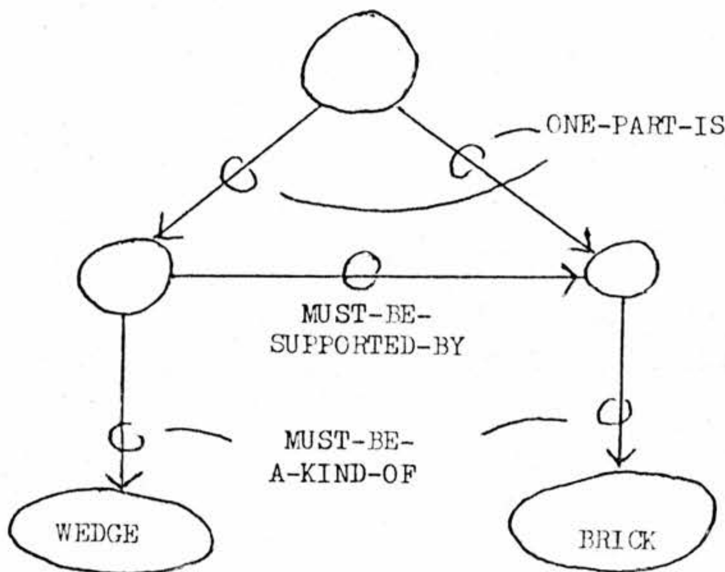


Fig. 9.6 Internal Model of a House

9.1.4.2 Using a Model in a more elaborate situation

An important example in Winston's thesis is the ARCADE, illustrated in fig. 9.7 with accompanying network. It is important because, apart from the poorly expounded ANALOGY problems discussed above, it is the only example of a synthesised network being used at a higher level than the most basic. It will be shown below that it cannot be done by the means Winston provides and that the claim made on his behalf by Minsky and Papert (1972, end of section 4.4),

viz. "its descriptive mechanisms proceed from local to global aggregates using as much available knowledge as it can apply", is not supported in the thesis. Confirmation comes from the fact that the crucial ARCADE example has been omitted from a revised version of the Ph.D. thesis (Winston, 1975).

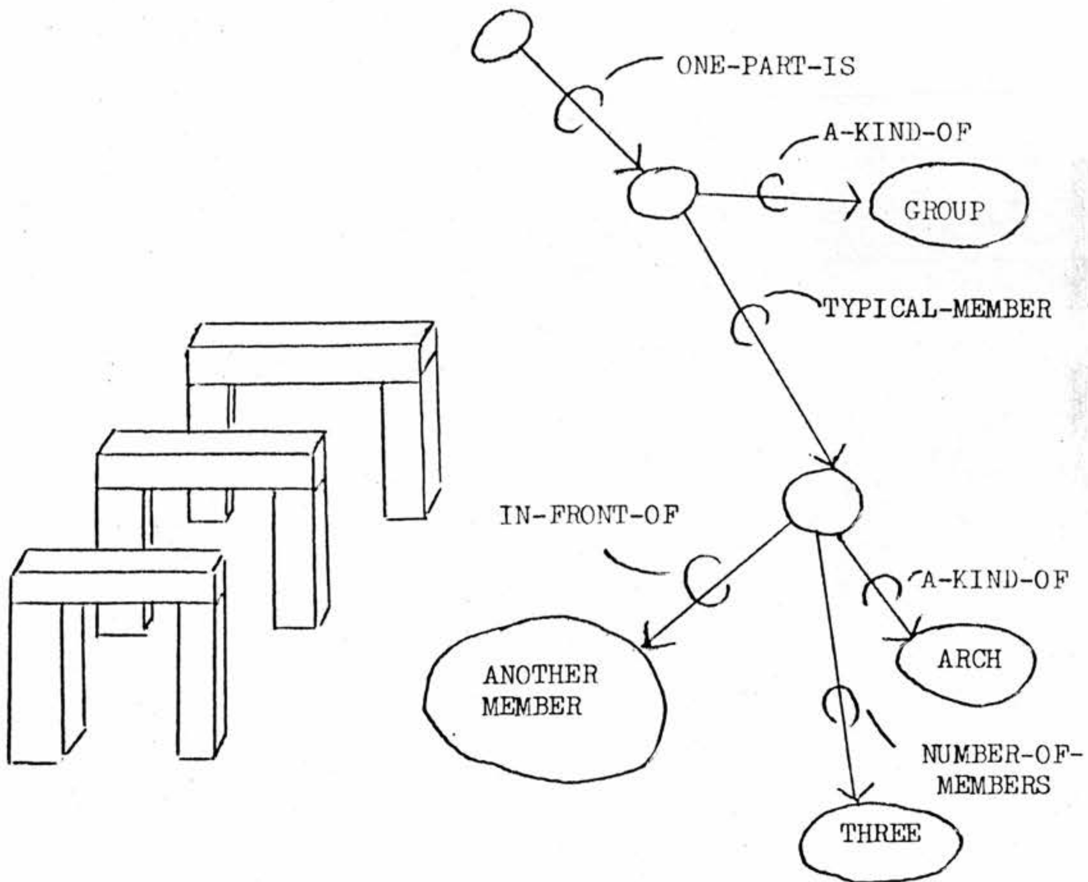


Fig. 9.7 The ARCADE from Winston's figs. 6-42 and 6-43.

The argument is in two parts. First, it will be shown that the necessary grouping must be ad hoc and second that the ARCH concept lacks an essential element for the learning.

9.1.4.2.1 Grouping

After the visual processing, which is outside the scope of this discussion, Winston posits a grouping process to produce a net characterising groups of objects in a scene. This is paralleled to some extent by the Perception process of DISCO described in section 6.1. Detailed comparison is impossible owing to inconsistencies and lack of information in Winston's exposition. For example, he shows a network reproduced here in fig. 9.8 as a representation of the scene illustrated.

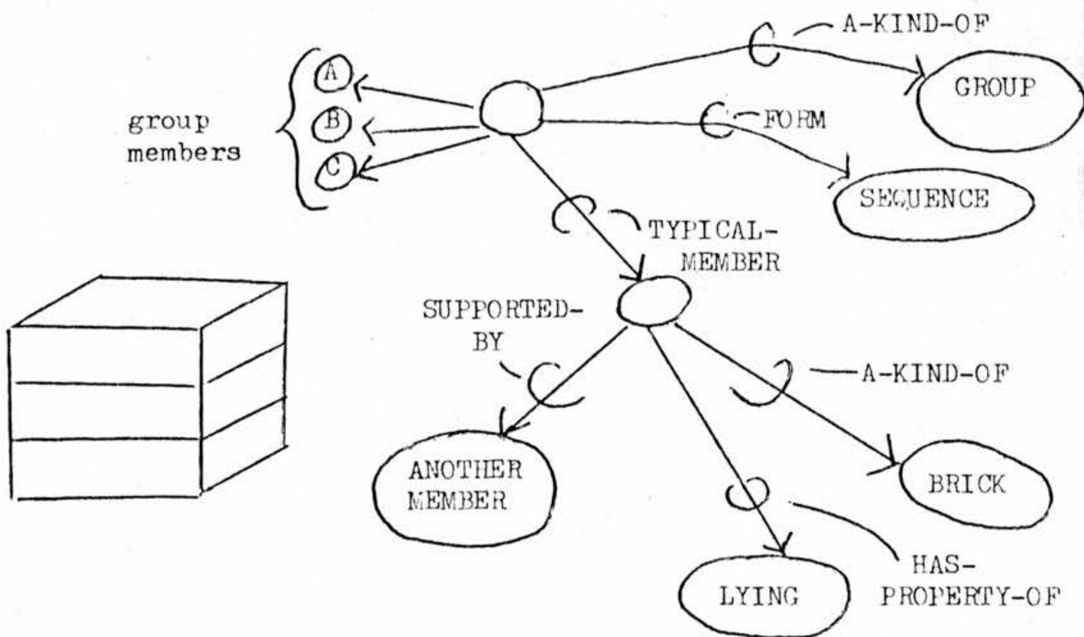


Fig. 9.8 Network representing the group illustrated at the left

The FORM arc and the SEQUENCE node are not explained; one is merely called upon to notice them (p.98). ANOTHER-MEMBER is not derived in the text. On page 251, the following remark is made: "The schemes for recognising reasonable clusters of objects is particularly primitive and has undergone too little testing. Mechanisms must be found for producing and handling alternatives to the first partition devised." "Partition" is apparently a reference to the two methods of grouping mentioned in his Chapter 3, neither of which could produce the ANOTHER-MEMBER node.

Since ANOTHER-MEMBER is a nice idea, elaboration would have been interesting and helpful. It would be particularly interesting to see how to cope with the problem that the bottom brick is not supported or the rear arch is not in front of another member in the arcade. In view of the 80% criteriality level (p.94) this seems a difficult and important problem.

Another disturbing omission is the failure to explain why in Chapter 3 an arch is considered to be a group (fig. 3-1) whereas in Chapter 6 it is not, as implied by his fig. 6-54 (see below). Perhaps that is why the exposition of the arch in section 6.5 carries no networks. Instead section 6.10, somewhat misleadingly entitled "The Arch in Depth", merely refers the reader to the rather complex network of fig. 6-54 without a word of explanation. That diagram contains the arc GROUP-OF and the nodes ABUT and MUST-NOT-ABUT that are not mentioned anywhere in the text but does not contain an A-KIND-OF pointer to GROUP.

9.1.4.2.2 <u>Learning the ARCADE</u>

The description of the arch is a vital omission because of its use in the ARCADE (fig. 9.7). On page 186 we are told: "The des-

cription programs identify arches using the previously assimilated arch model. This leads to the description partially shown in figure 6-43." For such a very important step, this is not much information to go on. Later on that page he admits that deducing the IN-FRONT-OF relation is an unsolved problem for structures: "I have thought about (it) only enough to write programs which can build these few examples." But he does not mention any of the other unsolved problems.

Neither does he explain how the subsequent learning from examples of non-arcades takes place when the generalised ARCH concept is participating. An example of a non-arcade is a series of three bricks replacing the arches. From this, the A-KIND-OF pointer to ARCH is supposed to be strengthened to MUST-BE-A-KIND-OF. This process is reasonably clear where the object is a WEDGE and the example is a BRICK, as happens in one of the HOUSE counterexamples (fig. 6-4 resulting in fig. 6-9). It works because WEDGE and BRICK have A-KIND-OF pointers to OBJECT and the matcher (fig. A-1, fig. A-4; pp.255-6) can link them. This generates an A-KIND-OF-MERGE C-note (as p.135; the same happens with the TENT counterexample of two bricks (fig. 6-17) as mentioned on p.167; it is subtly different from the INTERSECTION described on p.105) and reference to the table of actions on page 146 shows that the MUST-BE version of the pointer is called for. ARCH, however, has no such pointer; no A-KIND-OF pointer from the concept is shown in the complex fig. 6-54.

Winston (1976) is of the opinion that a pointer to a node called CONSTRUCTION was in the network in a late version of the program although he is not sure.

9.1.4.3 The Question of Generality

It would not seem possible to generalise Winston's method because it admits of an open-ended set of primitives. He remarks on this open-endedness himself on p.121: "Each type of satellite is associated with a type of C-note forming an open-ended family." Merely associating two finite sets does not produce an open-ended one. In his case, there is no completeness in the basic sets. A programming language is capable of generating an open-ended family from a small number of primitives and this claim is made also for DISCO: the STM and LTM primitives and a fixed number of associated joining operations (see chapters 4 and 5) constitute a universal programming language and so, in a logical sense, DISCO is a compiler - as a program synthesiser would also be - but it is a much more subtle one than would conventionally be meant by the word compiler. Winston, on the other hand, introduces new types of node and arc constantly and often with no explanation. For example, on page 203 he introduces the nodes OCTAGON, HEXAGON and SHAPE.

A count reveals at least seventy different types of node or arc, in addition to twenty-one types associated specifically with visual processing. DISCO, of course, requires domain-specific operations (Reader and Output Phase with a set of characters). It is the non-specific primitives which must be finite in number and well defined if generality is to be achieved. Of course, generality is not sufficient in itself to provide interesting or useful behaviour as Newell (1973, p.51) points out and the question of what else one might require of a learning system was discussed in Chapter 2. But it is clearly necessary if computer software is to relieve us of the necessity for expert "surgery" of ever increasing complexity. Because the arguments to STM-LTM in DISCO are procedural, the same

primitive is used to identify a word (LTM: EXECPROC = "Controller", IDENTIFIER = "RESULTANT", SEARCHPROC = procedure for printing the word) as to specify a context for a particular meaning (e.g. associated with "3" in "3 and 2": LTM: EXECPROC = procedure for meaning of "and 2", IDENTIFIER = "FOUND", SEARCHPROC = "Empty" as explained in Chapter 7) or to anticipate feedback (LTM: EXECPROC = "Feedback Analyser", IDENTIFIER = "RESULTANT", SEARCHPROC = procedure predicted).

9.1.4.4 Generalisation Learning

Winston describes two kinds of learning: generalisation from positive examples and a "near miss" idea that gives rise to the insertion of emphatic pointers and negative relations. Generalisation is by means of pre-defined classes and is comparable to the class generalisation performed by Waterman's program (as distinct from replacing constants by variables which is another method that Waterman employed and which was later used in STRIPS and in HACKER). He was able to define conditions in production rules numerically for the game of poker and generalisation was to allow ranges of numbers. In Winston's nets, nodes like BRICK and WEDGE point to OBJECT and can be replaced by the class descriptor if an appropriate example is given. DISCO does not have such class knowledge pre-defined. It is able to induce class from function by the generalised matcher defined in Chapter 5. The examples of "character" and numerals were spelt out in Chapter 7. Learning in terms of function is highly desirable as Winston himself remarks (pp.197-8).

The trouble with his method of generalisation is that the resultant model is unusable by the matching algorithm. We are only given a cursory introduction (p.254) to the matcher which presumably copes with many examples ad hoc. The algorithm itself does not cope

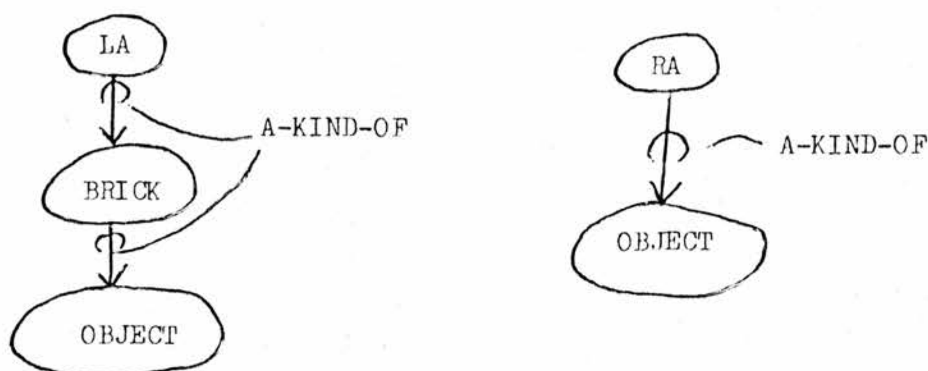


Fig. 9.9 Problem for Winston's matcher

with the match between the two nodes LA and RA in fig. 9.9 although this would be essential after the generalisation of the top of an arch (his fig. 6-25), the components of a column (fig. 6-36) and the legs of a table (fig. 6-49). It is also prerequisite to the a-kind-of-chain C-note (p.124).

The algorithm fails because in comparing two nets it always moves down one level in both nets at the same time. Thus, in attempting to link LA and RA in our fig. 9.9 it would attempt to establish LINKED pointers between BRICK on the left and OBJECT on the right. As these are unequal and the right-hand has no daughters, no link will be formed between them or between LA and RA.

9.1.4.5 The Near Miss

There are two interpretations of the near miss: the literal and the sensible. Taken literally, it would imply that children learn such early concepts as table, chair and door by means of carpentry, and depend on the ability to comprehend negation in order to complete their instruction or "self-programming" as Winston styles it

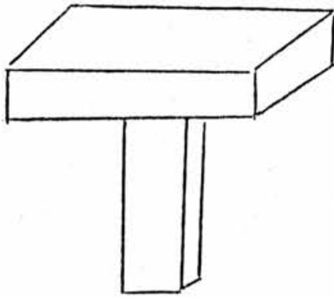
(p.132). Negation, however, emerges in child language later than words like "window", "car" and "truck" (Bloom, 1970, p.171ff. and p.104) and it seems most unlikely that the order of conceptual development for language comprehension (obviously prerequisite to the Winston scenario) should be otherwise.

There is no doubt that human beings are able to learn from the near miss and there is a whole tradition of such experiments in psychology, beginning with Bruner, Goodnow and Austin (1956). The question is whether the ability is a prerequisite for primitive learning or whether it is comparatively sophisticated. The evidence from child language is that negation is not necessary for elementary structural learning of the kind Winston describes.

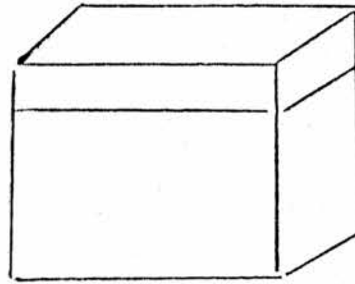
The key to the question lies in how a system is to use the fruits of a near miss situation. Winston posits two possible effects that such a situation can have on a model: the insertion of an emphatic relation or of a negative one. The purpose of such models, since they are not functional like DISCO's, can only be for perception, or identification as Winston calls it (p.199ff) and he gives proposals about how that should be done. Here we come to the more sensible interpretation of "near miss" because one could say that a positive example of a concept is a near miss to those models that come close to it. Winston comes near to this realisation in his suggestion (pp.228-33) that a machine should spend idle time computing differences between its internal models.

The important point to realise, however, is that a positive example of something else is enough to define the first concept without altering it at all. The mere presence of another model in memory is sufficient to constrain the first. So, for instance, the

counterexample to a pedestal in fig. 9.10 could be introduced as a pillar or a post. A program would not then need to be told that it is not a pedestal; that is implied by the existence of a more appropriate model and no emphatic or negative pointers are needed.



PEDESTAL



NEAR MISS

Fig. 9.10 From Winston's fig. 6-11 and 6-15

DISCO works this way. A concept is defined by its first instance and may then be generalised by other positive occurrences. Typically, over-regularisation will occur and then discrimination (differentiation) comes into play. Always the aim is to be as specific as possible.

The trouble is that Winston's proposed representation is so cumbersome that it would lead to a tremendous combinatorial explosion. To begin with, identification of a scene demands an exhaustive search of all networks. The provision of difference relations between models in order to reduce the search is of dubious worth because it involves massive expansion of the data base since difference descriptions are larger than the nets that they compare. They also are specific to each pair of models linked so that a searching algorithm would have no way of preferring one above another. In any case,

there is no guarantee that a relatable model would be found very quickly and the difference relations would only be useful thereafter. If we allow, on average, five difference relations per model, memory space increases more than five-fold and the search is still only reduced by a fifth (from N/2 to N/10 on average).

The next weakness to be dealt with is the dependence on training sequence. On page 132 he stresses this point. It is unclear just how important the sequence would be had he used the identification techniques of his Chapter 7 but it is manifestly clear that children are hardly ever presented with well ordered examples in learning about basic concepts like houses and doorways. In face, one of the most striking results of the work of Piaget and his many followers (e.g. Minsky and Papert, 1972, section 4.1) is the relative invariance of the child's development across cultures, teaching methods and generations. The child selects an order from his experience. In Chapter 7 above instances of variation in training sequence were cited but certainly DISCO has not yet been exposed to the rich variety of sensations that the infant experiences. The question of DISCO's robustness in a benignly indifferent environment is not finally answered but it does need asking and Winston ought to allow it too since he is interested in psychological modelling (p.15).

9.1.4.5.1 The Final Weakness

The last objection to the near miss is that Winston never gives enough detail to convince one that there is any general way of matching the two nets involved in such comparisons. Fig. 9.3 above shows a simple example that typifies many near misses. Nowhere does Winston work through a case like this that differs by a horizontal pointer. None of the diagrams in the appendix (pp.254-63) are like this. I

have supplied the linkages in fig. 9.3 by interpreting Chapter 4. Winston never explains the process and one must therefore presume that the method is ad hoc, dependent on the details of these particular examples.

If Winston could not find a general solution to the matching problem for the near miss it does suggest that the idea may be unsound, at least for the representation he has chosen.

9.1.4.6 The Near Miss again

Recently Hedrick (1976) has published details of a learning program that employs the idea of a near miss in two instances, although not as extensively as Winston did. They are used for a somewhat different purpose. Whereas Winston employed them to insert negative relations or make existing relations emphatic, Hedrick uses them to create a condition or relation in order to compensate for over-generalisation.

Given a network of facts about relations between the letters of the alphabet, Hedrick's program builds production systems that characterise series. In dealing with the example A X B X C X ... it over-generalises to allow any letter after A X B and must be given the counter-example A X B Y. This causes it to restore the over-generalised production to its former state and inhibit its future change. A subsequent example A X B X is then forced to generate a new special rule which happens to be appropriate in this case.

If the ideas embodied in DISCO were applied to this problem differentiation would supply the rule after over-generalisation because of the principle of "explaining" or accounting for the unspecified attributes of a situation. The need for a near miss would then disappear. This would appear to be closer to human performance;

we do not require such a counter-example to solve this problem.

The other activity in which Hedrick's program engages is interpreting English sentences. It is presented with a sentence, such as "Cats love Jill" which is given in the format: "cat s love # Jill" where # indicates the absence of a suffix on "love". This is paired with a network representing its meaning with respect to a pre-defined network of class relations. Both Siklóssy (1972) and Harris (1972) (see section 9.2) present sentences to their programs paired with an internal representation of the meaning. It is, of course, a wholly unnatural device.

Hedrick has made an advance, however, in being able to induce a context-sensitive rule, whereas Harris' program only induces context-free rules. He uses a counter-example such as "cat s like s John" to force the program to notice the requirement of number agreement between noun and verb. DISCO's grammatical component should cope with this problem without recourse to near misses. However, the nearest that has been tested is the agreement between "-s" and a number word as in "three dots". A further point is that DISCO would not reject an ungrammatical sentence as Hedrick's program would and in this respect is more natural. Hedrick, by the way, does not deal with the problem of utterance at all.

The biggest defect of Hedrick's program is that it employs a variation of the GPS paradigm, exploring all possibilities and thus leading to an exponential explosion. His variation of the method is to seek the amendment that leads to the least change, rather than the greatest improvement as Newell, Shaw and Simon proposed.

Both Hedrick's and Winston's methods of generalisation rely on classes pre-defined to the program in a network. They lack DISCO's ability to generalise by function.

9.1.4.7 A Proposal by Becker

In conclusion a proposal by Becker (1973) that has some features in common with DISCO is worthy of mention. Of course, being a proposal it leaves many unanswered questions but, following current ideas in psychology, he suggests an STM and an LTM together with the twin-learning processes of generalisation and differentiation. The latter is hardly defined at all but he suggests that the former could take place by keeping statistics of use and eliminating rarely used conditions. This would work quite differently from generalisation in DISCO and there probably is a case for introducing such an idea - not to effect generalisation as he suggests but to inhibit it for frequently used items. This would provide a "sense of proportion", a protection mechanism against random mistakes. For instance, the following would cause the meaning of "three" to be generalised to include "two".

: **

: THREE ASTERISKS

If "three" had only just been learnt, this generalisation to "two" would be reasonable but after frequent use of the word in correct situations it would no longer be so. Alternatively, the program might by this time be sufficiently endowed with knowledge of the numbers to rule this out without recourse to any built-in statistical inhibition. This is an unanswered question.

The contents of Becker's proposed LTM he calls schemata and they have the active property of DISCO's LTM records and of the condition-action pairs of production systems. He expects retrieval to be a deliberate process and requires the schema that initiates retrieval to cope with a list of possibilities found by a pattern matcher much as CONNIVER does. This is a distinct short-coming

compared with the automatic or notionally automatic retrieval of production rules and DISCO's LTM records because it would require existing schemata to possess information about schemata as yet uncreated if new learning is ever to be retrieved from LTM.

Becker's suggestion for STM is that insertions to it be automatic like insertions to DISCO's STM. He is non-committal about the means or function of retrieval, except to remark that it might be an alternative to searching LTM when trying to satisfy goals (p.423). Here, of course, he enjoys the luxury of the proposer although he loses the rigour which many would feel to be the raison d'etre of Artificial Intelligence.

9.2 Natural Language Acquisition

A proposal to study natural language acquisition on a computer was made by Schwarcz (1967). He suggested that a program might be written to accept sentences in a natural language together with a representation of their meanings as statements in the logic of Tarski (1956). First it would employ a clustering process to distinguish lexical items; next it would associate them with classes, relations or operators; thirdly induce a grammar; fourthly generalise grammatical rules and relate them to their semantic counterparts; and finally learn transformation rules (Chomsky, 1965).

These suggestions are all rather obvious in the context of the mid- and late-sixties. Quillian (1968) had recently devised the semantic net which would conveniently represent formal statements and Chomsky's formal methods seemed very promising. It is not surprising, therefore, that Harris (1972), who must have started in the late-sixties, opted for a similar scheme, apparently independently. (At least, he does not credit Schwarcz). Harris, however, produced a working program and so made a more reliable contribution to the body

of knowledge.

After Winograd (1971), he simulated a robot to provide a simple domain of discourse and the representation of knowledge was procedural. Harris's robot moves around an eight by four rectangle of squares discovering the position of certain objects, such as a "big square table" or a "piano", and updating its records accordingly. The program possesses concepts like "move", "step" and "is" and it learns to relate words to them and subsequently induces a grammar. Thereafter it is able to carry on a respectable conversation, obeying commands, answering questions and paraphrasing input in a constructive way from the semantic representation using the acquired grammar.

The grammar copes with nested clauses like the following.

STEP TO WHERE THE TABLE THAT IS SMALL IS!

The program constructs the reply:

THE ROBOT IS IN POSITION 84.

Like Winograd's SHRDLU, Harris's parser reduces the combinations of possible interpretations by appealing to the state of the world to eliminate unlikely constructs during syntactic processing. Harris also taught the program French (in a separate session).

Of course, like all successful programs in A.I. there are limitations. The human tutor has to switch the program into one of three modes: semantic learning, grammatical induction and dialogue whereas DISCO is able to distinguish these situations itself. Harris's program has all its concepts built in whereas DISCO constructs many of them (e.g. "after", "is", the numbers, "and" and so on). Moreover the way in which his program associates word and referent is by a crude statistical correlation between groups of words and concepts. This method has obvious limitations since every word is

related to every concept by a numerical weighting factor. There may be some justification for statistical techniques in learning the very first few words: DISCO's method is open to the criticism that it is too sensitive to chance in a case like:

: *ASTERISK

for teaching the word "asterisk" although generalisation and discrimination do provide a way of remedying accidents later. Once basic nouns are established, however, DISCO's learning is essentially discrete and structural, embodying the views of Minsky and Papert (1972, section 4.3) in this respect.

The most interesting phase in Harris's program is induction of a context-free grammar. One of these is equivalent (Hopcroft and Ullman, 1969, p.51) to a phrase structure grammar (Chomsky, 1957) which consists of a set of productions of the form $A \rightarrow BC$ and $A \rightarrow a$ where A, B and C are variables (e.g. <Noun Phrase> , <Adjective>) and "a" is a constant (i.e. a word in the language). According to Chomsky the grammar of a natural language is best described by the addition of a set of transformation rules to a phrase structure grammar. Transformations convert the deep structure to various surface forms. Harris gives the following examples.

1. A small chair is right-of the piano.
2. There is a small chair right-of the piano.
3. Right-of the piano there is a small chair.

The first sentence parses directly into a convenient form for semantic representation and the others are transformations. Harris primes his grammars with two productions (these names are related to others by the induction process).

<SENTENCE> → <SUBJECT> <PREDICATE>
<PREDICATE> → <VERB PHRASE> <MODIFYING PHRASE>

Semantic processing is endowed with the capability of permuting the components of the derived meaning.

Hence no transformations are learnt by the program - only a context-free grammar is induced. The program begins by writing a production that crudely corresponds to the form of the first example. It then applies two operators, grouping and folding, which make more general rules. Subsequent examples can be assimilated to these. Details are in his section 3.4.2. He is able to prove that repeated application of these operators can generate any phrase structure grammar (which one depends on the examples). Unfortunately an equivalent result for Turing machines would involve proving Church's thesis and so such a proof in DISCO's case is not available in the present state of automata theory (Hopcroft and Ullman, 1969, p.80). It would be necessary to show that the STM-LTM formulation has a semantics, as Scott (1970) has done for the lambda calculus, Gordon (1973) has done for LISP, and Kowalski and van Emden (1974) have done for the predicate calculus when viewed as a programming language. Anyway, even Harris's result does not prove that his program will produce them all. It should be pointed out that a context-free grammar is a grammar of type 2 and as such is two steps removed in power from a Turing machine, which is equivalent to a grammar of type 0 (type 1 is context-sensitive). DISCO's enhanced capability is therefore of considerable significance.

The most unsatisfactory aspect of Harris's work is that the built-in concepts have to be tagged with grammatical markers for the induction to be possible. This violates the requirement of linguistic universality which is prerequisite to acquisition (as was discussed in Chapter 2). For instance, Harris might set up a noun concept "light" and a verb concept "flash" but in the Hopi language

"The light flashed" is rendered by the single word "Reh-pi" (Whorf, 1956, p.viii) and the verb-noun classification breaks down. The difficulty is evidence of the undesirability of separating syntax and semantics at all. On this matter the present approach is closer to those of Schank (1973) and Wilks (1973) than of Winograd and Harris. The matter is taken up in section 9.3.

Meanwhile there remains Siklóssy's (1972) work on language acquisition. Inspired by books that teach foreign languages through pictures he wrote a program to accept paired sentences, one in Russian or German, the other in a formal language supposed to represent the pictures. The program initially stores every pair and searches the entire collection on each example, producing a sentence in the natural language when given a formal sentence. The longest sentence has four words in Russian, means "it is in the boy's hand", and is defined as (BE (IN(BOOK) HAND OF [BOY])). Since HAND and OF [BOY] are found in patterns stored during previous examples the program only needs to store information about the rest of the sentence. Although providing some economy in storage, this provision of embedded matching forces the program to search for all subpatterns of the input in all patterns and subpatterns in memory. It is clear that the program carries out only a simple rote learning task.

9.3 Natural Language Understanding and the Representation of Knowledge

A theory of natural language acquisition that is sufficiently precise and complete to be programmed on a computer necessarily entails a theory of the comprehension and utterance of natural language as well. As such, it bears comparison with programs embodying other theories and three of them are discussed here. The emphasis is entirely on comprehension, reflecting the emphasis in natural language research within Artificial Intelligence generally.

Similarly, a theory of learning entails a statement about the way knowledge is represented as a result of the acquisition process. Indeed the necessity of acquisition is a severe constraint upon the form that the representation should take. Happily that very severity has led to a formulation that has serious claims to generality, parsimony and efficiency.

Winograd's (1972) work is as important a statement about knowledge representation as it is a working theory of the process of understanding. The work of Schank (1973) is even more biased toward the former and he left the natural language processor to Riesbeck (1974). Winograd's thesis title: "Procedures as a representation for data in.." is made manifest in his work in at least four ways. The first is through the arrangement that the meaning of each word be a procedure to be executed during the process of interpretation. Riesbeck took this idea further and arranged that the process of interpreting a sentence was to proceed from left to right executing the procedure for each word as it was encountered by a supervisor which merely performed a few housekeeping functions. DISCO also works this way. Winograd's approach is less radical. His program SHRDLU possessed four distinct, albeit interrelated, components: syntax, semantics, world knowledge and utterance. The system of Schank, Goldman, Rieger and Riesbeck (1973) combines the first two. In fact they give the impression of trying to do without syntax at all and grudgingly admit as little of it as possible (see for instance Riesbeck (1974) p.87).

The second manifestation of procedural knowledge in Winograd's system is in the "semantic specialists", procedures charged with the responsibility of ascertaining the meaning of syntactic structures as soon as the presence of one has been tentatively confirmed

by the parser. These specialists invoke the meaning procedures of the words in the group or clause and attempt to construct a semantic structure. That also has a procedural interpretation (the third manifestation) and is converted to executable form in order to ascertain whether it is consistent with world knowledge. At any stage, failure can cause the program to back up to an earlier choice: perhaps to try an alternative meaning for a word or to get the parser to attempt an altogether different parsing.

Essentially, therefore, SHRDLU traverses a graph of possible interpretations, not considering meanings until the parser has had one of its hypotheses confirmed by the syntactic markers of the words in the group or clause and then continuing the search if the semantic processes fail. The thrust of Winograd's argument, and the fourth manifestation of procedural knowledge, is that instead of using general heuristics to guide the search, specific ad hoc abilities for the particular domain should be incorporated at every stage in the system. It goes beyond GPS (Newell, Shaw and Simon, 1960) because their domain specific differencing operations were numerical in output whereas Winograd often includes arbitrary choice procedures at decision points. The trade-off is between efficiency on his side and extensibility on theirs.

Like the work of Schank et al., and Wilks (1973), DISCO does not follow the paradigm of separable syntactic and semantic components with its consequent use of ad hoc methods to prune a combinatorial search space. Instead, Riesbeck (1974) in particular is followed in applying the idea of words as procedures to syntactic as well as semantic processing. In section 6.5., DISCO's use of STM and local contexts within the constraint of left to right retrieval from LTM was justified as a framework for a grammar by reference to Halliday's

(1960) general theory of grammar. (Notice that this theory is about grammar itself and is not to be confused with Halliday's systemic grammar as used by Winograd although obviously there are links).

Like DISCO, Riesbeck's natural language processor (a component of MARGIE (Schank et al., 1973)) also permits words to establish expectations for what follows during their interpretation. Of course, there is no question of learning in his system; the word-procedures are all specified in advance. He has not been able to use the same primitives for expectations and matching as are used for other parts of the MARGIE system. Whereas word-procedures in DISCO seek left by way of STM and seek right via LTM, Riesbeck's program needs special primitives for language processing such as CHOICE, CHOOSE, REPLACE and IMBED in order that word-procedures may construct a "conceptual graph" that represents expectations (and the final meaning). Subsequent word-procedures must then interrogate this graph for matches to their own (smaller) graphs.

DISCO's local contexts have three additional advantages over this conceptual graph formulation. One is that DISCO's matching can be done by means of an efficient direct accessing algorithm which is defined in Chapter 4 and which avoids the combinatorial explosion when the number of expectations is large. The second advantage is in the universal nature of procedural representation; it is shown below how that subsumes networks and graphs. The third is that Riesbeck's procedures must establish their expectations afresh on each application whereas DISCO's local contexts are stored and can be instantiated (and augmented) without any overhead.

9.3.1 The Frame

The last point above is exactly the difference between frames

and the use of demons explained by Charniak (1975). In his essay "A Framework for Representing Knowledge" Minsky (1975) refers to the description by Schank (1973) of Riesbeck's scheme of local expectations as an example of his rather elusive notion of a frame. Charniak is more precise in pointing out the advantage over demons (see Chapter 4, section 4.4 above). The present work goes further not only by implementing them (as Schank and Abelson (1975) have done in a different way) but by providing a means for them to be acquired through learning. (Schank and Abelson propose to synthesise scripts from plans but not by learning from experience). As was explained in Chapter 4, the LTM primitive is powerful enough to include stored contexts as a special case.

The terms "local context" or "stored context" are preferred to "frame" because Minsky seems to have in mind something much more pervasive when he uses that word. For example, on page 229 he likens Piaget's concrete operations to transformations between frames. In the spirit of DISCO, such operations would be procedures that act on other lower-level procedures, where lower-level means closer to the sensori-motor procedures of physical action and perception and higher-level procedures are those that transform them and thus represent differences between them.

DISCO's framework for representing knowledge fulfills two of Minsky's requirements: the first is speed (Minsky, 1975, p.215) which has already been elaborated and the second is provision for default assignments (p.228). Examples of the latter appeared in Chapter 7 where DISCO learnt to supply suitable objects for the numerals in the absence of a syntactic object.

The gulf between Minsky's formulation and that presented here is his preference for network representation. Less important is

the fact that most of Minsky's examples are of three-dimensional vision about which nothing has been said here. A proposal to represent line drawings to DISCO will appear in Chapter 10 but the state of the art in 3-D vision by computer is too primitive (Minsky's word, p.216) for an exact theory to be applied to it.

The network representation which Minsky prefers is subsumed by DISCO's procedural form. The proof is by construction. A node in a net can be represented by a procedure consisting of a sequence of assignments of subroutines (see Chapter 3) to a dummy variable. The first item assigned gives the type of the node and the rest of the procedure is a sequence of pairs of arc types and subprocedures. (They are the adjacent nodes because pointers are used). This proof also shows the disadvantage of the network point of view, namely that the node has no meaning unless acted upon by some outside entity. By contrast, a difference description for DISCO is a procedure that will join the extra portion back on to whatever is presented to it.

Minsky several times refers to Winston's ideas on representing differences between networks. However, Winston states that the content of a difference description is a skeleton containing a copy of the common portion of the two networks and a set of comparison notes (Winston, 1970, pp.103-5). The trouble is that he never says how the difference information could be removed from the copy of the original networks and so compared to similar differences from other examples in the way that DISCO's difference representation can. Indeed the nature of Winston's representation appears to make this kind of generalisation impossible.

The principal advantage of DISCO's procedural representation is that one only uses networks when they are actually needed because of the structure of the problem. Equally important is the advantage

that matching of atoms, which in a net is the exception (Minsky, 1975, p.250), is now the rule and therein lies the efficiency. A third advantage is that instead of defining an open-ended set of node and arc types one can leave the (procedural) nodes to define their own meaning in the programming language.

It is also possible to account in outline for the phenomenon of noticing connections that Minsky writes of on page 258. The theory would be that during the (possibly subconscious) process of recalling impressions (from LTM) the most recent ones would be in short-term memory where more elaborate matching is possible. The "inspiration" would then be to detect a match between two procedures with differing subroutines (with the generalised matcher, see Chapter 5, section 5.2.3). The quotation from Poincaré, reproduced here, captures this rather nicely: "Elements are so harmoniously disposed that the mind can embrace their totality while realising the details."

9.3.1.1 Clustering

This is quite distinct from the clustering described in the immediately preceding pages which is largely a technical device to reduce the problem of massive memory searches brought about by the network formulation. However, although DISCO does not need clustering, the ability to classify by function is a central feature which derives naturally from procedural representation.

Clusters are not necessary because there is a generalised concept of pointer (known as a key pointer) to which Minsky attaches little weight. Instead of all instances of a class possessing direct address pointers to a class node, the same effect is achieved simply, elegantly, economically and efficiently by a key with partial match and direct access. Moreover, the key-pointer is far more flexible

since new entities are automatically (i.e. effortlessly) classified in terms of existing knowledge. This requirement is crucial for learning.

A related point is Minsky's implied criticism (p.276) of those who try to explain behaviour "in terms of unstructured elementary fragments" which apparently (though not certainly) refers back to his references to production systems and CONNIVER (p.264) and might be applied also to the homogeneous appearance of DISCO's long-term memory. The point about key-word pointers should forestall such criticism. The memory possesses an elaborate potential structure which above all is highly sensitive to novelty.

9.3.1.2 Control

The final point to be made about frames leads on from production systems and CONNIVER to the proposal by Scott Fahlmann that is embedded in Minsky's essay (pp.264-7). DISCO's stored contexts meet his proposal in that any number of such "packets" can be active at one time and lower-level ones can override the effects of higher-level ones. Whether these are adequate to handle the medical and visual problems he suggests remains to be seen.

There is a particular difference in that he proposed that if a packet is active its contained packets should be active too. This would be rather pointless although there may be some confusion over the word "active". For DISCO, if an LTM record possesses a stored context (frame) then that context comes into effect when the record is activated. If now a record in that context is activated and it too has a frame (stored context) then that comes into effect at that moment and so on recursively. Viewed in this way, the global LTM is the highest frame and Fahlman's proposal, if the author understands it correctly, would then cause all frames to be active all the time

which is obviously not what is intended.

The missing capabilities in CONNIVER are

(a) to be able to restart a suspended process in an environment other than that in which it began; and

(b) to be able to instantiate a context in an environment other than the one in which it was created.

Capability (a) was present (in limited form) in PROCESS 1 (1972) and in full form in PROCESS 1.5 (Knapman, 1973). Capability (b) is present in DISCO: moreover it arises naturally out of the formulation of long-term memory.

Hewitt (1975) is also implementing a version of the frame idea. He proposes the notion of "world-directed invocation" whereby more than one assertion may contribute to a data base search. The LTM of DISCO could be described as giving "situation-directed invocation". Two procedures (which might sometimes be equivalent to assertions) can participate in LTM matching and retrieval (i.e. EXECPROC and SEARCHPROC).

A related criticism of languages like CONNIVER is the lack of explicit control (Minsky, 1975, p.264) which is due to the flexibility of the matching algorithm. The contextual mechanism, even in its elaboration to frames, does not answer this difficulty unless the matching is restricted in some way. DISCO seems to get the balance right: the kind of matching that relinquishes control (i.e. in LTM) is more restricted than the kind where control is retained (i.e. STM and generalisation matching).

9.3.2 Procedural Representation of Knowledge

The present work comes down firmly in favour of procedural representation remarking in Chapter 3 that statements in a formal language, and in the above discussion networks, may be subsumed to

the rather powerful construct of procedure as data structure expounded in the present work. So DISCO uses procedures not merely to limit the search tree like SHRDLU did but to eliminate it in principle.

It is illuminating to observe that neither Schank nor Wilks, both of whom are exponents of network representation, have attempted to construct programs which undertake question answering or imperative action as Winograd did. The MARGIE system produces paraphrases and inferences and Wilks's program translates into French. It is very hard to imagine them doing anything else. How could an imperative be obeyed by a computer other than by the construction of a procedure containing reference to the motor capabilities that are needed to carry them out?

Wilks employs some sixty primitive semantic units such as MAN, STUFF, FLOW, HOW, GOOD, IN, etc. Schank only propounds a set of primitive acts e.g. PTRANS (physical translation), MTRANS (transfer of mental state, i.e. communication), INGEST, etc., and has even used them in a frame construct (Schank and Abelson, 1975). Minsky (1975, p.246) casts doubt on the completeness of Schank's set, a doubt which should apply equally to those of Wilks and Winston (1970), and Minsky prefers the alternative of a very large collection of "primitives" with comments about how they are related.

That idea begs the question of how such knowledge primitives are "related to objects and processes in the world" (Lyons, 1970, p.166) which is just as important as "the way in which they are related to one another in terms of such notions as 'synonymy', 'entailment' and 'contradiction' " (ibid., p.166). This is where the procedural representation scores, as Hewitt (1975) also argues. The relation with the world can only be via mechanisms for action and perception.

In constructing a learning system one soon discovers that the only primitives available are the experiences and action capabilities of the individual. All conceptualisations must be manipulations of these. This view fully accords with Piaget's theory that the sensori-motor schemata are the primitives on which higher cognitive functioning is performed (see the discussion in Chapter 2). The hyphen between "sensori" and "motor" is expressed in DISCO by the initial conversion of sense data to procedural form for the sake of uniformity. This corresponds to Liberman's proprioceptive theory in the case of speech perception (Liberman, Cooper, Harris and MacNeilage, 1962). An idea for similarly processing visual information will appear in Chapter 10.

Notice that it is not necessary for both the active and passive aspects of all procedures to be exploited but in most cases they are.

CONCLUSION

Chapter 10

As indicated at the outset the work is far from complete. The medium-term objective is to teach such a computer program enough to allay all doubts about the advantages of the approach through learning compared to writing performance programs. To do this it would have to perform convincingly in at least two different domains. Naturally one would hope to go much further.

10.1 Directions for future research

There are three substantial omissions from DISCO's basic equipment which are apparent from a general consideration of its organisation without appeal to specific examples of the program's use. They have been mentioned before. One is the manner in which the Perception process (see section 6.1) synthesises procedures that do not use the STM and LTM primitives; another is the incompleteness of the rules for handling interracting or conflicting STM specifications where several processes contribute to an STM reference. The latter situation arises during the interpretation of sentences, when each word-procedure contributes something to the meaning as embodied in a reference to short-term memory (see Chapter 5). A third omission is the ability to verbalise process-like attributes of events being described in utterance.

10.1.1 Perception

In the course of perception, the program scans STM for another occurrence of each entity (as identified by a match in LTM) in the sense data (i.e. the input line). Where the match happens to be with the front of the remaining, unanalysed, portion of the input it detects this condition explicitly and sets up a repetition procedure

(CALL X; CALL X;) as the structured representation of the input. It would be a simple matter to extend that method to recognising more complex patterns; indeed the author has programmed such an ad hoc collection of capabilities without using STM at all.

The standard rules (Chapter 5) for synthesising procedures corresponding to all the different kinds of match that can arise in STM could readily be applied to the perception problem as well. The difficulty is that for some cases, like repetition, it is desirable to re-format the input to represent the newly discovered structure while in other cases, it is not.

An example is the teaching of the word "say"

: SAY DOT [DOT]

DOT

If the program were to substitute for DOT in the input line a procedure representing its occurrence in the printing activity then the reference to the word itself would be lost because the procedure representing the relationship would be independent of the actual word. There may well be a simple solution, viz. to propose a rule that if the "relationship" procedure does not contain the original input datum then do not treat it as something to be matched to LTM but retain the original. This rule takes care of repetition because the repeated item must be included in those procedures. However, the question remains of what to do with the relationship procedure. In fact, DISCO already contains an answer to that one: the Synthesiser builds the relationship expression only if there is no meaning result from the rest of the sentence as happens in the alternative example

: SAY SPLODGE [SPLODGE]

SPLODGE

where the word is not known by DISCO to have any meaning. Notice that the same issue is raised in puns. How do we note the relationship between the two similar words and still identify their meanings?

The pun also raises another issue which is that of context, since normally in a pun the word is used with different meanings. The system of stored contexts introduced for grammatical purposes (see section 6.5) copes with the problem of particular meanings being ascribed to pairs (or indeed n-tuples) of words but the expectation keys are sensitive to the meanings of the following words, not to their surface form. Thus a record in such a context may adapt a meaning to a particular purpose (as in the arithmetic examples) but it cannot cause a word to have a different meaning. For that, perceptual frames (implemented exactly like the others) are proposed which are capable of ascribing alternative meanings to words in context.

Implementing them presents no difficulty. The existing framework can handle it without modification. As to their acquisition, it seems possible that once a suitable way of representing perceptual relationships using STM is found, the complementary procedures derived by substituting LTM for STM should create the contexts in the same way that LTM references create the other variety of context (see Chapter 4, section 4.4). The same notion could prove helpful in visual scene analysis. The initial cues or one's own expectations would provide a context for the ensuing perceptions.

10.1.2 <u>Interracting STM specifications</u>

The second major area of investigation mentioned above is the problem of interacting STM specifications. A simple illustration is the tense conflict in the word "printed" between the present tense "print" (fundamentally imperative) and the past tense morpheme "-ed".

It is a simple case because, in generating the meaning of "-ed", the Synthesiser receives an imperative result on the left and a past tense result on the right (details in Chapter 7) and carries over the common parameters (i.e. everything except tense). In other words the tense information is overridden by the presence of "-ed" and, moreover, this is done by exactly the same process that acquires the meaning of other words such as "what" and "before".

The emphatic, and archaic, form of "you printed an asterisk" is "you did print an asterisk". This looks quite different to the Synthesiser if this time "did" is the morpheme to be learnt instead of "-ed". The reason is that "did" is now outside the phrase "print an asterisk" and so it will never receive the reference to the actual event located by the word-procedure for "asterisk" because this will have been masked by the imperative implication of "print". This time the Synthesiser only receives one result (from the right) which is an imperative. However it is able to detect that the program formed the imperative on its own initiative, so to speak, and so it investigates by searching STM again with only the SEARCHPROC argument specified. Thereafter, the override convention is adopted again.

That convention is adopted only when the discrepancy is solely one of time (i.e. involving SUBORDINATE and SUPERORDINATE arguments). Then it seems appropriate but when the other parameters conflict, vital information would be lost thereby. For example, reference to a nonsense word, as in "I said splodge" (I being the human tutor), can be correctly interpreted within the override convention because the word "splodge" has been given no meaning to the program. Hence the procedure for printing the word "splodge" is the SEARCHPROC argument to STM and the parameters laid down by the procedure for "said", causing reference to the Output Phase of the program, are re-

placed with others which refer to input from the teletypewriter when the word "I" is given rein. Consequently, the sentence "I said dot" would be misinterpreted as referring to an event where the tutor had entered the character "." rather than the word "dot" as intended.

An improved convention is suggested, namely that when there is a danger of loss of information the right-hand result should be assigned an imperative specification and executed in a special mode so that any output to the world (i.e. to the teletypewriter or the blackboard) should be trapped and absorbed into the SEARCHPROC of the left-hand result specification. Depending on the tenses, the latter should then be invoked against STM or transformed to LTM. The last part of that rule has been implemented and was used in the example

: SAY TRIPLE AFTER I PRINT THREE CHARACTERS

where the discrepancy arises between "I" and "print".

All the examples of this discrepancy problem so far encountered have arisen because of the pronoun "I" and it is quite possible that the wrong sort of induction is being made from these few cases. They are all instances of the problem of relating the actions of another to one's own actions, just as in transferring knowledge from comprehension of language to utterance or from seeing counting to doing it. That is what the principle of complementarity between STM and LTM is all about and it may be that the above-mentioned problems are better viewed in that light.

10.1.3 Utterances from Process-like Meanings

The third problem to be dealt with is that of using process-like attributes of a situation in the course of forming utterances. At the moment, the procedure referenced by an event is all that can be verbalised so, for instance, the program can utter "three dots" but not "printed dot". This is because the built-in procedure for

forming utterances (the Output Phase) only accepts procedures and so words like "say" or "what" assign the procedural component of the result they receive from the right to be indirect output (see Chapter 5). A conventional perception operation is then applied by the Output Phase to the supplied procedure in order to produce the utterance. Enhancing this facility to accept processes as well would not be as simple as it might seem. In the case of temporal relations ("before", "after", "-ed") the only way it could be done is by constructing procedures that represent the relation so that they would match the meaning procedures of the relevant words. That would require a different kind of LTM insertion when learning such words in order to allow matching with the text of a meaning procedure itself, as opposed to matching the LTM attributes specified by executing the meaning procedure in complementary mode. It also makes the process of uttering them much more like the process of learning them. No insuperable difficulty is envisaged in implementing such a scheme. In fact, it has already been done within a cruder framework (Knapman, 1974).

10.1.4 Technical Improvements

In addition to these three more or less substantial areas of investigation there are one or two technical matters deserving mention. DISCO is written in PROCESS 1.5 (Knapman, 1973) which is implemented as an interpreter written in the programming language POP_2. It would benefit from a re-write to make it faster and to compress the size of procedures (for matching purposes) to a minimum.

Another practical measure would be to rewrite some of DISCO's facilities in the base language (POP_2 or whatever) so that the amount of code executed interpretively is kept to a minimum. This would also save the STM search scanning the contents of working

storage variables. Alternatively that could be achieved by better use of subroutines within PROCESS 1.5. Because of the extreme generality of the process handling capabilities of this language, the tricks presented by Bobrow and Wegbreit (1972) and by Wegbreit (1975) for facilitating variable look-up and data base retrieval cannot be used by PROCESS 1.5, as was explained in Chapter 3. (Their formulation is not general enough to give rise to the PROCARG problem, the process analogue of the FUNARG problem).

In order to realise the full benefit of the theoretical efficiency of LTM and contexts, it will also be necessary to implement the direct access algorithm outlined in Chapter 4, taking due note of hardware configuration for large-scale application.

10.1.5 Vision

To support the claim that the framework used by DISCO is general enough for wider use a suggestion is given as to how visual information might be expressed in procedural form, exploiting both the structural and the active aspects of this representation. The primitive entities are taken to be lines. If sometimes more complex objects can be recognised innately (and there is evidence that they can - for example, reports identifying cells in a chimpanzees brain that fire only when a chimp's hand is in the field of view) that presents no problem because built-in procedures are handled just like synthesised ones. Certainly, mammals do have sets of nerve cells that each fire when a line of a particular orientation and length is being viewed (Pribram, 1971, p.126).

One method for representing lines is as a procedure to draw them. The LOGO programming language (Abelson, Goodman and Rudolph, 1973) intended for use by children immediately springs to mind. A drawing would be compiled as a sequence of FORWARD, TURN, PEN-UP and

PEN-DOWN instructions. This does not mean that a visual apparatus must actually traverse the scene in this fashion; merely that its output could be given in this form. A perception process based on the principles stated above would then perform recognition activities using LTM, contexts, procedural difference descriptions, and all the other methods that have been presented above.

There are several objections. No provision is made for colour or textural information although that could be inserted in the form of dummy assignments provided the hardware is capable of detecting it. More serious is the problem that LOGO primitives require numerical parameters (degrees of arc and units of distance). These would have to be replaced by a more flexible notation analogous to the frequency modulated pulse chains passed by neurons to the muscles (Pribram 1971, p.3ff. and p.74). For instance, if MOVE(A) means traverse a distance of half the visual field (say 0.5) then MOVE(AA) means 0.25, MOVE(A); MOVE(AAA) means 0.625 and so on. This idea could be extended to curves which would be represented by primitives XMOVE and YMOVE specifying the corresponding orthogonal components of the move at given times. A numerical representation closer to the form of the neuronal impulses might be preferable for curves but the logarithmic representation given above has the benefit that partial matches to an ordered LTM can be made because the fine-grain information comes last and the coarser front portion can be matched to a comparatively small set of records in LTM or in some frame (stored context). Hence a learning system could very quickly make crude distinctions and with more practice progressively refine them where needed.

It is difficult to see how the requirement for procedural representation can be ignored because, although the eye does not

apparently have to scan every line of a scene in order to encode it, there is no doubt that it is capable of scanning any particular line and therefore the necessary motor code must be available to it. Why therefore should nature bother, so to speak, to introduce another representation for perception, especially when procedures can again be used to describe (constructively and abstractly) the difference relations and higher conceptual knowledge as well?

10.1.6 Further work in the existing domain

Even within the limited domain in which the computer program now operates more work could be done on number. A study of negation could also be undertaken as well as teaching it the conditional word "if" which is closely related to negation. More utterances should be possible when the problems alluded to above concerning process-like attributes have been overcome. Dialogues could also be devised on the subject of time: telling the time of day and relating the passage of time measured on a clock to the temporal relations.

It is to be hoped that such things could be done without changing the program beyond solving the three substantial problems outlined above and carrying out minor debugging of a technical kind. In that ideal case, the object of trying further dialogues would be to produce a system capable of more and more useful things while at the same time giving insights into more and more advanced linguistic and conceptual processes. More realistically, the medium-term objective of trying further dialogue is to validate and revise the principles on which the program is grounded.

10.1.7 Social Behaviour and Practical Application

Looking now to the long term, the most obvious difficulty in extending the program's abilities to practical application is the choice of a suitable way of representing human social behaviour.

Even the simplest computer program in commercial application, probably the payroll, requires of its author considerable knowledge of taxation and people's working habits. One can inform a human programmer that the Chancellor of the Exchequer has abolished the earned income allowance and he will know how to change his program. A computer program that could accept information in that form would presumably be capable also of conducting a large part of the administration of the organisation - commercial, governmental or otherwise - of which it was a part.

Yet such a computer system would not appear to need any kind of visual or tactile capability, provided that there were some way in which it could experience human beings and their interactions and be enabled to grasp the elements of geography for coping with transportation and communications. The use of some kind of simulation would be expedient.

It is important to avoid confusion as to the purpose of such simulations, which would be to bypass the immense technical difficulty inherent in constructing a robot capable of anthropomorphic interaction with the world. The objective would be to enable a learning system to construct a model of its world (expressed as a collection of condition-action and condition-expectation pairs) in terms of its own activity and experience augmented by the fruits of linguistic interaction with a human tutor. The simulation must therefore be entirely external to the learning program with communication along well-defined paths. It is important to distinguish that from the kind of emulation exemplified by Stansfield's (1974) model of climate where he attempts to emulate the understanding of climate that might exist in the mind of a geography teacher or his pupil. Such a model should be the end product of an interaction between a

learning system and a simulation of events, coupled with teaching. The confusion between simulation of world and emulation of mind is nowhere more obvious than in Sussman's (1973) work where the world is simulated by preconditions and the know-how is represented by a library of routines specifying what to do when those preconditions fail. The connection between these two is unnaturally close and that is why this aspect of his work is more properly termed "acquisition of skill" than "learning" (see section 9.1.2 above).

It is conceivable that the simple scenarios of social interaction presented by Power (1974) and by Gullahorn and Gullahorn (1963) could be adapted to the present need but it is by no means clear how this should be done.

10.2 Eventual Use of a Learning System

Once simulations such as those just mentioned have been conducted the result should be a representation of knowledge and of ability which is versatile and extensible, compatible across many domains and organised on an efficient basis of "what to do if...". Once the constraints of the learning and extensibility requirements have been fully realised, there will be no need always to proceed *ab initio* if it appears easier to program a system to behave as if it had learnt. The advantages of parsimony and flexibility could still be retained.

Nearly all the effort of those who design computer software today - be it control software or applications of most kinds - is directed toward extending, modifying or integrating what is already there. That is where the learning approach wins hands down because a learning system is capable of accommodating new information in the light of what it already knows.

10.3 Some Principles of Artificial Learning

1. Learning underlies intelligence and is essential to it.
2. The importance of formulating general principles of learning, as opposed to *ad hoc* methods, is paramount.
3. An important component of learning is remembering and the achievement of an effective organisation for memory is a crucial step towards realising a system that learns and possesses cognitive abilities.
4. Memory can be divided into the short term and the long term. The most important reason for making the division is to provide for rapid access to a large body of information in LTM while allowing a more flexible kind of search and match in a limited area (STM).
5. An interface is necessary to facilitate reference to memory by the program. It turns out that a standard interface is possible for referring to both STM and LTM. The interface allows the program to express succinctly types of events.
6. Short term memory is a record of the most recent experience and behaviour (affects and effects). A call to the STM interface either verifies the recent occurrence of some event or causes a particular event to happen. In the latter case it is known as an imperative call.
7. Long term memory contains the accumulation of knowledge and ability. A call to the LTM interface creates a new record which consists of a suspended process which will be re-activated in the future whenever an event of the specified type occurs.
8. The memory interface is a universal programming language (equivalent to a Turing machine). LTM records are effectively

conditional statements; recursive calls are possible; and STM calls can cause the execution of other procedures. Moreover, LTM calls provide a means to create procedures, an essential requirement of an automatic programming system.

9. The mechanism of learning is to construct pairs of procedures, one referring to LTM and the other to STM. The principle is to relate the unspecified or unexplained attributes of two or three events in a situation. Typically this involves associating a word in language with its meaning or a cause with its effect, taking account of other factors such as the rest of the words in a sentence. So the LTM call will establish a suspended process to be re-activated whenever the program reads the new word in future.
10. Because of the standard interface, reference to STM and LTM in a procedure are complementary and can be interchanged. Thus the same procedure can be used for utterance as for comprehension and for causal prediction as for explanation *a posteriori*.
11. The LTM interface can create frames or stored contexts of expectations. These provide a vehicle for grammar and are probably also important in perception. The STM complement of such an LTM call is an imperative.
12. The learning program has five components: perception, causality, meaning, differentiation and grammar. All follow a similar pattern.
13. Differentiation, or discrimination, is the means whereby a procedure may be extended either conditionally or unconditionally. There is no theoretical limit to the extensions that can be made in the course of experience.
14. The opposite of differentiation is generalisation which is a method of relaxing conditions. Often overgeneralisation will occur and subsequent differentiation becomes appropriate.

DESCRIPTION OF PROCESS 1.5

Appendix 1

The following description adds to the detail given in Chapter 3. The first part deals with procedures and the second with processes.

A1.1 Procedures

Details are given of the format and types of instructions and the means provided for creating and modifying procedures.

A1.1.1 Basic Instruction Set

The primitive instruction types from which procedures are formed are the following.

CALL, RETURN, RUN, RISE, RRUN, RRISE, NOOP, ASSIGN and GOTO.
The primitives CALL, RUN and RRUN can all initiate execution of a procedure while RETURN, RISE and RRISE will suspend execution and transfer back to an earlier one. The control structure is generalised and details appear in section A1.2.

A LIFO stack is provided for passing arguments from one instruction to another and to subprocedures. Any primitive may be followed by a sequence of arguments. For example

CALL '*' PRINT;

causes '*' and PRINT to be written to the stack. The PRINT procedure will be executed and an asterisk printed. The primitive NOOP simply causes the arguments to be stacked. ASSIGN removes one item from the stack for each of its arguments, which must be variables, and causes the item to become the new value of the variable. As in most programming languages, a variable is a word (a group of characters) with which an item (its value) can be associated.

With a generalised control structure a branch instruction (GOTO)

is logically unnecessary but is nevertheless highly desirable, since the alternative is to fragment a procedure into smaller ones and this makes textual matching more difficult. The word following GOTO must appear in the set of labels belonging to the procedure; associated with the label will be a pointer to the instruction to which execution is to pass.

Any of the nine primitives may appear in the conditional form, indicated by the extra initial letters C- or NC-. C- indicates that the instruction is to be performed if and only if the value at the top of the stack is "true" and likewise NC- for "false". As in POP_2, "false" is represented by the integer 0 and anything else is "true".

An illustration may clarify these details.

```
CALL ARGUMENT 1 =;
NCGOTO  LOOP;
CALL '1' PRINT;
RETURN;
CALL ARGUMENT 1 >;
CCALL SUB;
CALL ARGUMENT 2 +;
ASSIGN ARGUMENT;
GOTO LOOP;
```

In the set of labels, LOOP is associated with the instruction

```
CALL ARGUMENT 1 >;
```

Talking through it line by line, if ARGUMENT equals one it prints one and returns to the caller. Otherwise it enters a loop (the GOTO at the end is what makes it a loop) wherein it will successively increment the argument by two until it exceeds one. At that point the procedure SUB is called or, more exactly, the execution is com-

menced of the item that is the value of the variable SUB.

A1.1.2 Variables

Variables are of two types: local and non-local. Local variables may be free or they may be for input or output. A procedure has associated with it an ordered set of free locals and two ordered sets of input and output locals whose values are assigned from or to the stack, respectively, at entry to or exit from the procedure. Apart from that, all the local variables behave in the same way. In simple cases, the behaviour of variables is as in LISP and POP_2. (The ordering of free locals is significant only for STM search: see Chapter 4).

A procedure also carries a set containing those variables which are referenced in it and are non-local. It has two purposes, as explained in section A1.2 when variable binding is discussed in relation to the control structure.

A1.1.3 Compatibility with POP_2.

As has been said, the interpreter executes procedures encoded in the above form. It establishes the control structure to be described in section A1.2. The other component of PROCESS 1.5 is the compiler. The internal form of procedures, while suitable for updating by a program and for textual matching is not convenient for people. The compiler will convert code written in a language based on POP_2 to the internal PROCESS 1.5 format.

It will, apart from one or two omissions, convert any POP_2 program. It will also accept the RUN, RISE, RRUN and RRISE primitives in the form specified in Knapman (1973), e.g.

RUN FUN (X,Y), 3 ;

In addition it will accept instructions in the PROCESS 1.5 internal form as illustrated above and these may be mixed with the other forms.

The POP_2 features not included are CANCEL and SECTION. POPVAL still calls the POP_2 compiler. There are one or two other minor shortcomings which could be corrected were there a demand.

A PROCESS 1.5 procedure may call a POP_2 function but not vice versa.

The compiler is written in POP_2 but could now be used to convert itself to the PROCESS 1.5 form should that prove desirable. A would-be implementer without POP_2 would therefore only have to re-write the interpreter which is comparatively small. Performance, however, would be poor.

A1.1.4 How Procedures are written

There are three ways to write procedures. The first can be done only by people and involves starting PROCESS 1.5 from a timesharing terminal and writing them in any mixture of the approved formats. The second way is to initialise a skeletal procedure by calling the primitive INITPROC and then to insert code by calling ADD with the text in any format supplied as an argument in a simple list (optionally nested). That method is open to both people and programs - as is the third, which is simply to use the basic operations for creating and updating the data structures of which a procedure is composed. The latter two methods may be mixed.

For instance, the following two are equivalent if entered at the top level (i.e. immediate execution) on a timesharing system. Here is the first.

```
VARS DISPL;
INITPROC ( ); ASSIGN DISPL;
ADD ([PRINT (DBL); RETURN]);
"DISPL" → PFNAME(DISPL);
```

The last line is just a nicety. The code is hybrid as can be seen. Below is the equivalent.

```
FUNCTION DISPL;
PRINT (DBL)
END;
```

The ADD method is far more flexible thereafter because it may be used repeatedly to append more text. (It can be done regardless of the manner in which the procedure was initialised.) There is also provision for editing and ADD then performs insertions instead of appending the text. Four standard variables CURRPROC, CURRCONT, CURRINST and CURRPOS control editing. When INITPROC is invoked it places a skeletal procedure in CURRPROC and there is another primitive EDIT of one argument that places a procedure in CURRPROC ready for changes. The pointers CURRCONT, CURRINST and CURRPOS govern the placing of the insertion when ADD is called. Normally they point to the end of the procedure but CURRINST can indicate any instruction and CURRPOS any position within it. CURRCONT points to the corresponding position in the list of structurally contained subroutines, which are not kept within the text itself. The text contains one marker for each subroutine in the list, indicating where each one is to be used in the procedure. The three pointers must be kept consistent.

Standard procedures are provided to position the pointers but they can also be accessed directly. The main ones are listed below.

FS : Find Start - positions at beginning of the procedure.

FINDN: Find next - moves to start of next instruction.

BN : Back Next - moves to end of previous instruction.

BNP : Back Next Position - moves to preceding position within current instruction or as BN if already at start.

DN: Delete Next - deletes the current instruction and positions at the next.

DNP: Delete Next Position - deletes the current item and positions at the next, thus shortening the current instruction by one place.

FP: Find Predicate - moves forward from current position seeking item that satisfies a predicate; returns truth value.

EDPOSITION: positions at current point of execution of procedure.

FP takes one argument which must be a procedure or a POP_2 function having one argument and one result (a truth value).

EDPOSITION takes one argument which should be the data structure (a process or state) that governs the execution of the procedure.

A1.2 Processes

PROCESS 1.5 implements Stansfield's régime for level numbering as well as a more flexible version. A comparison was given in section 3.2.2 but the exposition here deals exclusively with the more flexible version, known as relative level numbering.

A1.2.1 Control Structure

A procedure may be invoked by CALL or by RRUN. The effect of the difference between the two is only apparent when a procedure is to pass control back to a caller. It returns to its immediate caller by means of a RETURN instruction; in this case it matters not whether the invocation was by CALL or by RRUN. Alternatively, a procedure may return to an earlier state, bypassing the intermediate ones, by issuing RRISE n. The number n indicates a count of the number of RRUN invocations to be bypassed on the backward chain of all invocations (CALL or RRUN).

When a procedure invokes another, the state descriptor for the new process points back to that of the caller, which is known as the return state. The pointer can be one of two sorts: same-level or new-level. CALL will cause a same-level pointer and RRUN makes a new-level pointer. Thus in the course of time, a two-dimensional structure of return states emerges, as illustrated in fig. A1.1

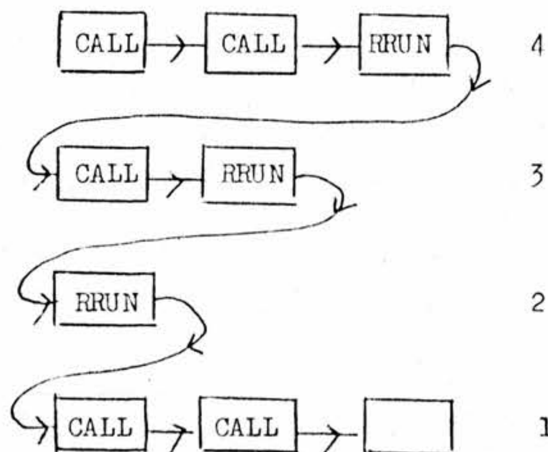


Fig. A1.1 A structure of return states

The numbering is relative to the empty box which denotes the execution of the current procedure. In order to pass back to its caller, that procedure could issue RETURN or RRISE 1; RRISE n would pass control directly back to the final state on level n.

At that point, the lower level states are not lost. The state descriptor for the returning process is placed in a standard global variable called CONTINUE and the user may save it by assigning it to another variable. The object saved bears the level number of the RRISE and is linked to all the intermediate states. It therefore represents the whole process below level n and it is called a process descriptor. It may be the argument to a RRUN instruction whereupon it will be resumed from the point of suspension.

The process descriptor is similar to the environment descriptor of Bobrow and Wegbreit while a state descriptor matches their "frame" and Sussman's frame in CONNIVER. Bobrow and Wegbreit's control structure, being more efficient, is not garbage collected and so the method of obtaining environment descriptors is different (using ENVIRON). They do not have a level structure, of course; return states are simply numbered consecutively from the current one.

PROCESS 1.5 has a refinement which often allows one to forget the numbering altogether. It is to define a routine to locate the level on which a particular procedure has been executed. A typical set-up is shown in fig. A1.2 which is a real example from the application program, involving interpretation of a procedure supplied in square parentheses as mentioned in Chapter 1.

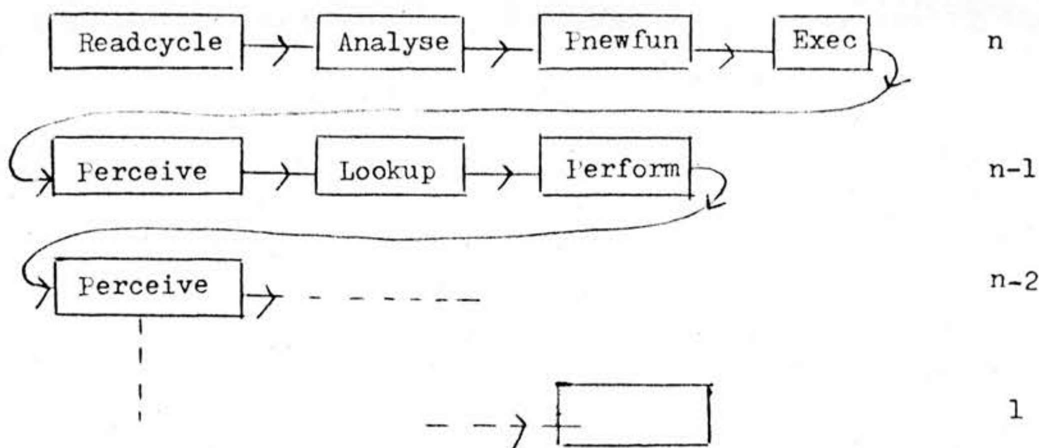


Fig. A1.2 RRISE to the level of Analyse

The need is not to return to "Analyse" itself but to "Exec". On other occasions, it may be directly to "Analyse" or to something else on the same level. The number of intervening levels is variable, depending on the nature of the input. In the returning procedure,

whose execution is represented by the empty box, one writes:

RRISE PFINDP(ANALYSE);

The user (of PROCESS 1.5) can easily write such facilities for himself. The state descriptors are implemented as ordinary data structures, just as procedures are, and nothing is hidden or inaccessible, including the cells that contain variable values and bindings.

A1.2.2 Variables

Bindings are carried explicitly for those variables mentioned as referenced non-locals in the set associated with the procedure. Compiled procedures have all such variables noted automatically except for those known to be global (i.e. not local to any procedure).

The value of a variable in a process may be accessed by the primitive VALUE and updated also. Thus

VALUE (PRA, "ABC")→VALUE(PRB, "XYZ");

will assign the value of "ABC" in the process PRA to "XYZ" in PRB. Similarly the primitive BIND may be used to access the value cell and update the binding of a variable in a process. This can only be done if the variable appears in the set of referenced globals of the procedure for the state descriptor that is the argument of BIND, whereas VALUE will perform a search of the return states if necessary to find the variable named.

The convention for variable binding is dynamic as in LISP and POP_2. There is a primitive FREEZE which is like FUNCTION in LISP (Moses, 1970). Optionally one can specify that only certain variables be frozen into a process other than the current one. The result of FREEZE is an initial state descriptor. Similar to the FUNARG problem there arises the "PROCARG" problem when a suspended process is to be restarted (using RRUN) in an environment other than its original one. To cope with this, the primitive FREEZEPROCESS

will bind all or some (if specified) of the variables in the process descriptor, which may consist of one or more states, into any specified process. Otherwise, the variables will be rebound dynamically at RRUN time if the environment has changed.

EXCERPT FROM THE SYNTHESISER

Appendix 2.

It may be helpful for some readers to see an excerpt from the actual coding of the Main Synthesiser in order better to understand its functioning. The following illustrates the kernel of the learning process. A procedure is being generated by means of the ADD function. A result has been received from a call of the STM interface and it happens to be a reference to a process. The result is held in the variables named FOUNDR and REFR, where REFR contains the name of an identifier within the process addressed by FOUNDR. The variable MSTM addresses the STM interface process. The Synthesiser interrogates the values of the arguments defined within MSTM and, for the undefined arguments, generates instructions that specify the appropriate value to MSTM. These instructions form part of the procedure that is to become, for instance, the meaning of a new word. The syntax is that of POP_2.

```
IF    VALUE(MSTM,[IDENTIFIER])= UNDEF   THEN
      ADD([%   """,REFR%]<>[->VALUE(MSTM,[IDENTIFIER])])
      CLOSE;
IF    VALUE(MSTM,[EXECPROC])= UNDEF
AND    REFR /=    EXECTOKEN    THEN
      PROCEDURE(FOUNDR) -> TEMP;
      IF    TEMP = NEWENTITY    THEN    NEWPROC -> TEMP    CLOSE;
      ADD([% TEMP %]<>[->VALUE(MSTM, [EXECPROC])])
      CLOSE;
IF    VALUE(MSTM,[SUPERORDINATE]) = UNDEF
AND    ISSUBORD(FOUNDR, PFINDPRO(CONTROLLER))  THEN
      ADD([PFINDPRO(CONTROLLER)
```

```
            →VALUE(MSTM,[SUPERORDINATE]))]) CLOSE;
    IF     VALUE(MSTM,[SUBORDINATE]) = UNDEF
    AND     ISSUBORD(PFINDPRO(CONTROLLER),FOUNDR)     THEN
          ADD([PFINDPRO(CONTROLLER)
            →VALUE(MSTM,[SUBORDINATE])]) CLOSE;
```

Legend:

UNDEF:	The value undefined.
[% %]:	List parentheses, the contents being evaluated.
[ ] :	List parentheses containing text.
< > :	Join of two lists.
VALUE:	The value of the variable named within square parentheses in the given process.
/=:	Not equal.
EXECTOKEN:	The execution token to the STM interface.
PROCEDURE:	The execution procedure of a process.
NEWPROC:	The procedure currently being created by the ADD function.
ISSUBORD:	True if the first argument is subordinate to the second; otherwise false.
PFINDPRO:	The first execution of the given procedure on the return chain.

REFERENCES

1. Abelson H, Goodman, N. and Rudolph, L., 1973. "LOGO Manual", LOGO Memo. 7, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass.

2. Anisfield, M. and Tucker, G.R., 1968. "English Pluralization Rules of Six-Year-Old children" Child Development, 38, pp 1201-17.

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

4. Bartlett, F.C., 1932. "Remembering: A study in experimental and social psychology". Cambridge University Press.

5. Becker, J.D., 1973. "A Model for the Encoding of Experiential Information" in Schank, R.C. and Colby, K.M. (ed's) "Computer Models of Thought and Language", San Francisco: W.A. Freeman & Co.

6. Bennet, R.W., 1975. "Proactive Interference in Short Term Memory: Fundamental Forgetting Processes". Journal of Verbal Learning and Verbal Behaviour, 14, 2, pp 123-144.

7. Berko, J. and Brown, R., 1960. "Psycholinguistic Research Methods" in Mussen, P.H. (ed.) "Handbook of Research Methods in Child Psychology", New York: Wiley, pp 517-557.

8. Bloom, L., 1970. "Language Development: Form and Function in Emerging Grammars". The MIT Press, Cambridge, Mass.

9. Bobrow, D.G., 1968. "Natural Language Input for a Computer Problem-Solving System", in Minsky, M.(ed.) "Semantic Information Processing", The MIT Press, Cambridge, Mass.

10. Bobrow, D.G., and Wegbreit, B., 1972. "A Model and Stack Implementation of Multiple Environments". BBN Report No. 2334, Bolt, Beranek and Newman Inc., Cambridge, Mass.

11. Bobrow, D.G., and Wegbreit, B., 1973. "A Model for Control Structures for Artificial Intelligence Programming Languages". Third International Joint Conference on Artificial Intelligence, Stanford University, Stanford, California.

12. Brown, R., "The First Sentences of Child and Chimpanzee". in Brown, R. (ed) "Psycholinguistics: Selected Papers". The Free Press (a Division of the Macmillan Co.)

13. Bruner, J.S., Goodnow, J.J., and Austin, G.A., 1956. "A Study of Thinking". Wiley.

14. Burstall, R.M., Collins, J.S., Popplestone, R.J., 1971. "Programming in POP2". Edinburgh University Press.

15. Burstall, R.M., and Darlington, J., 1976. "A Transformation System for Developing Recursive Programs". D.A.I. Research Report No. 19, Department of Artificial Intelligence, University of Edinburgh.

16. Caplan, D., 1972. "Clause boundaries and recognition latencies for words in sentences". Perception and Psycholinguistics, Vol. 12(1B), p. 73.

17. Charniak, E., 1972. "Toward a Model of Children's Story Comprehension". (Ph.D. Thesis) AI TR-266, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass.

18. Charniak, E., 1974. Unpublished Tutorial Lecture. AISB Summer Conference, University of Sussex, Brighton, Sussex.

19. Charniak, E., 1975. "Organization and Inference in a Frame-like System of Common Sense Knowledge", in Schank, R. and Nash-Webber, B. (ed's) "Theoretical Issues in Natural Language Processing: An Interdisciplinary Workshop", Cambridge, Mass. pp. 42-51.

20. Chomsky, N., 1957. "Syntactic Structures". The Hague: Mouton.

21. Chomsky, N., 1964. "Formal Discussion", in Bellugi, U. and Brown, R. (ed's) "The Acquisition of Language", University of Chicago Press, pp 35-39.

22. Chomsky, N., 1965. "Aspects of the Theory of Syntax". The MIT Press, Cambridge, Mass.

23. Chomsky, N., 1972. "Language and Mind". Enlarged Edition: Harcourt, Brace, Jovanovich Inc. New York.

24. Dahl, O-J., and Nygaard, 1966. "SIMULA - An ALGOL based Simulation Language". C.A.C.M. 9, pp 671-8.

25. Evans, T.G., 1968. "A Program for the Solution of Geometric-Analogy Intelligence Test Questions", in Minsky, M. (ed) "Semantic Information Processing", The MIT Press, Cambridge, Mass.

26. Feigenbaum, E.A., 1963. "The Simulation of Verbal Learning Behaviour", in Feigenbaum, E.A., and Feldman, J. (ed's) "Computers and Thought", New York: McGraw-Hill, pp 297-309.

27. Feldman, J.A., 1970. "Some Decidability Results on Grammatical Inference and Complexity". Stanford Artificial Intelligence Project. Memo. AIM 93.1, Stanford University.

28. Fikes, R.E., Hart, P.E. and Nilsson, N.J., 1972. "Learning and Executing Generalized Robot Plans". Artificial Intelligence 3, pp 251-288.

29. Flavell, J.H., 1963. "The Developmental Psychology of Jean Piaget". New York: Van Nostrand Reinhold Co.

30. Gardner, B.T., and Gardner R.A., 1971. "Two-way Communication with an Infant Chimpanzee", in Schrier, A., and Stollnitz, F. (ed's) "Behaviour in Nonhuman Primates", New York: Academic Press.

31. Goldstein, I.P. 1975. "MYCROFT: A System for Understanding Simple Picture Programs", Artificial Intelligence, Vol. 6, No. 4, pp 249-88.

32. Gordon, M., 1973. "Models of Pure LISP", Ph.D.Thesis, Dept. of Artificial Intelligence, University of Edinburgh.

33. Good, D.I., London, R.L., and Bledsoe, W.W., 1975. "An Interactive Verification System". IEEE Transactions on Software Engineering, 1, 59-67.

34. Gullahorn, J.J., and Gullahorn, J.E., 1963, in Feigenbaum, E.A., and Feldman, J., "Computers and Thought", New York: McGraw-Hill.

35. Halliday, M.A.K., 1961. "Categories of the Theory of Grammar". WORD, Vol. 17, No. 3, pp 241-292.

36. Halliday, M.A.K., 1975. "Learning how to Mean - Explorations in the Development of Language", London: Edward Arnold.

37. Hardy, S., 1974. "Automatic Induction of LISP functions". AISB Summer Conference, University of Sussex, Brighton, Sussex, pp 50-62.

38. Hardy, S., 1975. "Synthesis of LISP functions from examples". Fourth International Joint Conference on Artificial Intelligence, Tbilisi, U.S.S.R., pp 240-5.

39. Harris, L.R., 1972. "A Model for Adaptive Problem Solving applied to Natural Language Acquisition" (Ph.D.Thesis) TR 72-133, Department of Computer Science, Cornell University, New York.

40. Hayes, P., 1970. "A Logic of Actions", in Meltzer, B., and Michie, E. (ed's) "Machine Intelligence 6", Edinburgh University Press.

41. Hedrick, C.L., 1976. "Learning Production Systems from Examples". Artificial Intelligence 7, No. 1. pp 21-49.

42. Hewitt, C., 1972. "Description and Theoretical Analysis (using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot". (Ph.D. Thesis) AI TR-258, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass.

43. Hewitt, C., 1975. "How to use what you know". Fourth International Joint Conference on Artificial Intelligence, Tbilisi, U.S.S.R. pp 189-198.

44. Hewitt, C., Bishop, C., and Steiger, R., 1973. "A Universal Modular ACTOR Formalism for Artificial Intelligence". Third International Joint Conference on Artificial Intelligence, Stanford University, Stanford, California, pp 235-45.

45. Hopcroft, J.E., and Ullman, J.D., 1969. "Formal Languages and their Relation to Automata". Reading, Mass. Addison-Wesley.

46. Klahr, and Wallace, 1972. "Class Inclusion Processes", in Farnham-Diggory, S. (ed) "Information Processing in Children", New York and London: Academic Press, pp 143-72.

47. Kowalski, R. and van Emden, M.H., 1974. "The Semantics of Predicate Logic as a Programming Language", D.C.L. Memo no. 73, Dept. of Artificial Intelligence, University of Edinburgh.

48. Knapman, J.M. 1973. "Process 1.5: Description and User's Guide". Bionics Research Report No. 11, Department of Artificial Intelligence, University of Edinburgh.

49. Knapman, J.M., 1974. "Programs that write programs and know what they are doing". AISB Summer Conference, University of Sussex, Brighton, Sussex, pp 80-89.

50. Knapman, 1975. "Some Principles of Artificial Learning that have emerged from examples". Fourth International Joint Conference on Artificial Intelligence, Tbilisi, U.S.S.R. pp 253-9.

51. Ladefoged, P., 1959. "The Perception of Speech", Proc. Symp. Mechanisation of Thought Processes, London: Her Majesty's Stationery Office.

52. Lenneberg, E.H., 1964. "A Biological Perspective of Language", in Lenneberg, E.H. (ed) "New Directions in the Study of Language". The MIT Press, Cambridge, Mass.

53. Liberman, A.M., Cooper, F.S., Harris, K.S., and MacNeilage, P.F. 1962. "A motor theory of speech perception". Proc. Speech Communication Seminar - 1962 - II. Stockholm: Speech Transmission Laboratory, Royal Institute of Technology, 1963.

54. Lyons, J., 1970. Introduction to a paper by Manfred Bierwisch in Lyons, J. (ed) "New Horizons in Linguistics", Harmondsworth, Middlesex: Penguin.

55. McCarthy, J., Abrahams, P.W. Edwards, D.J. Hart, T.P. and Levin, M.I. 1962. "LISP 1.5 Programmer's Manual", MIT Press.

56. McCarthy, J., and Hayes, P., 1969. "Some philosophical problems from the standpoint of artificial intelligence", in Meltzer, B., and Michie, D. (ed's) "Machine Intelligence 4", Edinburgh University Press.

57. Manna, Z., and Waldinger, R., 1975. "Knowledge and Reasoning in Program Synthesis". Artificial Intelligence, 6, pp 175-208.

58. Michie, D., 1968. "Memo. functions and machine learning" Nature, 218, pp 19-22.

59. Michie, D., 1974. "A theory of evaluative comments in chess". AISB Summer Conference, University of Sussex, Brighton, Sussex.

60. Miller, G.A., 1956. "The Magical number seven, plus or minus two: some limits on our capacity for processing information", Psychological Review, 63, No. 2. pp 81-97.

61. Miller, G.A., Galanter, E., and Pribram, K.H., 1960. "Plans and the Structure of Behaviour". Holt, Rinehart and Winston, Inc.

62. Minsky, M., 1968. "Semantic Information Processing". The MIT Press, Cambridge, Mass.

63. Minsky, M., 1975. "A Framework for Representing Knowledge", in Winston, P.H. (ed) "The Psychology of Computer Vision", New York: McGraw-Hill.

64. Minsky, M., and Papert, S., 1969. "Perceptrons: An Introduction to Computational Geometry". The MIT Press, Cambridge, Mass.

65. Minsky, M., and Papert, S., 1972. "Artificial Intelligence: Progress Report". Memo No. 252, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass.

66. Moses, J., 1970. "The Function of FUNCTION in LISP", SIGSAM Bulletin, July 1970, pp 13-27.

67. Newell, A., 1972. "A Note on Process-Structure Distinctions in Developmental Psychology", in Farnham-Diggory, S. (ed) "Information Processing in Children", New York and London: Academic Press, pp 125-39.

68. Newell, A. 1973. "Artificial Intelligence and the Concept of Mind", in Schank, R.C., and Colby, K.M., (ed's) "Computer Models of Thought and Language", San Francisco: W.H. Freeman & Co.

69. Newell, A., Shaw, J.C., and Simon, H.A., 1960. "A Variety of Intelligent Learning in a General Problem Solver", in Yovits, M.C., and Cameron, S. (ed's), "Self-Organising Systems", New York: Pergamon Press.

70. Newell, A., and Simon, H.A., 1956. "The Logic Theory Machine: A Complex Information Processing System". IRE Transactions on Information Theory, Vol. 1 T-2, No. 3, pp 61-79.

71. Newell, A., and Simon, H.A., 1972. "Human Problem Solving" Englewood Cliffs, N.J. Prentice-Hall.

72. Newell, A., and Tonge, F.M., 1960. "An introduction to information processing language IPL-V". Communications of the ACM, 3, preprints of the Conference on Symbol Manipulation, pp 205-211.

73. Papert, S. "Process Models for Psychology", Nuffic.

74. Piaget, J., 1952a "A Child's Conception of Number". New York: Humanities.

75. Piaget, J., 1952b. "The Origins of Intelligence in Children" New York: Int. Univer. Press.

76. Piaget, J., 1971. "Structuralism" translated by C. Maschler, London: Routledge and Kegan Paul.

77. Popova, M.I., 1958. "Grammatical Elements of Language in the Speech of Pre-preschool Children", translated by Slobin, G., in Ferguson, C.A., and Slobin, D.I., 1973, "Studies of Child Language Development", Holt, Rinehart and Winston, Inc.

78. Power, R., 1974. "A Computer Model of Conversation", Ph.D. Thesis, University of Edinburgh.

79. Pribram, K.H., 1971. "Languages of the Brain". Prentice Hall, Englewood Cliffs, N.J.

80. Quillian, M.R., 1968. "Semantic Memory", in Minsky, M. (ed) "Semantic Information Processing", The MIT Press, Cambridge, Mass.

81. Riesbeck, C.K., 1974. "Computational Understanding: Analysis of Sentences and Context". Fondazione Dalle Molle per gli studi linguistici e di comunicazione internazionale.

82. Samuel, A.L., 1959. "Some studies in machine learning using the game of checkers". IBM Journal of Research and Development, Vol. 3, No. 3, pp 210-223.

83. Samuel, A.L., 1967. "Some studies in machine learning using the game of checkers", Part II. IBM Journal of Research and Development, Vol. 11, No. 4, pp 601-618.

84. Schank, R.C., 1973. "Identification of Conceptualizations Underlying Natural Language", in Schank, R.C., and Colby, K.M. (ed's) "Computer Models of Thought and Language", San Francisco: W.H. Freeman & Co.

85. Schank, R.C. and Abelson, R.P. 1975. "Scripts, Plans and Knowledge". Fourth International Joint Conference on Artificial Intelligence, Tbilisi, U.S.S.R. pp 151-7.

86. Schank, R.G. Goldman, N., Rieger, C.J., Riesbeck, C., 1973. "MARGIE: Memory, Analysis, Response, Generation, and Inference on English". Third International Joint Conference on Artificial Intelligence, Stanford University, California, pp 255-61.

87. Schwarcz, R.M., 1967. "Steps toward a Model of Linguistic Performance: A Preliminary Sketch". Mechanical Translation, Vol. 10, Nos. 3 and 4, pp 39-52.

88. Scott, D., 1970. "Outline of a Mathematical Theory of Computation", Proceedings of the Fourth Annual Conference on Information Science and Systems, Princeton, N.J., pp 169-76.

89. Shaw, D.E., Swartout, W.R., and Green, C.C., 1975. "Inferring LISP programs from examples". Fourth International Joint Conference on Artificial Intelligence, Tbilisi, U.S.S.R., pp 260-7

90. Siklossy, L., 1972. "Natural Language Learning by Computer" in Simon, H.A., and Siklossy, L. (ed's) "Representation and Meaning: Experiments with Information Processing Systems". Englewood Cliffs, N.J.: Prentice-Hall.

91. Smith, B., and Hewitt, C., 1974. "Towards a Programming Apprentice" AISB Summer Conference, University of Sussex, Brighton, Sussex.

92. Solomonoff, R.J., 1975. "Inductive Inference Theory - A Unified Approach to Problems in Pattern Recognition and Artificial Intelligence". Fourth International Joint Conference on Artificial Intelligence, Tbilisi, U.S.S.R. pp 274-280.

106. Winograd, T., 1975. "Frame Representations and The Declarative - Procedural Controversy" in Bobrow, D.G. and Collins, A. (ed's) "Representation and Understanding: Studies in Cognitive Science". New York: Academic Press.

107. Winston, P.H., 1970. "Learning Structural Descriptions from Examples". (Ph.D. Thesis) MAC TR-76, AD 713 988, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Mass.

108. Winston, P.H., 1975. "Learning Structural Destriptions from Examples" in Winston, P.H. (ed) "The Psychology of Computer Vision", New York: McGraw-Hill.

109. Winston, P.H. 1976. Personal Communication.

110. Woods, W.A., Kaplan, R.M., and Nash-Webber, B., 1972. "The Lunar Sciences Natural Language Information System: Final Report". BBN Report No. 2378, Bolt, Beranek and Newman, Inc. 50 Moulton Street, Cambridge, Mass.

93. Stansfield, J.L., 1972. "PROCESS 1: A Generalisation of Recursive Programming Languages". Bionics Research Report No. 8, Department of Artificial Intelligence, University of Edinburgh.

94. Stansfield, J.L., 1974. "Programming a Dialogue Teaching System". Bionics Research Report No. 25 (Ph.D. Thesis), Department of Artificial Intelligence, University of Edinburgh.

95. Sussman, G.J., 1973. "A Computational Model of Skill Acquisition" (Ph.D. Thesis) AI TR-297 Massachusetts Institute of Technology, Cambridge, Mass.

96. Sussman, G.J. and McDermott, D., 1972; revised 1974. "The CONNIVER Reference Manual", Artificial Intelligence Memos. Nos. 259 and 259a, Massachusetts Institute of Technology, Cambridge, Mass.

97. Sussman, G.J. Winograd, T., and Charniak, E., 1970. "Micro-Planner Reference Manual". AI Memo 203, Project MAC, Massachusetts Institute of Technology, Cambridge, Mass.

98. Tarski, A., 1956. "The Concept of Truth in Formalised Languages", translated by Woodger, J.N., Oxford: The Clarendon Press.

99. Waterman, D.A., 1970. "Generalization Learning Techniques for Automating the Learning of Heuristics". Artificial Intelligence, 1, pp 121-170.

100. Waterman, D.A., 1975. "Adaptive Production Systems". Fourth International Joint Conference on Artificial Intelligence, Tbilisi, U.S.S.R. pp 296-303.

101. Wegbreit, B., 1975. "Retrieval from Context Trees". Information Processing Letters, Vol. 3, No. 4, pp 119-20.

102. Wilks, Y., 1973. "An Artificial Intelligence Approach to Machine Translation" in Schank, R.C., and Colby, K.M. (ed's) "Computer Models of Thought and Language". San Francisco: W.H. Freeman & Co.

103. Winograd, T., 1969. "PROGRAMMAR: A Language for Writing Grammars". AI Memo 181, Project MAC, Massachusetts Institute of Technology, Cambridge, Mass.

104. Winograd, T., 1971. "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language". (Ph.D. Thesis). MAC TR-84, Department of Mathematics, Massachusetts Institute of Technology, Cambridge, Mass.

105. Winograd, T., 1972. "Understanding Natural Language", Edinburgh University Press.