A CONVERSATIONAL PROBLEM SOLVING SYSTEM REPRESENTED AS PROCEDURES WITH NATURAL LANGUAGE CONCEPTUAL STRUCTURE

Laurence Thomas Shafe

Ph.D. Thesis

Queen Mary College

University of London

1976

ABSTRACT

PIDGIN is a conversational computer programming language with a structure that facilitates the construction of computer systems that accept statements, answer questions and obey commands in natural language. It also incorporates a deductive problem-solving capability to enable such systems to solve non-trivial application problems. PIDGIN is intended to form a base for natural language problem-solving systems that can be used directly by the people with the problems, for example, designers, managers, engineers and scientists.

Because any system constructed using PIDGIN consists entirely of PIDGIN statements it may be conversationally updated to alter fundamentally its capabilities within the limit of the basic PIDGIN primitives. It also enables the system to answer questions about its own structure and workings and so assist the user to improve its capabilities. By working with the system in this way the user should be motivated to teach the system new heuristics for improving its performance.

The syntax of PIDGIN is based on the representation language developed by R. Schank and the semantics on the PLANNER language of C. Hewitt. PIDGIN incorporates some novel and powerful programming features such as success-failure backtracking; meaning-invoked rules; meaning restricted variables; the ability to specify the requirements and results of any command; the ability to generate programs automatically using this information; and the ability to generate automatically knowledge about the system's workings.

The design of PIDGIN has been worked out in detail and a subset of the language has been implemented using the programming language POP-2. The

limitations and possibilities of PIDGIN have been investigated by
working through the design of a chess endgame system, and the translation between English and PIDGIN has been investigated and PIDGIN equivalents for many semantically difficult English constructions have been worked out.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. Alan Bond and my former supervisor Professor Peter Landin for allowing me to develop the ideas in this thesis.

I am indebted and deeply grateful to Dr. Tom Westerdale for cultivating my initial enthusiasm and for helping me to unify my evolving ideas by long and painstaking discussion.

I am grateful to Professor Donald Michie for enabling me to visit the School of Artificial Intelligence at Edinburgh University and for suggesting the chess endgame problem. I would further like to thank Dr. Sou Tan for providing me with one of his chess endgame programs.

I am appreciative of the help given to me during 1974 by Dr. Eric Wagner and of the continual help and support maintained by Dr. Rod Macbeth, Dr. Ken MacCallum and David Mott.

Finally, I would like to thank all those authors whose books and articles have been the source and inspiration of my work, especially R. Schank, T. Winograd, W.V.O. Quine, L. Wittgenstein, G. Spencer Brown and P.M. Roget.

And Val, for putting up with it for so long.

CONTENTS

CHAPTER 1 INTRODUCTION

1.1. Outline

The world confronts us with a series of increasingly complex problems. Computers are helping us to solve these problems by taking over more and more of the mundane clerical work. Because of their speed, accuracy and efficiency they enable repetitious and tedious clerical work to be handled automatically. But computers can also be used to manipulate complex patterns and because of this have been used to model structures and control and optimize processes, all of which previously required skilled personnel. This is because much time and effort has gone into the precise, formal solution of each of these problems.

Research is being done into the way in which complex problems can be solved by computers. A major part of AI (Artificial Intelligence) research work has been involved with this type of investigation. One particular branch of the investigation is concerned with creating a computer system which understands natural languages, such as English. Achievements in this area would have many applications, for example:

i) If computers "understood" English they would become available to a far wider range of users, such as managers and designers, people without the time or inclination to learn a conventional programming language.

ii) People would be able to help computers solve difficult problems by interacting with them in some natural language.

iii) The linguistic nature of much information suggests many applications for computer programs that understand language. For example, information retrieval, index construction, machine translation, précis writing and report writing.

iv) Voice communication with computers would be of benefit in many applications especially if the person did not need to learn a special language.

v) Language itself is one of the most complex of human abilities and investigating its structure may help us understand more about the way the human brain works.

This thesis is concerned mostly with the first two points above.

It has long been known that the straightforward approach to the solution of complex problems, such as chess playing, immediately comes up against what is called the "combinatorial explosion". This is the uncontrollable increase in the time the computer must take to investigate all the possibilities of each new step. For example, if a chess program tried to examine ten different moves for 20 moves ahead, and each move takes one micro-second to analyse, then it would take about 10 million years to make a single move. One solution to this problem has been to find ways of rejecting most of the possibilities. For example, if only six moves ahead were considered the above chess program would need only one second to make a move. The rules used to limit the search are called "heuristics". Professor Sir James Lighthill (1973) states:

"It is important to understand the meaning attached to this adjective 'heuristic' which increasingly permeates the Artificial Intelligence literature: it means that the program stores and utilises a large amount of knowledge derived from human experience in solving the type of problem concerned."

It is clear then that the investigation of methods for improving the storage and utilisation of human knowledge is important to AI. The more human knowledge that can be incorporated in a program the more the combinatorial explosion can be curtailed and the better the program's performance will be. Thus it is important to find better ways of allowing humans to communicate their knowledge and experience to computer programs.

We are used to communicating this information to others using our natural language. When this type of information needs to be communicated to computers, however, it must first be translated into a computer language. If computers could be programmed to extract such, information from natural language then they could be taught the heuristics necessary to cut down the combinatorial explosion when solving complex problems.

One approach to this problem is the investigation of computer models of linguistic memory. M. Ross Quillian (1968) has proposed such a memory model with a number of attractive features. It is simply constructed with an appearance analogous to neural networks, yet it has a good inductive ability and ~ behaviour which corresponds with human linguistic behaviour in areas such as word association. However, Quillian's "semantic memory" is a static structure with no deductive or planning capabilities. After some initial attempts to construct a more comprehensive system based on S. Lamb's stratificational grammar plus networks analogous to those of Quillian, I abandoned this approach in favour of a procedure oriented system similar to that of T. Winograd(1971).

Winograd's system is a complete, working conversational system that understands English concerned with a simple "world" containing coloured blocks on a table, plus a crane for moving the blocks. The system will accept new information about its world, answer questions about the position of the blocks and what is has done to them and obey commands concerned with moving the blocks around. The idea of setting up a question-answering system to investigate the problems of language is not new, but the idea of precisely defining a "toy world" to restrict the language without otherwise arbitrary restraints and the methods he used to translate and store the assertions, questions and commands were first brought together in his system. His system translates English into a tree structure using a syntax analyser based on the ideas of systemic grammar developed by M.A.K. Halliday.

This structure is then translated into the theorem-proving language PLANNER. Because the English is translated into PLANNER the full power of this programming language is available to help it make complex deductions and create plans. However, Winograd's system is still rigidly limited to its toy-world and any extension to this would require substantial reprogramming. I hoped to simplify this problem by setting up a system that used the same language for the result of syntax analysis, the deductive program and programming the system.

C. Hewitt's PLANNER programming language (1972) is the deductive base and "deep structure" of Winograd's system but it was not designed for the latter purpose. It was designed as a programming language to enable people to write theorem-proving model-building goal-directed programs. G. Sussman and others have objected to the way that PLANNER leads the user into writing programs that thrash around for the solution and they have defined the language CONNIVER to overcome these objections by removing PLANNER's automatic backtrack control scheme and providing the user with more primitive operators. I have defined a language which takes another approach.

The reason that PLANNER programs thrash is that the system does not know what it is doing. One way to overcome this is to remove the automatic control and force the user to program it explicitly. The approach that I have taken is to provide the system with more information about what it is doing so that the automatic control can stop itself thrashing. This I did by basing a language upon the deep structures of natural language in the form proposed by R. Schank. The essential difference between this approach and that of PLANNER and CONNIVER is that the latter are programming languages designed for people whereas the former is an implementation language for conversational systems. An implementation language is a language designed to simplify the problem of implementing some other language, in this case English. It is not designed specifically as an easy language in which to program but as a language into which natural language may be easily translated. However, it is not just a static data-structure like Quillian's semantic memory or Schank's representation but a deductive goal-directed procedural language like PLANNER.

R. Schank has been refining a representation for the deep structure of natural language for a number of years, together with a translation scheme based on Conceptual Dependency theory. This theory was developed by Schank as a computer orientated approach to the problem of natural language analysis and translation. He claims the bulk of what people talk about can be reduced to his representation. It therefore seems like an ideal base upon which to build a conversational system. I planned to design a language with a PLANNER-like control structure" intelligently" directed by a data-base constrained to Schank-like deep structures. This initial language was called Conceptor. Conceptor was similar to PLANNER but the patterns were not arbitrary lists but conceptions. Conceptions were, roughly, one-dimensional representations of Schank's C-diagrams that is, unambiguous representations of the meaning of natural language sentences. The introduction of syntactic and semantic restrictions on the patterns of PLANNER meant that Conceptor had a meaning-directed data-base search, and a meaning-directed invocation of procedures, and this eliminated some of the thrashing of the system by restricting the matches found.

However, Conceptor was only a half-way stage. Although it had a data-base of conceptions the program that manipulated the data-base was separate from that data-base and still constructed from programming features such as labels, gotos, if-thens and loops rather than natural language features such as needs, schemes, states and plans. This meant that the relations between conceptions could not be manipulated by the program in the same way that the conceptions themselves could. Replacing these features involved abandoning the distinction between the data-base and the program and replacing the usual programming features by features related to natural language. Many advantages result from taking this step:

The system may answer questions concerned with its own workings. As the system can explain its current problem-solving rules to the user it becomes much easier to modify the system conversationally.

Because the problem-solving system is programmed in the same language that the English input is translated into a conversational user may extend and modify all parts of the system, not just the data-base.

The translation from natural language is simplified because the artificial step of translating into a programming language is removed.

The system may be conversationally extended to cope with new problems.

The generation of output is simplified because the system works in the deep structure of English. For example, all its plans and actions are expressed in this deep structure.

I have called the language incorporating these ideas PIDGIN. PIDGIN has been implemented using the programming language ABC, a language that was specially designed and developed for the task. PIDGIN statements are either conceptions, with a syntax based on Schank's conceptualizations, or relations between conceptions, called thoughts. The way in which conceptions can be related together is based on R. Abelson's (1973) description of how Schank's notation may be extended to build belief systems. However, Abelson does not consider how these relations may be made to realise a computer deductive problem-solving system. I have approached this problem by trying to implement a chess-endgame program called EIKASIA by combining chess board predicates and action schemes using relations similar to those discussed by Abelson. Before discussing the historical background to PIDGIN in more detail I would like to more clearly define PIDGIN as a computer language. The following types of language can be distinguished:

(1) Natural languages - used for communication 'between people for unlimited discourse. E.g. English, Chinese.

(2) Artificial languages - specialised formal languages used for precise description.

(a) Logic languages - used for precise human communication. E.g. predicate calculus.

(b) Computer languages - used for man-machine communication.

(i) Programming languages - used for algorithm communication. E.g. Algol, LISP, PLANNER

(ii) Representation languages - used for data or knowledge description. E.g. Schank's representation

(iii) Realisation languages - used for constructing knowledge algorithms, a combination of a programming and a representation language .e.g. PIDGIN

The above division is not meant to be exhaustive or definitive; for example, Esperanto is an artificial natural language, Algol is used for human communication. It is merely meant to show the difference between PIDGIN and other programming languages.

Throughout this thesis when I talk about natural language I am referring to that part of natural language that can be translated into Schank's notation, and thus PIDGIN. The extent of this is discussed by Schank (1969b). Similarly when English is referred to I mean that subset of English that can be handled by PIDGIN. Just how large this subset is discussed in Chapter 4 and suggested by the examples throughout the thesis.

This thesis consists' of five chapters numbered from 1 to 5 and three appendices, I to III. Chapters are divided into sections which are numbered and can be referred to by chapter.section number (e.g. 1.2, 3.2). Sections are divided into sub-sections which are numbered (e.g. 1.2.1, 2.3.1); both sections and sub-sections will be referred to as sections. Sections are divided into lettered divisions (e.g. A, B, C...) which are further divided 'into numbered parts (e.g. 1, 2, 3...). Parts are divided into lettered segments (e.g. a, b, c...) and these into pieces (e.g. (i), (ii), (iii) ...). If a sequence of short points is given in a section, part or division then piece numbering is used.

1.2 A History of Question-Answering Systems

Attempts to set down algorithms for translating English into logic have been made for hundreds of years. When computers were developed it was natural to consider using them for this purpose, but the first practical systems were not developed until about 1960. During the last fifteen years many such systems have been developed. There have been many different motives for developing these systems, for example, some are more concerned with the syntax analysis of English, some with setting up deductive systems, some with the problem of representing knowledge in computers, some with relating such systems to other problems such as picture processing or information retrieval, and others with building complete question-answering systems.

In the brief historical summary below a number of these projects are described, but although they are presented as separate self-contained units occurring at a particular time, it must be remembered that they all extended over a number of years and borrowed heavily from each other. The summary is extracted from a number of sources but mainly from the summaries of R.F. Simmons (1965, 1969), the Machine Intelligence series (annually from 1967), the Proceedings of the International Joint Conference on Artificial Intelligence (1969, 1971, and 1973) and the following three books:

Computers and Thought, ed. Feigenbaum and Feldman, 1963

Semantic Information Processing, ed. Minsky, 1968

Computer Models of Thought and Language, ed. Schank and Colby, 1973.

One of the earliest papers on question-answering systems was by McCarthy (1959) and in 1960 one of his students, A.V. Phillips, developed a working system. It was written in LISP and answered simple questions in a subset of English. The sentences were analysed by a simple context-free phrase structure analyser into five constituents - subject, verb, object, place and time; everything else was ignored. To answer a question it made a linear search through its data-base of stored sentences until a matching entry was found, which was then output. If no match was found the programs gave up without attempting to make any deductions. The system was in advance of some other systems developed at the time in that it could add new sentences to its data-base as it proceeded and thus it had a simple rote learning ability.

One of the systems with a fixed data-base was developed by B.F. Green (1961) at Lincoln Laboratories and was called BASEBALL as it could answer questions about the time, place, teams and score of all the American League baseball games for one complete season. Questions had to take the form of a single clause without logical connectives, negation or certain complex relations such as "most" and "highest". The question was translated into a specification list which was a pattern that could be matched against the built-in data-base to find the answer. The system's dictionary contained the part of speech and "meaning" of most of the words used to ask questions about baseball. There was no interaction with the user but it did show that with a limited field of discourse natural language questions could be analysed and answered.

J.L. Darlington (1963) developed a system, in COMIT, that took a slightly different approach to the problem. Darlington was interested in translating English into symbolic logic and by this means solving complex logical problems posed in English by using formal theorem provers. His system read in typical elementary problems in logic and translated them into statements in symbolic logic whose universal truth was then tested. The system was not interactive and the output was restricted to simple built-in phrases or single word replies.

In the same year R.K. Lindsay (1963) published a paper describing a memory structure called inferential memory. A program written in IPL-V called SAD-SAM (Sentence Appraiser and Diagrammer, and Semantic Ana1ysing Machine) accepted sentences in Basic English (Ogden 1933) from which it extracted information concerning kinship. This information was then added to one of the family trees the system maintained. Although the program was not interactive it was a simple matter to extract family relationships from the tree to answer questions. The system was heavily syntax orientated and made no use of the semantic information available in the family trees to help it to analyse the input. Instead it first tried to produce every possible parse tree and then it used the result to build up the family trees.

A. Newell and R.A. Simon had been interested in computer problem solving since their work on the logic theory machine (Newell 1957) and by 1963 they had developed a program called GPS (General Problem Solver) that tried to incorporate their theories on the pyscho1ogy of human thinking into a working problem solving program. Although it is not a question-answering system it contains many ideas that were later incorporated in the problem-solving part of many such systems.

B. Raphael (1968) developed a system called SIR (Semantic Information Retrieval), in 1964 that was one of the first truly interactive systems. If asked a question it could not answer directly it asked the user questions until it could deduce the answer. It accepted as input simple sentences in anyone of about twenty fixed formats useful for expressing relationships between objects. The relationships included set membership and inclusion, left-right position and ownership. They were stored in a semantic network in which nodes represented objects and labelled links the relationships. Raphael states as his primary interest the ability to store and utilize relational information and for this reason his system ignored the problem of analysing general English sentences.

In the same year D.G. Bobrow (1964) developed a system, called STUDENT that accepted and solved algebra problems posed in a restricted subset of English. The system was developed to investigate the problem of building interactive problem solvers. It could be "programmed" with a problem in English and would ask questions until it could solve the problem. The internal semantic model was based on one relationship (equality) and five basic arithmetic functions (from which others could. be constructed). It was more advanced than many other systems in that new transformations could be introduced into the running system. At General Electric F. Thompson (1966) and J. Craig (1966) developed a question-answering system called DEACON. Thompson investigated formal languages as a basis for a question-answering base language and Craig developed a working sys tem based on these ideas. This system used "rings" as the system's primitive data structure and had interpretation rules defined as programs that manipulated these rings. Unfortunately the project had to be abandoned through lack of funds.

Based largely on the ideas and theoretical work of McCarthy and his "Advice Taker" and Raphael's SIR, J .R. Slagle (1965) developed a deductive question-answering system called DEDUCOM (DEDUctive COMmunicator). It could answer a wide variety of questions and its deductive power could be increased by telling it more facts. However, the input had to be carefully prepared before giving it to DEDUCOM; the facts had to be in the right order, some redundant facts were required and some facts had to be carefully worded to enable the correct answer to be deduced.

In 1967 M.R. Quillian completed his thesis on semantic memory. As this will be described in Section 1.2.1 it will not be further discussed here.

In 1962 R.F. Simmons developed a system, in JOVIAL, called SYNTHEX. This formed the basis of the first Protosynthex system which through various stages of modification and extension became Protosynthex III (R.M. Schwarcz and Simmons, 1970). The original system answered questions using a children's encyclopedia. The text of the encyclopedia was stored on magnetic tape and analysed whenever a question was asked. The system analysed questions by separating function words ("the", "do", "is" etc.) from content words (nouns and verbs). The text was indexed on content words and the question's content words were used to retrieve all relevant sentences from the encyclopedia. The sentences retrieved were then reduced in number by comparing them more carefully with the question. The text of the encyclopedia was not pre-analysed except for the word index and so the system spent a large part of its time analysing the text. By 1970 the Protosynthex III system was a sophisticated deductive question-answering system containing a formalized data representation language based on triples, a translator to and from English and the formal language, and a powerful deductive and inference system.

The system contains many ideas incorporated in PIDGIN. However, it is not discussed in detail as it did not form part of the historical development of PIDGIN because I did not read a detailed description of Protosynthex until after completing the design of PIDGIN. It is thus interesting to note that the proposals made for removing some of the' limitations of Protosynthex III are those that form the basis of PIDGIN, namely that the deep structure was not deep enough and a structure based on Fillmore's case system was suggested; "how" and "why" questions could not be answered and a system for automatically adding to the data-base was suggested. Also the system ran into the combinatorial explosion with large data-bases and to try to solve this, a basic unit larger than the triple was suggested together with a "partitioned"

data-base. PIDGIN has a deeper deep structure based on Schank's
which in turn incorporated an improvement of Fillmore's case system. PIDGIN does automatically add information to its memory when it answers questions and makes plans and this information can be used to answer "how" and "why" questions. PIDGIN also incorporates the two suggestions for reducing the problem of the combinatorial explosion.

By 1969 Bursta1l and Ambler (Ambler 1969) had developed a system at the Department of Machine Intelligence and Perception, Edinburgh University, called QUAC (Question Answerer C). It was a deductive system that accepted sentences stating a relation between two objects and between other relations. New relations could also be defined in the system. It accepted sentences in restricted English and extracted their meaning, and it could also generate true sentences about the objects it had been told about. It was written to develop the ideas of Raphael (1968) but still suffered from a lack of generality. The internal model used by both became too specia1ised though Jinich (1971) developed the system in a number of interesting ways.

Vigor (1969), also at DMIP, developed a system which, although it was syntax rather than semantic based did adaptive1y improve its language capability by conversation. Initially the program has a dictionary containing about 100 words and six relations. From sentences read it increased the size of its vocabulary and the range of sentence forms handled. It was originally developed from a program called GASP. GASP was a hierarchy of subroutines that returned words in certain classes, e.g. noun-abstract. It was extended and combined with an English parser called SPUD (Bratley, 196$). SPUD was a dependency grammar parser. Words are divided into classes - binding (bound or loose), determinacy (determinate or recursive) and negative dependency (global or local), and these classes were used to determine where a word could occur in a sentence. The program reads in pieces of text to build up its memory structures and then randomly generates English sentences from these structures. The system is thus not really interactive. However, it was possible to "converse" with it in so far as when the sentence was output the user could score the result to modify the system's future behaviour.

In the last five years, from 1970, the work of Schank and Winograd is of most importance to the design of PIDGIN, and both of these are discussed in the next section. Schank's work has been associated to some extent with the belief systems developed by Colby (1969a, 1969b, 1973). Colby's system was originally closely related to J. Weizenbaum's ELIZA (1966) conversational system. Both of these systems attempted to carryon a conversation by always keeping the initiative that is by always asking questions and never answering them. They did this by recognising key words and phrases in the user's replies and then they used these to generate related questions. Colby went on to extend his system to incorporate his theories about neurotic human behaviour.

The above computer systems cover the major part of the work that has formed the basis of the structure of PIDGIN. However, a wide range of other work motivated the overall design philosophy behind PIDGIN. Of this other work the most notable is that done in the philosophy of language by L. Wittgenstein (1922), W.V.O. Quine (1960), M.J. Cresswell (1973), and R. Montague (1969, 1970 and 1973). This work attempts to describe a logical notation equivalent to natural language in order to explicate certain difficult problems that arise in natural languages. It is of most use in relation to PIDGIN in the way that certain semantic problems are brought out and discussed because it is important that question-answering systems do not fall into the associated traps. However, a logical notation is not a programming language and the solutions proposed are therefore difficult to assimilate directly; they can at best act only as guide lines.

Other related work has been done in the field of the psychology of language (Carol1 1964, Slobin 1971, Piaget 1929), problem solving (Po1ya 1945), human communication (Chapanis 1975), linguistics (Chomsky 1968, Lamb 1966), the physiology of the central nervous system (Hebb 1949, Geschwind 1973) and computer models of memory and cognition (Hunt 1973, Becker 1973).

1.2.1 Quillian's Semantic Memory

Quillian (1968) showed how knowledge might be stored in a single interconnected network that he called "semantic memory". It enables the association between semantically related concepts to be discovered. This provides a simple form of inference and is also used by the system to control the parsing of English into the network.

Semantic memory was later incorporated into a system called Teachable Language Comprehender (Quillian 1969) that translated a subset of English into the semantic memory representation. Teachable Language Comprehender, or TLC, regards each input sentence as a specification for assembling the parts referred to in the sentence. These parts are called units by Quillian and they correspond to what are called concepts in PIDGIN. TLC is not a question-answering system; it is a mechanism for assembling new units of semantic memory from English sentences. The simple syntax used by TLC during its analysis is secondary to the main mechanism for assembling new units from old. Thus the analysis is semantically rather than syntactically driven and it makes use of extensive implicit information. TLC is historically related to TEMPO (Thompson 1966) and SYNTHEX (Simmons 1962). It does not deal with a restricted environment like Bobrow (1964), Raphael (1968) or Winograd's (1971) system but attempts to handle general unrestricted English sentences. In many respects TLC is similar in scope and purpose to Schank's system (1969b).

The TLC memory consists of facts and form tests. A fact is either a unit or a property and a form test is a syntax routine that recognises simple phrases and specifies what action to take if found. A unit corresponds to a concept, noun phrase or sentence and in semantic memory it consists of a superset pointer plus zero or more delimiting properties. This idea of a concept being defined by a suitable restriction of a superset concept is also used by PIDGIN. A property is an attribute value pair plus zero or more refining properties. Units can also be modified by quantifiers and defined as sets of other units. The memory is thus one big interconnected network in which each unit is connected to its superset unit and, via properties, to modifying units.

Because the units are interconnected it is often possible to trace a path between two units along the superset and property pointers. Two units are related if a path exists between them and by considering the properties along that path they can be semantically related. So given any two units they can be semantically related by looking for a path between them. This is done in TLC by a breadth first or parallel search from each of the units. Each unit reached is "tagged" and the first common (intersecting) unit found gives the shortest path and thus the "closest" semantic link between them.

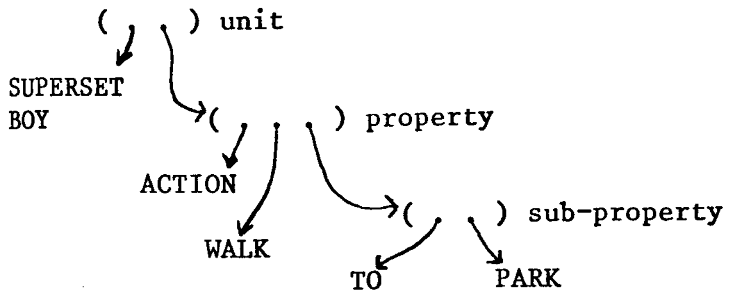
To analyse a piece of text each word in the text is associated with a list of all of the words' meanings plus possible anaphoric ("backward") references. A semantic link is then looked for between each of the meanings of the words close together in the text. If a link is found and a form test succeeds then the part of memory through which the link passes is copied as part of the new unit being generated. However, if any superset pointers are involved then the particular values are copied rather than the superset unit. This corresponds to the way that a question and statement may be matched in PIDGIN even though one involves concepts that are supersets of those in the other.

The form tests are introduced to check that the syntax agrees with the semantic link found. For example, though "lawyer t s wife" and "wife's lawyer" are both connected by the fact that a person may employ a lawyer the form tests associated with "employ" only allow the second syntactic arrangement to agree with such a semantic link. Each attribute and value in the system is associated with a set of form tests which pass or fail possible semantic links by checking the syntax (word order, inflexion, agreement and so on between the words involved). The form tests are applied one by one until one is found that succeeds. If a form test only fails because of intervening words it is provisionally succeeded and if the intervening words are all later incorporated in the analysis then the form test succeeds.

When the analysis is complete the original sentence will have been transformed into a single unit made up of copies of parts of the memory with new units substituted. The new unit created is linked into the complete semantic memory and can then be used to form part of a later new unit. TLC regards a sentence as being about the subject of the sentence. The subject forms the basis of the new unit and determines the superset link of the whole new unit. The subject is then modified by properties such as the verb group, qualifying clauses, adjectives and so on. Some of these properties are further modified, for example an adjective may be modified ("light blue") and a verb may be modified by an adverb ("walk quickly"). In PIDGIN the subject and verb are regarded as together forming the basis of the structure and each of these may be further modified as well as the combination of the two being modified.

TLC might generate the following unit from:

A boy is walking to the park.



Whereas PIDGIN would generate the conception

A BOY TRANSFER SELF A PLACE THE PARK.

TLC regards the sentence as being about "boy", in fact as specifying a new unit which is its old "boy" unit modified by the property of "walking", which is further modified by "in park". PIDGIN regards the sentence as being about "boy transferring" (explained later) and it handles the prepositional clause by incorporating it in with the particular case system it associates with the act TRANSFER. TLC handles the sentence in a very uniform manner. PIDGIN uses a more complex representation based upon Schank's work and also upon the requirements that a conception is a program statement that can be evaluated for its effect upon the system. The advantage of the more complex representation is mentioned by Simmons (see last section).

1.2.2 Schank's Dependency Representation

Schank developed Hay's conceptual dependency grammar into a linguistic theory organized from a computational point of view. Linguistic utterances are regarded as devices used by the utterer to guide the formation of a conceptual structure in the receiver by modifying and guiding expectations. Schank states that there exists a conceptual base into which utterances in natural language can be mapped. He proposes a syntax and semantics for such a conceptual base and describes mapping rules for generating conceptual structures from language utterances and vice-versa. The conceptual base is claimed to be language independent and meaning based, that is to say any sentences, in any language, with the same meaning, can be translated into the same conceptual structure and any two sentences with a different meaning can be translated into different structures. Further, it is claimed that a conceptual base containing a small, fixed number of "acts" is sufficient to allow the meaning of all the verbs in natural language to be expressed.

Schank's system will be described starting with his idea of a concept, how concepts may be combined and how utterances may be translated into such structures.

a. Concepts

The basic structure in the conceptual base is called a conceptualization. This consists of concepts and certain relations between the concepts. There are three types of concept, a nominal, an action, and a modifier. Nominals can be thought of by themselves without needing to relate to other concepts, i.e. a word that is the realization of a nominal concept tends to produce a picture of that real-world object in the hearer's mind. For this reason they are also called PPs (Picture Producers), for example, man, book, John and London are all PPs.

An action is what a PP can be said to be doing. Schank has reduced the actions required in the conceptual base to less than twenty. These actions are called ACTs and the reduction and rationalization of the ACTs is one of the most important features of Schank's system.

A modifier is a concept that makes no sense without the PP or ACT to which it relates. It describes the PP or ACT to which it relates ~d serves to specify an attribute of the nominal or action. Modifiers of nominals are called PAs (Picture Aiders) and of actions, AAs (Action Aiders).

In computer systems based on his theory Schank suggests a number of "files" of information that should be kept in order to produce a working system. One of these files is called the semantic information file; this contains information about which concepts may modify which other concepts. For example, the concept "pebble" might be associated with the information that it is, by definition, a physical object that is round in shape; usually smooth textured and may be modified by the attributes of colour, size and consistency. The semantic file is used to check concept combination during translation to help disambiguate a sentence and during generation to restrict the conceptualizations generated.

The concepts are language independent meaning units and to explain the differences that exist between the meanings of words in different languages Schank develops the ideas of under and over-naming. For example, "mare" is the over-named variant of "female horse" and "flat rectangular surface raised from the ground by four legs" is the under named variant of "table". He suggests that similarity can be reduced to equality, between PPs, in so far as two PPs refer to the same picture. For example, "mare" and "female horse" are the same concept as both refer to the same picture. So, if there are 40 words for different types of rice in one language and only one word in another then the language independent conceptual base for the first would have 40 named concepts for rice while the second would have just one. However, both systems would implicitly contain the same range of rice types because this depends not upon what concepts are named but on the allowed concept modifiers and the allowed concept combinations. Thus if both systems contain the information that rice can have texture, and possible textures are smooth, gritty, course and rough, then both contain the implicit concepts "smooth rice", "gritty rice" and so on. By including all the possible modifiers of rice possibly hundreds of implicit rice concepts are contained in both systems. Most of these will be unnamed, for example:

"Very small, slightly gritty, soft, yellow rice".

Of course different particular conceptual bases differ in the range and size of their information files but this is the essence of a conversational system, i.e., one that learns new concepts and increases its range of conceptual knowledge through discourse.

b. Conceptualizations

The conceptual categories (PP, ACT, PA and AA) can relate in specified ways to each other. These relations are called dependencies and are the conceptual analogue of the syntactic dependencies described by Hays and others. A dependency relation between two concepts indicates that the dependent concept predicts the governing concept. A dependent must have a governor. The fundamental base of any conceptualization (except for certain special cases) is what is called a two-way dependency link between the main PP and ACT. From these two inter-dependent concepts all the other dependent concepts of the conceptualization hang. Associated with the act will be a case dependency which can be either objective, recipient, directive or instrumental and will link one or more PPs to the ACT as conceptual cases. Any PP may be modified by a PA or by the relation of containment, location, or possession to another PP.

The conceptual act cases are the basic predictive mechanism available to the conceptual processor, and dialogues are often partly concerned with filling in the case slots in a conceptualization.

Conceptualizations can relate to other conceptualizations by dependencies called conceptual relations. The most important of these is causality as this is present in the conceptual structure representation of many natural language utterances in which causality is not explicitly mentioned. For example, verbs such as "kill", "fly", "comfort", and "prevent" in English.

Schank claims that his conceptualizations, with their small fixed number of ACTs, are sufficient to represent adequately the information underlying English verbs. Since there are thousands of verbs and only a few ACTs this amounts to a tremendous saving, especially when considering the inference rules that must be associated with each one.

c. Natural Language Analysis

The translation of natural language into conceptualizations, as described by Schank, takes place at two levels, the conceptual and the sentential. The conceptual level is concerned with creating syntactically well-formed conceptualizations incorporating all the information from the utterance and the sentential level is concerned with the particular syntax rules of the natural language being translated. The words of the utterance are used as indicators of the top-level structure to be searched for. The translator thus combines a bottom-up with a top-down parser and avoids spending a great deal of time looking for combinations of syntactic categories that do not exist and looking to see if what is wanted at the conceptual level is present at the sentential.

At any point in a translation the system will be working at a number of levels of prediction:

i) Syntax: at the conceptual level the rules determine what concepts are required and at the sentential level what part of speech is allowed at any point.

ii) Context: the preceding part of the sentence and the preceding sentences limit what is expected at any point in terms of what particular verbs or nouns are most likely. A similar prediction can be made at the conceptual level.

iii) Conversational: because people talk for a reason and because the hearer is usually aware of the reason it is possible for prediction to be made about large parts of sentences. This fact is usually taken into account by the person speaking who will truncate an otherwise lengthy utterance.

iv) World view: as well as information associated with a particular conversation, each individual also has knowledge of the situation within his entire view of the world. This includes the speaker's knowledge about the hearer and vice-versa. This information is used by hearers in order to predict what is likely at any point in an utterance and by speakers to shape and control their utterance.

All this information and more should be used by a natural language translator to help it predict and thus disambiguate an utterance and also to enable it to fill in any information assumed and omitted by the speaker. This is done by Schank's translator by making use of the system's memory of past conceptualizations and rules about likely intentions and behaviour.

The translator works by using surface level heuristics to help it find the main verb and the subject. From this a verb-ACT dictionary allows the system to set up a rudimentary conceptual structure which can be used to direct further analysis.

d. Conclusions

Schank's conceptual dependency framework provides a good base for holding the meaning, or deep structure, of natural language utterances in a computer conversational system. However, Schank's notation was not designed to be interpreted and therefore suffers from some of the disadvantages claimed by Winograd to exist with such static systems. The setting up of PIDGIN attempts to overcome this disadvantage by defining a language plus an interpreter so that utterances can be translated into procedures and thus take advantage of the benefits of doing this as described by Winograd (1970).

The main advantage of Schank's scheme is the rationalization of conceptual actions and their case dependencies. Reducing the number of actions to below twenty allows a conversational system to be able to incorporate all the basic actions and their associated 1nference rules in an initial system and conversationally extend this base to include more nominal concepts and more sentential level verbs and nouns.

Unfortunately the interpretation of Schank's structures is not well defined and there are a number of omissions. PIDGIN attempts to rectify some of these faults, first by defining its syntax clearly, then by defining an interpreter and lastly by unifying and extending the possible structures. Some of the main features of PIDGIN omitted by Schank are the clear treatment of structures above the conceptualization (see Abelson, Section 1.2.4), a full treatment of relations and properties, the inclusion of arithmetic expressions, a pattern matching ability, the consistent treatment of quantifiers including scope rules, and the deeper analysis of concepts.

Schank's ideas have been implemented as a system called MARGIE (Memory, Analysis, Response Generation and Inferences on English, Schank 1973c), which includes an ana1yser that translates English into conceptualizations, and a generator that translates conceptualizations back into English (Goldman, 1975).

The existence of such an implementation obviates the need for a PIDGIN translator implementation because there is a simple syntactic correspondence between Schank's conceptualizations and PIDGIN statements. Of course, in order to demonstrate those features of PIDGIN not available in Schank's system a complete translator for PIDGIN would be an extension of Schank's translator but for a large part of the facilities it could be essentially the same. However, because of the differences the design of a PIDGIN translator has been described in Chapter 4.

1.2.3 Winograd's Procedural Deep Structure

The greatest influence in AI in the last few years has undoubtedly been the system designed and developed by T. Winograd. He brought together many different techniques and implemented an impressive system that could remember information, answer questions "and obey orders concerning a simulated "toy world" of co1oured blocks, boxes and pyramids on a table top.

Winograd's system uses a parser that is based on the ideas developed in Halliday's systemic grammar (1970). The parser is a program in a language called PROGRAMMER, a language developed by Winograd specifically for the task. The parser does not work independently of the rest of the system but can call upon any of the other parts to check what it is doing as it goes along. For example, when each phrase is formed complex deductions can be performed to check that the phrase is correct before continuing.

Transformational grammar deals with a context-free base grammar over which transformation rules operate to produce the actual "surface structure". Systemic grammar works with a context-free tree but each node of this tree may be associated with "features". These features are, for example, "transitive", "question", or "major" and by their use the context dependent aspects of natural language can be handled in an economical way. Further, there is a high degree of correlation between these features and the semantic interpretation of the constituents that exhibit them. The parser extracts the features by the use of special routines that check agreement, word endings, word order and so on. Context sensitive aspects are handled by special checks, for example, the verb phrase routine might climb back along the parsing tree to find the main subject in order to check that the subject and verb endings are either both singular or both plural.

As the parse tree is built up semantic routines are called to generate the PLANNER program that is equivalent to the parse tree produced. PLANNER is a programming language developed by C. Hewitt (1971, 1972) to enable goal-directed theorem-proving programs to be simply written. For example, if the noun group "a red cube" has been parsed the tree formed is passed to the noun group semantic routine to produce the corresponding Micro-Planner (the implementation of PLANNER used by Winograd, see G.J. Sussman 1970) statements:

```
(THGOAL (#IS $?Xl #BLOCK))
(#EQDIM $?Xl)
(THGOAL (#COLOR $?Xl #RED))
```

The above program would fail of there were no blocks of equal dimensions and coloured red, otherwise it would succeed with the variable Xl bound to the first red cube found in the data-base. The above piece of program would form part of the complete procedure generated for a sentence. For example, the question "How many red cubes are there" could use the above program repeatedly to find every red cube and at the same time keep a count of the number found. The imperative "Put a red cube in the box" could use the same program to find a red cube and then it could be moved into the box by a further piece of PLANNER program. The assertion "I like red cubes" could generate the theorem

```
(#LIKE $?X)
```

with a body which is the same as the above program. This would fail if the object given to the theorem were not a red cube (e.g. (#LIKE :PYRAMID)), otherwise it would succeed.

Winograd has taken the sophisticated data-base theorem-proving programming language PLANNER as the structure in which the meaning of a sentence is represented. This structure is not a static tree structure but an actual PLANNER program. By representing meanings as procedures Winograd argues that greater power and flexibility is obtained than with any static representation such as that of Schank. However, the advantage of Schank's representation is that it was designed around the conceptual base of natural language whereas PLANNER was designed as a theorem-proving language. It was to try to combine the advantages of both these systems that PIDGIN was developed.

Winograd developed the powerful parsing language, PROGRAMMAR, based on systemic grammar theory, in order to write a parser that could produce a parse tree for any meaningful English sentence in the vocabulary used. This parse tree then had to be translated into a PLANNER procedure. By designing the meaning structure to incorporate the underlying rules of natural language the parser can be driven by the syntax of the meaning structure and, further, this meaning structure can be built up as the parsing proceeds, word by word.

Most question-answering systems, including Winograd's, generate output by a form of "slot-and-fi11er" technique. The system contains various skeleton sentences into which the answer or query is put. Winograd's system uses a number of special rules to improve the form of the output by the use of pronouns and word endings. Each type of question has a model output statement into which the answer is put. There are also a number of built-in error messages to reject questions that cannot be answered, and ambiguous and other ill-formed sentences. The advantage of Schank's scheme is that every possible legal conceptualization can be mapped into an English sentence, whereas in Winograd's system there is no attempt to map PLANNER back into English. Schank discusses how random sentences may be generated within his system and he goes on to say that to generate conversation it is necessary to introduce motivation into the choices made. However, he does not describe how this might be done.

PIDGIN generates a conception as an answer to a question, and this conception is then mapped back into English. Consider an example to compare Winograd's use of PLANNER with PIDGIN. The question:-

What is in the box?

will be parsed by Winograd's system to form a parse tree which will then be translated into a PLANNER procedure that will contain a statement of the form:

(THGOAL (#IN :BOX $?Xl))

The object found will then be slotted into a standard reply suitably modified to distinguish it from all the other objects. If no object was found then a suitable "error" message would be output.

PIDGIN would analyse the question into:

AN OBJECT <IN BOX> ?

and this question would then be evaluated to return say:

PYRAMID <IN BOX>

This then forms the basis of the reply.

PIDGIN can be described by analogy with certain features of PLANNER. In PLANNER there are a large number of primitives that manipulate arbitrary patterns stored in a data-base. One of these primitives is "GOAL" (or "THGOAL" in Micro-Planner). This primitive will succeed if the pattern which is its argument matches a pattern in the data-base or if the pattern matches a consequent theorem and then that theorem succeeds. This is the essence of the evaluation of PIDGIN. Every PIDGIN statement either succeeds or fails when it is evaluated. If it matches a statement directly or if it matches an IF-rule that succeeds then it succeeds; otherwise it fails. PIDGIN statements can be conjoined or disjoined with other statements to form programs in which the flow of control is determined by the success or failure of its individual statements. A complete description of the evaluation of a PIDGIN statement is given later but the above summary serves as a useful guide to the overall flow of control.

In conclusion, Winograd's system is a PROGRAMMAR program that translates English sentences into PLANNER procedures that manipulate a data-base which is a description of the simulated toy-world. A PIDGIN conversational system is a PIDGIN program that translates English sentences into PIDGIN statements that manipulate other PIDGIN statements that together form a description of the world.

1.2.4 Abelson's Belief Structures

Whereas Schank describes the details of the structure of conceptual dependencies, Abelson considers the relationships between them. Abelson is interested in how conceptualizations can be put together to form larger coherent systems. Schemes such as Schank's and Winograd's are sometimes called "knowledge systems" because there can be little argument over the truth or falsity of the facts stored within the system. However, when relationships between conceptualizations, such as purpose and cause, are considered, disagreement may arise. Such systems are therefore called "belief systems".

Abelson distinguishes between three types of conceptualization, which he calls "atoms", i.e. action atoms, state atoms, and purpose atoms. He describes how these may be put together to form "molecules" and how these can be assembled into "plans". Plans of two individuals can be related as "themes" and these can be built up into complete ideologies or "scripts".

For the purpose of building a problem-solving system the complete range of Abelson's work is not required. However, his technique for assembling atoms into plans is directly applicable to building a chess playing program and his higher level techniques for building themes and scripts could be applied to constructing a chess program that played with purpose to achieve certain states by particular actions and which could play with say aggression or care (themes) or possibly in a way that was related to the opponent's style. It must be remembered that although the chess problem was mentioned above and this particular problem is considered in more detail later it is not being studied as m end in itself but only in so far as it is a well defined problem that is difficult to solve. This means that although the ideas presented below are later applied to a particular chess endgame problem they were added to PIDGIN with more general aims in view.

A brief summary of Abelson's terms and techniques will be given and then related to the specific requirements of a PDIGIN chess system in order to show how they may be used in a practical system. Abelson bases his work on Schank's Conceptual Dependency Analysis with minor changes to the notation for the sake of convenience within Abelson's overall system. Abelson calls conceptualizations, "atoms" and distinguishes three types, A (action) atoms, which are Schank's PP-ACT conceptualizations, S (state) atoms which are Schank's attributive PP-PP or PP-PA conceptualizations and P (purpose) atoms which have the Schank representation:

Actor <=> **want**

ℵ

Actor <=> Poss (X)

These atoms can be assembled into plans. The simplest type of 'plan is called a molecule and it has the form:

P - A - S

where the S-atom is the state connected to the "want" of the P-atom, the A-atom is causally connected to the S-atom and the actor in the A-atom is an agent of the actor in the P-atom. A molecule captures the idea of an action undertaken in order to attain a goal desired by the sponsor of the action.

With each type of actor-action conceptualization of the "TRANS" class (as described by Schank, 1973b) Abelson lists those states that must hold before or while the action is performed. For example, before A can take X from B using Y three states must hold, B must possess X, A must have access to Y, and A, X and Y must be in close proximity. These conditions can be simplified, for example, if the instrument Y is A's hand and the object X is taken from a place rather than a person then the only condition is that A and X are in close proximity.

The simple P-A-S molecule can be extended to form a "serial plan" (or "chain") by interposing alternating A and S atoms following the initial P-atom. Then the final S-atom must be the "want" state of the P-atom, each A-atom must be causally bonded to the following S-atom, the actor in each A-atom must be either the actor in the initial P-atom or the agent of the actor in the preceding A-atom, and each S-atom must "enable" the following A-atom. Networks may be built up to form complex plans because an action may require a number of states before it is enabled and one state may enable a number of actions. Abelson distinguishes fourteen types of linkage between atoms in a network:

i) P --- A	Purposive action	The action A serves the purpose P.
ii) A --- S	Casual linkage	Action A causes the outcome S intended by the actor.
iii) S --- A	Enablement	The state S enables action A.
iv) $\begin{tikzcd} S_1 \arrow[r] & A \\ S_n \arrow[ur, dash] & \end{tikzcd}$	Multiple enablers	All of states $S_1 ... S_n$ are required to enable action A.
v) $\begin{tikzcd} & A_1 \arrow[d, dotted, dash] \\ S \arrow[ur, dash] \arrow[r, dash] \arrow[dr, dash] & {} \\ & A_n \end{tikzcd}$	Multiple enablement	State S enables all of the act1ons $A_1 ... A_n$.
vi) $\begin{tikzcd} & A_1 \arrow[d, dotted, dash] \\ P \arrow[ur, dash] \arrow[r, dash] \arrow[dr, dotted, dash] & {} \\ & A_n \end{tikzcd}$	Concurrent action	Actor with purpose P concurrently undertakes actions $A_1 ... A_n$.
vii) $\begin{tikzcd} & S_1 \arrow[d, dotted, dash] \\ A \arrow[ur, dash] \arrow[r, dash] \arrow[dr, dash] & {} \\ & S_n \end{tikzcd}$	Multiple consequences	Action A causes each of the intended consequences $S_1 ... S_n$
viii) $\begin{tikzcd} A_1 \arrow[dr, dash] \arrow[dd, dotted, dash] & \\ & S \\ A_n \arrow[ur, dotted, dash] & \end{tikzcd}$	Alternative causation	S is caused by either A_1 or $A_2 ... A$
ix) A -x- S	Casual blockage	Action A prevents state S
x) S -x- A	Vitiation	State S inhibits performance of A
xi) A --- S	Unavoidable consequence	Action A causes state S, not intended by the actor
xii) A -x- S	Unavoidable blockage	Action A unwittingly prevents outcome S
xiii) $A \underset{S2}{---} S1$	Positive gating	State S_2 enables action A to lead to state S_1
xiv) $A \underset{S2}{-x-} S1$	Negative gating	State S_2 prevents A from leading to state S_1

All of the linkages can be combined to form a network, with the additional restriction that all networks must begin with a single leftmost P-atom and end with one or more rightmost S-atoms.

If the atoms correspond to statements in a normal programming language then the above linkages correspond to the control structure. It is interesting to compare a network with a PLANNER program to contrast the two ways in which plans are constructed. A comparison highlights the view expressed in this thesis that the program structure of a conversational system should reflect the structure found in natural language rather than that taken from programming languages. For this reason the control structure of PIDGIN is based on a scheme similar to the above description.

In PIDGIN a state can be specified as desired and if it is connected by enabled actions to the current situation then a plan can be constructed that consists of those actions. Every action is associated with a set of enabling states and if all these hold then when the action is performed a new set of states will arise which will enable further actions to be performed until the desired state holds.

Abelson goes on to consider how two plans can be combined into a theme. A theme is a relationship between the plans of two actors. One actor is either the agent, goal object or interested party of the other actor's plan. Each actor has an "attitude" (positive of negative) towards the other and each is able to interfere with the other's plan in a set number of ways. Depending on whether the actor's relationship: is positive or negative a theme will be one of admiration, devotion, cooperation, alienation, betrayal, rebellion, antagonism, oppression or conflict, plus a number of other related themes.

Themes can be combined to form theme sequences or scripts. A script is a network of themes with linkages analogous to those between atoms in a plan. For example, the "romantic triangle" script runs:

$$\begin{tikzcd}
& \text{Love} \ (E,F) \arrow[dr] & & \\
S \arrow[ur] \arrow[dr] & & \text{Antagonism}(E,G) \arrow[r] & \text{Conflict} \ (E,G) \\
& \text{Love} \ (G,F) \arrow[ur] & &
\end{tikzcd}$$

And the rescue script might run:

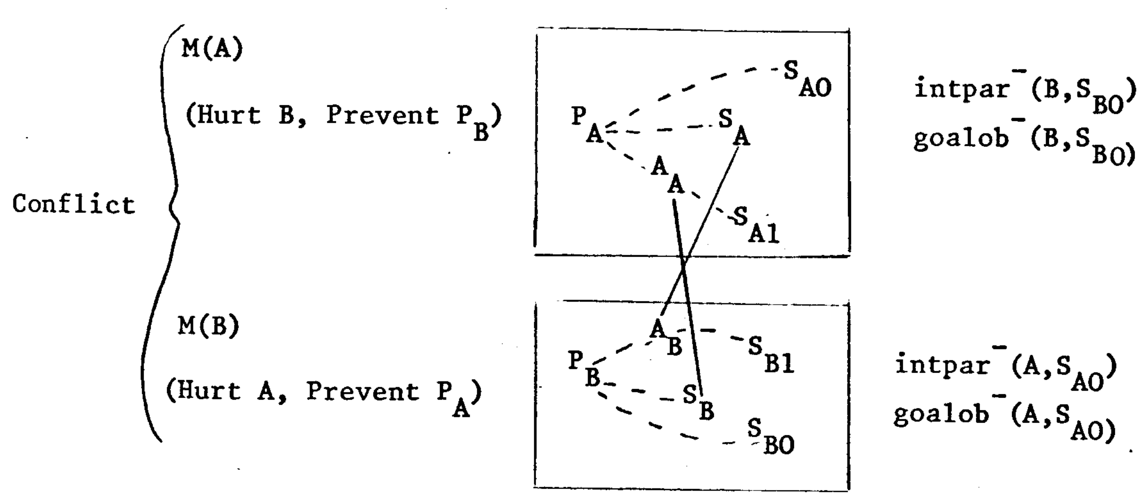
$$\begin{tikzcd} & \text{Conflict (E,F)} \arrow[rr, "\times" marking] & & \text{Oppression (E,F)} \\ S \arrow[ur] \arrow[dr] & & & \\ & \text{Dominance (G,E)} \arrow[uur, shorten >=40pt] \arrow[rr] & & \text{Oppression (F,E)} \end{tikzcd}$$

Abelson gives the script of the current Cold War (detente disregarded) based on a hypothesised belief system or ideology of what he calls a Cold Warrior.

PIDGIN has not been extended to contain primitives that correspond to themes and scripts as from the point of view of the chess problem these are simple and fixed. In chess the theme is conflict leading eventually to oppression. The script is simple; for player A it would be:

$S_{chess} \otimes$ Conflict (A, B) $\otimes$ Oppression (A, B)

The chess ideology is simple and brutal. However, it might be interesting to postulate an ideology for say, the white king, involving themes corresponding to the interplay of all the pieces. What is interesting about chess is the nature of the theme of Conflict. Abelson diagrams this as:



It might be worth while to consider the chess problem from this point of view. Each side can carry out actions that either damage the position of the other side or prevent the other side from doing damage, both from the point of view of the desired end state. Such a computer program would need to consider its own plans, to hypothesise about its opponent's plans and to consider how they interact.

CHAPTER 2 PIDGIN - A REALISATION LANGUAGE

2.1 The PIDGIN Language

PIDGIN is a programming language into which natural languages, such as English, may be easily translated. Thus corresponding to every PIDGIN program there is some sequence of English sentences. The unit of a PIDGIN program is called a conception and this corresponds approximately to 'a simple English sentence. Conceptions can be combined to form what are called thoughts, as sentences can be combined in English to form complex sentences, rules, suggestions and so on. There are eight types of thought in PIDGIN, each associated with a different conception connector. For example, the following PIDGIN thought uses the IF connector to join a single conception to a rule consisting of two conceptions joined by conjunction:

```
A WOMAN <AUNT A PERSON1> IF
            THE WOMAN <SISTER A PERSON2> AND
            THE PERSON2 <PARENT THE PERSON1>.
```

This corresponds to the English definition:

A woman is the aunt of somebody if she is the sister of another person who is the parent of the first person.

The above PIDGIN thought is given in a language called Input PIDGIN. This is assembled by a program called the PIDGIN assembler into the Strict PIDGIN language. Input PIDGIN is suitable for human consumption and Strict PIDGIN for machine interpretation, for example, the above thought would be assembled into the Strict PIDGIN thought:

```
<IF  [1 LTS 50 TRUE <EQUAL 123> MANNER PERIOD MOD]
     <BE <WOMAN <EQUAL 1> SUBJA      <AUNT <PERSON1
<EQUAL 1> ATTR SPEC>>>
            [2 LTS 50 TRUE <EQUAL 123> MANNER PERIOD MOD]>
     <BE <WOMAN <EQUAL 1> SUBJA <SISTER <PERSON2
<EQUAL 1> ATTR SPEC>>>
            [3 LTS 50 TRUE <EQUAL 123> MANNER PERIOD MOD] >
     <BE <PERSON2 <EQUAL 1> SUBJA <PARENT PERSON1
<EQUAL 1> ATTR SPEC>>>
```

```
                [4 LTS 50 TRUE <EQUAL 123> MANNER
PERIOD MOD] >
      ⌋
>
```

If the interpretation rules given later are referred to it can be seen that this PIDGIN thought will result in the function IF being applied to its three arguments. The IF function will store the rule in memory so that it can be used later to make inferences. For example, consider the three Input PIDGIN conceptions:

```
MARY <SISTER JOHN>. JOHN <PARENT BILL>.
MARY <AUNT BILL>?
```

The first two assertions would be stored in the memory in their expanded Strict PIDGIN form. The question, however, must extract information from the memory. This is done by matching the question with all the conceptions stored in memory. If one matches then that is the answer, but if none match then the system will try all matching IF thoughts in order to try to answer the question using deduction. If the first conception of some IF thought matches, as it does in this case, then the other conceptions making up the IF thought are treated as questions. If these can be matched (possibly involving further IF thoughts) then the original question succeeds and is answered, otherwise the answer cannot be inferred from the memory.

It can be seen from this example that there are two ways the PIDGIN system treats conceptions and thoughts as they are input. If it is an assertion then it is added to the memory and if it is a question then it is matched against the memory to retrieve information from it. Together with commands (which are like assertions but may add more than one conception to the memory according to matching ENABLE and PRODUCE thoughts) these form the basis of all PIDGIN. It remains necessary to describe the exact syntax of both Input and

Strict PIDGIN, the matching algorithm, the general structure of the system, and implementation questions such as the structure of memory.

2.1.1 The Strict PIDGIN Language

A description of the syntax of Strict PIDGIN will be given next, and it is convenient to describe first the foundation of the complete PIDGIN system - the interpreter. The basic interpreter is called ABC (Associative Backtrack Computer) and the language over which ABC is defined is called ABL (Associative Backtrack Language). Each Strict PIDGIN thought is a legal ABL expression, so each Strict PIDGIN thought can be interpreted according to the rules of ABC. The complete PIDGIN interpreter is ABC plus a number of ABL expressions to be described later. The syntax in this section is a type of Backus-Naur notation with the following features:

i) The string "::=" can be read as "is defined as".

ii) The vertical bar can be read as "or".

iii) Lower case letters and the hyphen are used to name syntax categories. Each category is defined once by placing it on the left-hand side of the "::=" string with its legal replacements on the right-hand side.

iv) Upper case letters, digits and punctuation characters are terminal symbols.

v) If a defined syntax category (say, X) is used terminated by the letter "s" then this means that is may occur one or more times (i.e. Xs ::= X | X Xs).

vi) If a defined syntax category (say, X) is used terminated by the string "-choice" then this means that it may occur between band or class brackets (see syntax). That is:

X-choice ::= X | (X-choices) | ⌞X-choices⌟ | [X-choices] | ⌊X-choices⌋

vii) A string in quotes is used to describe the category in English.

First the ABC language ABL is defined and the interpretation rules given. This is followed by the syntax of Strict PIDGIN together with some notes justifying the structures described by the syntax by reference to the way the system is used. It should be noted that the set of strings defined by the PIDGIN syntax rules are a strict subset of the set of strings defined by the ABL syntax rules. Thus the interpretation rules given for ABL also apply to each legal PIDGIN string.

2.1.1A Associative Backtrack Computer

The following sixteen syntax categories define a language called ABL (Associative Backtrack Language). After this the evaluation of each legal ABL string is given.

expression ::= application | band | class | concept | number
application ::= <expressions>
band ::= ordered-band | unordered-band
class ::= ordered-class | unordered-class
concept : : = name | name subscript
number ::= integer | real | negator number

ordered-band ::= [expressions]
unordered-band ::= ⌊expressions⌋
ordered-class ::= (expressions)
unordered-class::= ⌞expressions⌟

name ::= "one or more letters"
subscript ::= integer
integer ::= "one or more digits"
real ::= integer point integer
point ::= "a full stop"
negator : : = "a hyphen"

To define the evaluation of an ABL expression the evaluation of each of the five types of expression is described separately. In general when an expression is evaluated it either returns a value, and is said to succeed, or else it fails.

i) Concept. In order to describe how a concept is evaluated it is necessary to describe some other features of ABC first. Every pair of concepts is associated with an expression, which is initially taken to be the first concept of the pair. The pair is ordered, that is, the pair A/B need not be associated with the same expression as the pair B/A. The second concept of the pair is called the "aspect" and the expression is called that aspect of the first concept. For example, if the pair PERSON/SUB is associated with MALE then MALE is called the SUB aspect of PERSON. The REF aspect of a concept is also called the reference of that concept. The aspect of a subscripted concept is always equal to the same aspect of the unsubscripted concept except in the case of the REF aspect. That is, the X aspect of A equals the X aspect of A1, A2, A3 and so on, but the REF aspect of A may be different from that of A1, which may differ from that of A2 and so on. For example, the SUB aspect of PERSON3 is the same as the SUB aspect of PERSON but their references may be different.

A concept is evaluated by evaluating its reference, if this is equal to the concept itself then evaluation fails.

The association of an expression with a concept pair may be "frozen" so that it can no longer be altered by the PIDGIN processes of matching and binding described later. In PIDGIN the only association that is frozen is that with the concept ENTITY, if this is frozen as the reference of a concept then that concept is called an entity concept. Any other concept is called a group concept.

During the evaluation of ABC if the association of an expression with a concept pair is altered then the previous association is remembered until the expression currently being evaluated either succeeds or fails. If it fails then the previous association is reinstated, this is called automatically undoing a side-effect after failure.

The set of all the associations of expressions with concept pairs is called the data-base. During evaluation the data-base is continually changing and it forms a context for resolving ambiguities and anaphoric references during natural language translation, as well as a guide to control memory searching and a "local variable" mechanism when making deductions. The memory of PIDGIN is a subset of the data-base consisting of those statements stored by the running system. The ABL programs making up the PIDGIN system determine the particular way the memory is "partitioned" into the data-base and this is described in more detail later.

ii) Application. An application is an ordered sequence of one or more expressions, the first of which is called the function and the others of which are called the arguments. To evaluate an application (i.e. apply a function to its arguments) all the arguments are formed into an ordered band (any argument that is itself an application being evaluated first).

This ordered band is then made the reference of the concept ARGS. The function is then examined; if it is one of the 42 ABL primitives listed in Appendix 1 then the action taken is as described in Appendix 1 (note that if its ARCS aspect is QUOTE then the arguments are not evaluated first as described above). If it is not a

primitive then it is evaluated according to the rules given in these five pieces. In particular if the function is a concept then its reference is evaluated. The value of the application is the value obtained by applying the function to its arguments.

iii) Band. A band is either an ordered or an unordered conjunction of all the expressions it contains. It is evaluated by evaluating each expression until all succeed or until one fails. If an expression fails then the evaluation of the band terminates immediately and any side-effects are automatically undone and the complete band fails, otherwise the band succeeds and the value returned is the value of the last expression evaluated. If the band is unordered then the order of evaluation of its expressions is not defined.

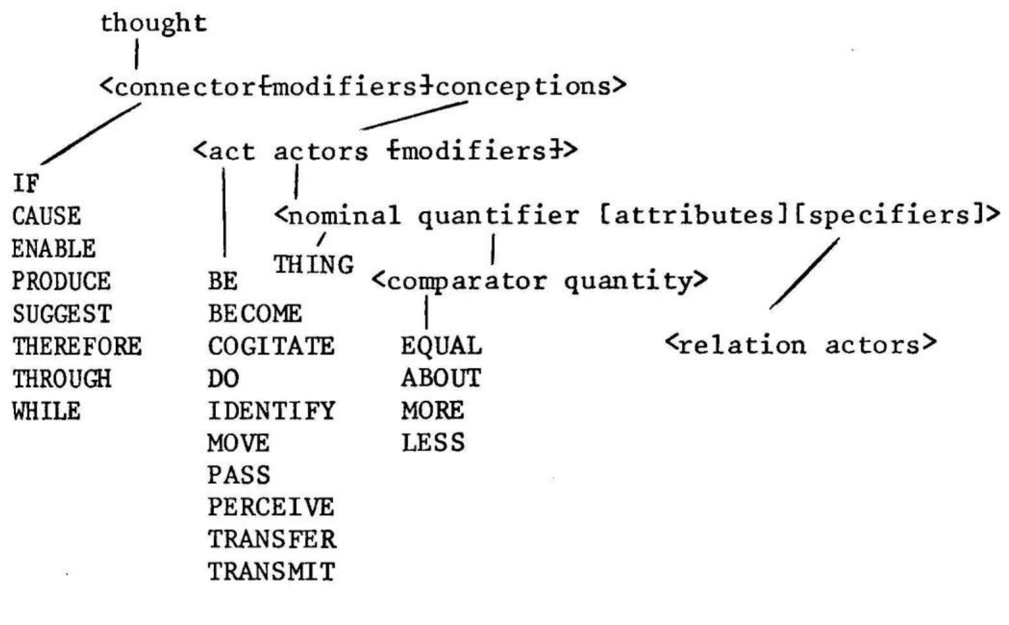
iv) Class. A class is either an ordered or an unordered disjunction of all the expressions it contains. It is evaluated by evaluating each expression until one succeeds or all fail. If an expression fails then any side-effects are automatically undone. If all the expressions fail then the complete class fails. If an expression succeeds then the evaluation terminates immediately, the side-effects of any other expression in the class being evaluated are undone and the class succeeds, returning the value of the successful expression. If the class is unordered then the order of evaluation of its expressions is not defined.

v) Number. The evaluation of a number always fails.

2.1.1B Strict PIDGIN Syntax

A Strict PIDGIN program is a sequence of statements the syntax of which is a particular restriction of the more general syntax of ABL. The syntax of ABL allows any number of bracketed expressions or concepts or numbers to be enclosed in anyone of five types of bracket. The syntax of PIDGIN allows only specific numbers and types of concept and bracketing to make up a program. As the ABC interpreter is defined over all ABL expressions it follows from the above that it is defined over any PIDGIN program.

A PIDGIN thought is a conception connected to a choice of conceptions by one of the eight connectors. A conception consists of one of the ten acts plus one to four actors plus a modifier. Before giving the syntax of thoughts and conceptions the following diagram of a simplified version of the syntax may make their structure clearer:



```
program ::= statements
statement ::= thought | conception
thought ::= <SUGGEST modifier state-choice state>
        | <ENABLE modifier state-choice action>
        | <PRODUCE modifier action state-choice>
        | <CAUSE modifier action actions>
        | <THEREFORE modifier
```

```
                <CAUSE modifier action actions> action>
          | <THROUGH modifier action action-choice>
          | <WHILE action-choice action-choice>
          | <IF modifier conception rule>
rule ::= conception-choice
```

The following two categories do not form part of the syntax but are given here because they are referred to later:

```
plan ::- action-choice
scheme ::= state-choice
```

The above syntax defines the connections allowed between the conceptions whose syntax is described next. Conceptions correspond approximately to simple English sentences and the above syntax defines all the ways that these may be put together in PIDGIN. The function part of a thought application is called a connector (see Division 2.3A).

```
conception ::= state | action
state ::= <state-act actor modifier>
action ::= <action-act kernel modifier>
kernel ::= subject object
         | subject object source destination
subject ::= actor-choice
object ::= actor-choice | thought | pattern
source ::= actor
destination ::= actor

actor ::= <nominal quantifier [attributes]
                                 [specifiers] >
nominal ::= entity-concept | group-concept
entity-concept ::= SELF
group-concept ::= THING

state-act ::= BE
action-act ::= BECOME | COGITATE | DO | IDENTIFY | MOVE | PASS
                | PERCEIVE | TRANSFER | TRANSMIT
                | TRANS | TROW

quantifier ::= <comparator quantity> | ALL
comparator ::= EQUAL | ABOUT | MORE | LESS
quantity ::= <operator quantity quantity> | number
                | variable | actor | ALL
operator ::= ADD | SUB | MULT | DIV
variable ::= NUMBER
attribute ::= ATTRIBUTE
specifier ::= <relation actors> | <<relation relmod> actors>
relation ::= SUB | EQUIV | INVERSE | OPPOSITE | POSS | HAS
           | FEEL | VALUE | PRIORITY | CLASS | space-relation
space-relation ::= LOC | NEAR | ABOVE | BELOW | BACK
           | FRONT | LEFT | RIGHT | BETWEEN | DIST
```

```
relmod :: = MAX | MIN | MEAN

modifier ::= [index author priority truth time manner
              period [mods]]
index ::= integer
author ::= entity-concept
priority ::= integer
truth ::= NOT | TRUE | POSSIBLE | DEFN
time ::= quantifier
manner ::= CAN | INTEND | ACCIDENT | DISPOSED
period ::= START | WAX | CONTINUING | WANE | STOP | EVENT
              | REPEAT
mod ::= <DEGREE actor> | <LOC actor>
              | <INTERVAL actor>
```

This completes the syntax of Strict PIDGIN except for the description of the category "pattern". It can be seen that a number of features of PIDGIN are taken from Schank's notation for conceptual deep structures. For example, consider:

Schank

$$\begin{tikzcd}
& \text{modifier} \arrow[d] & \\
\text{subject} \arrow[r, Leftrightarrow] & \text{act} & \text{object} \arrow[l] \\
\text{subjmod} \arrow[u] & & \text{objmod} \arrow[u]
\end{tikzcd}$$

```
PIDGIN      <ACT <SUBJECT SUBMOD>
                  <OBJECT OBJMOD> MODIFIER>
```

The equivalence is only approximate because PIDGIN contains features not present in Schank's system. The differences between the systems arise from the fact that Schank's notation is a static picture of the deep structure whereas PIDGIN is a programming language. Thus the justification for Schank's pictures lies with the reader because a precise interpretation is never given. The final part of the syntax describes a structure that enables general pattern matching problems to be handled in PIDGIN (see IDENTIFY, Part 2.3B5).

```
pattern ::= lattice | grid | line | actor
lattice ::= <grids>
grid ::= <lines>
line ::= <actors>
match-degree ::= MATCH | SIMILAR | MIRROR
match-type ::= SAME | RSUB | VAGUE | TYPE | LIKE
```

The ability to set up and compare patterns of actors is very useful for many problems, for example, the chess endgame problem considered later. The complete syntax of Strict PIDGIN has now been described.

2.1.2 The Input PIDGIN Language

The syntax described in the last section would be tedious to use as an input language because of the bracketing, the large number of modifiers required, the dissimilarity with English and the occasional necessity to repeat conceptions. For these reasons an input language is defined below to reduce these problems yet maintain a clear correspondence with the Strict language.

If the Strict language is imagined as the machine-code of a PIDGIN machine then the input language is like an assembler language. It is in one-to-one correspondence to Strict PIDGIN, is simple to translate and is user oriented.

The syntax below modifies the syntax of Strict PIDGIN. If a category is not redefined it retains the definition given in the last section. Curly brackets are used to enclose optional items.

i) Comments. These are expressly forbidden. Any information that would go in a comment should be given to the PIDGIN system. In some ways statements in PIDGIN are like comments in most programming languages.

ii) Labels. Any conception or thought can be labelled by following it with a colon and the label name. The conception or thought is made the reference of the label and if the label is used later it will be replaced by its reference.

iii) Program.

```
statement ::= thought terminator | conception terminator
thought ::= rule {modifier} connector thought | rule
connector ::= SUGGEST | ENABLE | PRODUCE | CAUSE
              | THEREFORE | THROUGH | WHILE | IF
rule ::= conception | conception choices
choice: : = AND conception {,} | OR conception {,}
terminator ::= . | ? | !
```

If a comma is used all the conceptions to the left of the comma are bracketed together, otherwise bracketing is from the right, e.g. :

```
A AND B OR C AND D. gives [A (B [C D] )]
A AND B, OR C AND D. gives([A B] [C D] )
A AND B, OR C, AND D. gives[([A D] C) D]
```

iv) Conception.

```
conception ::= subject {modifier}{act {object
               {source destination}}}
object ::= actor-choice | <thought> | *pattern
act ::= state-act I action-act
```

If the act is omitted then the state-act (BE) is assumed.

v) Actor.

```
actor ::= {quantifier}{[attributes]} concept
                     {specifiers}
quantifier ::= number | A | AN | THE | NO | SOME | MOST | ALL
             | EVERY | ANY | <comparator quantity>
             | = variable
```

where:

NO is translated as <EQUAL 0>
THE, A, AN is translated as <EQUAL 1>
SOME is translated as <MORE 0>
MOST is translated as <MORE <DIV ALL 2>
EVERY, ANY is translated as ALL

If the actor starts with a quantifier it is assumed to be a group concept, otherwise it is assumed to be an entity concept.

The entity concept SELF has a special meaning. If it is in the subject position then it is taken as referring to the PIDGIN system and the conception is treated as a command; otherwise it refers to the complete subject of that conception. It must always be used where it can be used. Its use is explained in Section 2.4.1.

vi) Modifier

```
modifier ::= [mods]
mod ::= POSSIBLE | DEFN | NOT | CAN | INTEND
          | ACCIDENT | DISPOSED | START | WAX
          | CONTINUING | WANE | STOP | REPEAT | EVENT
          | <DEGREE actor> | <LOC actor>
          | <TIME actor> | <INTERVAL actor>
```

The index, author and priority cannot be specified and are added automatically by the input assembler. If the mod POSSIBLE is used the priority is set to 20; if DEFN is used it is set to 10 greater than the user's current priority; otherwise the user's priority ~s used. The author is made the current user, and the index is incremented by one for every new conception read. If the conception is terminated by"?" then the assembler sets up the modifier:

[index author priority TRUTH TIME MANNER PERIOD MOD]

where TRUTH matches any truth, TIME any time and so on. However, if any modifier is stated explicitly it replaces the default value. It will be seen later (Division 2.4.2C) that the index, author and priority are ignored when matching.

If the conception is terminated by"." then the default value . for truth is TRUE and for time is NOW. The following abbreviations are accepted in the modifier:

PAST is translated as <LESS t>
FUTURE is translated as <MORE t>
NOW is translated as <EQUAL t>

where "t" is the current time as maintained by the PIDGIN system.

vii) Patterns. A pattern may not contain more than one occurrence of any entity concept. On input they must be preceded by the character "*".

viii) ABL. To input a structure not complying with the syntax of Input PIDGIN it should be preceded by the character "$". The structure following the "$" must conform to the syntax of ABL and it will be taken as satisfying the PIDGIN structure currently being searched for, either a thought or an actor. From the above descriptions of Input and Strict PIDGIN it should be clear how one is translated into the other. An example is given at the start of this chapter.

All the syntax of both Strict and Input PIDGIN has now been covered. It is next necessary to consider how PIDGIN statements may be used to represent conceptual knowledge and thus enable assertions and rules to be stored from which questions can be answered. This will be done by first considering the overall structure of the working system. But first a few examples are given to suggest how the PIDGIN statement corresponds to the English sentence.

2.1.3 Examples of PIDGIN

The following examples give an English sentence with the corresponding Input PIDGIN deep structure plus a possible English paraphrase of the deep structure. The way in which the translation is performed is described in Chapter 4.

i) John walked to the park.

JOHN [PAST] TRANSFER SELF A PLACE THE PARK
THROUGH JOHN [PAST REPEAT] MOVE 2 LEG
A PLACE$_1$ A PLACE$_2$.
John transferred himself from some place to the park by moving his two legs from one place to another.

ii) Bill hit John.

BILL TRANSFER AN OBJECT A PLACE JOHN.
Bill transferred an object from somewhere to John.

iii) Will John give his wife an expensive present on her birthday?

```
JOHN [<TIME A BIRTHDAY <BELONG A PERSON <WIFE JOHN>>>]
        PASS AN [EXPENSIVE] PRESENT SELF A PERSON
                  <WIFE JOHN>?
```
Will John pass possession of an expensive present from himself to a person who is his wife at the time equal to a moment which is the birthday of the person who is the wife of John?

iv) Since smoking can kill, I stopped.

```
A PERSON TRANSFER SOME SMOKE A SMOKEABLE LUNG
[CAN] CAUSE THE PERSON BECOME [DEAD] SELF,
        THEREFORE SELF [STOP PAST] TRANSFER
                  SOME SMOKE A SMOKEABLE LUNG.
```
A person who transfers some smoke from a smokeable object to their lung can cause that person to become dead, therefore I stopped transferring smoke from smokeable objects to my lungs.

v) John grows roses.

```
JOHN [DISPOSED] DO AN ACTION
        [INTEND] CAUSE SOME ROSE <HEIGHT =N INCH> BECOME
                  SELF <HEIGHT <MORE N> INCH>.
```
John is disposed to carry out an action that intentionally causes at least one rose of height n inches to become more than n inches tall.

iv) Yesterday, the boy in that chair stopped the girl by the window going to the park with the dangerous swings with John.

```
BOY <LOC CHAIR> [PAST] DO AN ACTION
        [INTEND <TIME YESTERDAY> ] CAUSE
        (JOHN GIRL <LOC WINDOW>] [STOP] TRANSFER
                  SELF A PLACE PARK <CONT SOME
                            [DANGEROUS] SWING>.
```
The boy located at the chair did an action in the past that yesterday intentionally caused the girl located at the window and John to stop the event of transferring themselves from somewhere to the location of a park containing at least one dangerous swing.

The above translations are at best very approximate because the translation actually performed by the PIDGIN system depends on the vocabulary of concepts and the entries in the translator's dictionary, and these change as the system is used. For example, in case (i) "walk" could have been translated simply as the first conception (in which case the information that the method of transfer involved feet would have been lost). In the second example the tense of the verb is ignored, as it might be in "a simple PIDGIN system. In the fifth example the actions involved in growing roses are not specified and the intended result is simplified to merely increasing their height. All of these limitations in the translation can be improved by giving the system more information about the meaning of the English words.

2.2 The PIDGIN System

2.2.1 The Components of the System

The best way to describe the complete system is to first distinguish between its major components, then to consider each component in detail and finally to show how they work together to produce a conversational problem-solving system.

The following components may be distinguished:

A. Associative Backtrack Computer

(i) Interpreter (42 primitives)

(ii) Assembler (Meta-ABL to Strict-ABL translator)

B. PIDGIN

(i) Interpreter (10 acts and 8 connectors)

(ii) Assembler (Input to Strict PIDGIN translator)

(iii) Disassembler (Strict to Input PIDGIN translator)

(iv) Resolver (matcher, binder and deduction mechanism)

C. Translator

(i) Analyser (English to PIDGIN)

(ii) Synthesiser (PIDGIN to English)

D. Knowledge Base

(i) Primary knowledge

(ii) General knowledge

(iii) Specialist knowledge

2.2.1A Associative Backtrack Computer

This is the heart of the entire PIDGIN system. All of the basic PIDGIN features are written in ABL, the language of ABC. The syntax of ABL has already been described in order to introduce PIDGIN but the description of the primitives of ABC is relegated to Appendix I as they are more concerned with the implementation of PIDGIN than its structure and interpretation.

ABL has been fully implemented using the programming language POP-2 and it has been used to implement the basic features of PIDGIN. In order to simplify programming in ABL a language called Meta-ABL or MABL was developed, and all the examples given in ABL are written in MABL. MABL allows much of the nested bracketing of Strict-ABL to be omitted. All the modules described later are implemented either using MABL or using PIDGIN itself.

ABL can be regarded as the "micro-code" of the PIDGIN machine in the sense that it was used to write expressions to implement the basic PIDGIN features and once the design of PIDGIN was fina1ised these expressions were never changed. That is, all further extensions to the system in terms of adding to its knowledge, both of facts and rules, can be done in PIDGIN and eventually, once the translator has been extended, in English.

It should be remembered that ABL is not the deep structure of, natural language. It is at a level which is below the lowest level that can be altered by the natural language level. No natural language input is translated into ABL expressions and no ABL expression can be translated into natural language unless the expression is a valid PIDGIN statement. Very crudely, the part of the system written in ABL can be thought of as the hardware or the neurophysiology of the complete PIDGIN system. For example, one important capability possessed by PIDGIN is the ability to answer questions concerned with its own workings. Such questions however can only be answered to the level of the PIDGIN programs that make up the system. For the system to produce more detailed answers it would be necessary to construct a "model" of the ABL programs in PIDGIN. Such a model would be a collection of PIDGIN facts and rules whose interpretation reflected the inner workings of the system, rather than a direct analysis of the system itself.

2.2.1B PIDGIN

As every PIDGIN statement is a member of the set of ABL expressions the ABC interpreter will interpret any PIDGIN statement. If the syntax of a PIDGIN statement is examined it will be seen that both a thought and a conception are applications whose first member is a concept (a connector or an act). When evaluated this will result in the reference of that concept being evaluated. Thus the reference of each of the eight connectors and ten acts should be an ABL expression whose evaluation results in the changes to the system associated with that act or connector. For example, if a PIDGIN conception is input as a statement then it should be added to the memory, and if a PIDGIN conception is input as a question then a matching conception should be retrieved from memory as the answer, and if a command then the appropriate action should be taken. The details of the actions performed by these expressions are described later in Section 2.3.

The user who interacts with the system may do so at a number of levels:

(i) Operating system command level

(ii) Implementation language (POP-2)

(iii) Associative Backtrack Language

(iv) PIDGIN language

(v) English

Commands are available at each level to allow the user to ascend or descend one or more levels, but the exact form of these implementation details will not be discussed here.

At each level the syntax of the output from the system reflects the syntax of the legal input. This feature of the system becomes more marked at the higher levels. At the PIDGIN level the output from the system has the same syntax as the legal input to the system. However, certain errors at the PIDGIN level can result in a return to the ABL level together with an ABL error message. At the English level all errors are trapped and all output has a syntax related to the input syntax that can be translated by the system into PIDGIN. This is because both the English-to-PIDGIN Analyser and the PIDGIN to-English Synthesiser use the same dictionary. By observing the system's output the human user can learn what input the system will be able to translate.

The program that reads, translates and evaluates PIDGIN is called the driver and this program is written in ABL. The simplest form the driver could take would be to read repeatedly a single PIDGIN statement in the syntax of the Input language, translate this into the Strict language and then evaluate it. The driver in this case thus repeatedly calls the assembler followed by the evaluator. The more complex driver actually used carries out tests and traps errors to provide a better interface to the PIDGIN user. If the system were used entirely at the English level then a PIDGIN driver and assembler would not be required. They are necessary in order to carry out the initial "bootstrap" to the English level and to provide a primitive level of control and adjustment to the working English level sys tem.

At the PIDGIN level it is also necessary to provide a disassembler for translating the Strict PIDGIN back into Input PIDGIN so that the output from the driver has the same syntax as the input typed in by the user.

When the user is working at the English level the dialogue is translated immediately to and from Strict PIDGIN so the assembler and disassembler are not used. However, at the English level, all the other parts of the system are required, i.e. the ABL expressions associated with the connectors and acts and the resolver to compare statements. The resolver is used to search for a match between two statements. This search may involve further searching to match other statements. This mechanism is used to check input for consistency and to answer questions and carry out deductions. It is thus a fundamental part of the whole system. It can be divided into two parts, the matcher and the deducer. The matcher can be described by giving a whole series of rules that can be used to determine whether any two statements match. This is done in Section 2.4.2. The deducer is used to answer questions that cannot be answered directly from the memory by matching but require further matching to be performed first, this is described in Section 2.4.3.

2.2.1C Translator

The translation from English to PIDGIN is carried out by an ABL program called the Analyser and the reverse translation from PIDGIN to English by another ABL program called the Synthesiser. These programs are described in more detail in Chapter 4 but the ideas have not been fully implemented as equivalent translators have already been implemented by R. Schank and others.

The translators can be sub-divided into parts corresponding to the various levels of translation. An important aim in designing the translators is to try to separate those parts of the translation process that are language dependent from those that are language independent (PIDGIN dependent) and to try to design the language dependent part in such a way that it can be easily updated and amended. Ideally such extensions to the natural language syntax that can be handled by the system should be possible at the natural language level by a series of definitions. The translators described in Chapter 4 make use of a dictionary in order to provide an easily amendable interface between the natural language and PIDGIN levels.

2.2.1D Knowledge Base

The complete system consists of a fixed core of ABL programs plus a collection of PIDGIN statements called the knowledge base. This knowledge base grows as the system converses with the user. All the facts and rules that determine the system's range and depth of knowledge are stored in the knowledge base. The knowledge base can be roughly divided into a collection of PIDGIN statements that initialise the system called the primary knowledge, a collection that provides the system with a rough knowledge of a wide range of facts and rules called the general knowledge, and a large number of specific facts and rules concerned with the particular application that the system is being asked to converse about called the specialist knowledge.

Appendix II includes a typical set of primary knowledge statements. These create a primary set of concepts and provide a model for all future PIDGIN statements created by the Analyser (see Chapter 4). They also form the basis of the dynamic structure of the system by enabling certain commands and specifying the states produced.

The general knowledge (see example in Appendix II) extends the set of concepts, facts and rules started in the primary knowledge base. The difference between the two is that the primary knowledge remains the same no matter what application the system is used for, but the general knowledge is orientated to the application. The general knowledge forms a "world view" that places any particular application within a context.

The general knowledge can be roughly divided into the following parts:

i) a structured set of group and entity concepts.

ii) their attributes,

iii) the relations among them,

iv) a set of conceptions defining the possible combinations of the concepts,

v) a set of thoughts defining suggest and action information,

vi) a set of "core beliefs" and inference rules.

The "core beliefs" and rules incorporate knowledge that guides the system's behaviour and enables it to make predictions and answer questions about the likely behaviour of others. Colby (1969a) has estimated the core beliefs of a person as under 50. Typical such core beliefs inc1ude:

i) Avoidance. If one person does something that makes another person angry then the second person will often avoid the first.

ii) Retaliation. If one person does something that hurts another person then that second person will often want to hurt the first person.

iii) Taking sides. If one person does something that hurts a second and a third person thinks the first did a good thing then the second person will often be angry with the third.

iv) Alliance. If one person likes a second and a third person also likes the second then the first person will often like the third but

v) Triangle. If one person loves a second and a third person also loves the second then the first person will often hate the third.

Other general information concerns basic enable and produce information, for example, for a person to transfer an object requires the person to be near the object, the object to be moveable, and the person to want to move it. Once moved the object will no longer be at the location it was at but will be at the location it was moved to.

The world view forms a framework into which the system can slot new knowledge and from which it can make deductions to answer questions and solve problems (see Section 2.4.4). The setting up of the world view is the second step in the initialisation of a working PIDGIN system. The third and final step involves setting up a natural language to deep-structure dictionary to enable PIDGIN to translate to and from the deep structure. This final step is called "dictionary set-up" and is described in Chapter 4. The three steps create a working question-answering problem-solving system to which new concepts, new rules, new beliefs and new dictionary entries may be added at any time by conversing with the system in English.

2.2.2 The Initialisation of the System

The basis of a working conversational PIDGIN system is an ABC interpreter and an ABL assembler. These are currently implemented in the POP-

2 programming language and they are described in Appendix 1. The initialisation consists of the 'three parts:

A. Defining the PIDGIN system in ABL.

B. Defining the English translator in ABL.

C. Creating the Knowledge base in PIDGIN.

2.2.2A Defining the PIDGIN System

There are seven parts that must be defined:

1. the eight connectors
2. the ten acts
3. PIDGIN assembler
4. PIDGIN disassembler
5. PIDGIN resolver
6. PIDGIN driver
7. initialising certain concepts

Each of these consists of associating a complex ABL expression with some aspect of certain concepts. For example, a simplified version of the driver in MABL could be:

<REPEAT [<ASSEMBLE> -TP. TP]>-DRIVER.

This associates the expression on the left with the reference of the concept DRIVER. Appendix 1 gives the MABL expressions that form part of this first stage in the initialisation of the system.

2.2.2B Defining the English Translator

The other major part of the system that is defined in ABL is the translator to translate between English and PIDGIN. This consists of three parts :

1. the Analyser
2. the Synthesiser
3. the Dictionary

Chapter 4 discusses these segments in more detail. The general approach is taken from the methods described by Schank although the implementation differs because of the advantage that can be taken of certain powerful primitives in ABC and because of the structure of PIDGIN.

2.2.2C Creating the Knowledge Base

Once the seven parts of the PIDGIN system have been assembled the system may be switched from working at the ABL.1eve1 to working at the PIDGIN level by evaluating the PIDGIN driver. Basically, this repeatedly calls the Input PIDGIN assembler and then evaluates the result. Further initia1ising of the system is done at the PIDGIN level and it can be separated into three parts :

1. primitive knowledge
2. general knowledge
3. specia1ised knowledge

These are described in the last section and in Appendix II. As each PIDGIN statement is read it is assigned a priority by the assembler. This priority specifies how important that statement is and it is used to try to resolve inconsistencies within the system (see Section 2.4.1). At any moment the reference of the concept USER gives the current author and the PRIORITY aspect of that author is a number between 0 and 100 giving the priority associated with that author.

If the priority is 100 then no checks for consistency are performed and the statement is stored immediately in the memory. Any statement with a priority of zero is regarded as no longer required by the system and if the system exhausts all the storage space available for the memory then such statements are automatically deleted. Any other priority is used to determine which of two conflicting statements the system will reject. For example, if the memory contains a conception with a priority modifier of 80 and a user with priority 50 tries to tell the system a fact that contradicts that conception then the user's fact will be rejected. In general a statement will be rejected if the memory contains a contradictory statement of higher priority. If the statement in memory is of equal or lower priority then the new statement is added to the memory in such a way that when the memory is searched it will always be found first. Further, it is possible for a user to alter his priority by using the modifier DEFN, this stores the statement with a priority 10 greater than the user's current priority and this enables the user to get the system to check his statements for self-consistency.

The above simple priority system is sufficient to enable a simple linear hierarchy of counter checks to be maintained. Users with the responsibility for creating and maintaining the system will have a high priority whereas users who simply make use of the system will have a low priority to prevent them from destroying the consistency of the complete system.

During Part 7 of the PIDGIN initialisation the following associations are made:

```
    90-GENERAL, PRIORITY .
    100-SYSTEM, PRIORITY.
SYSTEM-USER.
```

This means that when the primitive knowledge is read the author is SYSTEM and the priority is 100. At the end of Part 1 (primitive

knowledge) the user is changed by the statement :

GENERAL-USER.

and subsequent statements are assigned priority 90. The specialised knowledge may be set up by a number of users who may have different priorities (usually less than 90).

Checks are required to prevent a user from altering his own priority. This is done by associating a class concept with the CLASS aspect of the author concept. There are three classes of user - SYSTEM, GENERAL and USER, and each class is associated with a different set of checks. A user working in SYSTEM class can alter any part of the system including class and priority; a user with GENERAL class cannot alter class or priority and cannot return to the ABL level (where such checks could not be made); a user with USER class is further restricted in that such a user may only work at the English level and cannot return to the PIDGIN level. The user name thus acts as a password and each user is required to state his name when first entering the system.

Appendix II gives listings of parts of all three segments. The final segment, specialised knowledge, merges into the knowledge obtained from the end-user, the user who uses the system at the English level in order to solve a practical problem. All of Parts I and 2 and some parts of Part 3 are at the PIDGIN level but eventually it becomes possible to switch up to the English level in order to teach the system new facts and rules. The switch is carried out by entering a PIDGIN statement consisting solely of the user name. This special statement is recognised by the BE act and it causes the system to use the English driver. This driver repeatedly calls the Analyser to read an English sentence and translate it into PIDGIN. It then evaluates the PIDGIN which results in a PIDGIN answer which is then translated back into English by the Synthesiser. This process is described in more detail in the next section.

2.2.3 The Construction of the System

The overall construction of the PIDGIN system is diagrammed in Fig. 4.1 (Section 4.2.1). It can be seen that it is possible to regard it as consisting of three parts - the processor. the memory and the translator.

2.2.3A The Processor

The processor is the name given to the PIDGIN driver plus the ABC evaluator with its associated primitives. At any moment while the system is running the processor contains the PIDGIN statement being. evaluated. Because the evaluation of a single statement may result in further statements being evaluated before the first has completed evaluation a number of statements may be in the middle of being evaluated at any moment. The stacking and unstacking of these partially evaluated statements is taken care of by the PIDGIN driver.

The evaluation of a PIDGIN statement either results in the statement being stored in memory or a matching statement being retrieved from the memory. A comparison between the evaluation of PIDGIN and the evaluation of a more conventional programming language, such as POP-2, may make the evaluation of PIDGIN clearer. In POP-2 a variable may be associated with a function body (a lambda expression) by defining a function with that name.

When that name is followed by round brackets containing expressions. the expressions are first evaluated and then the function is applied. In PIDGIN the distinction between function name and parameters is not so distinct. Functions do not have names but instead each function is associated with a description of the parameters it requires. When any statement is evaluated all the functions whose parameter description matches that statement are evaluated one by one until either one succeeds or all fail. Returning to the comparison with POP-2, it is as if functions with the general form, e.g. :

```
function f a, b, c;
      j(a, b); k(c); l(b, a, c);
end;
```

were interpreted as something like:

```
function {a, b, c};
      if {a, b} and {c} and {b, a, c} then
              true else false close;
end;
```

where, for example, {a, b} means apply all functions whose parameter description matches {a, b} until one returns true and if all return false then the result is false. If {a, b} , {c} and {b, a, c} all return true then {a, b, c} returns true. Anyone familiar with PLANNER will immediately recognise this goal-oriented success/failure mechanism.

The advantage of this evaluation scheme is that it is closely related to human question-answering conventions. New functions may be defined to extend the system capabilities without destroying previous definitions. The new function will automatically be tried if a matching statement is evaluated. The evaluation mechanism can be thought of as trying to prove a statement by searching for a fact (or disproving by counter-example in PIDGIN) or a rule. A fact is simply a function that is always true, to return to the analogy with POP-

2, and a rule is a function that is true if a further series of statements is true (like function {a, b, c} above).

One disadvantage of this method of evaluation is that, unlike conventional programming languages, functions cannot be called directly. The apparent inefficiency of this is overcome by the nature of a PIDGIN statement as a structure incorporating knowledge. The matching that must be performed before applying any function is an essential part of the question-answering process.

The evaluation scheme actually used by PIDGIN is more complex than the above description suggests, for example, it involves the conjunction and disjunction of statements and negation. But the idea that matching is the fundamental evaluation process is equally true. A more complete description of the evaluation scheme of PIDGIN can be found in Section 2.4.1, and a complete description of the matching process is given in Section 2.4.2.

Although it is possible to draw a close analogy between aspects

of PIDGIN and PLANNER it is important to recognise the differences because it is in these that the extra power of PIDGIN lies.

2.2.3B The Memory

The memory can be divided into two parts - the immediate and the long-term memories.

The two basic memory processes are" transmitting a statement to memory and retrieving a statement from memory. These can be performed explicitly using the TRANSMIT act (see Part 2.3B10) and they occur implicitly as evaluation proceeds (see the last division).

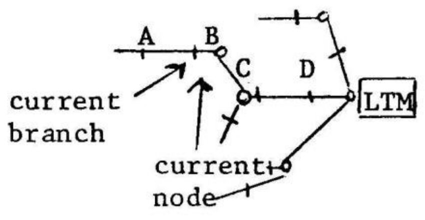
The efficiency of the memory largely determines the overall efficiency of the system. One efficient way of implementing the memory is to avoid multiple copies of a structure. This could be done during translation when new structures are created by first searching the memory for the structure. If this were done then the memory search for the complete structure would be much simpler. In such an implementation the actual structure representing a statement might be considerably different from that suggested by the syntax. For example, an actor could be set up as a basic data structure which combined all the qualifiers of its main concept, although it would be necessary to include references to the conceptions containing the actor in order to allow the modifiers to be checked. This flexibility of the PIDGIN implementation is not present in systems based on PLANNER where the data-base pattern is not restricted to any fixed format.

All the above details of implementation obscure the central problem of efficiency, which is to find a way to overcome the combinatorial effect of a growing memory on the time taken to solve a given problem. Schwarcz (1970) points out that this combinatorial explosion comes not from the memory search for facts but from the investigation of inferences during problem solving. He suggests that a structure larger than his triples might mitigate and PIDGIN does comply with this suggestion but this only pushes back the explosion to slightly larger memory sizes. A complete analysis of the problem has not been done for PIDGIN but it is hoped that the increasing complexity of the memory as its size increases can be used to offset the inefficiencies resulting from the increasing size. By matching the implementation with the nature of the use of the deep structure it is hoped that the knowledge contained in the memory can be used as a heuristic to speed up searching by guiding the search to the right place. For example by controlling this search a1goritlunusing the "substitutable" relation (SUB) it will improve as the system is supplied with more concepts because this will automatically limit the number of concepts that are substitutable for any particular concept used.

2.2.3B1. Immediate Memory (IM)

a. Short-term Memory (STM)

STM enables conflicting facts (or different "possible worlds") to be stored in the memory without the inconsistencies interfering with each other. This is done by storing them outside the long-term memory (LTM) in the form of a tree structure so that any memory search always proceeds down a single path from the current leaf to the root. If the fact is not found in the STM tree then the LTM is searched, this can be diagrammed as:



In this situation the facts A, B, C, and D will be examined before LTM is searched.

As execution proceeds the STM tree grows and at any given point in the evaluation there is a current node and a current branch. A new node is set up in the following circumstances:

i) When a rule is evaluated a new node is added to the tree so that any statements added to the memory while inside that rule can be easily removed if it fails.

ii) Similarly when evaluating a class of statements (i.e. a disjunction of statements) a new node is set up and any member of that class that stores a statement will do so to a different branch,

iii) and when evaluating a band of statements (i.e. a conjunction of statements) a new node is set up so that any statements stored can be easily removed if the band fails.

If a rule, band or class succeeds then the current branch and node are not altered but if they fail then the current branch and node are set to the value they had before evaluating the structure. When an evaluation is complete all the statements stored between the current branch and the LTM can be added to the LTM.

b. Suspended Evaluation Memory (SEM)

Complete evaluation may be suspended, for example, to ask the user a question. This is done by saving the STM in a band called SEM. A new STM can then be started and used to evaluate the new input. Every saved STM in SEM is associated with what is called a reviver. This is a statement which if matched will restore the suspended STM and continue evaluation.

For example, if the sentence:

John flew to London.

was input the system might generate the question:

Was John a pilot or passenger?

and suspend the current STM with the reviver:

JOHN < SUB (A PILOT A PASSENGER».

When any sentence is input it is first stored and then compared with all the revivers in SEM. If any match the associated STM is re-activated. As the required information will then be in the memory evaluation will be able to continue. To avoid confusing the user by reviving old evaluations the size of SEM is arbitrarily limited to five members, any new member after the fifth will replace the oldest member.

c. Current Evaluation (CE)

The current context of evaluation is determined by:

(i) the current STM tree plus the current branch and node.

(ii) the current references of all concepts.

(iii) the current statement plus all suspended statements.

(iv) the current LTM.

The manner in which the references and the nesting of statements are stored is partly described in Appendix I when the ABC system and the PIDGIN driver are discussed. To a large extent it is handled by the ABC system in a manner which is transparent to the workings of PIDGIN.

d. Context Information (CI)

The translation from English to PIDGIN and vice-versa requires context information to resolve and to create various references such as definite article and pronoun references. This information is held as the global reference of concepts, for example, the global reference of HUMAN might be JOHN. The global reference of certain special concepts is also held by the translator, for example, HE, SHE, IT, ONE, THEY and WE, in order to provide a sentence context for resolving anaphoric references.

If the sentence being analysed is the reply to a question then it might be elliptical if it immediately follows the question. The information for replacing the missing parts of the elliptical reply is obtained by the Analyser from the reviver of the last entry in SEM. For example, the reply to the question in segment (b) above might be:

A pilot.

The reviver supplies the missing information.

Definite article references may need to be resolved from LTM. For example, "the girl next door" might cause the Analyser to ask the question A [YOUNG] WOMAN <LOC NEXTDOOR> which might find MARY <LOC NEXTDOOR> in the memory and resolve the reference.

2.2.3B2 Long-term Memory (LTM)

LTM contains all the statements that have been input to PIDGIN. They can be imagined as a linear list, in index-number order, to which new statements are added at the high-index number end. At the start are the low index statements of the primary knowledge, followed by the general knowledge and the specialist knowledge. The dictionary used by the translator is also stored in the LTM although its format differs from that of normal statements. When LTM is searched the search starts at the high index-number end and proceeds towards the low index-number end. So if the memory contains two statements that would match, the one with the higher index-number will be found first and terminate the search; the higher numbered statement is said to "hide" the lower.

Although this simple description of the LTM provides a clear model of the system's behaviour when searching it would be very in

efficient if implemented this way and further it does not exhibit all the properties desired of the memory. Instead the LTM is organised as what is called a "merged" memory rather than a linear memory. The LTM is first partitioned into eighteen parts corresponding to the eight connectors and the ten acts. Each partition contains a merged list of all the statements which have that connector or act. The structure of the merge lists is best described by explaining how a new statement is added; there are three possibilities:

i) the statement matches nothing in the list; in this case the statement is added to the end of the list at which the search started, i.e. the newest are found first.

ii) the subject of the statement is more specific than a matching statement in the list; in this case the new statement is added before the old so that it will be found first in future searches.

iii) the subject of the statement is less specific than a matching statement; in this case the search continues until either a less specific or a miss-matching statement is found when the new statement is inserted before it.

One actor is less specific than another if it is either substitutable for the other or they are the same (or equivalent) and the second is more qualified than the first. For example, ANIMAL is less specific than CAT and [BLACK] CAT is more specific than CAT.

It can be seen that a merge list is a linear list if none of the statements in the list matches another.

The merge list for the act BE is called the "world model" because it contains all the states that describe the attributes of the objects known to the system.

From what has been said it can be seen that the memory only grows. It would be useful in practise to have a "garbage collection" system that destroyed unnecessary statements when computer storage was full. This would be possible with the above merged lists by destroying specific statements if more general statements where present. This would result in the system losing specific information but retaining general information. It would also be possible for the system to generate general statements in order to replace two or more specific statements with a single general one. For example, if the statements:

JOHN <POSS A CAT>.
JOHN <POSS A DOG>.

were stored in the memory they could be replaced by:

JOHN <POSS A PET>.

A similar approach to garbage collection is to destroy the attributes and specifiers of statements. The advantage of these approaches to garbage collection is that they do not simply destroy knowledge but gradually make it more imprecise and general.

2.2.3C. The Translator

During translation to and from PIDGIN a number of buffers are used to contain intermediate structures, the main sequence is : character buffer, word buffer (surface structure), item buffer (shallow structure), and then the processor (deep structure, PIDGIN). The output synthesis works in the reverse sequence using a different set of buffers. The precise way in which the buffers are used is described in Chapter 4.

2.3 The PIDGIN Concepts

Before describing how questions are answered and problems solved by the process of matching statements it is first necessary to consider the simplest unit in PIDGIN, the concept.

In all the examples where English is compared to an equivalent PIDGIN statement (see Section 2.1.3) it looks as if the PIDGIN statement is a sequence of words taken from the English sentence, re-ordered, bracketed and written in upper-case. This is misleading as it hides the fundamental distinction between the words in the English sentence and the concepts in the PIDGIN statement; it is more than just a matter of upper or lower-case. The upper case words in the PIDGIN statements are symbols of "universal human concepts". A universal human concept or concept is a language independent meaning. Two English sentences with the same meaning or in fact any sentences in any language with the same .meaning will translate into the same PIDGIN statement. The reason for using English words for some of the concepts is simply for convenience, but it can lead to confusion if the distinction is not borne in mind.

There are some schools of thought that maintain that no two sentences have the same meaning and that the same sentence used on two separate occasions will have a different meaning on each occasion. I am not using the word "meaning" in this way. Two sentences have the same meaning if they convey the same explicit information. One practical method for judging if two sentences have the same meaning is to consider the circumstances in which one would be true and the other false. If there are no such circumstances then they have the same meaning and if the only such circumstances depend upon the disposition of some human actor referred to in the sentences then they are said to have nominally the same meaning. This definition is used to try to simplify the problem sufficiently to produce a working question-answering system. More subtle difficulties can be considered later by the modification of the working system.

During the analysis of English into PIDGIN, concepts may be introduced that do not occur explicitly as words in the English sentence, and words in the sentence may be lost. During synthesis of PIDGIN into English some concepts may not be realised as words and some concepts may be translated as complete English phrases. Further, a single English word might be the realization of a number of concepts; this is called "lexical" ambiguity. For example, the word "light" may mean LIGHTWEIGHT, LIGHTCOLOUR or LIGHTBULB. The converse is not true, that is PIDGIN is not ambiguous, no two concepts with different meanings have the same name. Some concepts are realised in English as word endings and inflexions, for example, those of number, gender and tense.

The rest of this section deals with all the different classes of concept handled by PIDGIN - connector, act, nominal, relation and modifier.

2.3A The Connectors

The connectors are those concepts that are used .to link together conceptions in order to form thoughts.

They are used to state knowledge and to control problem solving. There are eight connectors - SUGGEST, ENABLE, PRODUCE, CAUSE, THEREFORE, THROUGH, WHILE and IF. Each connector is associated with an ABL program that is evaluated if a thought containing that connector is evaluated (see Section 2.1.1 for the evaluation rules). In general if a thought is asserted then that thought will be stored in the memory and if a thought is evaluated as a question then the memory will be searched for a matching thought.

Before describing how the knowledge stored in the form of thoughts is made use of by the rest of the system it is necessary to explain the justification for just the eight connectors chosen. The connectors arise from a simple world description that arose from a consideration of the types of connectors described by Schank (1973b) and Abelson (1973). The world description is as follows:

i) At any moment of time, t, the world can be described completely by a band of states. This band is called the world model.

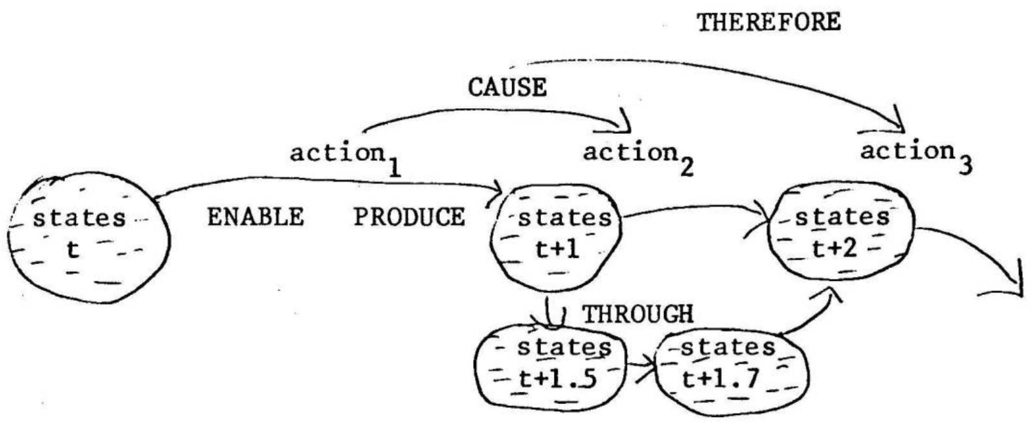
ii) Two moments of time are described by different band of states.

iii) An action is a function from one world model to another. An action can be enabled by zero or more states in a world model, that is, the action cannot occur unless those states are in the first world model. An action can produce zero or more states, that is, it can add those states to the second world model.

iv) Any action may cause or block another action.

v) A cause or block may cause or block an action.

vi) Between any two moments of time there is another moment of time. Thus as an action takes a finite length of time it can be analysed into sub-actions, that is, functions between world models occupying intermediate moments of time. These sub-actions may be specified by saying how an action is achieved through or by means of one or more other actions. The above points can be diagrammed as:



This shows how five of the connectors are related; the other three are fitted into the above model later in this section.

2.3A1. SUGGEST (state-state)

All the connectors playa dual role in the system; they enable natural language sentences to be stored in the memory in a suitable form for later interrogation and they act as controlling and guiding information for the PIDGIN problem solver described later.

The SUGGEST connector is used to link two states (a state is a conception in which the act is BE) together. The linkage roughly corresponds to implication or causation, thought not logical implication which is handled by the IF connector. An example of a natural language sentence which would be translated into a thought involving SUGGEST is:

"In chess control of the centre often leads to victory."

It will be seen later how such information is used by the PIDGIN problem solver to form schemes and from these, plans.

In fact SUGGEST may be used to join state-choices, that is, a conjunction and disjunction of states. For example:

THE QUEEN$_1$ <BELONG A PLAYER$_1$> <ON THE BOARD> AND

```
THE QUEEN2 <BELONG A PLAYER2> <ON THE
BOARD>[NOT]
      SUGGEST THE PLAYER1 <WIN THE GAME>.
```

SUGGEST is like CAUSE only between states rather than between actions.

2.3A2. ENABLE (state-action)

This connector is used to specify what state(s) must hold before an action can be performed as a command (see Section 2.4.1). Abelson (1973) tries to systematize the basic states that must hold for each of the acts described by Schank to be performed. He distinguishes between two types of enab1ement - instrumental control, where the state represents the main actor of the action being in a position to use the instrument(s) of the action, and social contract, where the state represents the actor of the action being the agent of a prior actor in a sequence of actions. PIDGIN does not distinguish between these two types but regards both as examples of a more general notion of enab1ement.

There are no enabling conditions built into the system and if no enabling conditions are specified for an .action then that action cannot be obeyed as a command. However, if a matching action has the modifier CAN then the action is allowed regardless of enabling states.

If some action is inappropriate in certain circumstances then it is necessary to define explicitly those circumstances using the ENABLE connector. For example, enabling conditions can be defined to prevent the system from making illegal chess moves:

```
A [COLOUR] PIECE <ON A SQUARE1> [NOT] BE
      ENABLE
      THE PERSON <POSS A [COLOUR] PIECE1>
             TRANSFER THE PIECE1 A SQUARE2 THE SQUARE1.
```

That is, for a person to move a piece to a square the square cannot already be occupied by a piece of the same colour.

Before the system carries out any action it first searches the memory for a matching action with the modifier CAN; if none is found it searches for all matching enabling thoughts, if any are found then a further check is made to confirm that all the states enabling that action are true.

Although some enabling conditions apply in most circumstances, for example, for a person to grasp an object they must be physically close to that object, most enabling conditions depend upon the circum stances. For example, the above chess rule does not apply if there is no game of chess taking place. To cope with this problem the enabling condition must be made more complex. The state requirement of an enabling condition may be in fact a state-choice, so the above condition could be modified to:

A PERSON DO A CHESSGAME AND
A PIECE<BELONG THE PERSON><ON A SQUARE>[NOT]BE
ENABLE
THE PERSON TRANSFER A PIECE$_2$<BELONG THE PERSON>
A SQUARE$_2$ THE SQUARE.

This also illustrates the fact that it is possible to state a condition in a number of different, but equivalent, ways.

2.3A3 PRODUCE (action-state)

This is used to specify those states that result from obeying an action as a command. Like enabling conditions there are no produce conditions built into the system but there are a number of basic enable and produce conditions in the general knowledge. Produce information is in some ways analogous to PLANNER's antecedent or asserting theorems in that it specifies how the world model is to be amended after a command has been obeyed. As this determines both how fast the memory grows and the range of questions that the system

can answer about its actions, a compromise must be reached.
Certain states do not need to be specified as they can be deduced from the information available in order to answer a question, but the total set of states, the world model, must reflect a true picture of the world otherwise inconsistencies will not be detected and commands will not be enabled. This is because when the system checks for inconsistencies and when it checks enabling conditions no deductions are performed, they are simply checked against the current world model. So, if more enabling conditions are specified then more produce conditions must also be specified in order to ensure that the world model contains the necessary states.

A typical primitive produce condition that would probably be in the general knowledge is:

A PERSON TRANSFER AN OBJECT A PLACE$_1$ A PLACE$_2$
PRODUCE THE OBJECT <LOC THE PLACE$_2$>.

2.3A4 CAUSE (action-action)

The cause connector has been thoroughly discussed by Schank (1973b). It is realised in English in a great many ways, some explicit such as "because", "when" and "since" and some implicit e.g. "fly", "prevent" and "want". It is the combination of the few acts defined by Schank with the cause connector that enables him to encompass the meaning of so many English verbs with so few acts. For example:

"John killed his teacher."

becomes:

"John does an action that causes his teacher to change from the state of being alive to the state of being dead."

2.3A5 THEREFORE (cause-action)

This connector is used to link a cause with an action, for example:

"As John stopped Mary going I'll go."

becomes:

"John carried out some action that caused Mary to not transfer herself from some place to another and therefore I did transfer myself from some place to that place."

or, in PIDGIN:

JOHN DO SOME ACTION
CAUSE MARY [NOT] TRANSFER SELF A PLACE$_1$ A PLACE$_2$
THEREFORE SELF TRANSFER SELF A PLACE$_3$ A PLACE$_2$.

2.3A6 THROUGH (action-action)

This connector is used to specify the means by which an action is carried out. It corresponds to what Schank calls the instrumental case. It is treated here as a connection between conceptions rather than as a part of a conception because every action can be divided into sub-actions and these sub-actions into further sub-sub-actions and so it is more convenient to deal with it in the same way as other relation ships between conceptions, namely by means of the thought structure. Many verbs translate to the same basic act but with a different THROUGH action, for example, walk, run, fly, drive and ride are all concerned with the basic action of transferring an object from one place to another, but they all differ in the means by which this is achieved. This is expressed in the deep structure by the use of different THROUGH actions, for example, walk is transfer THROUGH moving one's legs, run is transfer THROUGH moving one's legs quickly. Further, some verbs correspond to very complex deep structures if their full meaning is to be extracted, for example, "drive" refers to a whole series of complex actions the total of which is to do with controlling a car. However, this is not a limitation of the notation but a positive advantage. The complexity of the structure corresponds to the level of the systems knowledge of the concept.

Driving involves a whole series of interrelated actions, and what any person understands by the word depends upon that person's knowledge of these activities. A child, an experienced driver and a racing-car driver all have a different understanding of the verb "drive". In fact it could be argued that every individual has a slightly different under standing because of their different experiences, but common to all of these there is the basic notion that driving concerns moving from one place to another using some vehicle. Similarly the structure that PIDGIN generates for the ver9 "drive" represents the systems knowledge of driving. This may change as the system is given more knowledge, for example, the system could be given more information concerning the sub-actions THROUGH which the basic action of transferring is carried out in the context of driving.

An interesting aspect of the THROUGH knowledge of the system is that it provides a basis for adding the capability for actually manipulating the world by means of some robotic facility. The breaking down of actions into sub-actions is precisely the analysis required in order to determine the basic actions that the robotic facility is capable of performing. In such a system there would be a continuous linguistic link between the complex actions that the system was capable of talking about and the simple actions that it was capable of performing. This link would be provided by the THROUGH connector. For example:

Lift block THROUGH Find block and Move hand to
block and Grasp block and
Move hand up.
Find block THROUGH If not Perceive block Move
visual receptor

2.3A7 . WHILE (conception-conception)

This connector is used simply to join two conceptions that took place at the same time. It could be incorporated in the conception by extending the possible time modifiers to include complete conceptions. However, this would subordinate one of the conceptions to the position of a modifier and it is often the case that both conceptions are of equal importance. It would also be possible simply to store both conceptions together with the fact that they both occurred at some particular time, t. However, the WHILE connection implies that not only did the conceptions occur at the same time but that they were also connected in some undefined way. For example, "I ran the bath while the child got undressed" implies the bath was run for the child, "I made the tea while the others prepared the sandwiches" implies a communal meal, and "Nero fiddled while Rome burned" implies something other than simply that they occurred at the same time. The WHILE connector enables the system to store this knowledge without needing to explicate the implication.

2.3A8 IF (conception-rule)

This connector forms the basis of the system's deductive capability. It corresponds to the consequent theorems of the PLANNER programming language. The way in which IF rules are used is described later (Deduction, Section 2.4.3), but their use can be simply illustrated by means of an example:

A PERSON$_1$ <TALLER A PERSON$_2$> IF
THE PERSON$_1$ <HEIGHT =N METRE>
AND THE PERSON$_2$ <HEIGHT <LESS N> METRE>.

is one of the possible ways of defining the relation "taller". Then if:

MARY <HEIGHT 1.7 METRE>.
JOHN <HEIGHT 1. 8 METRE>.

it is possible for the system to deduce the answer to the question:

JOHN <TALLER MARY>?

by using the IF rule. This is done by first matching the question against all the conceptions in the memory to see if it can be answered directly and then, if none match, to see if it can be matched against any IF rule. In the above example the question does match the IF rule (the exact rules for matching are described in Section 2.4.2), in this case the specific information contained in the question is substituted for the more general concepts in the rule giving:

JOHN <HEIGHT =N METRE>
AND MARY <HEIGHT <LESS N> METRE>?

that is, has Mary a height that is less than the height of John. This

new question is answered from the memory using the same procedure as before but in this case it can be answered directly from the two assertions. This results in the IF rule succeeding and thus the original question succeeding. This would normally result in the controlling program generating the response:

JOHN <TALLER MARY>.

or, in English:

Yes.

2.3B The Acts

The ten acts of PIDGIN (including BE) are based on the acts defined by Schank (1973b). They are not intended to exhaust the possibilities of all English verbs but to enable enough verbs to be handled to build up a reasonable vocabulary for a question-answering system. The question as to the minimum number of acts required has not yet been answered. However, Schank suggests that the number may be very small, perhaps less than twenty. Less than twenty certainly seems sufficient to encompass a large part of everyday

English. Schank has 14 - 16 acts in his notation but five of these are concerned with the senses and I have compressed these into one act (PERCEIVE). Also he has the acts PROPEL and GRASP which I have omitted; the idea of applying a force is included as a modifier and GRASP is included in MOVEing the hand around an object. I have introduced the act IDENTIFY as it seems necessary to cope with the type of pattern matching comparisons required in many problem domains, for example, the chess end-game considered later. Further I have changed the names of some of Schank's other acts in order to try to make their meaning more obvious from their name.

It is interesting to compare with the above the ten operation words . of Basic English (Ogden 1933), "make", "put", "take", "keep", "let", "give", "get", "go", "come" and "do" which, with the auxiliaries and directives, immediately give the equivalent of 200 English verbs. It can be seen that nine of the ten are roughly handled by the two acts PASS and TRANSFER in PIDGIN.

The acts are the base of the system as they define all the possible ways in which actors may be combined. Because there are so few and because no more are required, the complete description of these acts is simple to set out and will remain valid no matter how far the system is extended. All growth of the system's knowledge proceeds through new combinations of the ten acts with a growing number of actors. Although the acts have names which are English verbs this is only a mnemonic convenience to help the reader. The actions performed by the acts are close to just one of the many different meanings and shades of meaning of the verbs. The following diagram shows a simple framework that expresses one way in which nine of the acts are related:

$$\begin{tikzcd}
 & \text{THINKER IDENTIFY} \arrow[rr, leftrightarrow, dashed] & & \text{THINKER TRANSMIT} \arrow[d, leftrightarrow] & & \\
 & & & \text{THINKER COGITATE} \arrow[d] & & \\
 & \text{LIFE PERCEIVE} \arrow[uu] \arrow[urr] \arrow[rr] & & \text{LIFE DO} \arrow[dr] & & \\
 & & & & \text{LIFE MOVE} \arrow[dr] & \\
\text{state}_1\ \text{BE} \arrow[uur] \arrow[urrrr] \arrow[rr] \arrow[drr] & & \text{BECOME} \arrow[rrr] & & & \text{state}_2\ \text{BE} \\
 & & \text{FORCE TRANSFER} \arrow[urrr] & & &
\end{tikzcd}$$

This should be seen together with the description at the beginning of the last division where the idea of an action as a state change is put forward in order to discuss the possible connections between actions. The above diagram shows the possible actions. The only act not included is PASS; this is because this act expresses the complex idea of ownership and this involves at least two thinkers. The diagram shows the possible ways in which any state may be transformed into another. The time axis is horizontal with the later time on the right, although COGITATE may extend over a long time period and involve many TRANSMIT acts. The diagram represents the way in which the acts fit together and it is shown simply to give the reader some idea of the motivation for choosing the ten acts described next.

There are two other acts shown in the syntax (Section 2.1.1), TRANS and TROW, these are not acts but stand in place of acts. TRANS may be substituted for any of the acts MOVE, PASS, TRANSFER, and TRANSMIT, and TROW for any of COGITATE, IDENTIFY, PERCEIVE and DO.

2.3B1. BE

ANY THING BE.

This act is separate from the others in that it is used to construct a state rather than an action; in this sense it is not really an act at all. It is used to give an actor the status of a conception. The above conception gives the most general form of a state, where the concept THING can be substituted for any other entity or group concept. In general it is used to specify the attributes or relations of an entity or group concept at any particular time. For example:

MARY <AUNT JILL>.
A [RED] BOX <ON THE TABLE>.
[HAPPY] JOHN .
FIDO <SUB A DOG>.

The set of all such states known to be currently true by the system is called the "world model" because it represents inside the system a picture of the current static state of the external world. Note that the input language syntax allows BE to be omitted.

2.3B2. BECOME

ANY THING BECOME SELF.

This act is the most primitive of all the acts and the most general. It enables the initial and final state of a transformation to be specified without needing to describe how the transformation was achieved.

It is usually used to specify some change in one or more of the qualifiers of an entity or group concept. The act is restricted in that the object must always be the same nominal as the subject, so by the rules of the input language the object nominal is always SELF. For example:

A [RED] BOX <LOC FLOOR> BECOME SELF <LOC TABLE>.
BOX_1 BECOME SELF <ABOVE BOX_2>.
BOOK <BELONG JOHN> BECOME SELF <BELONG BILL>.

BECOME is useful when an action is required but it is not known or not important which act is involved. For example, the CAUSE connector requires an action and BECOME can be used to supply it in the cases where it is only the final state that is important, for example:

Matches cause fires.
SOME MATCH <LOC A PLACE> BECOME [ALIGHT] SELF
[CAN] CAUSE THE PLACE BECOME [ALIGHT] SELF.

2.3B3. COGITATE

ANY THINKER COGITATE A THOUGHT.

Like all the acts COGITATE can be used in two ways by the PIDGIN system, as "data" forming part of a deep structure representation of some linguistic knowledge and as "program" when that same deep structure is evaluated. The action taken depends upon the context of evaluation; if the conception is an assertion then the acts add the conception to the memory; if a question then a matching conception is retrieved from memory; but if it is a command then the action taken depends on the particular act. In the case of COGITATE there are a number of possibilities, depending on the object:

i) Judging. If the object is a conception then it is checked to see if it is positive or negative (see Division 2.4.4A), if positive the conception succeeds, if negative it fails.

ii) Decision. If the object is the concept ACTION then if its reference is a conception the action taken is the same as in (i), if the reference is a choice of actions then the best (most positive value) is selected and made the reference of ACTION. If the reference is anything else then the action taken is as in (i) and the conception found is made the reference of ACTION.

iii) Planning. If the object is the concept PLAN then COGITATE will form a plan (see Section 2.4.4) and make it the reference of PLAN. If the reference of PLAN is already a plan (a choice of actions) then the plan will be checked and corrected if unsatisfactory.

iv) Scheming. If the object is the concept SCHEME then a new scheme is formed (see Section 2.4.4) and made the reference of SCHEME. If the reference of SCHEME is already a scheme (a choice of states) then the scheme will be checked and corrected if unsatisfactory.

v) Generalising. If the object is the concept POSSIBILITY then a conception is selected at random and generalised (see Section 2.4.1) and made the reference of POSSIBILITY. If the reference is already a conception then that conception is generalised.

In the above where ACTION, PLAN, SCHEME and POSSIBILITY are mentioned any concept for which these are substitutable is accepted.

If a plan or scheme cannot be formed, or if a conception cannot be consistently generalised or an action is negative then the COGITATE conception will fail.

As "data" the act is used to incorporate the meaning of many English mental-action verbs, for example, "ponder", "consider", "plan", "think", "wonder", and "decide", although some uses of these verbs can be better realised by TRANSMIT or DO. Some examples of the use of COGITATE are:

John wondered whether to go to the cinema.
JOHN COGITATE <JOHN TRANSFER SELF A PLACE CINEMA>.
John loves Mary.
JOHN COGITATE <MARY BE>
    CAUSE JOHN BECOME [LOVE] SELF.
John wondered if he'd done the right thing.
JOHN COGITATE <JOHN [PAST] DO AN ACTION>.
John decided to give Bill the book.
JOHN COGITATE <JOHN PASS BOOK SELF BILL>.

2.3B4. DO

ANY LIFE DO ANY THOUGHT.

This act is used to carry out a thought, which may be a conception, program (i.e. a conception-choice) or plan:

i) Conception: The conception is evaluated; this is equivalent to:

SELF TRANSMIT A THOUGHT HERE SELF.

ii) Program: the program is evaluated; if it fails then any side effects are automatically undone.

iii) Plan: the first action of the plan is removed from the plan and evaluated.

This act occurs in the deep structure of verbs involving a number of unspecified actions, for example, "make", "build", "grow" and "drive" and also verbs in which the actions involved are even less specific, such as "like" and "want". In some cases it may be possible to say what some of the actions are but not all. In these cases DO expresses the fact that not all the actions involved have been specified, for example:

John grows roses
JOHN DO <JOHN TRANSFER FERTILISER A BAG
SOME GROUND <NEAR A ROSE>
AND JOHN TRANSFER SOME WATER A PLACE
SOME GROUND <NEAR A ROSE>>
CAUSE SOME ROSE BECOME [GOOD] SELF.

In this way it is possible to add to the systems knowledge of what growing involves and yet at the same time to include the fact that growing is more than that.

2.3B5. IDENTIFY

(ANY THINKER ANY LIFE) [CAN] IDENTIFY ANY PATTERN.

The object of IDENTIFY is compared by the subject with its perceived view of the world. If the subject is not a THINKER then this view must be the visual field of the subject as perceived by the PERCEIVE act. Otherwise, it may be an internal mental view. For example:

FIDO<BELONG JOHN>IDENTIFY JOHN<NEAR DOG>.

but:

JOHN IDENTIFY FIDO<BELONG JOHN><FAR JOHN>.

That is, JOHN can compare an internal mental image of his dog with other internal mental images (say to describe his dog or to identify a picture of his dog), but his dog can only compare its internal mental image of John with its visual impression of John. Thus IDENTIFY is used to represent the deep structure of verbs such as "recognise", and "identify".

However, the IDENTIFY act was not introduced so much for its use in allowing the deep structure of certain verbs to be represented more accurately, but because it could be associated with a new range of capabilities for the PIDGIN system, namely a powerful pattern-matching ability. This pattern-matching ability was found to be desirable when tackling complex problems such as the chess endgame problem because without it a long-winded linguistic description was required together with an even more complex set of IF rules so that the linguistic pattern matcher could deduce that "above and to the left" is the same as "left and above" and so on, on the chess board.

From the syntax rules (Section 2.1.1) it can be seen that a pattern may be either a lattice, a grid, a line or an actor. A lattice is a three-dimensional pattern, a grid is a two-dimensional pattern, and a line is a one-dimensional pattern of actors. When the IDENTIFY act is evaluated as a command the pattern which is its object is compared with the pattern which is the current reference of the concept VIEW. Depending on the modifiers of IDENTIFY, its object pattern and the VIEW pattern the IDENTIFY conception will either succeed or it will fail. A simple pattern is:

<<A B C> <D E F> <G H I>>

Which represents the grid:

```
A B C
D E F
G H I
```

As the members are actors they may be bands or classes of actors, for example:

```
<<WALL       WALL       WALL>
<TABLE [CUP SAUCER] TABLE>
<TABLE        TABLE  (SPOON FORK)>>
```

If this pattern was the reference of VIEW then the command:

```
SELF IDENTIFY SOME CUTLERY.
```

would succeed leaving the reference of the concept CUTLERY equal to (SPOON FORK). Of course, if the conception had been:

```
SELF IDENTIFY A FORK.
```

it would have failed (according to the matching rules for actors, Division 2.4.2B). A pattern can be set up in VIEW by the TRANSMIT act (see Part 10), for example:

```
SELF TRANSMIT * <<A SQUARE WKING A SQUARE>
                <A SQUARE WPAWN A SQUARE>>
                HERE VIEW
```

would transmit the 2x3 grid to VIEW.

The view (the pattern that is the reference of the concept VIEW) can be regarded as the systems "camera" picture of the world and also as its internal imagination. However, it will become outdated as soon as any actor in the view is moved. To prevent this if any object is transferred all view are examined and suitably updated. These changes are side-effects so if the rule containing the transfer later fails then they will all be undone automatically. This is useful during problem solving because it enables the system to tryout possibilities "in its imagination" so that they can be rejected if unsuccessful.

The matching ability of IDENTIFY is based on the PIDGIN conception matcher described in Section 2.4.2. A conception can itself be thought of as a pattern that is a linguistic picture of the world, each actor in it is a quantified and qualified nominal concept (called Picture Producers by Schank), and the act specifies the framework in which the actors sit in certain relationships (subject, object, modifier and so on).

If the object pattern is a single actor then it is simply compared with each actor in the view and will succeed if anyone succeeds. However if both the object and view have dimensionality then the possible comparisons become more complex. To begin with the object pattern must be the same or a lower dimensionality than the view and further, if they have the same dimensionality then the object pattern must have the same number or fewer members. The object pattern is then "passed over" the view in the following order:

A lattice is a series of applications which are grids; the first (that is the first to be matched) is the FRONT-most, the last the BACK-most.

A grid is a series of lines; the first is the ABOVE-most, the last the BELOW-most.

A line is a series of actors; the first is the LEFT-most, the last the RIGHT-most.

FRONT, BACK, ABOVE, BELOW, LEFT and RIGHT are relations used to specify the relative positions of two actors. If the view is a lattice all six apply, if a grid then only ABOVE, BELOW, LEFT and RIGHT, if a line only LEFT and RIGHT, and if an actor then none of them.

IDENTIFY may be modified by a degree and a type modifier in order to further describe the matching algorithm required. The degree modifier may be

MATCH, SIMILAR, ROTATE or MIRROR and MATCH is assumed if the degree modifier is omitted. The type modifier may be SAME, RSUB, VAGUE, TYPE or LIKE to specify how each actor is to be compared. RSUB is assumed if the type modifier is omitted. Thus a total of 20 different styles of matching may be performed.

a. Degree Modifiers

i) MATCH. This is the default if the degree modifier is omitted. It specifies that the object pattern and view are to be compared in all the possible ways that, without rotation, the object pat tern can be entirely "covered" by the view. One pattern entirely covers another if and only if every member of the second pattern has a corresponding member in the first pattern. Another way of describing this is to imagine the view as an infinite lattice (or grid, or line) every point of which contains the concept THING except for those points explicitly specified by the pattern itself. The concept THING can be substituted for any other concept so, as it is being matched against, it will fail (unless matched by the concept THING). Similarly, the object pattern should be imagined as an infinite lattice (grid, line) of THING points. Matching starts at the front, top left and the object pattern is "slid" over the view in the order described above (cf. reading a book). This continues until the back, bottom, right is reached, when the IDENTIFY conception will fail, or until a match is found, when it terminates the search and immediately succeeds. A 2x2x2 object pattern can thus be compared at eight different positions on a 3x3x3 view pattern if the degree modifier is MATCH.

ii) SIMILAR. This is the same as MATCH but will succeed the first time that more than half the actors of the object pattern are matched.

iii) ROTATE. This is like MATCH but at each position the object pattern will be rotated to try to match it. If the object pattern is a line this will involve a maximum of the same number of comparisons as a MATCH; if it is a grid it will involve four times as many and if a lattice twenty four times as many. iv) MIRROR. This is like ROTATE but it also tries rotating the object pattern in the next higher dimension. This is only used typically for a line, when twice as many comparisons are possible, and for a grid, when twice as many comparisons as for ROTATE are possible. To MIRROR compare a 31<:3 object grid with an 8x8 view grid involves a maximum of 6x6x4x2 or 288 positions for comparison.

b. Type Modifiers

i) SAME. The above degree modifiers are used to specify how the two patterns are to be moved over each other, the type modifiers are used to specify how each pair of actors is to be compared for a successful match. SAME indicates that two actors should only match if they are the same or equivalent (see Section 2.4.2).

ii) RSUB. This is the default assumed if the type modifier is omitted. RSUB specifies that the comparison to be made should be the same as that used by the matcher when comparing actors in conceptions (see Section 2.4.2).

iii) TYPE. One of the facilities of the matcher is to be able to check that two actors are "vaguely" the same; two sorts of vagueness are handled, TYPE and LIKE, plus their combination VAGUE.

Most general terms in English are vague in terms of their extension that is in terms of what does and what does not count as a member of the set named by the general term. An analogous feature in PIDGIN is created by regarding a concept to be TYPE vaguely the same as another concept if it is a member or subset of the concept which is the immediate superset of the first concept. To put this in PIDGIN terms means rephrasing "subset" and "superset" in terms of substitutability. If there are two concepts, A and B, related by the substitutable relation SUB in the form:

A <SUB B>

then B is substitutable for A and is called the category of A. So for example, the category of JOHN might be MAN, and the category of DOG might be ANIMAL. One concept is a TYPE-vague match to another if the category of the first is substitutable for the second. The only restriction is that if either concept is an entity concept then the other must be. For example:

```
FIDO <SUB A DOG>.
ALL DOG <SUB AN ANIMAL>.
ALL CAT <SUB AN ANIMAL>.
ROVER <SUB A DOG>.
```

then ROVER is TYPE-vague the same as FIDO, and DOG is TYPE vague the same as CAT, but FIDO is not TYPE-vague the same as CAT.

Note that as the number of group concepts increases so TYPE vagueness decreases. For example, if dogs are distinguished as ALSATION and POODLE then Alsatian dogs will only be TYPE-vague the same as other Alsatian dogs.

iv) LIKE. With LIKE-vague comparisons only the qualifiers are compared. Two concepts are LIKE-vague if more than half of the qualifiers of the concept with the least number of qualifiers (but at least one) match some qualifier of the other concept. For example, if:

```
ALL PEBBLE <SUB A [ROUND] STONE
               <TEXTURE SMOOTH>>
ALL EGG <SUB A [ROUND] OBJECT <BELONG A HEN>
               <TEXTURE SMOOTH>>
```

then PEBBLE and EGG are LIKE-vague. As the number of defining characteristics increases so the number of LIKE-vague concepts will probably decrease.

v) VAGUE. This type combines TYPE and LIKE. Two actors are VAGUEly the same if the are TYPE-vague the same or if they are LIKE-vague the same or if their category concepts are LIKE-vague the same.

2.3B6. MOVE

ANY LIFE MOVE ANY BODYPART A PLACE$_1$ A PLACE$_2$.

This act is used to express verbs involving bodily movement, such as "punch", "walk", "eat" and "breathe". The body part specified as object must be part of the subject carrying out the action, and the places must be situated inside or close to the body of the subject. The act is used to express the verbs concerned with ingestion ("drink", "eat", "breathe") as well as expellation ("breathe"). For example, breathing consists of inhaling ("transferring air from near the body to lungs by moving chest out") and exhaling ("transferring air from lungs to near body by moving chest in"). The movement of this act involves two places, both outside but near the body, for example, eating may involve moving the hand from the food source to the mouth, walking involves moving each leg alternately forward.

```
John punched Bill.
JOHN [<DEGREE VIOLENT>] MOVE FIST A PLACE1
                   A PLACE2 <LOC BILL>.
John is eating a sandwich.
JOHN TRANSFER A SANDWICH A PLACE1 STOMACH
    THROUGH JOHN MOVE HAND A PLACE2 THE SANDWICH
                   <LOC THE PLACE1>
    AND JOHN MOVE SOME FINGER A PLACE3
         <FRONT THE SANDWICH> A PLACE4
         <BACK THE SANDWICH>
```

AND JOHN MOVE HAND THE PLACE$_1$ MOUTH.

The last example illustrates how the THROUGH connector can be used to fill in the details of an action to virtually any level of detail. In both examples the object and any body part mentioned are assumed to belong to the subject unless otherwise stated.

Evaluated as a command it is this act that would be responsible for controlling any robotic activity. For example, in a Winograd type of block moving environment the MOVE act would be used actually to control the crane. The THROUGH connector would be used in such circumstances to describe the required activity down to the level of individual movements that the devices attached to the system were capable of being commanded to do. It is assumed that with any set of devices there would be a certain group of primitive commands for controlling those devices and that higher level commands could be constructed using the THROUGH connector.

2.3B7. PASS

ANY THINKER PASS ANY OBJECT ANY OWNER$_1$ ANY OWNER$_2$.

This is one of the transfer acts (MOVE, TRANSFER, TRANSMIT and PASS). In this case it is the possession of an object that is transferred. This transfer is performed by the subject, who may be the donor (OWNER1) the recipient (OWNER2) or neither. The act does not imply ownership in a legal sense but possession in a social sense. It is the possession expressed by the relation POSS.

The act will usually be enabled by OWNER$_1$ possessing (POSS) the object OBJECT and it will usually produce OWNER$_2$ possessing the OBJECT and OWNER$_1$ not possessing it. Alternatively, the act may be enabled by OWNER$_1$ having (the HAS relation is usually defined in the general knowledge) the OBJECT and produce OWNER$_2$ having it without OWNER$_1$ losing it. The second type of transfer is that associated with diseases and knowledge and it is also the type of "passing-on" transfer that occurs with the act TRANSMIT.

The act occurs in the deep structure of verbs associated with transfer such as "give", "take", "sell", and "buy". These will often be accompanied by a corresponding change in location but this is not necessary. Some examples are:

John gave his book to Bill.
JOHN PASS BOOK SELF BILL.
John took a book from Bill.
JOHN PASS BOOK BILL SELF.

2.3B8. PERCEIVE

ANY LIFE PERCEIVE PATTERN.

This act is the instrument (specified using the THROUGH connector) of all acts concerned with sensing the external world, namely those TRANSMIT acts in which the source is one of the sense organs. I t can also be used to specify that the subject "sensed" the object without needing to specify the particular sense involved. For example:

```
Bill saw John kissing Mary.
BILL TRANSMIT <JOHN TRANSFER MOUTH A PLACE
                                     MOUTH<BELONG MARY>>
                                 EYE SELF
       THROUGH BILL PERCEIVE [JOHN MARY].
John smelt the rose.
JOHN TRANSMIT [SMELL] ROSE NOSE SELF
    THROUGH JOHN PERCEIVE ROSE.
```

2.3B9. TRANSFER

ANY FORCE TRANSFER ANY OBJECT ANY PLACE$_1$ ANY PLACE$_2$

This act refers to the physical transfer of an object from one location to another by the application of some force (animate or inanimate). The act would typically be enabled by the object being located at the source location (PLACE$_1$) and would produce the object being located at the destination location (PLACE$_2$) and not at the source location. However, more complex conditions and results could be specified, for example, that the force is able to apply itself to the source place and is sufficient to reach the destination location, that the object is transferable, and that the force has access to any instrument required to carry out the action. Some simple uses are:

```
John went to school.
JOHN TRANSFER SELF A PLACE SCHOOL.
John and Mary walked to the park.
(JOHN MARY] TRANSFER SELF A PLACE PARK
      THROUGH (JOHN MARY] [REPEAT] MOVE 2 LEG
                         A PLACE1 A PLACE2.
```

In the current system the TRANSFER act evaluated as a command automatically searches the current view for the object specified and if found updates the view to describe the object's new position.

2. 3B10 TRANSMIT

ANY THINKER TRANSMIT ANY THOUGHT ANY MIND$_1$ ANY MIND$_2$.

This mental transfer act is used to transfer between locations in one mind ((i), (ii) and (iii) below) or between minds ((iv), (v) and (vi) below). The following list gives all the possible source and destination locations for this act:

i) HERE. This entity concept can be used in the source position to transmit from the object position, or in the destination position to transmit to the object. This can be used in PIDGIN in a way that is ana1agous to assignment in a conventional programming language, for example:

SELF TRANSMIT A THOUGHT USER HERE.

would read (and trans1ate if in English) a thought from the user and make it the reference of the concept THOUGHT.

ii) MEMORY. Can be used to recall from memory (source position) or to store in the memory (destination position).

iii) VIEW. This concept allows the view to be accessed or altered.

iv) SELF. In the subject position of a conception this concept refers to the subject of that conception. If a thought is transferred to the SELF of the PIDGIN system then this is equivalent to transferring to HERE and to MEMORY; similarly if SELF refers to some other THINKER it is assumed to be transferred to the MEMORY of that THINKER.

v) USER. This is a group concept that includes all minds that PIDGIN can transmit to. Therefore, before PIDGIN can communicate to a person the name of that person must be defined as one for which USER may be substituted, for example:

JOHN <SUB A USER>.

vi) THINKER. A group concept that includes all minds that can transmit and be transmitted to, so:

ALL USER <SUB A THINKER>.

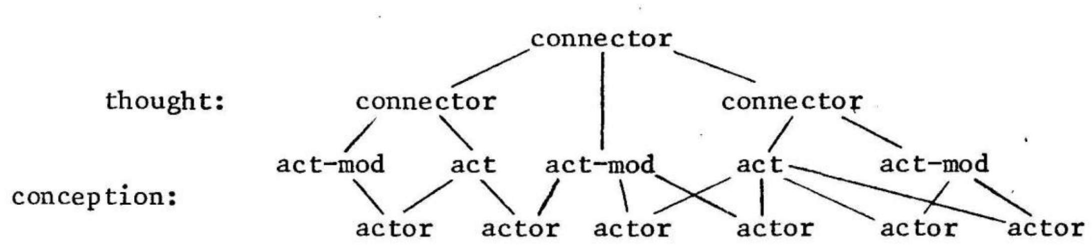
Verbs that include TRANSMIT in their deep structure are, for example, "remember", "learn", "speak", "teach" and "listen".

John recalled the time they met.
JOHN TRANSMIT<SELF<NEAR PERSON>>MEMORY SELF.
John told Bill to go.
JOHN TRANSMIT<BILL TRANSFER SELF A PLACE$_1$ A PLACE$_2$>
SELF BILL.

When a thinker is specified as destination it is assumed that the thought object is transmitted to that part of the mind of the thinker concerned with comprehension, in the case of the PIDGIN system the evaluator.

2.3C Actors

Actors are the PIDGIN equivalent of what in Schank's notation would be fully qualified picture producers or PPs. An actor is a qualified entity-concept or a quantified and qualified group-concept. The distinction between these two types of actor corresponds to the distinction between singular and general terms as described in Section 4.2.1. The memory of PIDGIN is thus organised at the lowest level into actors and these are brought together by the acts to form conceptions which may be further organised into thoughts by the connectors, thus:



An actor corresponds to a noun group in systemic grammar that is it is a complete description of an object set, for example:

John some apples

some apples

all the other ten very worn school books in the library

However, an actor is a conceptual unit in PIDGIN and a noun group is a syntax unit in English, therefore there are a number of differences. For example, definite descriptions are replaced by the entity-concept that is being described, and rank-shifted qualifying clauses are usually replaced by a separate conjoined conception.

2.3C1 Entity and Group Actors

An actor is essentially a qualified concept. There are two types of concept, entity and group concepts. An entity concept names or purports to name a single object and may not be substituted for any other concept (except an equivalent entity concept). A group concept is true of each, severally, of any number of objects. Corresponding to these two types of concept there are two types of actor, an entity actor which is an entity concept qualified by other concepts, and a group actor which is a group concept quantified (to define the number of objects being mentioned) and qualified by other concepts. The distinction between entity and group concepts runs through the complete system.

A subscripted group concept is treated as the same concept as its unsubscripted form except that it may have a different reference, but each subscripted entity concept is treated as a different (entity) concept, with a fixed reference.

A new concept is defined by specifying the concept that can be substituted for it, using the SUB relation. This concept must be a group concept which has itself been previously defined in this way. The complete set of definitions serves to define the relation of each concept to another and the complete set forms a tree, at the root of which are the group concepts which 'are built into the initial system, for example, the group concept THING. The built-in concepts have special features associated with them, for example, the acts, which are entity concepts, have already been described. The relations are group concepts with certain properties (described later) that they pass on to any concept for which they may be substituted. The primitive knowledge will usually contain a series of definitions that set up a range of new concepts that are then used to define the possible subject, object, source and destination

actors for each of the acts as shown in the last division. These
concepts form the basis of the world view set up later by the general and specialist knowledge.

The following parts deal with the possible quantifiers and qualifiers of these concepts in order to try to show the range and limitations of the actor structure in PIDGIN.

The following definitions show part of a typical primitive know ledge base that would organise the nominal concepts in a manner consistent with the model conceptions given in the last division:

```
ALL FORCE <SUB A THING>.
ALL OWNER <SUB A THING>.
ALL PLACE <SUB A THING>.
ALL OBJECT <SUB A PLACE>.
ALL BEING <SUB A THING>.
ALL THINKER <SUB A BEING>.
ALL LIFE <SUB A BEING>.
ALL BODYPART <SUB AN OBJECT>.
ALL BODYPART <PART A LIFE>.
ALL MIND <SUB A PLACE>.
ALL MIND <PART A THINKER>.
ALL MEMORY <SUB A MIND>.
ALL USER <SUB A MIND>.
ALL VIEW <SUB A MIND>.
HERE <SUB A MIND>.
SELF <SUB A USER>.
```

2.3C2. Quantifiers

This structure occurs only within a group actor. It defines the number of the nominal concept as being equal to, more than, less than or approximately equal to an arithmetic expression. The simplest form of an arithmetic expression is a number, thus:

three books	**3 BOOK**
more than four books	**<MORE 4> BOOK**
about one hundred books	**<ABOUT 100> BOOK**

More complex expressions involve the addition, subtraction, multiplication and division of numbers, numeric "variables" and actors:

Mary is twice John's age.
MARY <AGE <MULT =N YEAR <OLD JOHN> 2> YEAR>.

That is, the number of years of Mary's age is equal to two multiplied by the number of years of John's age (N), so if:

20 YEAR <OLD JOHN>.

Then the system can calculate that:

40 YEAR <OLD MARY>.

The other quantifier, ALL, plays a special role, it is not used to signify a particular numeric quantity but is used refer to the group as a whole, for example, in setting up definitions and attributes of the complete group. In order to discuss the adequacy of the quantifier construction in PIDGIN a few English quantifiers will be examined to see if, and how, they might be translated into PIDGIN. The following list gives a few English quantifiers:

very
nearly, almost
all, both, at least, half, third, quarter
a, no, the, some, any, each, another, neither, every, several
other, same, few, certain
one, two, three ... first, second, third

The quantifiers concerned with approximation ("very nearly", "nearly", "almost", "approximately", "aboutl1 and so on) are handled by the quantifier comparator ABOUT. This comparator assumes an (arbitrary) variation of 25% either side of its argument. If necessary PIDGIN could be altered to allow an explicit variation to be stated to cope with the difference between say "very nearly" and "nearly". However, it is very difficult to quantify such variations as they seem to depend on the magnitude of the quantity, the concept being quantified and more particularly with the use to which the assertion is being put (for example, a casual remark opposed to a scientific report). These difficulties though are difficulties of translation, of deciding on the implied variation, not limitations of the method of representation. If PIDGIN were extended to allow a variation to be explicitly stated then this could probably best be done by removing the ABOUT comparator and allowing the EQUAL, MORE and LESS comparators to take another parameter that specified the variation as a percentage. The present ABOUT comparator would then correspond to the EQUAL comparator with an explicit variation of 25. A single number as quantifier can be used to cope with the translation of "a", "an" and cardinal numbers:

a man	**1 MAN**
two men	**2 MAN**
half an apple	**0.5 APPLE**

In order to express averages, superlatives and fractions of groups the special relation modifiers MAX, MIN, and MEAN are introduced:

```
The oldest person died.
A PERSON <AGE <EQUAL =N YEAR <<OLD MAX>
                   ALL PERSON>> YEAR>
    BECOME [DEAD] SELF.
John is average height.
JOHN <HEIGHT <EQUAL =N METRE <<TALL MEAN>
    ALL PERSON>> METRE>.
The average salary of men is twice that of woman.
<DIV =N POUND <<SALARY MEAN> ALL MAN> 2>
                   POUND
              < <SALARY MEAN> ALL WOMAN>.
Over one third of the people in the world are hungry.
<MORE <DIV ALL 3>> [HUNGRY] PERSON.
```

It will be seen later that only certain relations may be modified by MAX, MIN and MEAN and these relations are those in which it is the quantity of the first actor that is of concern. Thus .the first example above may be read - there is a person such that his age is the maximum for all people and that person has become dead. In the second example the difference between this and:

=N METRE <<TALL MEAN> JOHN>.

should be recognised, the second suggests that John's height changes and it is currently average for John.

The third example states that the number of pounds of a woman's salary is that of a man's salary divided by two. The final example shows the other use of ALL within an expression to stand for the numeric quantity which is the total number of individuals making up the set described by the group actor.

Other English quantifiers such as "the", "other", and "same" are not really quantifiers, in the PIDGIN sense, but directives to the analyser to find and substitute entity concepts for group concepts wherever possible. In the deep structure noun groups qualified by these determiners will usually be replaced at the analysis stage by the entity concept that they describe.

Most of the other quantifiers reduce to the usual logical universal and existential quantifiers. Universal quantification is expressed in PIDGIN using ALL, and existential quantification by referring to a single member of a group concept. Some examples are given below:

Croydon is near a city.
(∃ x) (x is a city ^ Croydon is near x).
CROYDON <NEAR A CITY>.

There are no five-legged cows.
-(∃ x) (x is five-legged ^ x is a cow).
5 LEG <PART A COW> [NOT].

Something pleases Bill.
(∃ x) (x pleases Bill).
A PERSON DO AN ACTION CAUSE BILL BECOME
[HAPPY] SELF.

2.3C3. Attributes

An attribute is a concept that corresponds to what Schank calls a picture-aider (PA) and what in English is a general term that can be used adjectivally. For example, "red book" is true of all those things which it can be said that they are red and they are books. A new attribute is defined in the same way as a new nominal but by making it a concept for which the group concept ATTRIBUTE can be substituted:

ALL COLOUR <SUB AN ATTRIBUTE>
ALL RED <SUB A COLOUR>
ALL PINK <SUB A RED>.

However, it is difficult to build the attributes into a hierarchy that clearly defines their inter-relationships, as can be done for nominals. This is especially true for those attributes concerned with the feelings of living things and those concerned with the value that living things place upon objects. For example, Schank uses the attributes "comfortable" and "upset" (Schankl973b) without bringing in any mechanism for relating them together when it is clear that in order to use these attributes correctly the system must be able to do this.

Psychologists have tried to determine these connections between attributes experimentally. C E Osgood (Carroll 1969) took 50 dimensions or axes named by different pairs of adjectives and then got college students to rate a large number of nouns along these axes. By factor analysis it was found that the 50 dimensions could be reduced to three, which could be roughly described as an evaluation, a potency and an activity dimension. There are a number of dimensions that do not readily fit this trichotomy, for example serious-humorous, but the experiments have been repeated many times in different ways with the same general conclusion.

In PIDGIN all nominals can be rated along three dimensions which describe how "good", how "'strong", and how "active" that concept is. These three dimensions can be roughly related to the basic psycho logical processes of a concepts average reward value, the effort required to produce or resist it and the rapidity of movement associated with it. Thus it is assumed in PIDGIN that all nominals can be placed at some point within a three-dimensional "semantic space" in which closeness implies closeness of some aspect of the concepts meaning. For example, some actual experimental averages bring the following concepts into the groups shown:

(i) abortion, divorce, bad, feverish, crooked

(ii) calm, chair, table, statue

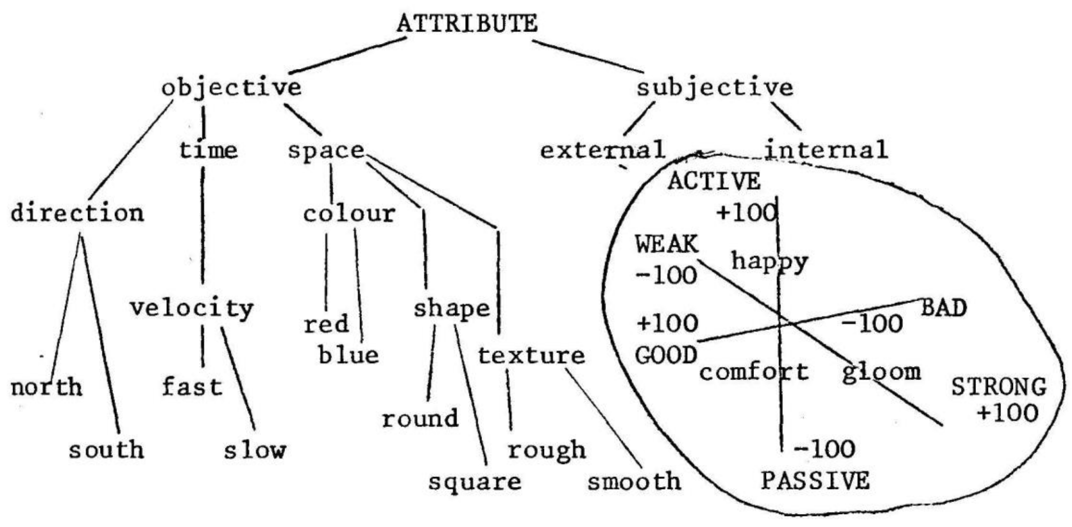
(iii) happy, patriot, leadership, brother, progress

(iv) art, nice, food, sky

Of course, any such table of groupings represent a single individual's view of the world. The point at issue here is whether it is possible to find some common method for finding such a table. The obvious way is to put together those nominal concepts that are associated with the same attributes. However, if there is no limit to the number of attributes and no connection between them then this method is of little use. In PIDGIN it is assumed that all nomina1s can be placed somewhere on a three-dimensional graph and that each area of that graph may be associated with some attribute in the sense that the attribute names that area of the graph. This has been incorporated into PIDGIN using two relations, FEEL and VALUE, and three dimensions GOOD, STRONG and ACTIVE rated (arbitrarily) between -100 and +100. So for examp1e:

```
ALL HAPPY <SUB AN ATTRIBUTE <FEEL [100 GOOD
                                   70 STRONG
                                   80 ACTIVE]>>.
ALL GLOOM <SUB AN ATTRIBUTE <FEEL [-80 GOOD
                                   -50 STRONG
                                   -50 ACTIVE]>>.
ALL COMFORT <SUB AN ATTRIBUTE <FEEL[50 GOOD
                                   20 STRONG
                                   -20 ACTIVE]>>.
```

Any attribute described as a feeling is concerned with the subjects own view of themselves (a subjective internal view), any attribute described as a



value is concerned with the subjects view of some object in the external world (a subjective external view) and all other attributes are concerned with describing the external world (an objective external view). This can be diagrammed as:

2.3C4. Specifiers

A specifier is a relation, or modified relation, followed by one or more actors and used in an actor structure to modify the nominal.

A specifier corresponds to a relative term in English, it is a concept that relates or links together an actor with one or more other actors. The twenty relations (ten spatial relations and ten others) built into PIDGIN are described later in this part.

New relations are introduced by using SUB to define their category. A new relation may be substituted for by one or more of the group-concept relation-describers RELATION, MEASURE, REFLEX, SYM, TRAN, LSUB, and RSUB with the following effect:

i) RELATION. If a concept, R, is a relation it can be used to connect two actors as A <R B> but it has none of the properties described below unless it is explicitly stated as having them by defining the relation as capable of being substituted for by the appropriate describer.

ii) MEASURE. If a concept, R, is a measure then it is a relation and can be modified by the relmods MAX, MIN, and ME~. For example:

ALL OLD <SUB A MEASURE>.
ALL SALARY <SUB A MEASURE>.

iii) REFLEX. If a concept, R, is reflexive it is a relation that is always true if used to link an actor to itself A<R A>. For example, everything is equivalent to and located at itself.

iv) SYM. If a concept, R, is symmetrical then if A<R B> then this implies B<R A>. Only a relation between two actors can be symmetrical, an example is the relation NEAR.

v) TRANS. If a concept, R, is transitive it is a relation such that if A<R B> and B<R c> then this implies A<R c>.

vi) LSUB. If a concept, R, is left-substitutable then it is a relation such that if A<R B> then A can be substituted for B in any conception containing B, without altering the truth value of the conception. Further A is said to be the "category" of B.

vii) RSUB. If a concept, R, is right-substitutable then it is a relation such that if A<R B> then B can be substituted for A in any conception containing A without altering the truth value of that conception, and B is said to be the "category" of A. The built-in SUB relation is of type RSUB.

The following describes each of the built-in relations and specifies its type.

a. SUB (TRANS, RSUB)

This is the basic relation for defining new entity and group concepts, including nominals, attributes and relations. A definition takes the form:

entity <SUB A group>
or ALL $group_1$ <SUB A $group_2$>.

where this declares "entity" to be of category "group", and "$group_1$" to be of category "$group_2$". Because SUB is RSUB this implies that "group" may be substituted for "entity" and "$group_2$" for "$group_1$". Further, because SUB is TRANS if "$group_1$" were the same as "group" then "$group_2$" could be substituted for "entity".

b. EQUIV (SYM, TRANS, LSUB, RSUB)

This relation is used to specify that two entity concepts or two group concepts are inter-substitutable in any conception containing them without altering its truth value.

c. INVERSE (SYM)

This relation is unusual in that it is only used to relate two concepts that are themselves relations. It specifies that they are inverse relations (brother-sister, part-contain, father-child). For example, if:

MARY <SISTER JOHN>.
and BROTHER <INVERSE SISTER>.
then JOHN <BROTHER MARY>.

If R_1 <INVERSE R_2> then if R_2 is REFLEX then so is R_1 if R_2 is SYM then so is R_1 if R_2 is TRANS so is R_1 if R_2 is LSUB then R_1 is RSUB and if R_2 is RSUB then R_1 is LSUB. However if R_2 is MEASURE R_1 is not.

d. OPPOSITE (SYM)

This relation, like INVERSE, is used to relate two relations. It is defined as, if R_1 <OPPOSITE R_2> then R_1 is the same type as R_2 and if A<R_1 B> is true then A<R_2 B> will be false and vice-versa. For example:

FAR <OPPOSITE NEAR>.
NOTLEFT <OPPOSITE LEFT>.

e. PART (REFLEX, TRANS)

This is used to express the English notion of "part of", thus:

A PISTON <PART AN ENGINE>.
AN ENGINE<PART A CAR>.

Therefore, because PART is TRANS:

A PISTON<PART A CAR>.

f. POSS (TRANS)

Used to express the notion of "possession" (without any legal associations), thus:

JOHN <POSS A DOG>.

g. FEEL, VALUE

These are used to relate concepts to the three-dimensional semantic space discussed in Part 3. FEEL is used to relate a THINKER to a point in the space or to define a concept as being that point. VALUE is used to assign an object some point in the space in order to express how some THINKER values that object.

Both relations take a single actor that may be one, or a band of two or three actors, either GOOD, STRONG, or ACTIVE, with a quantifier between -100 and +100. If any dimension is not defined (less than three actors) it is assumed to be zero but is free to vary, thus:

```
ALL BEAUTY <SUB A RELATION <VALUE [50 STRONG -80
ACTIVE>>.
A VASE <BEAUTY 50 GOOD>.
```

h. PRIORITY, CLASS (MEASURE)

PRIORITY is used to assign a priority (between 0 and 100) to, or to discover the priority of, a user name (see Division 2.2.2C). This relation may only be used by a user with a priority over 90 or more. It is used when setting up the system in order to create a number of user names (or "passwords") so that the system can associate every user with a priority and thus assign an "importance" to anything that user tells the sys tem.

CLASS is used to define the type of user - SYSTEM, GENERAL or USER (see Division 2.2.2C).

i. LOC (REFLEX, SYM, TRANS)

This relation corresponds to the LOC relation of Schank. It is used to express the location of one actor with respect to another. If a spatial relation (LOC, NEAR, FAR, ABOVE, BELOW, BACK, FRONT, LEFT, RIGHT, BETWEEN, DIST) relates two actors that match two objects in a current view then the view is used to answer questions about the objects. The spatial location of an object in a view cannot be altered by simply stating a new location but it is altered by using the TRANSFER act.

j. NEAR (SYM)

NEAR means that the two actors are either at the same location or they are next to each other. Note that, unlike LOC, it is not transitive.

k. ABOVE, BELOW, BACK, FRONT ,LEFT, RIGHT (TRANS)

These can be used to express the three-dimensional relation ships between two objects (see IDENTIFY Part 2.3B5). The relations express the strict meanings of the words and so are not opposites of each other, for example:

A			
	B		
	C	D	
			E

A <LEFT B>. B <ABOVE C>. E <BELOW D>. are true.

but C <RIGHT B>. D <ABOVE C>. are false.

If the opposite relations (NOTABOVE, NOTBELOW ...) were defined in the general knowledge, then:

C <NOTLEFT B>. D <NOTBELOW C>. would be true.

l. BETWEEN

This is used to express the idea that one actor is spatially between two others. The notion of numerically between is already built into the way quantifiers are handled (using a conjunction of MORE and LESS). The relation takes one argument which must be an unordered band of two actors, the symmetry of these is thus automatically built in:

LONDON <BETWEEN [PARIS NEWYORK]>.

will match

LONDON <BETWEEN [NEWYORK PARIS]>.

m. DIST (MEASURE)

This MEASURE relation is used to specify the distance between two objects. The distance can be specified in pattern units by using the concept UNIT, for example:

5 UNIT <DIST [WPAWN BKING]>.

n. The Relation Modifiers

As well as the above twenty relations there are three relations modifiers (MAX, MIN, MEAN) built-in. These can be used to modify any relation of type MEASURE. One use for this facility is to be able to express the deep structure of comparatives and superlatives, for example:

oldest	**<<OLD MAX> =ALL PERSON>**
older than x	**<AGE <MORE =N YEAR <OLD x>> YEAR>**
old	**<AGE <MORE =N YEAR <<OLD MEAN> ALL PERSON>> YEAR>**
same age as x	**<AGE =N YEAR <OLD x>>**
young	**<AGE <LESS =N YEAR <<OLD MEAN> ALL PERSON>> YEAR>**
younger than x	**<AGE <LESS =N YEAR <OLD x>> YEAR>**
youngest	**<<OLD MIN> =ALL PERSON>**

2.3D. Modifiers

A complete conception may be modified in anyone of the thirteen different ways discussed below.

a. Index

This is an integer that uniquely identifies the conception. The index number modifier cannot be specified by the user for a PIDGIN assertion; it is automatically assigned by the system as one greater than any previously used index.

b. Author

This is the concept associated with the user that created the conception. This concept can be thought of as the user's name, or password, that he must specify before he is allowed to use the system. Every user name is associated with a priority that is used to assign a relative importance to all the assertions made by the user with that name. Any conception that is created by PIDGIN is assigned the author concept SELF.

c. Priority

This is a number between 0 and one hundred that assigns a relative importance to the conception it modifiers. That is if one conception contradicts another the conception with the highest priority is considered correct. The user cannot specify any of these first three modifiers, index, author or priority, directly in a conception. The index is assigned automatically by the system, the author concept is specified once when the user first starts to use the system and the priority is already associated with the author.

d. Truth

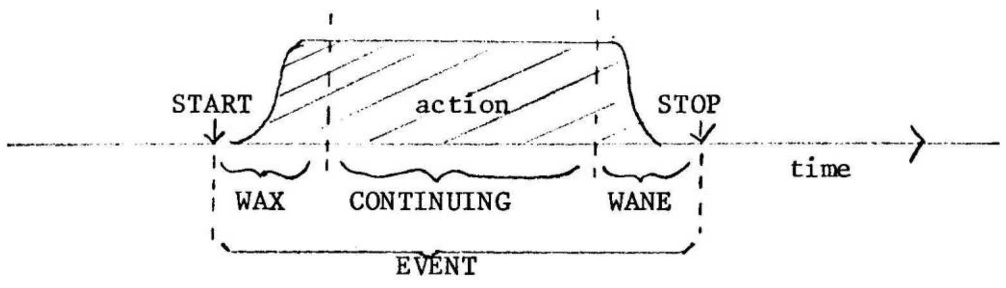
Any conception may be modified as to its truth value, either TRUE or NOT. NOT is used to indicate that the conception it modifies is false, TRUE that is true.

e. Modal

A conception may be modified as POSSIBLE, DEFN or neither. If a conception is POSSIBLE it is assigned a priority of twenty and will not be used to make deductions. A conception marked as DEFN is assigned a priority of 10 greater than the priority associated with the author concept, this allows a user to create conceptions that can be used to check the consistency of later (non-DEFN) conceptions asserted by the same user.

f. Period

This is used to specify any transitional aspect of the conception it modifies, that is, to specify that a particular phase or change is being mentioned. The possible modifiers can best be described by a diagram:



The other modifier, REPEAT, is used to indicate that the action is an event that is repeated (for example, moving the legs during walking) and this repetition is an essential aspect of the complete conception.

g. Manner

i) Intent

An action may be modified by INTEND or ACCIDENT to indicate whether the subject of the action intended the specified action or the action took place and involved the subject.

ii) Condition

If the conception is modified by CAN the subject has not or will not necessarily perform the action but could perform it in the appropriate circumstances if required. If an action is modified by CAN then any action that matches that conception can be performed without any enabling conditions being required.

iii) Disposition DISPOSED is used to mark those conceptions that express the idea of the subject being inclined to carry out the action from time to time, under the appropriate circumstances (for example, "John hunts lions", "Tabby eats mice").

h. Degree

This specifier modifier allows adverbs to be incorporated in the deep structure, for example:

<DEGREE SLOW>
<DEGREE IMPRESSIVE>

However, the meaning of these adverbs is not fully incorporated into the overall deep structure of the conception they modify; the DEGREE modifier merely provides a "slot" into which the concepts can be placed. This is an area in which PIDGIN could be extended to more fully incorporate the modification that an adverb has on its containing sentence.

<u>i. Location</u>

This is the same specifier that is used to qualify actors. If an action is associated with a particular location then it is incorporated in the location modifier. Any action that has a time always also involves some location but it does not need to be specified in the modifiers. Conversely any action with a location is associated with a time though once again this does not need to be specified even though the location is included. This modifier corresponds to what is usually called an "adverbial clause of place" at the English level. The location specified is the location of the subject when the act of the action is performed.

<u>j. Time</u>

Time is specified as an absolute date (relative to 0 A.D.) and time. Many relative times such as "now", "yesterday" and "next week" can be made absolute if the time of the utterance is known. However, there are relative times such as "John's birthday", and "Christmas" which may not be capable of being made absolute because the information they contain is lost if they are replaced by anyone of the absolute dates to which they refer. This is another area where PIDGIN could be usefully extended in the way it handles adverbial modification. The current way that such relative times are handled is to specify the time as the actor that describes the relative time. However, this does not fit into the way in which absolute times are handled.

Absolute time is specified as a number of years, months, days, hours, minutes and seconds, though they are not all required if the time is not known precisely, for example:

```
<TIME [22 DAY 8 MONTH 1947 YEAR]>
<TIME 1975 YEAR>
```

NOW is taken as the current date and time as known by the system. PAST is any time before (LESS) NOW and FUTURE any time after (MORE) NOW. The problem of how far in the PAST or FUTURE is resolved by always specifying the time NOW to the nearest second although often the past or future tense is used to signify before or after some more approximate period such as before or after today, or this year.

k. Interval

This modifier is only used if EVENT is also used. It gives the length of time of the complete EVENT (from START to STOP) in years, months, days, hours, minutes and seconds or some more approximate figure, such as:

<INTERVAL [2 YEAR 6 MONTH]>

That completes all the allowed modifiers for the conception. Thoughts may also be modified (except for WHILE) by the same modifiers, in which case the modifier applies to the complete thought.

2.3 E. Combining Concepts

Now that the concepts have been described it is necessary to consider how they are put together to form statements. The syntax of Section 2.1.1 does not rule out, for example, combinations such as [SAD] TREE or

A BEE TRANSMIT A THOUGHT A FLOWER A FLOWER$_2$.

These constructions are ruled out by the knowledge contained in the memory plus the rules for consistency. For example, the primitive knowledge might contain the conception:

ANY THINKER TRANSMIT ANY THOUGHT ANY MINDI ANY MIND2.

The system will only add a new conception to the memory if either it has a priority of one hundred or the memory already contains a conception that matches it. This rule implies that the priority one hundred conceptions define the limits of all possible future combinations. Putting this another way it means that the system can only learn new particular examples of what it already knows. For example, the above conception rules out : A BEE TRANSMIT A THOUGHT A FLOWER A FLOWERI. unless BEE is defined as THINKER and FLOWER as MIND. This forces the user to define the categories of the concepts used before they can be combined into conceptions. Any concept can be defined initially as the category pf any other because the primitive knowledge will usually contain the conception:

ANY THING BE.

As the system is given more knowledge so the range of possible combinations is restricted by those of higher priority. For example, the high priority general knowledge will restrict the specialist knowledge and this will still further restrict the knowledge that the user may give the system. In this way meaningless combinations of words at the sentential level cannot get through to the deep structure because the analyser will not be able to create the corresponding statement. Further, the analyser can use this mechanism to help it choose between alternative deep structures when the sentence is ambiguous.

Schank considers the problem posed by sentences such as:

John eats a light bulb.

With such sentences the analyser cannot create a deep structure because the dictionary entry for "eat" insists that only "food" can: be eaten and "light bulb" is not a food. Light bulb could be added to the list of concepts of type food but this would have undesirable consequences, for example, all questions concerning food would imply light bulb as a possibility and the unusualness of the event would be lost. The solution is to add the complete conception to the dictionary as a new possible translation of eat. . The dictionary would then contain the equivalent of :

Any person eat any food
or John eats light bulbs

When a new concept is defined it initially inherits all the possible combinations of its category concept. However, a new concept is usually distinguished because it differs from its category concept in some quality dimension. For example, "mare" is a subset of "horse" and it differs from "horse" in the "sex" dimension. Whereas, "horse" may have any sex the new category "mare" is, by definition, "female". This knowledge could be contained in the memory as:

ANY [SEX] ANIMAL.
ALL HORSE <SUB AN ANIMAL>.
ALL MARE <SUB A [FEMALE] HORSE>.

Here HORSE inherits from ANIMAL the possibility of being qualified by SEX (MALE or FEMALE) but when MARE is defined this possibility is excluded by explicitly stating its sex as FEMALE. To incorporate conceptual knowledge concerned not with a concepts defining qualities but with its typical properties the quantifier MOST can be used:

ALL BAIL <SUB A [ROUND] OBJECT>.
MOST [SMOOTH] BALL.

that is, the concept BAIL is by definition round and is usually smooth. The fact that they are usually smooth does not prevent the system creating the structure [ROUGH] BALL but it may help the analyser to disambiguate a sentence in which the texture is not mentioned but in which a knowledge of the most likely texture would distinguish between possible translations.

Finally, possible qualities can be specified:

MOST [WHITE] SNOW. SOME [YELLOW] SNOW.

although they cannot be used during analysis.

Entity concepts can only be combined with qualities which can be combined with their category concept.

2.4 The PIDGIN Statement

2.4.1 Assertions, Questions and Commands

PIDGIN statements can be divided into three types, assertions, questions and commands. These three types are distinguished by certain aspects of their structure and by the preceding dialogue. Assertions usually convey information, questions elicit information and commands instruct the system to carry out some action. However, the practical handling of these three types is not quite this straight-forward; assertions may convey information already known by the system or contradict what the system already knows, questions may require a simple yes/no answer, require complex deductions to be performed or even convey information and commands may not be able to be obeyed.

Assertions are indicated in Input PIDGIN by being terminated by a full-stop (see Section 2.1.2). The terminating full-stop sets a global indicator to signify that the deep structure currently being evaluated is an assertion. Both assertions and questions are treated in a similar way. They are matched against the memory in order to try and find a matching statement. In the case of an assertion if a matching statement is found this means that the system already knows the information that the assertion contains, in the case of the question this means that an answer has been found. If an assertion matches a statement already in the memory then that statement must either be identical to the assertion, more explicit than the assertion or a contradiction of the assertion. When checking for consistency of the complete assertion each actor in the assertion is separately checked against the world model. For example, the assertion:

JOHN PASS A [WHITE] RAVEN SELF MARY.

is contradicted by the state:

ALL [BLACK] RAVEN BE.

in the world model.

At the PIDGIN-level when an assertion is evaluated it will either succeed or fail. If the user is working at the English level then the English-level driver program will translate this into a suitable explanatory English reply. At the PIDGIN-level the action taken is determined by the ABC PIDGIN-driver program, this will, typically, store the assertion in the memory if it succeeds and output it back to the user if it fails. The following table shows the possible typical responses at the PIDGIN level : Assertion input:

i) No matching statement in memory

assertion stored in memory.

ii) Matching statement identical

output assertion.

iii) Matching statement more explicit

output statement.

iv) Matching statement contradictory and same or lower priority

assertion stored in memory.

v) Matching statement contradictory and higher priority

output statement.

Checking for consistency is a very important part of the way PIDGIN works. Rather than accept the assertions of the user, which can lead to inconsistencies arising, each one is checked with the current memory. Only if the assertion is consistent with the memory is it stored. This greatly aids the user because he is kept in contact with the contents and implications of the growing system. Of course the user must be allowed to store an assertion in the memory even if it is contradictory because he may wish to correct an earlier mistake (this can be done by using the DEFN modifier), and inconsistencies can arise, but this is preferable to having no checks.

Questions are treated in a similar way to assertions. In Input PIDGIN a question is signified by terminating the statement with a question-mark (see Section 2.1.2). A question is typically a general assertion that retrieves a more explicit statement from memory, however, in the case of the "yes-no" question the question is as explicit as an assertion. The only difference in the way that the system handles the two is that if no matching statement is found the assertion is stored in the memory but the question is not. So the question response table is the same as the above assertion table except for the first "no-match" case which, for a question, does not result in it being stored in the memory but some message, such as [I DONT KNOW.], being output instead.

Questions may be about the working of the system itself. For example, "how" and "why" questions are concerned with planning and scheming (see Section 2.4.4) respectively. The system automatically adds assertions to the memory when it carries out a plan and a plan will generate new states. Therefore, questions concerning actions that have been obeyed can be answered from the memory ("how" question from THROUGH sub-actions and "why" questions from the states produced). Questions concerning future actions can be answered from the current scheme and plan.

A command is a conception containing one of the action-acts with the concept SELF as subject and a time modifier of NOW. Unlike assertions and questions it does not cause the memory to be searched but it causes the ABL expression associated with the action-act to be evaluated. In Input PIDGIN a command is terminated by either a full stop or an exclamation-mark. The primary indicator of a command is a conception with SELF as subject. If the PIDGIN interpreter is given such a conception and the time modifier is NOW then it immediately evaluates the ABL expression associated with the act. This expression first checks that the command is enabled in the current memory. A command is enabled if either a matching conception has the modifier CAN or every ENABLE thought with a matching action has states that can be found in the current world model. If the command is enabled then the appropriate action is performed (see Division 2.3B), if not the command fails. After performing the appropriate action a check is made for any PRODUCE thought with a matching action part. All such thoughts that are found in the memory have their states added to the current world model. A conception with the form of a command but with a time modifier indicating the past is treated as an assertion (about some past command).

A conception with the form of a command but with a time modifier indicating the future is called a suspended command and it is stored in a special part of SEM, suspended evaluation memory (see Segment 2.2.3BIb), with a reviver based on the time it is to be obeyed.

The actual action performed by a command depends on the facilities of the complete system. For example, at one extreme the system may be connected to a number of peripherals in the real world, cameras (PERCEIVE and IDENTIFY acts), 'arms' (MOVE and TRANSFER acts) and so on, and at the other extreme it may be connected to simply a single input/output terminal. In the later case all the commands must be simulated inside the computer in such a way that the PIDGIN system is presented with an environment similar to that which would be presented by the real devices. This later system is that used by Winograd's question-answering system.

There are many types of sentence not covered by the above division into assertion, question and command, for example, jokes, greetings, exclamations, thanks and so on. This means that much of the subtlety of such sentence will be lost because the translator can only translate it to one of the three types provided. However, in the problem-solving environment in which PIDGIN is designed to work the information lost should not be relevant. Another difficulty is illustrated by the following example, if a headmaster says

"The boy responsible will tell me the reason why he did it."

then this can be taken as a hope by the staff, an assertion by the pupils and a command by the boy himself. It could also be taken in many other ways, such as a threat or a question. The way it is understood by each hearer depends on that hearer's world view, knowledge of the speaker and current world model. PIDGIN can handle some of the subtlety implied by this large number of sentences types by the use of more rules. For example, a threat can be treated as an assertion plus a rule that links such assertions to the possibility of the subject of the assertion harming the object. Such a rule would enable PIDGIN to deduce the possibility of harm but to recognise the first assertion as an example of a threat it would be necessary to define the meaning of the concept THREAT as any assertion for which there is a matching rule that implies the subject of the assertion may harm the object.

Another way that the PIDGIN system can be used is to run it in a mode that causes it to try to extend and improve its knowledge by asking the user questions. This can be done by taking any conception that is not fully modified and qualified. For example:

John takes a book.
JOHN PASS A BOOK AN OWNER JOHN.

is the source of the following questions, and many more:

Which book?
Who owned the book?
When did John take it?
How did John take it?
Why did John take it?

Questions can also be generated by forming hypotheses. A hypothesis is generated by taking a statement, making its concepts vaguer and checking the result for consistency with the current memory. For example:

John takes objects.
People take books.
People take objects.

The problem with allowing PIDGIN to generate such questions and hypotheses is to find some way of restricting them to the most likely possibilities. This requirement goes beyond the current scope of PIDGIN.

2.4.2 Substitution Rules

Matching is the most important basic process in the PIDGIN system as it determines conception equivalence, which determines what questions can be answered, what commands obeyed and what problems solved. This section describes all the rules used in the matching process. These rules are called substitution rules because they describe all the conditions under which one statement may be substituted for another.

Unlike most matchers, such as those of SNOBOL or PLANNER, PIDGIN does not match "blindly" but uses the semantic information available to limit the possible matches. Further, as all matches take place between two statements the matcher works with items that have limited syntax and fixed, known semantic relations. This enables an efficient implementation to be made. For example, the first item in a conception is always one of the 12 acts (nine acts plus BE plus the two general acts). The system can therefore not only divide the memory according to this first item but each act can be associated with a different matching algorithm that takes into account the syntactic and semantic restrictions that the act imposes on the whole conception. Further, the semantics of actors and attributes suggests that they may be efficiently related by means of tree structures. Thus, although the memory is regarded throughout as a linear sequence that is scanned sequentially it can in fact be implemented as an interconnected network, similar to memory structures such as that of Quillian. The form of this network is not described, however, as it is regarded as an implementation detail that should play no part in the design of the system. The rigid specification of the deep structure ensures that an efficient implementation is possible but the details of this are left to the implementor (one possibility is discussed in Appendix I). The justification that is some times given for describing networks is because of their obvious analogy to the central nervous system. However, so little is known of the structure of nerve nets involved with such complex activities as language and problem solving that little appears to be gained from such conjectures. PIDGIN can be regarded and described in terms of networks but for the purpose of presenting it as a programming language it is simpler to think of it as a sequence of statements.

The matching ability of PIDGIN centres on matching two actors together. This ability is available directly using the act IDENTIFY (see Part

2.3B5) which can be used to match actors or groups of actors with a group of actors called the view. The matcher is also used implicitly throughout the system, for example, to answer questions and to check for consistency. The implicit matching process always takes place between two statements and consists of three parts, the actor matcher, the modifier matcher, and the conception matcher which uses the other two.

2.4.2A The Matcher

The matcher takes a single statement and tries to find a matching statement in the memory using the substitution rules and the relations between the concepts. The single statement is called the picture and is typically a question and this is matched with each statement in memory in turn until either a successful match is found or the memory is exhausted and the match fails. As each statement in memory is examined it becomes known as the pattern or candidate, and is typically an assertion.

If a successful match is made then a "fail-point" is set up in the matcher so that a later failure can return to the same point and continue searching. If the memory is exhausted and the picture is a conception then the matcher examines all the IF-rules for one with a matching header (the first conception of the rule). If there are no IF-rules that match or all matching IF-rules fail then the complete match fails. If a matching IF-rule is found then a fail-point is set up and the rest of the IF- rule is evaluated to determine if the match is successful.

This fail-point mechanism is also incorporated in PLANNER for a similar purpose. However, it is hoped that the complexity of a single conception compared with for example Winograd's PLANNER data-base patterns will enable a more efficient system to be 'set up because of the reduction in failure backtracking. This follows from the fact that simpler patterns will match more often and so if one complex pattern is replaced by a number of simpler patterns each of which must be matched against the complete memory and each of which may involve a large number of backtracks then the complete process will take longer. For example, in Winograd's system to find "a large red cube" required the Micro-Planner commands:

```
(THGOAL (#IS $?Xl #BLOCK))
(#EQDIM $?Xl)
(THGOAL (#COLOR $?Xl #RED))
(THGOAL (#SIZE $?Xl #LARGE))
```

that is, first find a block, then check it is a cube, if not backtrack to find another block, then check it is red, if not backtrack to find another block, then check it is large, if not check for another colour and another block. This continual backtracking becomes even worse when the equivalent of a complete conception is considered.

It is interesting to note that R. Schwarcz (1970) and R.F. Simmons described a similar problem with their Protosynthex III system and suggest a similar solution. With a large data-base these backtracking memory searches become increasingly slow because of the combinatorial explosion and Schwarcz suggests two possible ways of alleviating this:

(i) "partitioning the data base into discourse units".

In effect this means not looking at parts of the data-base because it is known that they do not contain the answer. This amounts to the system having some idea of where to look before it starts the search and the problem is how to order or to divide the data-base so that the questions that occur can make use of the ordering and division. PIDGIN goes a small step along this path because its syntax allows the system to partition on the subject nominal and the act. For example, it can divide the data-base into those conceptions concerned with PERSON PASS, PERSON MOVE, PERSON TRANSFER, and so on. This is a graph-partitioned data-base which is strongly tree-partitioned because of the structure of the concept relations. That is, PIDGIN could be implemented with a meaning structured data-base so, for example, to find all conceptions concerned with "CHILD MOVE" it need only look first at the THING-ANIMAL PERSON-CHILD concept structure and then in detail through just the CHILD-MOVE conceptions. The advantage of this system is that the nominal concept structuring automatically increases in complexity with data-base size. The increased length of time required to search a larger data base should be offset to some extent by the finer partitioning resulting from the typical corresponding increase in the dividing of concepts. The extent to which one offsets the other has not been investigated by practical comparisons.

(ii) "use a basic unit larger than the relational triple".

Protosynthex, and to some extent Winograd's use of PLANNER, is based on a data-base unit consisting of three parts (relation plus two arguments). Schwarcz suggests that by increasing the size of the unit and thus reducing the number of sub-goals and the number of answers to each sub-goal the system would be quicker even though each individual comparison takes longer. He also states:

"Thus the substitution of the Fillmore case structure for the structure of event triples would yield substantial benefits for natural language deductive question-answering systems."

PIDGIN is based on Schank's case structure which is an improvement of Fillmore's (see Schank 1969b) because it is concerned with the conceptual deep structure not the surface syntax and also because it is designed with the computer in mind.

Although the PIDGIN matcher may be improved by the above techniques it is easier to understand if it is imagined that it searches the complete data-base for a match each time. If the statement is a thought only the corresponding thoughts need be examined (those with the same connector) and each one is matched by first comparing each corresponding conception and then comparing modifiers. If the statement is a conception only corresponding conceptions (those with the same act) need be examined and each one is matched by first comparing corresponding actors (subject, object, source and destination) and then comparing modifiers. This may involve recursively matching conceptions as the object of a conception may itself be a conception. When two actors are compared it is the reference of the picture actor concepts that are compared with the corresponding concepts of the pattern actor. The reason for this is described later.

If the matcher finds a match then the binder (see Division 2.4;2D) is used to make the pattern concepts the reference of the corresponding concepts of the picture statement.

2.4.2B Actor Matching

The essential point about matching two actors is that not only do identical actors match but if the picture is substitutable for the candidate then they also match. This division describes when one actor is substitutable for another.

2.4.2B1. Combinations of Actors

An actor may be a single actor or a band or class of actors. When matching two actors there are nine possibilities all of which, except the last, recursively call the actor matcher:

i) Picture actor is a band, candidate a band (conjunction conjunction) . Each member of the picture band is matched against the candidate band and must match at least one member. A complex case is illustrated by the following example: John, Bill and Jill went bowling. Did three people including two men go bowling? The above procedure would succeed with this example. Note that the question is ambiguous, it may mean exactly or at least the numbers specified. The two interpretations result in different deep structures, either EQUAL or MORE being used as the quantifier.

ii) Picture is a band, candidate a class (conjunction-disjunction). The complete picture band must match every member of the class.

iii) Picture is a band, candidate a single actor (conjunction actor). Every member of the band must match the single actor.

iv) Picture is a class, candidate a band (disjunction-conjunction). At least one member of the class must match the complete band.

v) Picture is a class, candidate a class (disjunction-disjunction). Every member of the picture class must match at least one member of the candidate class.

vi) Picture is a class, candidate a single actor (disjunction-actor). At least one member of the class must match the actor.

vii) Picture is an actor, candidate a band (actor-conjunction). The actor must match at least one member of the band.

viii) Picture is an actor, candidate a class (actor-disjunction). The actor must match all the members of the class.

ix) Picture is an actor, candidate an actor (actor-actor).

The quantifier, attributes, specifiers and nominals must match as described below. Nested combinations of the above cases can be matched using the above rules. Thus the following two questions would succeed:

(Two boys and three girls)or(two girls and three boys) went bowling.
Did five people go bowling?
Did at least two boys and at least two girls go bowling?

A single actor has the form (Input PIDGIN):

quantifier [attributes] nominal specifiers

therefore to match two actors involves matching four parts. The first part matched is the two nominals, they match if and only if:

i) they are the same concept or either is in the EQUIV relation to the other.

ii) the picture nominal is substitutable for the candidate nominal (see Part 2.3C4 for an explanation of concept substitutability).

iii) the candidate nominal has the quantifier ALL and is substitutable for the picture nominal.

If this comparison is successful the other three parts are matched in the order quantifier, attributes then specifiers.

Any world model state containing one of the nominals quantified by ALL may be used in order to check the general attributes and specifiers of that nominal.

2.4.2B2 Quantifier Matching

Each quantifier can be an expression, number, number concept, comparison or approximation. If it is expression it is evaluated, if possible, and the following table specifies the way in which the two quantifiers are then compared.

			Candidate	
		C	<MORE C>	<LESS C>
	P	P=C	fail	fail
Picture	<MORE P>	P<C	P≤C	fail
	<LESS P>	P>C	fail	P≥C

If a number is specified as approximate (using ABOUT) then the number is allowed to vary by 25%. The following examples illustrate these rules:

Picture	Candidate	Result
<MORE 2>	<EQUAL 3>	succeed
<MORE 30>	<MORE 40>	succeed
<ABOUT 10>	<EQUAL 12>	succeed
<ABOUT 10>	<EQUAL 14>	fail
<EQUAL 5>	<MORE 4>	fail
<MORE 5>	<MORE 4>	fail

If one of the quantifiers contains an expression it is evaluated to a number before the comparison, if it cannot be evaluated because it contains a number concept that does not have a number as its reference then the match fails. However, if two expressions are being compared and one or both cannot be evaluated then the expressions themselves are matched. To make the comparison the order of the arguments of both ADD and MULT is ignored and it

is sufficient for the two expressions to use the same number
concepts in the same places but not necessarily the same number concepts in both expressions. Thus:

```
<MULT X <ADD 2 X>>
      succeeds with <MULT <ADD Y 2> Y>
      but fails with <MULT X <ADD 2 Y>>
```

Also a number concept in the picture will match any self-contained expression (an expression containing no number concept used outside that expression), thus in the limiting case a picture quantifier consisting of a single number concept will match any expression, for example:

```
the picture      <MULT A B>
will match       <MULT <MULT X X> <MULT Y Y>>
```

2.4.2 B3. Attribute Matching

An attribute is a group concept being used to qualify a nominal. They are matched in the same way as nominals except that attributes always occur in an unordered band.

For example, COLOUR may be defined as RED, BLUE and GREEN, and RED as SCARLET, ROSE and PINK, then if the picture is PINK the candidate must be PINK, if the picture is RED the candidate must be PINK, ROSE, SCARLET or RED.

The order of the attributes is not significant. If a picture attribute does not match the candidate it is checked with the world model states that have a matching nominal and the quantifier ALL. If is holds in the world model then the comparison continues otherwise it fails.

2.4.2 B4. Specifier Matching

A specifier is a relation between two (or more) actors, it takes the form:

(i) <relation actor>

or (ii) <<relation relmod> actor>

The relations are matched first, in the same way as two nominal concepts.

If either relation has the second form above (where relmod,. is MAX, MIN or MEAN) then both relations must have the same form and the same relmod. If one relation is the INVERSE (see Segment 2.3C4c) of the other then:

if the picture actor has the form:- A <R_1 B>

and the candidate the form:- C <R_2 D>

then A must match D and B must match C for the two actors to match. If a picture specifier does not match it is compared with the world model states that have a matching nominal and the quantifier ALL, if a match is found the comparison continues otherwise it fails.

2.4.2 C. Modifier Matching

The modifier is checked last. If two modifiers do not match the conception may still be a satisfactory answer to the question. There fore a note is kept of the conceptions that match except for their modifiers and if no other match is found then they are used. However, if the only difference is the truth modifier of the two conceptions then they are regarded as matching and the match succeeds with the question denied (or the assertion contradicted). The following rules are used:

i) Index: not checked

ii) Author: not checked

iii) Priority: not checked

iv) Truth: see above

v) Period: the two periods are matched like two attributes, if the picture is EVENT it will match any candidate period, other wise they must be the same.

vi) Manner: must be the same, except that if the picture is CAN the candidate will match if no manner is specified.

vii) Degree: the degrees are matched by the specifier matcher.

viii) Location: the location specifier of a complete conception takes the same form as the location specifier of an actor and is matched in the same way.

ix) Time: this is matched by the specifier matcher. The time may consist of one or more actors each of which specifies the number of those time intervals, the accuracy of the result is taken as being specified by the smallest time interval used. The candidate must be as accurate or more accurate then the picture, for examp1e:

Picture <TIME [2 MONTH 1976 YEAR]>
matches <TIME [3 DAY 2 MONTH 1976 YEAR]>
but not <TIME 1976 YEAR>

x) Interval: this is only specified if the period EVENT is specified. The interval is specified as one or more of a number of time intervals. The two are matched in the same way as the time modifier with the same rules as regards the accuracy of the interval.

2.4.2D Binding Statements

If a picture matches a candidate then the references of all the concepts of the picture are altered to be the corresponding concept of the candidate. This process is called binding. For example, if the assertion candidate:

JOHN PASS BOOK SELF BILL.

is matched by the question picture:

JOHN PASS AN OBJECT SELF BILL?

then the concept OBJECT in the picture will have its reference altered to BOOK by binding.

Subscripted concepts may be used so that the same concept may take different references, for example, matching the question:

A PERSON1 PASS AN OBJECT A $PERSON_1$ A $PERSON_2$?

will bind $PERSON_1$ to JOHN, OBJECT to BOOK and $PERSON_2$ to BILL. If binding two statements would cause a concept (or one subscripted concept) to take more than one reference then the binding fails. Matching and binding occur in four circumstances:

i) When answering questions, this may involve evaluating associated IF-rules.

ii) When problem solving certain connections between conceptions (thoughts) are checked to determine what can be done.

iii) Consistency checking, this involves question-answering.

iv) Command execution, this involves checking for certain connections (for example, PRODUCE and ENABLE thoughts.)

These four circumstances can be reduced to two types of matching and binding, question-answering and connection checking. These two differ slightly in the way they are handled and they are described separately below.

a. Question-Answering

At the top-level conceptions are obtained directly from the user. If a question is asked a matching assertion is looked for in the memory and if found the information bound back into the question is used as the basis of the reply. This means that at the top-level the concepts have references that are determined by the questions that have been asked. For example:

Who took the book?
A PERSON PASS BOOK A PERSON2 SELF?

Might leave PERSON bound to JOHN (that is, JOHN as the reference of the concept PERSON). If the next question is:

Who took the money?

then if the same concepts are used the question generated will be asking if JOHN took the money. This problem is overcome by clearing all the concepts in each question before it is matched. A concept is cleared by making its reference equal to itself. An assertion in the memory may be associated with a rule (by the IF connector). If such an assertion matches a question then the rule must be evaluated before the question can be answered. Evaluating the rule may result in secondary questions being matched and these in turn may invoke further rules. The following sequence of steps describes the process of matching, binding and invoking a rule in a way which prevents two rules with a common concept from interfering with each other:

i) If the first conception of the rule (the header) matches the question then

ii) all the actors in the header are bound to their corresponding actors on the question, this is called forward binding and it is the only place that information goes from a high level (the user is the top level) to a lower level then

iii) all the concepts in the rule except for those altered by step (ii) have their reference set equal to themselves (they are cleared), then

iv) the rule is matched, this may involve further question-answering and rule processing. When comparing a rule question with the memory only conceptions with a priority greater then 20 are considered (see Segment 2.3De)

v) the references of all the concepts in the header are made the references of the corresponding concepts in the question (back ward binding, information is being carried back to the user) then

vi) all concept references altered by the above steps, except for those that occur in the question itself, are reset to their value before step (i).

The above steps are analogous to function application in many programming languages, step (ii) corresponds to passing the input parameters into the function body, step (iii) to setting the local variables to some initial value, step (iv) to executing the function, step (v) to returning the results and step (vi) to resetting local variables to their previous values. The above steps are also similar to the pattern-directed procedure invocation in PLANNER, but they can better be thought of as forming a meaning-directed rule invocation. The following example illustrates the above steps:

1. A PERSON <EARN <SUB <MULT H R> T>POUND>
2. IF THE PERSON < WORK = H HOUR>
3 . AND THE PERSON <RATE = R POUND>
4 . AND THE PERSON <TAX = T POUND>
5. JOHN <EARN =X POUND>?

i) Conception 1 matches question 5.

ii) PERSON in 1 is bound to JOHN in 5, i.e. the input parameter is passed.

iii) H, R and T are cleared, i.e. locals initialised.

iv) The rule (lines 2, 3 and 4) is matched to find HOURS H, RATE R and TAX T for JOHN, answering these questions may involve using other rules, i.e. function execution.

v) Conception 1 is bound into 5, the expression is evaluated and the result made the reference of X, the result is returned.

vi) PERSON, H, Rand T are reset, so their reference is the same as before step (i), i.e. locals are reset.

<u>b. Connection Checking</u>

This is similar to question-answering, consider:

1. A PERSON <POSS AN OBJECT>
2. ENABLE THE PERSON PASS THE OBJECT SELF A PERSON$_2$.
3. A PERSON PASS AN OBJECT SELF A PERSON
4. PRODUCE THE PERSON$_2$ <POSS THE OBJECT>

These thoughts give information used when a command is obeyed and when problem solving, for example:

5. JOHN PASS A BOOK SELF BILL.

will match conception 2 and then the following occur:

i) conception 2 is bound to 5.

ii) the conception 1 is matched, in this case

JOHN <POSS THE BOOK>.

is checked.

iii) any concept whose reference is altered by steps (i) of (ii) is reset to its value before step (i).

A similar series of steps occurs when matching 5 with the PRODUCE connection except that the conception matched at step (ii) is treated as a question for ENABLE and as an assertion for PRODUCE.

2.4.3 Deduction

The last section has described how IF-rules are invoked if a question cannot be answered directly from the memory. An IF-rule is an explicit statement of a deduction that can be made. It is based on the rule of modus ponens, namely given as premises a conditional proposition and the antecedent of that conditional, the consequence of the conditional may be drawn as conclusion. For example, given P ->Q and P,Q may be deduced, or in its PIDGIN form given Q IF P and P,Q may be deduced. This is also the method of deduction used by PLANNER in the form of goal-directed consequent theorems.

As well as this explicit deduction there is also a form of implicit deduction that occurs during matching in PIDGIN. All of this implicit deduction must be done explicitly in a system such as that of Winograd because the programming language used as the deep structure was not restricted to handle only those patterns required by the deep structure of language. It is this implicit deduction that justifies the phrase "meaning-directed" as applied to PIDGIN's evaluation as opposed to the "pattern-directed" evaluation of PLANNER. The following simple example illustrates implicit deduction. Consider the deduction:

Fido is a dog.
All dogs are animals.

so

Fido is an animal.

In Micro-Planner this becomes:

```
(THASSERT (DOG FIDO))
(DEFPROP THEOREMI
      (THCONSE (X) (ANIMAL $?X).
              (THGOAL (DOG $?X)))
THEOREM)
(THGOAL (ANIMAL FIDO) (THTBF THTRUE))
```

It can be seen that explicit deduction is required. In PIDGIN it becomes:

```
FIDO<SUB A DOG>.
ALL DOG<SUB AN ANIMAL>.
FIDO<SUB AN ANIMAL>?
```

Because the SUB relation is part of the basic system it is possible to define it in such a way that an assertion using .SUB causes the information to be stored in a way which may be efficiently utilised later (probably in some form of tree structure). Thus when the question is asked it can be answered directly from this structure with out needing to search the memory. Further, this information can be used whenever a question involving the concept ANIMAL is matched against an assertion containing the concept FIDO, for example:

```
JOHN PASS FIDO SELF BILL.
JOHN PASS AN ANIMAL SELF BILL?
```

This last question can be matched directly with the assertion after making the implicit deduction that Fido is an animal. Implicit deduction cuts down the amount of backtracking required to solve a problem especially when the conceptions involved are complex. However, there comes a point when it becomes more convenient to use explicit deduction. The following examples illustrate this. a. Example 1 The relation AUNT can be defined from the relations PARENT and SISTER by means of the rule: A WOMAN <AUNT A PERSON>

IF THE WOMAN <SISTER A PERSON>
. AND THE PERSON$_2$ <PARENT THE PERSON>.

Then whenever the relation AUNT is used the above rule can be applied to test if the relation is true by trying to find a third person who is the sister of one and the parent of the other person. For example, if:

JILL <SISTER BILL>.
BILL <PARENT JOHN>.

Then if the question:

JILL <AUNT JOHN>?

is matched against the memory a search will first be made for a conception that matches directly, if none is found then the IF-rules are searched. In the above case the IF-rule does match (assuming BILL and JOHN are PEOPLE and JILL is a WOMAN) and the question is forward bound into the rule. Note that the rule specifies that only WOMAN can be related to PERSON by the relation AUNT. This stops the system from trying the rule if the first person is not a WOMAN and thus it implicitly asserts that all AUNTS are WOMAN. After matching and binding the header the body of the rule is matched. The body of the above rule is a conjunction of two conceptions. Matching involves both of these conceptions, both must successfully match the memory for the complete rule to succeed. However, it is not clear in which order the conceptions should

be matched. The most straightforward way is to match them in the
order in which they occur but it will be shown below that in some cases this can be extremely inefficient. In this case the rule to be matched consists of the two questions:

JILL <SISTER A PERSON$_2$>?
A PERSON$_2$ <PARENT JOHN>?

Both questions contain two actors whose nominals are a group and an entity concept. The most efficient to match first is the question containing the group concept which has the least number of concepts that can be substituted for it. In this case the two group concepts are the same so the next step in trying to choose the most efficient order would be to estimate the average number of PERSON in the SISTER relation as opposed to the average number of PERSON in the PARENT relation. At this point the calculation of the most efficient order is becoming as lengthy as the search itself and so they are matched in the order they occur. With large data-bases and long search times it would probably become more efficient to spend longer on the estimate of the best order of evaluation and to do this some way of maintaining the necessary statistics would need to be developed. The first question matches the memory and PERSON$_2$ is bound to BILL. The second question will then succeed. Finally, WOMAN, PERSON and PERSON$_2$ (all the concepts not in the original question that have had their reference altered) are reset and the original question succeeds. If the question had been:

Who is John's aunt?
A WOMAN <AUNT JOHN> ?

then this question would also have matched the IF-rule but the two conceptions of the rule would then become:

A WOMAN <SISTER A PERSON$_2$>?
A PERSON$_2$ <PARENT JOHN>?

and in this case it is clear that it is better to match the second question first. In general the following rules give a good indication of the best order: i) match the conception with the least number of group concepts first. If this does not give an unique conception then ii) match the conception with the most number of more specific group concepts first. . If a concept, A is more specific than a concept, B, then B can be substituted for A. For example, if the group concepts involved in the conceptions are:

1. ANIMAL ANIMAL
2. PERSON ANIMAL
3. PERSON DOG

then the order of matching becomes 3, 2, 1 because PERSON and DOG are more specific then ANIMAL. That is, (1) comes last because its group concepts are the least specific (2) comes next because one of its concepts is more specific and (3) comes first because two of its concepts are more specific. If this does not give an unique conception then iii) match in the order they occur in the rule. For the case of the above two questions the second. should be matched first (rule (i». This then gives the question:

A WOMAN <SISTER BILL> ?

and this binds WOMAN to JILL. Finally, PERSON and PERSONZ (all the concepts not in the original question that have had their reference altered) are reset and the original question succeeds with WOMAN backward bound to JILL. The answer can then be framed from the changes that have occurred to the group concepts in the question. If there have been no changes (or no group concepts) then the answer is "Yes" if the question succeeds and "No" if it fails. Other wise the answer is constructed from the changes, in this case a suitable answer would be:

Jill.

b. Example 2

The following rule calculates the number of sub-parts of a part. This is followed by a detailed description of its use for an example which involves the rule being used recursively.

<MULT M N> OBJECT <PART OBJECT$_3$>
IF =M OBJECT <PART AN OBJECT$_2$>
AND =N OBJECT$_2$ <PART AN OBJECT$_3$>
5 FINGER <PART HAND>.
1 HAND <PART ARM >.
2 ARM <PART MAN >.
=X FINGER <PART MAN >

1 The memory is searched for a match to the question but none is found so the IF-rules are searched and a match is found binding OBJECT to FINGER and OBJECT$_3$ to MAN. The rule is then matched and the first conception matches binding M to 5 and OBJECT$_2$ to HAND. The question:

=N HAND <PART A MAN>?

is then asked causing the IF-rule to be used recursively (because the question cannot be matched with any conception in the memory). The first conception of the rule then becomes:

=M HAND <PART AN OBJECT$_2$> ?

and th1s matches and binds M to 1 and OBJECT2 to ARM. The second conception then becomes:

=N ARM <PART MAN>?

and this matches and binds N to 2. The rule is then complete and the expression can then be computed giving 2 which is made the reference of N. All the other changed concepts are reset to the reference they had on entry to the rule. The first level use of the rule is then complete and the expression can be computed (M has reference 5 and N 2) giving 10 which is made the reference of X and all the other concept reference changes are reset. The original question

thus succeeds leaving the number concept with the reference 10.
This can be used as the basis of the reply:

10

2.4.4 Problem Solving

The problem solving ability of PIDGIN is derived directly from the deep structure. Problems are typically solved with a computer by writing a program using the usual programming language features such as declarations, assignments, goto's, labels, looping and so on. If these were incorporated into PIDGIN much of its potential for answering questions about its own structure and the ability to teach it how to solve problems conversationally would be lost because such programming features do not combine easily with the linguistic deep structure. The problem is what features can be added to PIDGIN to take the place of these programming language features that will fit easily into the over all linguistic framework and yet still provide a "programming" type of problem-solving ability.

People use English to teach other people how to solve problems. This is done by explaining what situations to avoid, what types of situation to aim for, rules for recognising both situations and procedures for avoiding and achieving them. All this information can vary from the explicit rule ("Avoid...!") and procedure ("Always do this and this and this") to the vague rule ("Look out for something like...") and procedure ("It sometimes might help to..."). This section describes how this type of information can be incorporated in and used by PIDGIN. The way this is done can in some ways be regarded as the 1ingui~tic equivalent of the techniques used by General Problem Solver (Newell, 1961a and 1961b).

The basic idea is to use actions to reduce the difference between the current state and the desired state. But, unlike GPS, PIDGIN works within a particular linguistic framework that guides and improves the whole strategy. This is possible because of the large amount of knowledge contained in a conception in the form of its potential associations through memory compared to the sparse logical framework used by GPS.

Two problem solving techniques are used by PIDGIN, called scheming and planning. Scheming is the name given to the operations necessary to form a scheme and planning to those necessary to form a plan. A scheme is a sequence of states leading to the desired state and a plan is a sequence of actions leading to a state within a scheme (possibly the desired state). In anyone problem description there is one current desired state (for example checkmate in chess) and a scheme is a description of some of the intermediate states that it is thought necessary to reach before the desired state can be reached. When problem solving the first thing done i3 to form a scheme. This will consist of the desired state preceded by zero or more states that it appears necessary to achieve, in the order given, before the desired state can be reached. Once a scheme has been found a plan can be made, this will consist of a sequence of actions that should achieve the first state in the current scheme. Finally, the first action will be performed; this will change the world model. If the environment reacts to the action (for example the opponent makes a move in chess) then both the current plan and scheme may need to be reformulated before carrying out the next action.

2.4.4A Notation

Before describing the problem solving technique in more detail it is necessary to define a number of terms:

A state is a conception consisting of one subject actor and the act BE.

A current state is one that can be found in the memory.

The world model is all the current states.

A feeling is defined using a specifier with the FEEL relation.

A positive (negative) feeling is a GOOD, STRONG and ACTIVE (BAD, WEAK and PASSIVE) one. An actor has a positive (negative) feeling if it is qualified by the FEEL relation and the total of the co-ordinates specified is positive (any co-ordinate not specified is assumed to be zero) (negative). One feeling is more positive (negative) than another if the total is greater (less) than the other.

The feeling (of PIDGIN) is the (one) feeling currently associated with SELF (the PIDGIN system) in the world model.

A positive (negative) state is one that is associated by a thought with a positive (negative) feeling of the PIDGIN system.

The desired state is the most positive state.

An enabled action is an action that is enabled by the current world model (via the ENABLE connector or CAN modifier).

An action is positive (negative) if it produces a positive (negative) state.

A state is short-term positive (negative) if it enables a positive (negative) action.

A state is a long-term positive (negative) if it suggests a positive (negative) state.

The final state is the desired state and it terminates the problem. Similarly the final action is the action that leads to the final state and the short and long term final state and action are similarly defined.

A final world is a world model that contains the final state.

A positive (negative) world is a world model containing the positive (negative) feeling.

A step consists of obeying one enabled action. This results in a new world model.

A possible world is a world that can be reached from the world model by one or more steps.

The uncontrollable future consists of all the possible worlds that can be reached solely by steps in which the subject is not SELF (it being assumed that if the subject is SELF then the step is controllable).

A positive (negative) scheme is a sequence of positive (negative) states.

A practical scheme is one in which the first state is suggested by a state in the current world model and that state suggests the next state in the scheme and the last state of the scheme is a final state.

A positive (negative) plan is a sequence of positive (negative) actions.

A practical plan is one in which each action produces a world model that enables the next action in the plan and the first action in the plan is enabled by the current world model and the last action produces a state in a practical scheme.

2.4.4B. Scheming and Planning

Scheming involves producing a practical positive scheme and planning involves producing a practical positive plan. A scheme is generated by:

SELF COGITATE A SCHEME.

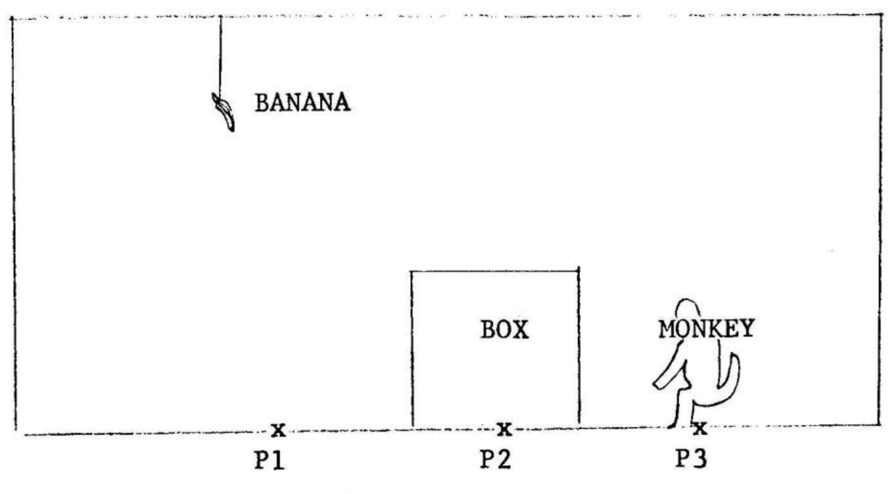
and a plan by:

SELF COGITATE A PLAN.

Only one scheme and one plan may be actively in use at anyone time. They are called the current scheme and plan and are the reference of the concepts SCHEME and PLAN. If there is no current scheme (SCHEME is clear) then the second command above must generate a plan that leads to a final world.

The way in which a scheme and a plan are generated will be described by means of examples.

a. Example I - Monkey Puzzle



The problem is how can the monkey reach and eat the banana. Monkeys can solve this problem, can PIDGIN? The following statements describe the problem:

1. Bananas are food

2. PI, P2 and P3 are places.
3. PI is below the banana, P2 at the box and you are at P3.
4. Eating makes you feel good.
5. Standing on the box under the banana enables you to eat it.
6. Standing by the box enables you to get on it.
7. You can move the box from place to place.
B. What do you do?

This description of the problem must first be translated into PIDGIN as follows:

1. ALL BANANA <SUB A FOOD>..

2. P_1 <SUB A PLACE>. P_2 <SUB A PLACE>. P_3 <SUB A PLACE>,
3. P_1 <BELOW BANANA>. BOX <LOC P_2>. SELF <LOC P_3>.
4a. SELF TRANSFER A FOOD A PLACE MOUTH..
b. CAUSE SELF BECOME SELF <FEEL 100 GOOD>.
5a. SELF <ABOVE BOX <BELOW BANANA>>
b. ENABLE SELF TRANSFER BANANA A PLACE MOUTH.
6a. SELF <LOC BOX>
b. ENABLE SELF TRANSFER SELF A PLACE
A $PLACE_2$ <ABOVE BOX>
c. PRODUCE SELF <ABOVE BOX>.

7a. BOX <LOC A PLACE>
b. ENABLE SELF TRANSFER BOX A PLACE A PLACE$_2$
c. PRODUCE BOX <LOC THE PLACE$_2$>
d. AND SELF <LOC THE PLACE$_2$>.
8a. SELF [CAN] TRANSFER SELF A PLACE A PLACE$_2$

b. PRODUCE SELF <LOC THE PLACE$_2$>

The problem is how to reach the desired state from the current world model. First a scheme is constructed by:

SELF COGITATE A SCHEME.

This command directs the system to formulate a scheme. The first step is to look for all occurrences of the final state. This will be a suggested state, a produced state or, as here, a caused BECOME action.

If the only occurrence is a suggest state then this is set up as the first step of the scheme and the search is repeated with that state as the required one. Otherwise the action producing or causing the state is examined and a search made for enabling states and causing actions. In this case the action (4a) matches 5b and this is enabled by 5a. The enabling state (5a) becomes the first state of the scheme because if it is reached the final desired state can be reached (by action 4a).

This procedure is repeated with this enabling state taking the place of the final state (that is, the enabling state is set up as a sub-goal leading to the final goal). This state (5a) says that SELF must be ABOVE the BOX and the BOX must be BELOW the BANANA. But the BOX is not BELOW the BANANA therefore this state must be reached before the scheme is complete. Also SELF is not ABOVE the BOX so this state is also added to the scheme.

This final sub-goal SELF <ABOVE BOX> has been generated by splitting the complex conception (5a) into two sub-goals. It matches the state at 6c which in the same way adds SELF <LOC BOX> (6a) to the scheme.

This matches the state 8b and the scheme is complete because the action that produces it has the CAN modifier and so is permanently enabled. This is the first time in the scheming that an enabled action has been found. When this happens the scheme is complete and practical because it leads from the current world to the final world. Scheming is also terminated if all the outstanding states required are in the world model. The complete scheme is:

[SELF <LOC BOX>
BOX <BELOW BANANA>
SELF <ABOVE BOX>]

Using this scheme the systems must next construct a plan for reaching the first state. This is done by the command:

SELF COGITATE A PLAN

A plan is constructed by searching for an action that produces the first state, this is found (action at 8a) and the plan is complete because the one action leads from the current world model to the one required in a single step. The plan is:

SELF TRANSFER SELF P3 BOX.

If this is carried out by:

SELF DO THE PLAN.

then the first state of the scheme will be deleted and the state:

SELF <LOC BOX>.

will be added to the world model by PRODUCE thought 8. The next plan will lead to the state BOX <BELOW BANANA> and as P_1 <BELOW BANANA> this is the same as BOX <LOC P_1> using the systems built in knowledge that BELOW specifies a location and LOC is symmetrical. This state can be reached by the one step plan:

SELF TRANSFER BOX P_2 P_1.

which when carried out leaves SELF <LOC P_1> as well as the BOX. As SELF <LOC BOX> (LOC is transitive and symmetrical therefore SELF <LOC PI> and BOX <LOC PI> implies SELF <LOC BOX>), the final state of the scheme can then be reached by the one step plan:

SELF TRANSFER SELF P_1 A PLACE$_2$ <ABOVE BOX>.

which produces SELF <ABOVE BOX>, which enables:

SELF TRANSFER BANANA A PLACE MOUTH.

which causes the BECOME action which gives the desired final state.

As soon as the desired state is reached the feeling of the system is reduced to zero so that it is then ready to solve another problem. The scheme and plan should be noted at each step of this problem solution because they are the systems understanding of what it is doing and are used to answer "how" and "why" questions. For example, the answer to:

Why did you move the box?

is

I wanted the box under the banana.

and the answer to:

How did you get the banana?

is

I went to the box, moved the box under the banana, got on the box and eat the banana.

It can be seen that these replies are obtained from all or part of the scheme ("why" questions) and plan ("how" questions) at each point. The following step by step descriptions of scheming and planning give a more detailed account of the processes:

i) Scheming.

1. Make the required state the desired state.

2. If all the required states are in the current world model then the scheme is complete and the scheming succeeds.

3. Search for a thought (suggest, produce or cause) that leads to (has as its second conception) the required state. If none can be found go back one step in the scheme and search for another thought. If no steps remain scheming fails.

4. If it is a suggest thought add the required state to the scheme and make the first state of the thought the required state, go to step 2.

5. Otherwise the thought (produce or cause) contains an action. Produce a scheme for each of the states that enable the action and add to the main scheme. Go to step 2.

ii) Planning

1. Make the required state the first state of the current scheme.

2. If all the required states are in the current world model then the plan is complete and the planning succeeds.

3. Search for an action that produces or causes a required state. If there are then no outstanding required states check the plan by carrying it out internally (resetting the world model back to its original condition when complete). If the plan succeeds add the action to it, otherwise search for another action. If no other action can be found go back one step in the plan and search for another action. If no steps remain then no plan is found and the planning fails.

4. Add the states that enable the action found to the required states, go to 2.

The scheming and planning algorithms outlined above work back wards from the desired state because there will typically be a large number of enabled actions causing a forward search to have to investigate a highly-branching tree of possible worlds. One problem with a back ward search is an action can produce a state that prevents a later action. To try to guard against this the plan being generated is checked whenever possible and" altered if this is found to occur. However, the method outlined can still fail to find a practical plan if it involves substantial changes to the world model.

An alternative would be to find some means of pruning the forward-search tree using a static evaluation function that could be combined easily with the features of PIDGIN. One way of setting up such an evaluation function, using the pattern matching ability and VALUE relation of PIDGIN, is mentioned in the next section.

<u>b. Example 2 - The Mikado Problem</u>

A more complex problem enables certain further features of the problem solver to be described. The following is a PIDGIN description of the Mikado problem:

```
1. ALIVE <SUB AN ATTRIBUTE <FEEL 100 GOOD>>.
2. DEAD <SUB AN ATTRIBUTE <FEEL -100 GOOD>>.
3. KOKO <SUB A MAN>. KATISHA <SUB A WOMAN>
4. WIFE <SUB A RELATION>. HUSBAND <INVERSE WIFE>.
5a. A MAN <HUSBAND A WOMAN$_2$ > [NOT]
b.  AND A WOMAN <WIFE A MAN$_2$ > [NOT]
c.       ENABLE THE MAN BECOME SELF <HUSBAND
                             THE WOMAN>
d. AND THE WOMAN. BECOME SELF <WIFE THE MAN>.
6. KOKO <HUSBAND A WOMAN> [NOT].
7. KATISHA <WIFE A MAN> [NOT].
8a. KATISHA <WIFE A MAN> [NOT]
b.  ENABLE KATISHA TRANSMIT
c.       <NANKIPOO DO AN ACTION
d.            CAUSE A PERSON$_2$ BECOME [HURT] SELF>
                   SELF MIKADO
e.       CAUSE [ALIVE] NANKIPOO
f.            BECOME [DEAD] SELF.
9a. [ALIVE] NANKIPOO
b.  ENABLE KOKO TRANSFER NANKIPOO A PLACE
                   A PLACE$_2$ <LOC MIKADO>.
10a. NANKIPOO <LOC MIKADO>
b .      ENABLE MIKADO PERCEIVE NANKIPOO.
11a. MIKADO [NOT] PERCEIVE NANKIPOO
b.       CAUSE KOKO BECOME [DEAD] SELF.
12. [ALIVE] KOKO. [ALIVE] NANKIPOO.
13. SELF <EQUIV KOKO>
```

This is more difficult than the Monkey Puzzle because of the increased complexity of the thoughts including the use of NOT, because the system is being asked to solve someone else's problem and because the system starts in a final state and the problem is to avoid any future deviation from that state.

The situation as described above enables Katisha to accuse Nankipoo, as she is unmarried, and this will cause Nankipoo to die which will prevent Koko from taking him to Mikado and so Mikado will kill Koko.

If the command:

SELF COGITATE A SCHEME.

is obeyed then it will be found that SELF (KOKO) is already in the desired state (ALIVE). In this case a check is made to see if this is likely to change in the uncontrollable future. In this case Katisha can denounce Nankipoo because she is not married and this will cause Nankipoo to die which will cause Koko to die because he will not be able to take him to Mikado. This is a bad thing in the uncontrollable future therefore an attempt is made to find a scheme that will prevent it happening. This is possible because although the future is uncontrollable the present can be controlled to some extent (as described by those enabled actions whose subject is SELF), and an action can change the uncontrollable future.

The sequence of actions outlined above must be prevented, this can be done by carrying out an action which removes some enabling state from the current world model. Katisha being unmarried (8a) is the first enabling state and this can be disabled by thought 5 because this enables action 5d. This is enabled for Koko and Katisha only therefore the solution is for them to marry.

Working backwards on a scheme Koko must transfer Nankipoo to the location of the Mikado (9b, 10 and 11). To do this he must be alive (9a) so Katisha must be stopped from accusing him (8b,c,d,e,f). This can be done by Katisha becoming a wife of a man (8a). For this to happen Katisha and the man must be unmarried (5a,b). Katisha and Koko are unmarried (6,7) therefore if they get married this will prevent Koko dying. The scheme is:

**[KATISHA <WIFE KOKO>
NANKIPO <LOC MIKADO>]**

and the action required is:

KOKO TRANSFER NANKIPOO A PLACE A PLACE$_2$ <LOC MIKADO>.

In both the above examples the problem could have been expressed in many different ways. For example, in practise the rules would probably be more general and contain more criteria. The above examples were set up with the bare minimum of knowledge required to express and solve the problem.

2.4.5 Teaching and Learning

If a computer modifies its output because of some input then it can be said to have learnt from or 'to have been taught by that input. Thus programming can be regarded as teaching the computer. This is the most important type of learning in the current PIDGIN system - being taught new facts and rules by the user in Input PIDGIN (and eventually in English). It is hoped that by this means the computer can eventually be used by the people with the problems in order to help them to solve those problems without needing to translate their problem into a conventional programming language. Conversational problem solving systems will provide people such as managers, designers, engineers and scientists with a means whereby they can explain their problem in English (plus mathematical notation} with conversational advice from the computer on inconsistencies and ambiguities so that these can be eliminated as they arise. In this way the computer becomes a tool to help the human problem solver unravel the solution and at the same time teach the computer the answer so that the solution and the program for solving the problem arise together. The computer is not being asked to, and is not expected to, behave "intelligently" it is there to do what it is told and to complain if it finds that it cannot do it. PIDGIN is a small step in enabling this to be done conversationally in English rather than involving special codes and languages, card punches and verification, turn-around and transcription errors, incomprehensible and useless error numbers, core dumps and cryptic messages, octal, hexadecimal, job control and all the other paraphernalia and secondary problems that separate the user from the real problems and require the intervention of human aid in the form of a programmer in order to sort it all out.

However, there is more to learning than programming. It is possible with certain problems to set up systems that learn from the short-fall between their behaviour and their expectations. Although little work has been done to investigate the potential of PIDGIN in different learning situations it is felt that the nature of the linguistic deep structure will enable some interesting learning possibilities to be investigated. In order to provide a framework for investigation four types of learning are distinguished - rote, rule, ratified and regulated.

a. Rote Learning

Rote learning is here taken to mean that the item learnt is simply added to some list or table of items. However, this is not intended to be a definition merely a description of how rote learning is usually implemented. From the point of view of a user a system which only rote learns can only answer questions that directly refer to the information learnt, no deductions are made. Rote learning is often called "parrot learning" when it is done by people, this term indicates that the material was not understood, that is, incorporated in the person's memory in such a way that it could be used to relate diverse other information together, but was simply remembered as a self-contained item. In fact quite often a large amount of material must be learnt like this before inter-relationships and connections can be pointed out and understood. A similar step occurs when using PIDGIN, simple definitions and states are rote learnt. It is not until the rules and connections that combine them are learnt that they can be related and so properly understood.

b. Rule Learning

Rule learning is the name given to the incorporation of sequences of PIDGIN conceptions (in the form of plans, schemes, rules and thoughts) in the memory. More generally programming is a form of rule teaching and in human situations much advice and teaching takes the form of rules. Although a rule is rote 1earnt,in use it expresses complex relationships between other items. This type of learning forms the basis of PIDGIN's learning ability as has already been described. In PIDGIN the word "rule" refers to "IF-rule", that is, a conception connected to one or more other conceptions by the connector IF. Such rules form the basis of PIDGIN's explicit deductive ability (see Section 2.4.3) and correspond in many ways to the consequent theorems of PLANNER. They are automatically invoked by question-answering if the question cannot be answered directly from the memory.

c. Ratified Learning

Ratified learning takes place when the system generates new statements and then confirms them with its environment. In people this type of learning perhaps corresponds to thinking, that is, ana1ysing known facts and thus discovering new relationships and predicting likely hypothesis. In PIDGIN there is a special mode in which the system can generate hypotheses (see Section 2.4.1) check them with its memory for consistency and then confirm them with the user for validity. The essence of this mode of working is hypothesis formation. In PIDGIN this is very simple and as a consequence the user gets inundated with trivial questions. It is done by choosing a conception and then creating a more general but consistent conception from it. This is done by making one or more of the actors in the conception vaguer by omitting qualifiers and by replacing the nominal by its category concept.

PIDGIN can also learn by asking questions about vague or unspecified parts of a conception. For example:

John went to the park.
JOHN TRANSFER SELF A PLACE PARK.

immediately leads to questions such as:

Where did he come from?

How did he get there?
What time did he go?
How long was he there?
How long did it take him?
Did he intend to go?

because these questions correspond to information missing from the original assertion. PIDGIN is thus always aware of what it does not know.

Rote and rule learning fill up the memory and slow down the system (except when one rule replaces a number of old rules or facts). But the essence of creative learning is - "to learn is to forget". That is, to learn is to forget what is not essential. Such learning frees the memory and speeds up the system. The usual price to be paid for replacing a precise rule by a more general one is that the general rule does not always work and has to be backed up by auxiliary rules for special cases. However, this is the way that science advances from data to special rules to general rules and then to general rules plus more data and special case rules and then still more general rules and so on. It is hoped that one day computers will be able to assist in the formulation and analysis of these rules.

d. Regulated Learning

Samuel's checker player (Samuel 1959) was one of the first computer programs to successfully learn to improve its performance by playing. The general scheme of the learning mechanism used is to associate each one of a

set of state or pattern recognisers with a weight in order to adjust
its importance relative to the others. The sum of the product of the states and weights can then be used to assign a figure of merit to any situation. For example, in a board game to rate each possible next move so that the best can be chosen. Learning is achieved by regulating the weights in such a way that good board positions are rated well and bad positions rated badly. In Samuel's program the state recognisers were carefully constructed using knowledge derived from human checker players and the weights were then discovered by the program through playing games with good checkers players. One limitation of Samuel's original program was that it simply added all the weighted states together so if one state was important only in combination with one or more other states such a technique would not be able to take advantage of the fact.

It would be possible to include such a learning technique in PIDGIN so that it could be utilised to improve scheming and planning. A state may be associated with a good, strong and active rating using the VALUE relation and this can be regarded as a weight. Further, in PIDGIN states may be combined using conjunction and disjunction and negated using NOT and the combination can be assigned a weight by specifying that is suggests some other state and assigning a weight to that state. A system can be envisaged in which these weights are improved in a forward search game-playing environment but this has not been investigated in detail.

A more interesting possibility is to combine ratified and regulated learning so that the hypotheses generated were new state recognisers and the ratification was not obtained directly from the user but by using them in a regulated learning environment. A number of improvements suggest themselves, for example, generating the hypotheses by generalising only the most successful state recognisers. However, this possibility has not been studied because the way in which hypotheses should be generated and ratified is such an enormous problem.

CHAPTER 3 EIKASIA - A PIDGIN BASED CHESS SYSTEM

Winograd chose to base his conversational system on a toy-world consisting of coloured b10cks that could be moved around a table-top by means of a crane. By thus sharply delimiting the context Winograd could construct a self-contained and complete system. His system accepts most grammatical English sentences directly related to this toy-world and framed in a particular vocabulary. By defining a toy world the limitation of vocabulary are made less arbitrary and the semantic relationships are well defined. Many conversational systems in the past have foundered on the cliffs of language. Those that have tried to reduce the size of the problem have usually done so in a way that does not effectively restrain the user. For example, although baseball seems like a well-defined single subject questions about baseball can quickly extend into many other areas. Winograd easily and neatly defines a small part of English by placing the user in a new world of just a few objects and relationships. The limits of this world are largely written on its face and though the awkward user might ask "What is under the table" or "How often do you oil the crane" he should not be surprised at the systems response.

In order to test some of the ideas in PIDGIN with a practical problem it was necessary to find a toy-world in which to work. It was decided that a two kings, one pawn chess-endgame was a suitable, well defined problem. It has fewer objects than Winograd's toy world and a better defined co-ordinate system but the relationships between the objects are much more subtle. Also, an algorithm for solving this chess endgame was available from S. Tan at Edinburgh University (Tan 1972). The predicates and action schemes defined in this algorithm were taken as the basic actions and states in a conversational

PIDGIN system called EIKASIA.

3.1 A History of Chess Systems

The history of computer chess divides, approximately into two camps, the first is concerned with writing a program that will beat as strong a human player as possible, the second is concerned with the chess problem only in so far as it embodies other more general problems. EIKASIA is in the latter camp. As a chess player EIKASIA is very poor but it is still of interest because of the way in which it is constructed. A program that played chess at grandmaster level would only be of interest in so far as it demonstrated general problem-solving techniques and EIKASIA is clearly a general problem-solver.

Some of the work in the latter camp has been done without actually writing a computer program that plays chess. An important early paper on the subject was written by C. Shannon (1950). In this paper he lays down features that he considers necessary for inclusion in any chess program. Some of these features are details of evaluation functions, static positions and minimax as well as forward pruning of the search tree. Later Turing (1950) published a paper extending Shannon's ideas and then Samuel (1959) produced a paper which, though it concerned a checkers playing program, described how many of these ideas had been built into a practical program.

The first chess program to incorporate these ideas was written by Bernstein (1957). This program aroused some public interest and it managed to play at "passable amateur" level taking an average of two minutes per move. Also at about the same time Newell, Shaw and Simon wrote a chess playing program based upon ideas from GPS (Newell 1958). This program was fundamentally similar to the other work that had been done but it approached the subject from a different viewpoint, that of the goal/sub-goal scheme. They also introduced further refinements to tree-searching including the alpha-beta algorithm.

From 1958 until recently the only notable contribution has been a paper by Good (1968) listing a number of features and ideas to bear in mind when writing a chess program. Good's paper is a major summary and analysis of the complete field and the fact that his idea do not seem to have been acted upon until recently suggests that most chess programmers in the first camp regard the problem as simply a programming one.

In 1973 A.L. Zobrist and F.R. Carlson published a paper (Zobrist 1973) describing a chess program that had been designed with a slightly different objective from usual. Their system was designed so that it could be given "advice" by expert human-chess players. This advice was given in the form of routines written in a simple language based on chess notation. Chess players with no previous computer programming experience could quickly learn this language and use it to correct faults in the programs play. One of the difficulties of writing chess programs has been the difficulty of translating chess knowledge into computer notation. By enabling the chess expert to "program" the computer directly it was hoped to overcome the translation problem. The routines that can be defined in their language consist of a pattern and a weighting. The system uses the pool of routines to try to find patterns on the board and associate weights with them. The system is set up so that the patterns are independent of the actual positions of the pieces but only depend on their relative positions. For example, a routine can be defined that matches bishops and knights on the back row and returns a negative weight. This would

tell the program to "get your bishops and knights off the back row".
This type of advice corresponds to SUGGEST connections in PIDGIN.

Their system uses this advice as a form of static evaluation function to prune the look-ahead tree. They go on to say that they believe computers will use advice-taking for the performance of tasks that demand more "intelligence than is needed for the simple clerical chores they now perform.

The language accepted by Zobrist's program is a very simple programming code. It would seem that a much more sophisticated input language is required to improve the performance. However, such a language cannot be a normal programming language otherwise no problem has been solved. I have taken PIDGIN as this language and have tried to justify this by considering the design of a PIDGIN based two kings, one pawn chess-endgame playing program called EIKASIA.

3.2 A Description of the Endgame Problem

Most chess playing programs let the endgame take care of itself. In fact the seemingly trivial single pawn ending is more difficult than most puzzle problems. It has the additional advantage that human players seem to use knowledge gained from chess literature to solve it but at the same time there is no obvious numerical evaluation function that can be used. The problem thus requires a solution that involves storing and efficiently utilising a great deal of specific chess knowledge.

The work done by S. Tan at Edinburgh University (Tan 1972) illustrates one method for storing such knowledge. Other work has been done on the endgame problem (Huberman 1968) but the approach taken by S. Tan fits in well with ideas incorporated in PIDGIN. The program developed by Tan is a computer representation of the knowledge contain ed in a number of books on chess endgame theory (Averbakh 1958, Fine 1941). This knowledge is stored as a decision tree, the nodes are predicates for recognising "good" board patterns and the leaves are action schemes. A similarity can be seen here with the advice-taker chess program of Zobrist but the patterns are combined to form a tree rather than being associated with a single integer weight. A path down the binary tree is determined by a particular board situation, at each node the result of applying the associated .predicate will select one of two paths corresponding to succeeding or failing to match the "pattern". The action schemes specify what is to be done in each particular board situation.

More exactly there is a board of 64 squares, each referenced by two co-ordinates, the first specifies the file, lettered from A to H, left to right, and the second the rank, numbered from 1 to 8, bottom to top. There is a white king, a white pawn and a black king on the board, and the white pawn always moves upwards. White wins if the pawn reaches the eighth rank and is not captured immediately. A game is also terminated by a draw, either because the black king takes the pawn or a stalemate situation occurs. A position is defined by the board configuration (the co-ordinates of the three pieces) together with the information about who is on the move.

A position defines a situation which is either intermediate or terminal if the one on the move is stalemate or black has captured the pawn or the pawn has been successfully promoted. A game may also be terminated by either side resigning or by mutual agreement. A game may be started in any legal chess board configuration by mutual agreement when all three pieces are on the board.

Let S be the set of situations. A (unary) predicate P is a partial function from S to true or false. Every predicate defines a subset of S, namely the inverse image of true. An action scheme is a partial function from S to S. Every action scheme defines, for each situation in its domain, a legal move applicable to the situation.

If A, A1, A2 are variables for action schemes and P, P1 P2 for predicates then:

```
A ::= <elementary action scheme>

      if P then Al else A2 close

<elementary action scheme> ::= PAWNSTEP PAWNJUMP. MOVE2
                                WGOPAWN BGOPAWN LETPASS
                                RUN SUPPORT MANOEUVI
                                MANOEUV2 MANOEUV3 MANOEU4
                                MANOEUV5 NOP

P ::= <elementary predicate>
      not PI
      P1 and P2
      P1 or P2
      if P then P1 else P2 close

<elementary predicate> ::= ADVANCE CAPTURE SELFBLK
                           MATCH1 MATCH2 WSTALEM
                           BSTALEM DOMINANT CANRUN
                           BKAHEAD NEEDSUP WSEE
                           BSEE LOOKAHD
```

The two kings, one pawn chess-endgame program is constructed from the above predicates and action schemes. The way they are put together represents the programs knowledge of playing chess. They are put together in sentences of the form:

```
<advice sentence> ::= dummy

<val, A>
if P then <advice sentence>
      else <advice sentence> close
```

where dummy is a do-nothing sentence, <val, A> is an ordered pair with val=WHITEWIN, DRAW or EVALUATE. If val is EVALUATE then the value of

the situation is obtained by executing A and searching further
through the game tree.

The knowledge structure can be seen as a binary decision tree with non-terminal nodes corresponding to predicates and terminal nodes to pair-wise disjoint subsets of S.

The action schemes and predicates listed above were extracted by Tan from chess literature on the endgame, they are the building blocks of the program and it is profitable to consider how some of them arose.

The predicates relate to the position or relative positions of the pieces. Some of these are fundamental, for example ADVANCE is true if and only if the pawn can advance one square without being taken, CAPTURE is true if and only if the pawn can be captured immediately and SELFBLK is true if and only if the white king is on the same file as and immediately in front of the pawn.

MATCH1 and MATCH2 are true if the pattern specified occurs some where on the board. Each takes two sets of co-ordinates relative to the pawn. MATCHI is true if the two kings are at the specified co ordinates, MATCH2 is true in this case and also if the left-right inverse of the co-ordinates matches. For example, if the configuration:

WK		
WP		BK

occurs anywhere on the board then MATCH1 (1, 3, 3, 1) and MATCH2 (1, 3, -3, 1) will both be true.

There are three other general predicates, WSTALEM and BSTALEM are true if the specified colour cannot make a legal move, and BKAHEAD is true if the black king is ahead of the pawn.

The other predicates relate more specifically to the one pawn ending. The concepts of critical or key squares and the "rule of the square" are stressed in the chess literature as being important in the endgame and these concepts are implemented by the predicates DOMINANT and CANRUN.

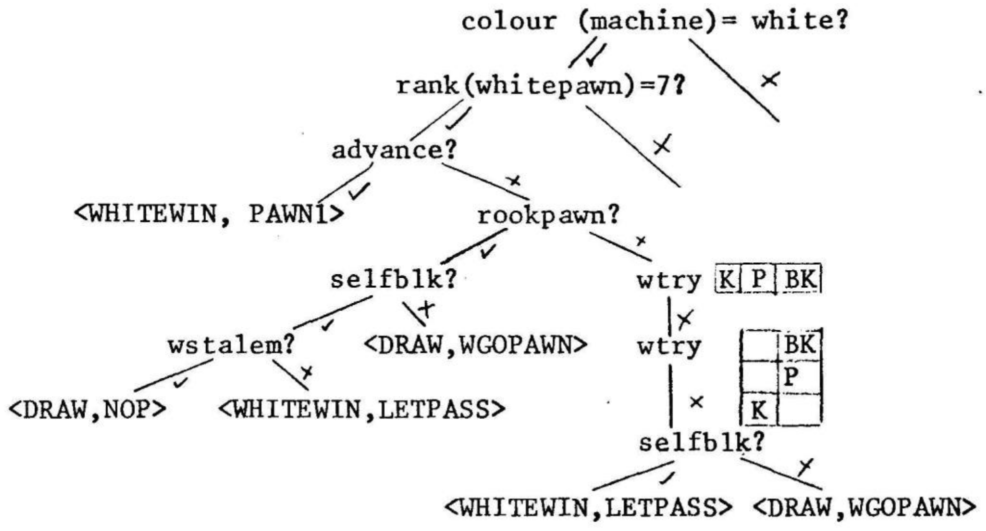
The following section of stylised program gives some idea of how the program is constructed by Tan in the programming language POP-2:

```
if colour(machine)=white then

    if rank (whitepawn)=7 then
         if advance then <WHITEWIN, PAWN1>
         elseif rookpawn then
              if selfblk then
                   if wstalem then <DRAW, NOP>
                   else <WHITEWIN, LETPASS> close

              else <DRAW, WGOPAWN> close
         else
              if wtry(-1,O,1,O) or wtry (-1,-1,0,1) then
              else if selfblk then <WHITEWIN, LETPAS>
                   else <DRAW, WGOPAWN> close
              close
         close
    close
else ...
```

This is clearer if drawn as a tree:



3.3 PIDGIN and the Endgame

EIKASIA is a particular, integrated extension of PIDGIN that is it is a PIDGIN system incorporating numerous concepts, rules, thoughts, beliefs and dictionary entries concerned with a particular problem. It is integrated in the sense that the extension is a complete solution to the problem. However, this does not mean that it is the best solution to the problem. By further dialogue with users the solution might be improved or extended to cope with a wider range of problems. EIKASIA as described here is only the design of a PIDGIN based one pawn chess-endgame system because of the limited nature of the PIDGIN implementation. Also it is not intended to be a practical chess endgame system in that it will not necessarily beat or play as quickly as a program designed specifically for the purpose. However, chess was not the motive for its development. It will have served its purpose if it shows that PIDGIN is capable of solving problems as complex as the one pawn endgame without losing its flexibility.

Having completed the initial design it seems that the problem chosen might not be the most suitable because the solution is so clearly and efficiently demonstrated by the program of S. Tan. It might have been better to have chosen an ill-defined problem and demonstrated how PIDGIN can be used as a conversational aid to the gradual development of the solution because this is really what PIDGIN was designed for. Nevertheless it is hoped that while reading the following description of EIKASIA the way in which the system could have been gradually developed will be borne in mind.

3.3.A PIDGIN Particular Extensions

The extension required to the initial PIDGIN system in order to construct a chess endgame system can be divided into four parts:

i) Chess concepts: The pieces, board and other objects, relations and attributes.

ii) Board states: The rules and thoughts required to recognise board patterns of significance in the endgame.

iii) Board actions: The legal moves, the good and bad moves plus their enabling states and the states they produce.

iv) Positive and negative states: The final states, suggestions for positive and negative play, and positive and negative board-pattern recognisers.

3.3A1. Chess Concepts

EIKASIA manipulates a black king, white king, and a white pawn which may be in a box or anywhere on an 8x8 board of 64 squares. The view is initially a 2x2 grid with the structure:

OPPONENT	SPECTATOR
BOARD	BOX

This can be set up by the following PIDGIN:

```
ALL PLAYER <SUB A PERSON>.

ALL OPPONENT <SUB A PLAYER>.
ALL SPECTATOR <SUB A PERSON>.
ALL BOARD <SUB AN OBJECT>.
ALL BOX <SUB AN OBJECT>
```

The view can be set up by the command:

```
SELF TRANSMIT * <<OPPONENT SPECTATOR>

<BOARD BOX>> HERE VIEW
```

However, when a game is being played the view is changed to an 8x8 grid representing the chess board. This is defined as:

```
ALL SQUARE <SUB A PLACE>.

NEWBOARD <SUB A BOARD>.

A1 <SUB A SQUARE>. A2 <SUB A SQUARE>.

... H8 <SUB A SQUARE>.
$ <<A8 B8 ... H8> <A7. ... H7> ...
             <A1 ... H1>> :NEWBOARD.

LASTRANK <SUB A PLACE>.

$ <A8 B8 C8 D8 E8 F8 G8 H8> :LASTRANK.
```

In a normal chess game NEWBOARD would contain all the pieces laid out correctly but in this endgame they can be set up anywhere therefore the board is initially empty, and the pieces are in the BOX.

```
ALL PIECE <SUB AN OBJECT>.

ALL KING <SUB A PIECE>.
ALL PAWN <SUB A PIECE>.
BKING <SUB A [BLACK] PIECE>
WKING <SUB A [WHITE] PIECE>.
WPAWN <SUB A [WHITE] PAWN>.
BOX <CONT WPAWN>.
BOX <CONT WKING>.
BOX <CONT BKING>.
```

The system keeps track of the entities in the view and a change of location of anyone is automatically recorded by a corresponding change to the view. This is the internal mechanism that simulates a change of location being performed by an "arm" and picked up by a camera.

3.3A2. Board States

S. Tan has fourteen elementary predicates over board positions. These can be incorporated in EIKASIA by defining the relevant states in terms of the positions of the pieces or by appropriate patterns. For example, Tan defines the predicate ADVANCE as:

```
i) function advance s; vars x y;

       wk(s(5),s(6)+1)->x;
       dbk(s(5),s(6)+1)->y;
       if x=0 or y=0 then false
       e1seif y=1 and x > 1 then false
       else true
       close

end
```

In English this is, the pawn may advance if the square above it is not occupied and the black king is more than one square away from it or the white king protects (is next to) it. In PIDGIN this becomes:

```
[ADVANCE] WPAWN IF

       A SQUARE <ABOVE WPAWN><CONT NO PIECE>
       AND BKING <FAR THE SQUARE>
              OR WKING <NEAR THE SQUARE>.
```

This definition will automatically fail if the pawn is not on the board or is on rank eight. The following examples illustrate other techniques:

```
ii) function capture s;

       comment true iff white pawn can be captured immediately
       if dbk(s(5),s(6))=1 and dwk(s(5),s(6))>1
```

```
                then true else false close
end

A PIECE <CAPTURE A PIECE₂>IF
SELF DO <A PLAYER TRANSFER THE PIECE A SQUARE THE PIECE₂>
AND SELF DO <A PLAYER₂ [NOT] TRANSFER A PIECE₁ A SQUARE₃
                                             THE PIECE>.
```

During a game a piece can only be transferred to another square by a legal chess move. The chess moves are defined by means of enabling conditions (see next page). Note that a piece can be treated as a place when it is actually the square occupied by the piece that is being referred to. This is possible because the pieces are in the view and can be found by searching.

```
iii) function selfblk s;

       comment true iff white king blocks the white pawn;
       if s(1)=s(5) and s(2)=s(6)+1 then true
       else false close
end


[SELFBLK] WPAWN IF
       WKING <ABOVE WPAWN><COL WPAWN><NEAR WPAWN>.


iv) function match1 a b c d;
       if s(1)=s(5)+a and s(2)=s(6)+b and
       s(3)=s(5)+c and s(4)=s(6)+d then true
       else false close
end
```

This matching ability is already built into PIDGIN for examp1e:

```
SELF IDENTIFY *<<WKING><A SQUARE₁><WPAWN A SQUARE₂
                        BKING>>.
```

will match:

WKING		
WPAWN		BKING

3.3A 3 Board Actions

The pieces are moved by the TRANSFER act, the actions of which are automatically reflected in the view by the system. If the view does not contain the piece transferred or the action is not enabled then the command will fail. The following requirements must be met:

i) when a game is not in progress any move is allowed;

ii) during a game moving a piece onto a square that is already occupied causes the piece that was there to be moved to the box;

iii) the moves of the pawn and king must be defined by the correct enabling conditions to reflect the rules of chess.

This can be done by the following PIDGIN ,statements:

```
A GAME [STOP] ENABLE

     A PERSON TRANSFER A PIECE A PLACE A PLACE2
A PERSON TRANSFER A PIECE A PLACE
                  A SQUARE <CONT A PIECE2>
     PRODUCE THE PIECE2 <LOC BOX>.
A [COLOUR] PLAYER [CAN] TRANSFER A [COLOUR] PAWN
     A SQUARE A SQUARE2        <COL THE SQUARE>
                               <NEAR THE SQUARE>
                               <CONT NO PIECE>.
A [COLOUR] PLAYER [CAN] TRANSFER A [COLOUR] PAWN
     A SQUARE A SQUARE2        <ABOVE THE SQUARE>
                               <NEAR THE SQUARE>
                               <NOTCOL THE SQUARE>
                               <CONT A [COLOUR2] PIECE>.
A [COLOUR] PLAYER [CAN] TRANSFER A [COLOUR] KING
     A SQUARE A SQUARE2 <NEAR THE SQUARE>
                               <CONT NO [COLOUR] PIECE>.
```

3.3M Game Playing

The knowledge contained in the program written by S. Tan can be divided into two parts, that within the statements making up the program and that implied by the ordering of the statements. The rules of PIDGIN are the equivalent of the statements of the program. Each rule independently recognises a particular situation and enables an action which produces a state suggesting either a win or a draw. S. Tan suggests that the strength of an experienced player is related to the organisational structure of his chess knowledge. This organisation could be built into PIDGIN by the construction of large IF-rules and it is this type of structure that would be built up if PIDGIN were told exactly what to do in each situation. This is the type of knowledge in S. Tan's program but it is not the type used to teach chess (for example, it is not the type found in chess books). Teaching chess usually consists of presenting a number of guides and rules and then leaving it up to the person learning to discover the way in which they must be combined together. This is usually done by playing chess and learning from experience.

One of the features of PIDGIN is the ability to develop the system by adding new rules and revising old ones. This implies a loose organisation of the rules and it would seem that the more rigid the organisation the more the user, or teacher, must be aware of it. At one extreme each rule is independent and the user may freely change or add a rule without being aware of the others. At the other extreme the complete system is one rule and to change any part the user must be aware of all the other parts. One extreme corresponds to PIDGIN's conversational but inefficient way of working and the other to S. Tan's program.

A compromise between the two extremes is possible if PIDGIN can be made to organise its own rules. For example, it could maintain the order it was told the rules and try the oldest and most detailed first, re-ordering them at each move in a way which would reduce the time it would take to repeat that move analysis. This would produce a simple linear ordering. As a next step it could build up a tree structure by searching for common sub-rules. For example, if two rules were:

[RULE1 RULE 2 RULE 3 RULE4]

[RULE1 RULE 2 RULE5]

then it could construct:

[[RULE1 RULE2] (RULE5 [RULE3 RULE4])]

Where square brackets form conjunctions and round brackets disjunctions (see Section 2~1.1). Note, that the new disjunction is unordered and so will automatically be ordered by the ABC processor in a way that will result in the one that has succeeded most often in the past being evaluated first. The other advantage of this restructuring is that common sub-rules are only evaluated once, though this could also be achieved by an alternative scheme in which the result of evaluating a rule was remembered and used whenever the rule was called with the same arguments or until the memory was altered.

The advantage of an organising algorithm is that the user can deal with the system at the level of the single simple rule yet the system is not thereby made unnecessarily inefficient. The above scheme has not been worked out in detail but it is suggested as the best way of extending PIDGIN to take advantage of the efficiency that results from organising the knowledge base.

S. Tan's program will find a move given any board configuration without needing to formulate an overall strategy or produce a detailed plan. This means that in PIDGIN no scheming (strategy) and little planning are required, for each configuration the rules simply give the move to make. However, with more complex chess problems it is likely that a great deal of effort would be put into producing and comparing different schemes and into constructing various plans for achieving the states outlined in those schemes. For example, some schemes might be:

Control centre,

Develop pieces

Make open file, Double-up rooks

Find passed pawns, Queen

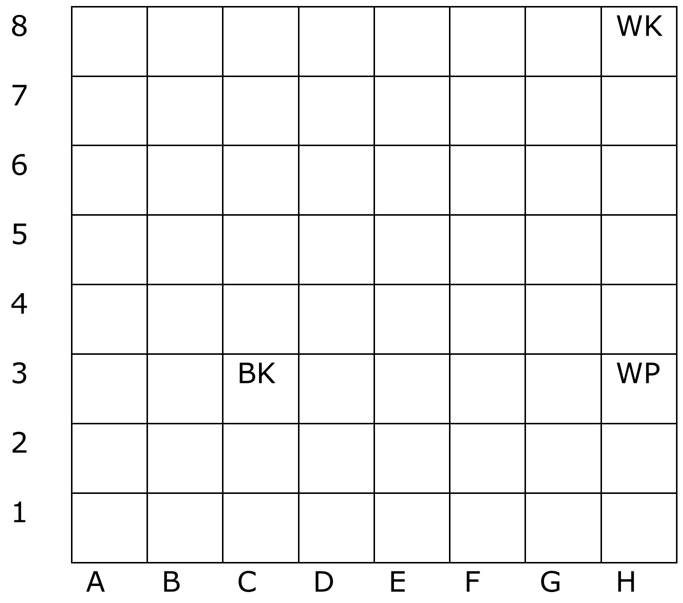
One disadvantage of not having such schemes with their associated plans is that "how" (planning) and "why" (scheming) questions cannot be answered. Without a scheme the only reply to a "why" question is "In order to win" (or draw) and consequently "how" questions are also reduced to either simply the last move made or the complete sequence of predicated actions leading to the win (or draw).

If schemes are produced they are likely to remain valid for a number of moves but plans are rendered invalid as soon as the opponent does not make the predicted reply. This typically means that a continual replanning is necessary and so it would be desirable to find some way of reducing this overhead. One way is to produce a structured plan that contains the best reply to a number of the opponent's possible replies. Whether the extra time required to produce this more complex plan would be offset by the subsequent saving in time at future moves depends, among other things, on how well the program can predict the opponent's moves. It would, for example, be very difficult if the opponent was aware of the programs attempt to do this. It is interesting to note that most chess programs carry no information from move to move and analyse each position afresh. This is because it has been found that the time required to check if the information carried forward is still relevant in the light of the opponents reply is about equal to the time required to re-analyse the position. This conflicts with the fact that people generally require longer to make a move if they have not seen the position before than if it is one that occurs during a game they are playing. This suggests that people do carry information forward and adds weight to the attempts to do likewise, for example, in the form of PIDGIN schemes. Another way of improving the efficiency of the system would be to remember past positions and the associated move made. Then if a similar position were to re-occur the same move could be used without needing to analyse the position. The problem is that it is useless to try to remember all positions as there are so many that the time to access anyone would soon exceed the analysis time. This problem would be solved if the system could save just the essentials of the position, however, this amounts to the system learning to play chess because whether two positions are similar depends on all the rules of chess. A program for detecting similar positions would be equivalent to the analysis rules themselves, and a program that extracted the essential features of a position would be one that was generating a chess playing program. All these techniques therefore, although interesting in their own right, are not useful ways of improving the efficiency of PIDGIN.

Appendix IIC gives some details of the PIDGIN statements required to set up EIKASIA.

3.3B A Typical Endgame

This division goes through the analysis of a complete endgame. The starting position and analysis are taken from S. Tan (1972) but the description is given in terms of PIDGIN. The starting position is:



White to move

In this position there are a number of attributes that are true and a number that are false. The attribute CANRUN is false (see Appendix IIC) because the black king is just inside the "promotion square" of the white pawn but the white king is also in the promotion square (NEEDSUP is false) and not blocking the pawn (SELFBLK is false) therefore the pawn is moved forward (PAWNSTEP). The only applicable attribute that is true is ROOKPAWN but as this is the simplest condition the others must be checked first, for example, if the conditions with their corresponding actions are:

[ROOKPAWN SELFBLK] LETPASS
[ROOKPAWN BKAHEAD] SUPPORT
ROOKPAWN PAWNSTEP

then obviously the more specific conditions must be checked first otherwise they would never be checked.

Black's best move is to try to reach the pawn and hope that white makes a mistake, therefore black moves to D4. The above analysis holds for the new position and results in a similar two moves, this is repeated once more leaving the following position:

F	G	H	
		WK	8
			7
BK		WP	6

White now has only two possible legal moves, one leads to a win and the other to stalemate. The winning move is found by a pattern match which matches the above position (or its inverse) and results in the king moving to G8.

The ADVANCE attribute is now true and this results in the white pawn being moved one square forward on each move, so whatever move black makes the game ends in two moves (the game ending when the pawn reaches rank 8).

CHAPTER 4 THE TRANSLATION OFA .SUBSET OF ENGLISH

4.1 Introduction

PIDGIN is a language in which a subset of English can be unambiguously represented. This chapter is concerned with the disambiguation of sentences in that subset of English by describing the equivalent PIDGIN statements. Question-answering systems differ greatly in the way they handle the analysis problem. Some, such as Colby's interviewing program (Colby 1969b), look for key phrases without attempting a detailed analysis. On the other hand Winograd's system carries out a detailed analysis of each sentence, using a special programming language called PROGRAMMAR for the purpose. Winograd's analysis program is based on M.A.K. Halliday's systemic grammar and it uses context and semantic information and the powerful deductive facilities of PLANNER to generate a parse tree for any sentence given to it. Another program then translates this parse tree into a PLANNER program whose action is equivalent to the meaning of the sentence. A disadvantage of this approach is that the syntax of PLANNER was not designed to represent the meaning of natural language sentences. It was designed as a theorem-proving language to enable people to write computer programs to solve certain classes of problem more easily. Schank shows that by designing a suitable representation language the problem of analysis is simplified because it becomes driven by the structure of that language. Because the syntax of PIDGIN is equivalent to Schank's representation language the ability to translate English into PIDGIN is guaranteed by the translator described by Schank. This chapter to some extent describes Schank's translation scheme applied to PIDGIN. However, a number of changes have been made in order to

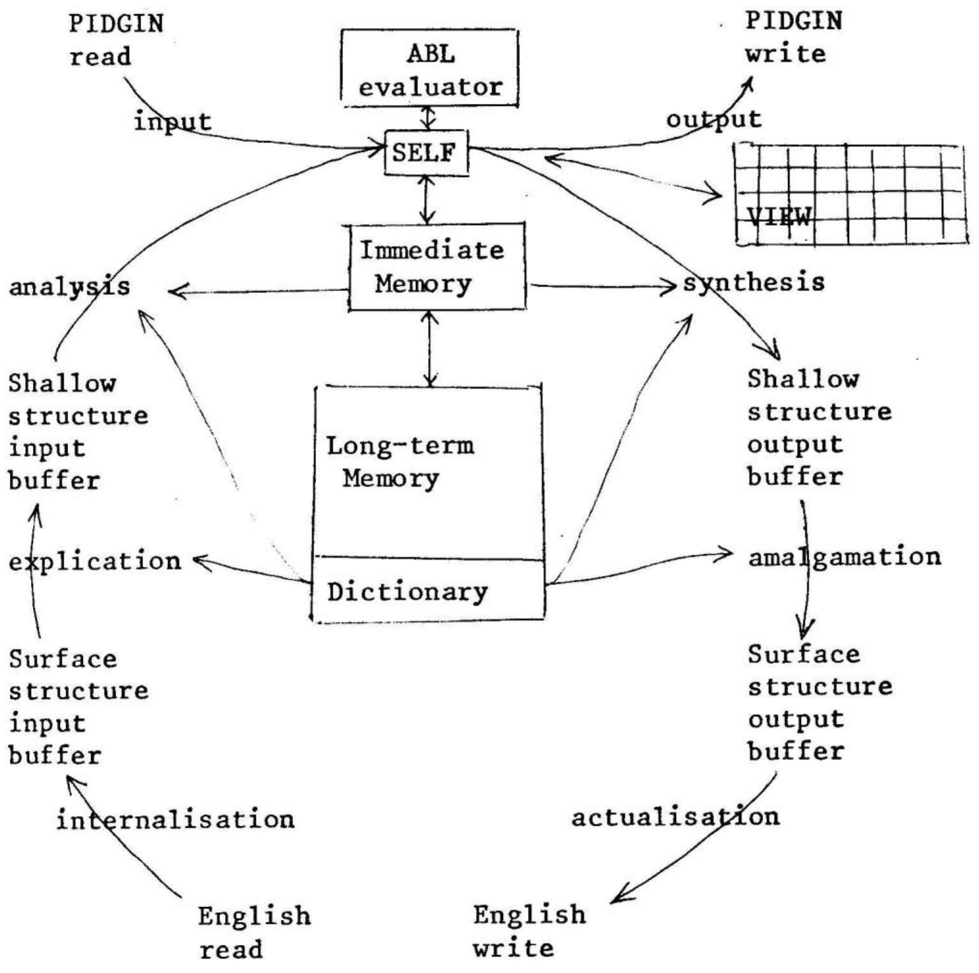
take advantage of the features of PIDGIN that simplify translation.
For example, the effect of context on translation, the resolution of cross-references, the generation of answers, the format of the dictionary, the suspension and re-activation of analysis and the fact that PIDGIN is implemented in ABL all simplify translation because of their special form in PIDGIN.

None of the ideas presented here have been implemented because of the equivalent translator described and implemented by Schank. This chapter describes both a PIDGIN version of Schank's translator and a number of more general points concerning the nature of the particular subset of English that can be handled by PIDGIN. The chapter is divided into a description of the analysis of a subset of English into PIDGIN, a description of the synthesis of PIDGIN back into English and a brief discussion of how these might be combined in a working system.

4.2 The Analysis of English

4.2.1 The Analysis Process

The data paths of the system are shown in the following diagram:



It can be seen that the translation of English into PIDGIN takes place in three steps internalisation, explication, and analysis. These three steps each perform one stage of the translation. Internalisation works at the character level and produces words in the surface-structure buffer. Explication works at the word level and produces items in the shallow-structure buffer. Analysis performs the major job of translating the standardised English items into PIDGIN.

4.2.1A Internalisation

This is the process of translating a sequence of characters, typed by the user, into a sequence of words. It becomes clear in the section on conversation (Section 4.4) that this translation cannot take place in isolation because of the possibility of typing errors and abbreviations. To cope with this it must combine with explication to ensure that the division into words takes place at the correct point. Apart from the practical problem of typing errors and spaces being omitted the interna1isation can take place at the character level. A word is a sequence of letters, a number is a sequence of digits (possibly with a decimal point) and other characters are treated as separate symbols. One practical problem is the detection of the end of the user's input. It might be terminated by the end of the line or the sentence might be run over more than one line or there may be more than one sentence on a line terminated by full-stops, question-marks or exclamation marks. One simple solution is to insist that all sentences are terminated in some standard way (symbol or end of line with the option of continuing over more than one line by the use of a continuation symbol). A better solution would be to do all analysis strictly from left to right word by word then if the end of the line was reached and the analyser had no outstanding syntax structures to be completed the input would terminate otherwise another line would be read, this could be combined with the ability to terminate by symbol if desired. The fact that people find trouble with this problem especially when communicating using a terminal indicates that no ideal solution exists. The output from interna1isation is a sequence of words, numbers, where numbers have been translated into the appropriate internal machine representation, and symbols, where the character has been translated

into a word {for example, "." into DOT, "?" into QUERY and so on).
Spaces are ignored.

4.2.1B Explication

This is the second stage of the translation process. Words are translated into a form that can be handled more easily by analysis. For example, affixes are removed and translated into separate qualifying items and idioms are replaced by the equivalent standard phrase. The result is a sequence of items all of which appear in the systems dictionary and which are loosely joined into structures (for example, word plus affix structure or quoted text structure).

a. Affixes

If a word does not occur in the dictionary a check is made to see if it is possible to remove affixes to produce a sequence of items that appear in the dictionary.

It is useful to distinguish a unit lower than the word in English grammar, it is called the morpheme. It arises through considering the structure of words, for example, many words occur on their own and with certain endings in particular contexts, for example, hunt, hunts, hunted, hunting and hunter. A morpheme that may be used as a word is called a free morpheme and the others bound morphemes, and the obligatory part of a word is called the base morpheme. In general the base morpheme is a free morpheme though such words as detain, contain and deceive, conceive may best be analysed as two bound morphemes.

Two types of bound morpheme can be distinguished, inflexional, which may only occur once in a word and always occur last, and derivational which do not form into sets and may occur more than once. For example, novel+ist+s consists of free base morpheme+bound derivational morpheme+bound inflexional morpheme. The items in PIDGIN correspond roughly with the morphemes of English grammar. The two types will now be considered:

i) Derivational. In general in PIDGIN derivational items are checked for if the word is not in the dictionary but if they are found they are removed and ignored.

Examples are: -ISH, -IST, -ISE.

ii) Inflexional. These form sets corresponding to the word classes of verb, noun and adjective.

Noun inflexional. Only "s" is checked for, if present it is made a separate item. Other plural forms are handled by the idiom facility.

Adjective inflexional. The comparative and superlative postfixes (-ER and -EST) are checked for and if present MORE or MOST is inserted before the adjective and the affix removed.

Verb inflexional. The endings -S, -ED, -ING and -EN are checked for and if present are separated. Irregular verb forms are handled by the idiom facility.

b. Idioms

It is possible for the user to define idioms, that is to specify one or more items to replace one or more input words. This facility can be used to cut down the number of dictionary entries required. The primitive IDIOM is used to extend the idiom dictionary, for example:

```
$ <IDIOM [APOSTROPHE S] POSSESSIVE>.

$ <IDIOM [S APOSTROPHE] POSSESSIVE>.
$ <IDIOM BEST [MOST GOOD]>.
$ <IDIOM MEN [MAN S]>.
$ <IDIOM FEET [FOOT S]>.
$ <IDIOM ATE [EAT ED]>.
```

4.2.1C Analysis

Analysis proceeds at two levels, the sentential and the conceptual. At the sentential level the Analyser uses a number of language dependent heuristics to look for the main nouns and verbs. It then brackets the dependent words with their governor using agreement rules based on word endings and function words. Thorne (1968) has shown that it is possible to do substantial syntax analysis without any information about "open class" words (nouns, adjectives and verbs). The sentential level analysis is carried out on the shallow structure of items constructed by explication. At the conceptual level the system is restricted by the syntax of PIDGIN as to the possible deep structure it can create. It is also limited by the memory as to the possible combinations of concepts allowed. Unless the user has a priority of 100 the system will not allow any combination of concepts unless some combination already in the memory can be substituted for it. Thus the conceptual level allows a top-down approach to the analysis because it knows what structures are possible, while the sentential level suggests a bottom-up approach because it is faced with the actual input sentence. The two levels are linked through a special memory known as the dictionary. The dictionary specifies the conceptual structure associated with each syntactic structure. Entries can be added to the dictionary by the primitive DICT. Each entry has the form:

DICT item type structure

where "item" is the shallow structure item, "type" is the syntax class and "structure" is the associated conceptual structure.

Possible syntax types that may be specified are:

INTRANSITIVE no sentential object, e.g. sleep, die.

TRANSITIVE a direct object, e.g. hit, like, wants.

INDIRECT no direct object, but a possible indirect object, e.g. go.

PSEUDO direct object to be treated conceptually as the actor in a relation of the form: X DO ... CAUSE Y ACT ... where X is the sentential subject and Y the sentential object, e.g. break, grow, open.

TRANSFER two objects with no preposition in front of them, in which case the first is recipient, the second the object, or in reverse order separated by "to", e.g. give, buy, sell.

DOUBLE two subjects and no sentential direct object, or an ordinary transitive verb followed by "with", e.g. fight, communicate.

STATE "that" following the verb or a noun-verb combination as object, possibly separated by "to", e.g. think, see, allow.

DETERMINER followed by a possibly qualified noun, e.g. the, some, this.

ADJECTIVE followed by a possibly qualified noun, e.g. red, tall, happy.

NOUN possibly preceded or followed by qualifying nouns, e.g. boy, John, we.

PREPOSITION followed by a noun group, e.g. with, to, on.

QUESTOR introduces a question, e.g. what, how, which.

CONJUNCTION includes special syntax checks for "and" and "or" constructions, e.g. but, since, although.

AFFIX special cases, e.g. plural "s", possessive.

ADVERB complex rules for position, e.g. slowly, almost, still.

Appendix II gives some examples of dictionary entries.

The sentential syntax analyser makes use of routines associated with the word types together with a set of built-in rules specifying the order in which the word types occur in English. This information is used to guide the Analyser when it looks through the sentence for concepts to insert into the conceptions it is building up.

The Analyser thus proceeds as follows:

i) Surface level heuristics (word endings, function words and so on) are used to find the main verb, and subject.

ii) The main verb is used to extract a model conception(s) from the dictionary.

iii) The model conception is used to guide the system in what to look for; the sentential order rules tell the analyser where to look first, where to look if that fails and so on.

iv) Each word in the sentence gives rise to some piece of deep structure that must be fitted into the statement being constructed according to the restrictions imposed by the conceptual knowledge.

v) As the analysis proceeds the memory and deductive facilities are used to resolve questions of ambiguity and consistency.

vi) The completed statement is moved to the thought processor with an indication as to whether the sentence was a question, command or assertion.

vii) The processor checks the statement with memory for consistency and then it takes the appropriate action (adds it to the memory, obeys the command or answers the question).

4.2.2. Word Analysis

The last section discussed the complete analysis process from the point of view of the implementation of the translator. This was not covered in any detail as it has been described by Schank else where. However, the more general consideration of the adequacy of PIDGIN for representing a large subset of English must be discussed. Although it follows, to some degree, from the adequacy of Schank's notation Schank does not discuss this in detail. The following discussion compares the capabilities of the PIDGIN deep structure with the requirements of the English language, mainly as described by Quine (1960). This is done by taking English constructions and considering the equivalent PIDGIN structure. This is not done from the point of view of the oddities of English syntax but by considering some of the main semantic features. First at the word level and then, in the next section, at the sentence level. The vaguer and more difficult features that arise beyond the sentence level are discussed in Section 4.4.

4.2.2A Divided Reference

The dichotomy between singular and general terms has been considered important throughout the history of the philosophy of language. A singular term names or purports to name one object while a general term is true of a number of objects. There is a third category which acts in some ways like singular and in some ways like general terms, these are the mass terms.

4.2.2A1 Mass Terms

Mass terms, such as "red" and "water" are best regarded as terms which do not engage in the sophistication of divided reference. In this sense they can be regarded as the remnants of our childhood use of terms (for example, "Mama", "Dada"). Grammatically mass terms are like singular terms in resisting pluralization and articles. Semantically they are like singular terms in not dividing their references but they do not name a unique object each. In PIDGIN it must be decided whether to represent mass terms by entity concepts and thus regard them as referring to one scattered object or by group concepts and thus regard them as referring to various shaped and sized portions of the object stuff. The later alternative is chosen (although it conflicts with the way Quine chooses to treat mass terms) because it enables the various uses of mass terms to be more easily handled in PIDGIN. The problem of exactly what size a single portion of, say, water or red is, is not defined, it is simply what people choose to make it from occasion to occasion. By their use of the demonstrative people do choose to talk about certain sized portions of stuff named by mass terms, for example:

This water is unfit to drink.

That red is a shade too dark.

The following examples illustrate the way mass terms are handled in PIDGIN by group concepts:

Water is a liquid. ALL WATER <SUB A LIQUID>.
Red is a colour. ALL RED <SUB A COLOUR>.
This red is suitable. A [SUITABLE] RED.
The water is lovely. A [LOVELY] WATER.

The decision to treat mass terms as group concepts allows, for example, "the water in this glass is a liquid" to be deduced from the first example above but prevents "water is unfit to drink" from being deduced from "this water is unfit to drink". There are many other examples:-

Most water is now polluted.

<MORE <DIV ALL 2>> WATER <CONT POLLUTANT>.

Some footwear has doubled in price.

```
=N POUND <PRICE SOME FOOTWEAR>
      BECOME <MULT N 2> SELF.
```

Lamb is expensive meat.

```
ALL LAMB <SUB A MEAT< COST
      <MORE =N PENCE <<PRICE MEAN> ALL MEAT>> PENCE>>.
```

4.2.2A2 General Terms

a. Absolute General Terms

Absolute general terms admit the definite and indefinite article and the plural ending. The plural form is handled by the quantifier <MORE 1> unless the quantity is stated more explicitly. The indefinite is handled by the quantifier <EQUAL 1>.

The definite article is used with a general term in order to describe a single object, rather than to name it as a singular term does. Whenever the definite article is used PIDGIN attempts to replace the description by a singular term with the same reference; this is usually known from the context otherwise the definite article would not have been used. However, there are cases where it is used when the object is only potentially knowable, for example, "the oldest man in England", "the winner will receive £10". In these cases PIDGIN automatically creates a new concept which refers to the individual described and the descriptive information is used to define it. For example, "the oldest man" would produce:

```
XYZ <SUB A MAN>.
XYZ <AGE =N YEAR <<OLD MAX> ALL MAN>>.
```

Many general terms also double as mass terms, for example "Mary had a little lamb" is ambiguous because "lamb" may refer to a single infant sheep or to a part of that single, scattered object consisting of lamb meat. The two views can be resolved by introducing a concept, such as MEAT, when the PIDGIN equivalent of the two meanings becomes:

```
MARY TRANSFER SOME MEAT <PART SOME LAMB>
                     A PLACE MOUTH.

MARY <POSS LAMB <WEIGHS <LESS =N KILO
```

```
<<WEIGHT MEAN> ALL LAMB>> KILO>>.
```

The central feature of a general term, that of dividing its reference forms the basis of what in PIDGIN is equivalent to the idea of a variable in many computer programming languages. For example, the general term "man" divides its reference among all male homo-sapiens.

The equivalent PIDGIN concept MAN divides its reference (in the PIDGIN sense of the word, see Section 2.1.1) in an analogous way. At any moment the PIDGIN concept may have as its reference some single example of an individual man or class of men. This corresponds to the ability to say "the man" to refer to the one man that is being talked about. Thus the PIDGIN group concept MAN may have as its reference any of the entity concepts JOHN, BILL or JOE and so on, which have been stated as being entity concepts for which MAN may be substituted. It may also have as its reference any of the group concepts, say FATHER, BACHELOR or UNCLE that have been stated as being group concepts for which MAN may be substituted.

b. Relative General Terms

A relative general term is true of one thing with respect to another, for example "part of", "bigger than", and "brother of". In PIDGIN these are translated into a structure with the form A <R B> where A and B are the things related by R. Relative terms can be paired off into mutual inverses so that if R1 is the inverse of R2 then A <R1 B> says the same thing as B <R2 A>. For example:

```
WEIGHS <INVERSE WEIGHT> .

      JOHN <WEIGHS 60 KILO>.
      60 KILO <WEIGHT JOHN>.

BELONG <INVERSE POSS>.
      JOHN <POSS FIDO>.
      FIDO <BELONG JOHN>.
```

OLD <INVERSE AGE>.
A PERSON <AGE 20 YEAR>.
20 YEAR <OLD A PERSON>.

The relation of a relative term is sometimes used derelativised in English as an absolute term true of anything x if and only if the relative term is true of x with respect to at least one thing. Thus "part", "brother" and "container". In PIDGIN a relation can be used as a nominal if it is being talked about as a relation (as in the: first line of each example above), but in the usual case of a derelativised term it will be expanded into the relation and two general concepts of the correct category, for example:

brother
A MAN <BROTHER A PERSON>.

part

AN OBJECT$_1$ <PART AN OBJECT$_2$>.

Composite absolute general terms can be constructed from a relative and a singular term, for example:

a brother of John

A MAN <BROTHER JOHN>.

a broken part of machinery

A [BROKEN] OBJECT <PART A MACHINE>.

<u>c. Relative Clauses</u>

A relative clause is usually an absolute term and is formed by substituting a relative pronoun such as "which", "who", "whom" or "that" for a singular term in what would otherwise be a complete sentence. It is true of just those things which would yield a true sentence if put in place of the relative pronoun. Relative clauses are adjectival but in PIDGIN they are translated into separate conceptions just as they are if used as unrestrictive clauses which are stylistic variants of co-ordinate sentences anyway. In many translation schemes the regimentation of relative clauses into "such that" clauses is suggest ed and this can be regarded as forming the first part of the conception split. For example:

A car which John bought is in the garage.

becomes:

A car such that John bought it is in the garage.

which in PIDGIN is:

A PERSON PASS A CAR SELF JOHN
AND THE CAR <IN GARAGE>.

4.2.2A3 Singular Terms

A singular term admits only the singular grammatical form and no article and it names or purports to name a single object. In PIDGIN they become entity concepts.

The problem of ambiguity of singular terms, for example the fact that the word "John" refers to millions of people, is not a deep problem, merely a question of resolving the ambiguity and careful distinction between the English name "John" and the internal concept JOHN so that multiple uses of "John" are distinguished internally, say $JOHN_1$ $JOHN_2$ and so on, or any equivalent scheme.

Singular terms can be obtained from general terms by the use of the demonstrative particles "this" and "that". By this means single objects can be specified without the need for naming .though, as was pointed out in the last part, PIDGIN .does carry out an automatic internal naming process. This is done so that extended discourse about a demonstrative singular term can be collated under a single concept. New singular terms can be introduced this way by using the "is" of identity, for example "This river is the Thames".

Mass terms also generate singular terms with the demonstratives, for example "this water", "that sugar". Thus

THISWATER <SUB A WATER>.

that is, the new concept THISWATER is identical to one part of the single, scattered water stuff.

The reference of demonstrative singular terms varies from occasion to occasion like the reference of the indicator words ("I", "you", "now", "here", "today" and so on) and PIDGIN handles them in the same way, that is by resolving the reference at translation time so that the PIDGIN deep structure produced does not contain any such unstable singular terms.

The definite article can be regarded as a weakened demonstrative particle used to form singular descriptions, and further the pronouns can be regarded as short singular descriptions ("he" = "the man", "she" = "the woman", "it" = "the thing"). In PIDGIN demonstrative singular terms, singular descriptions and pronouns are all translated into concepts that refer uniquely and stably to the current reference of the unstable English phrase.

Indefinite singular terms are formed from general terms with the indefinite article, for example "I saw a lion." This is translated to A LION in PIDGIN, that is, anyone member of the class of lions. Note the difference between this and the translation of "the lion" as ALBERT, where ALBERT is the particular single individual lion referred to. Also note the repeated use of the terms and the translation:

John saw a lion and Bill saw a lion.

JOHN PERCEIVE A LION$_1$ AND BILL PERCEIVE A LION$_2$

John saw the lion and Bill saw the lion.

JOHN PERCEIVE ALBERT AND BILL PERCEIVE ALBERT.

Note the use of concept subscripts in the first example to allow the particular lions seen to differ. It is because of this that pro nouns (always definite) cannot simply be regarded as standing in place of (can be totally replaced by) their antecedent, consider:

John saw a lion and Bill saw it.

JOHN PERCEIVE A LION$_1$ AND BILL PERCEIVE A LION$_1$.

This time the same subscript is repeated to force the lions to be the same individual.

Indefinite singular terms can also be formed from other particles, such as "every", "some", "each" and "any". The translation of these terms together with pronouns and especially with negation is erratic and will not be discussed here.

Abstract singular terms that purport to name qualities or attributes have long been a source of confusion. Every general term can be regarded as delivering an abstract singular term usually constructed by suffixing "-ness", "-hood" or "-ity". In PIDGIN they are avoided where possible by reducing them to concrete objects together with the single group of abstract objects, the numbers. For example:

Humility is rare.

<MORE ALL [HUMBLE] PERSON> PERSON.

That is the number of people is more than the number of humble people.

4.2.2A4. Composite Terms

The composition of the equivalent of singular and general terms to form new terms is the source of all PIDGIN's concepts. This part considers the rules that govern the composition. Composite general terms can be formed by adjoining one general term attributively to another, thus "red house", "iron bar". In PIDGIN if the term has been specified as ATTRIBUTE it can be used as a qualifier:

A [RED] HOUSE.

AN [IRON] BAR.

otherwise it must be linked using a relative term. A composite general term is true of just those things that both components are true of. If the relationship is obscure but significant then the general relation MOD is used. This can be read as "modified by" or "to do with", for example:

Water rat	**A RAT <MOD A WATER>.**
Fire Hydrant	**A HYDRANT <MOD A FIRE>.**
Steam engine	**AN ENGINE <MOD SOME STEAM>.**

Syncategorematic adjectives, that is those that cannot stand on their own as terms, are like adverbs in that they can be attached to terms to form further terms. In PIDGIN they are not treated in the same depth as in English. It is hoped that they can usually be handled in

PIDGIN directly (for example, adverbs of time and place and some of manner) or paraphrased or ignored or left for future analysis (for example, adverbs involving analogy such as "majestically"). General terms can also be obtained by combining a relative term and a singular term ("brother of John") as well as. by combining a relative term with a plural general term ("benefactor of refugees"). These composite general terms can be made into singular terms using "this", "that", or "the" and these can be combined with relative terms to form general terms again. For example: A part of the skin of a red apple

AN OBJECT <PART A SKIN <PART A [RED] APPLE»

In PIDGIN the combining of concepts is controlled by the previous occurrence of a similar combination. Unless the user has priority 100 a combination will not be set up unless some combination already present in the memory can be substituted for it. For example, if the combination:

A [PLEASURE] PERSON

has already been set up in the memory then the following combinations will be allowed:

[HAPPY] JOHN
A [SAD] CHILD
A [JOYFUL] WOMAN

(assuming PLEASURE can be substituted for HAPPY, SAD and JOYFUL, and PERSON for JOHN, CHILD and WOMAN). But the following will not be allowed:

A [HAPPY] BOOK

A [SAD] TREE

4.2.2B Ambiguity and Vagueness

The way in which language is learnt, by a process of piece-meal education arid induction, naturally leads to vagueness. Singular terms can be vague as to the spacio-temporal limits of the thing named; for example "the South Downs" specifies a particular range of hills but their precise limits are vague. General terms can be vague as to their extension, what counts as one, and as to the boundaries of each one. For example, "mountain" is vague as to when a hill becomes one, and thus how many there are, and as to the boundaries of each one. PIDGIN requires rules if such vagueness is to be resolved. For example, if told the heights of a number of hills and mountains and which are hills and which mountains PIDGIN will be unable to classify any new feature simply given its height. But if a rule is set up that defines a mountain as always being over 1000 metres and a hill as always being under then it will be able to deduce the class of a new feature from its height.

Another group of words that vary in meaning are words such as "big - small", "hot - cold", "high - low", "heavy - light" and so on. These words are not really ambiguous or vague but dependent on the governed term. In this sense they can be regarded as a type of syncategorematic adjective. In PIDGIN they Could be handled by either making them absolute or by making them clearly relative or by treating them as inseparable from the nominal they qualify. For example, "heavy" and "light" could be handled by substituting the actual weight to enable absolute comparisons to be made between say a "heavy woman" and a "light man". However, such a translation requires a lot of knowledge about the average and the distribution and it is extremely difficult to quantify for pairs such as "smooth" and "rough". The second method is to try to place every term on a relative rather than an absolute scale. Thus:

heavy man is heavier than heavy woman is heavier than average man is heavier than average woman is heavier than light man is heavier than light woman.

This method does not suffer from the disadvantage of quantification but it does require extensive knowledge of the relation between completely different terms if they are to be compared (for example, to judge that a heavy mouse is in fact lighter than a light elephant). If different terms are not to be compared then the final method can be used and PIDGIN merely needs to know:

heavy X is heavier than average X is heavier than light X.

The method used is a variation of the last method that allows some comparison. All such relative adjectives are related to the average, which is regarded as the sum divided by the total number. For example:

a heavy elephant

```
AN ELEPHANT <WEIGHS <MORE =N KILO
       <<WEIQHT MEAN> ALL ELEPHANT>> KILO>.
```

a smooth skin

```
A SKIN <ROUGHNESS <LESS =N AMOUNT
       <<ROUGH MEAN> ALL SKIN>> AMOUNT>.
```

The knowledge can be made absolute by specifying the weight of the average elephant or by specifying what amount (arbitrary) constitutes the average skin roughness. For example:

```
3000 KILO<<WEIGHT MEAN> ALL ELEPHANT>.

70 KILO«WEIGHT MEAN> ALL MAN>.
```

would allow average (and thus heavy and light) elephants and men to be compared. And:

ALL SKINWOMAN <SUB A SKIN>.

ALL SKINRHINO <SUB A SKIN>.

10 AMOUNT<<ROUGH MEAN> ALL SKINRHINO>.

1 AMOUNT<<ROUGH MEAN> ALL SKINWOMAN>.

The skin of an individual is substitutable for the appropriate skin class, for example:

Mary has smooth skin. .
ALL SKINMARY <SUB A SKINWOMAN><BELONG MARY>

<ROUGHNESS <LESS 1> AMOUNT>.

The problem of comparing more or less than average quantities depends on the distribution. There are many ways of incorporating this information, for example, specifying the standard deviation and assuming more than average means more than the standard deviation from average. Another method would be for the system to do a curve fit on all the information in memory and work out the standard deviation. However, these methods are not ideal, it requires a more detailed analysis of the interaction between the systems knowledge of the world and facts about the world, and how this could best be fitted into a statistical framework in which the system could perform useful interpolations and extrapolations. Sentence truth does not usually hinge on vagueness because people make sure it does not. For example, the height of a mountain is not vague, but the area is, for the very reason that it is the height that is of most interest to people when mountains are discussed. One area that does encounter vagueness with respect to sentence truth is deciding on records, the longest river, largest mountain and so on. On such questions PIDGIN relies on the simple numeric comparison of the in formation it has been given. Ambiguity is illustrated by the fact that a black feather is clearly light and clearly not light. Vagueness is associated with the exact placing of boundaries, ambiguity with striking divisions. Usually ambiguity is resolved by context, either within the sentence, within the broad context of discourse, or within the even broader context of social conventions and natural inclinations. The broader the context the more knowledge required to resolve the ambiguity and the more difficult it is for an automatic translation scheme to incorporate and make use of the necessary knowledge. This problem has been considered by Schank and others and is not discussed here.

The division between ambiguity and generality is usually resolved by considerations of etymology, intuitive sense and grammatical function. However, in a deep structure representation other considerations apply and a term is usually regarded as ambiguous if its incorporation as a single concept in the deep structure would require different interpretations from context to context. Thus the word "hard" as applied to chairs and questions would usually be regarded as generally true of both but in the deep structure two separate concepts would be used. Ambiguity arises with composite terms where a syncategorematic: reading is possible, for example, "a poor artist". It also occurs with the seemingly innocuous indefinite article:

i) A lion likes red meat.

ii) A lion escaped.

which leads to examples involving the plural:

iii) I like lions.

iv) I hear lions.

v) Lions are rare.

vi) I hunt lions.

vii) I am hunting lions.

Examples (ii), (iv) and (v) are straight-forward, "a" means a single unspecified thing, plural means more than one and plural in subject position without an article refers to the class of lions, the problems associated with these are discussed earlier. The other examples have a common thread, they all involve dispositions. How this might be used to translate the examples is discussed in the next section.

4.2.3 Sentence Analysis

The definition of what is and what is not a sentence will not be attempted. Sufficient to say that a sentence translates into one or more PIDGIN statements. Even elliptical sentences such as "Yes." or "The pawn." are translated into complete conceptions. This section discusses those sentences that alter the concept structure of the system and some special aspects of the translation problem. The translation of more general sentences is relegated to the discussion of a few examples in the next section.

4.2.3A. Predication

The basic combination of two terms at the sentence level is that of general and singular terms in predication: <singular-term> is a <general-term>. The sentence is true or false according as the general term is true or false of the thing, if any, to which the singular term refers. The particular form of a predication depends on the grammatical category of the general term. If it is an adjective the article is omitted and if it is an intransitive verb the article and "is" are omitted:

John is a boxer.

John is big.

John sings.

However, there are mechanisms in English for converting the verbal and adjectival forms to substantival form, namely the "-er" suffix and the use of the class name. For example, to translate the two non-standard cases above into a standard form:

John is a big person.

John is a singer.

This enables predication to be translated into the PIDGIN deep structure:

ALL group-concept <SUB A group-concept>.

or

entity-concept <SUB A group-concept>.

Mass terms can occur on either side of the predication. In the predicative position normally occupied by a general term a mass term is treated as referring to a portion of the stuff the mass term refers to ("That puddle is water."). In the other position it is treated as referring at once to all the scattered portions ("Water is a liquid."). Predication is the mechanism whereby new concepts are described to the PIDGIN system. In general the new concept occurs to the left of the SUB relation, an old concept or combination of concepts to the right and the complete conception has a modality of DEFN (see Division 2.3D). If this process is followed back then it can be seen that every concept on the right must have occurred in some previous definition on the left. Obviously all concepts "are eventually reduced to some initial group of concepts, namely those built into PIDGIN itself (see Section 2.1.1). Appendix II shows a typical sequence of definitions for building up a basic set of concepts from those built into PIDGIN.

4.2.3B Identity

Identity is also expressed in English by "is" in sentences with the form: <Singular-term> is <singular-term>. The old philosophical dilemma - "to say of two things that they are identical is nonsense, and to say of one thing that it is identical with itself is to say nothing at all" (Tractatus 5.5303, Wittgenstein 1922) is resolved in PIDGIN by distinguishing between the object an individual concept refers to and the definite description of that object in terms of other concepts. The system may contain any number of different descriptions of the same object each associated with a different concept. The fact that two descriptions describe the same object

is a statement of experience not something inherent in the descriptions themselves (for example "The evening star is the morning star"). When PIDGIN encounters any singular description describing an object not previously encountered then a new concept is formed and a predication is made between the new concept and the general term corresponding to the singular description. Thus:

The barn behind my house.

MYBARN <SUB A BARN <BEHIND MYHOUSE>>.
The barn behind your house.
YOURBARN <SUB A BARN <BEHIND YOURHOUSE>>.

The two concepts are assumed to refer to different barns. However, if PIDGIN is told they are the same: The barn behind my house is the barn behind your house.

MYBARN <EQUIV YOURBARN>.

then it will make the two entity concepts equivalent. The result of this equivalencing will be to use only one of the concepts in the future and to combine all the statements concerning each concept.

4.2.3C. Time

Time plays a large part in the structure of any sentence in English because it is necessary to give every verb some tense. The complex grammar of tenses can be greatly simplified in the deep structure by the use of explicit time, period and interval specifiers in the modifier of the statement. These specifiers are placed among the modifiers of the complete statement because it is only an event that can have a time and a duration and a PIDGIN statement is a description of an event. The temporal qualifiers "now", "then", "before t", "at t", and "after t" are systematised by translating them onto an absolute time scale. The time of day and the date (relative to the year 0 A.D.) are used to set up an absolute time scale stretching into the past and future. "now" moves along this time scale so that at each new moment it occupies a new position. A point on the time scale is expressed as a sequence of a number of each of the time units. Points on the scale can be associated with events and terns, "John's birthday", "last week", "the end of the year", "Christmas". When the term refers to a recurring event the term is kept in order to retain the ambiguity of reference, if it refers to an absolute event the term is kept in order to retain the description associated with the name of the event, otherwise it is translated to an absolute time point.. The various tenses are handled by the above time specifier, plus a period modifier to say whether it is an EVENT and if it is whether the sentence refers to the START, WAX, CONTINUING, WANE or STOP aspect of the event or to the complete event. If it is an EVENT then the interval modifier can be used to state the length of time occupied by the specified phase of the event. Other modifiers related to tense are concerned with intent and conditionality. The following examples show how these modifiers can be combined:

```
I am walking
    CONTINUING <TIME t2>

I walked.
    EVENT <TIME <LESS t2>>

I will walk.
    INTEND <TIME <MORE t2>>

I can walk.
    CAN <TIME t2>

I started walking yesterday.
    START <TIME t1>

I will stop walking this evening.
    STOP <TIME t1>

I will have been walking two hours.
    CONTINUING <TIME <MORE t2>><INTERVAL 2 HOUR>
```

where t1<t2<t3 and NOW=t2.

4.2.3D. Ambiguity

Sentence ambiguity is the central problem of machine translation. The problem is handled in recent question-answering systems (Winograd 1971, Schank 1970a) by writing translators that make use of the know ledge stored in the systems data-base together with the systems deductive capabilities. An example requiring the resolution of ambiguity is discussed in the next section. As well as the ambiguity of terms (see Division 4.2.2B) there can be ambiguities of grouping and scope. Grouping can best be illustrated by the use of brackets, for example:

(boy scouts) fish.
(boy) scouts (fish).

((pretty poor) girl) guides (light torch holder)).
(pretty (poor (girl guides))) light (torch holder).

etc. The bracketing in each case can usually be determined by the surrounding context plus more general knowledge. However, many examples are simply ambiguous and cannot be bracketed in anyone way but such examples do not often occur in conversation. Other true ambiguities are illustrated by "big European butterfly" in which the syncategorematic adjective "big" may include "butterfly" or "European butterfly". Such ambiguities can be resolved only by knowing common usage or by further conversation. Indefinite singular terms also give rise to problems of ambiguity concerned with scope, for example: "Each thing that glisters is not gold." If the scope of "each thing" is the whole sentence then the assertion denies that gold glisters, if the scope does not include the negator then the assertion is true because it is then equivalent to "it is not the case that everything that glisters is gold". Once again the ambiguity is resolved by convention. In PIDGIN the two possible deep structures would be:

A [GLISTER] THING <SUB A GOLD> [NOT].
ALL [GLISTER] THING <SUB A GOLD> [NOT].

Further ambiguity of scope occurs with "every" and "any" and in this case convention rules that "every" requires the smallest scope and "any" the largest. So:

I do not know any poem.

means

I know no poem.

but

I do not know every poem.

means

It is not the case that I know every poem.

4.2.3E Opacity

A purely referential position in a sentence is one in which any term designating the same object may be substituted without altering the truth value of the complete sentence. This is called substitutivity of identity and it is true for most of the terms in the examples given.; The most obvious case where it fails is with quotation, for example:

"Jekyll" has six letters.

the singular term "Hyde", although it refers to the same person, can not be substituted without destroying the truth of the sentence. PIDGIN handles quotation by keeping the quoted text in character form so substitution is prevented in all circumstances.

Contexts in which substitutivity of identity may not occur without the possibility of change of truth value are called referentially opaque, as opposed to those in which it may occur which are called referentially transparent. It is not necessary to discuss the problems that this subject brings with it because of the simplicity of PIDGIN statements compared with the subtlety of English. The following brief description shows how it is possible to handle the problem within PIDGIN.

Many verbs are associated with the relation between minds and the world, and with the internal models in minds. For example, "believe", "seek", "say", "wish", "fear" and so on. These -have been recognised as requiring special treatment with respect to the substitution of terms in the c1auses they govern and they have been christened propositional attitudes. All these propositional attitudes translate into one or other of the mental acts in PIDGIN, COGITATE and TRANSMIT. Each of these acts takes a complete statement as its object. If the subject of the resulting conception is the PIDGIN system (SELF) then concepts in the object statement can be freely substituted for any other concepts that the system knows are equivalent. This is because the statement forms part of the system itself and can be treated like any other part. If the subject is not the system however then it is reasonable that assumptions should not be made about the knowledge contained in the mind of the subject. For example: John believes Jekyll is dead.

JOHN TRANSMIT< [DEAD] JEKYLL> MEMORY SELF.

That is John can transfer the thought "Jekyll is dead" from his memory into his consciousness. If the conception:

JEKYLL <EQUIV HYDE>.

is known to PIDGIN this does not imply that John also knows this fact so it follows that it cannot be deduced that "John believes Hyde is dead". A similar justification can be found for the other propositional attitudes and the result is that the PIDGIN system has a working mechanism for deciding when a substitution can be made.

4.2.4 Examples of Analysis

The following examples are taken from Schank (1973) but have been modified in order to illustrate the features of PIDGIN.

1) "I want to go to the park with the girl."

The initial internalisation and explication has already been discussed. The explication process look for word endings and function words in order to discover the main subject and verb and any direct object(s). So by the time analysis begins these sentential features have been marked. The dictionary forms the basis of the analysis as it is this that contains the mapping from sentential-level words to the conceptual-level concepts. In the above example the verb "want" is found and looked up in the dictionary giving three possible structures:

i) STATE

A LIFE DO AN ACTION$_2$ CAUSE A PERSON$_1$ BECOME [PLEASED] SELF.

ii) TRANSITIVE

A PERSON PASS AN OBJECT A PERSON A PERSON$_1$ CAUSE A PERSON$_1$ BECOME [PLEASED] SELF.

iii) TRANSITIVE

A PERSON$_2$ TRANSFER SELF A PLACE$_4$ A PLACE$_5$ <LOC A PERSON$_1$> CAUSE A PERSON$_1$ BECOME [PLEASED] SELF.

Because the verb in the above example is a state verb only the first entry is applicable. Therefore another complete conception is required as the object of the act DO. The next verb in the sentence is looked up in the dictionary giving:

i) INDIRECT

AN ANIMAL$_1$ TRANSFER SELF A PLACE$_4$ A PLACE$_5$

As the conceptual subject of the second verb is "I" the concept ANIMAL is replaced by SELF. As the entry is a single conception the DO act is replaced by the conception rather than making it the object of DO. The next part of the sentence "to the park" is recognised as a possible case, "park" is looked up in the dictionary and found to mean PARK, which is a group concept for which PLACE may be substituted. The singular description is replaced by an entity concept (say, PARKX) and placed in the destination (because of "to") part of the act. This gives:

SELF TRANSFER SELF A PLACE P ARKX

CAUSE SELF BECOME [PLEASED] SELF.

The final part of the sentence is interesting because of the many ways in which it can be used:

"with+term"

i) term is object of the THROUGH statement (Schank's instrumental case), e.g.:

I hit the boy with a cane.

ii) term is an additional term of the conception, e.g.:

I went with the girl to the park.

iii) term is an attribute of the immediately preceding term, e.g.:

I hit the boy with long hair.

iv) term is an attribute of the act, e.g.:

I hit the boy with vengeance.

The possibilities are checked in the order given until a suitable one is found. In this case the first is ruled out because the only THROUGH statement involving SELF TRANSFER is SELF MOVE and the object must be a body part, which "girl" is not. The next case is checked and found to be satisfied, girl can be another actor for the act TRANSFER.

If the third possibility had been checked it would only succeed if the system knew about a particular park which had a girl in it. This brings up the question of handling the definite article and other anaphoric references such as pronouns. In all these cases the analyser uses the memory to try to find the entity concept referred to, in this case a park. This is done by using the context information contained in the reference of concepts, thus if a park had been referred to in the immediately preceding discourse then the group concept PARK would still have as its reference the entity concept which is the name of the particular park under discussion. If this is not the case then the most recent statements must be searched for some mention of a park and if none is found then more information is looked for in the sentence, in this case for example "with the girl" would be taken as a definition of the park described and the complete memory would be searched for such a park. Pronouns are resolved by searching for the most recent suitable object or person in the sentence being analysed or in previous recent statements. The final analysis will be:

[SELF GIRL] TRANSFER SELF A PLACE PARKX

CAUSE SELF BECOME [PLEASED] SELF.

Note that SELF as subject refers to the PIDGIN system but SELF in the object position refers to the complete subject.

2) "I saw Birmingham flying to Edinburgh."

The second example is described to indicate how syntactic ambiguity is resolved. This has always been a problem with the traditional approach to syntax analysis because of the lack of semantic information. The sentence above is a poorly constructed English sentence because it is nearly always parsed incorrectly on first reading. However, as it can be understood a conversational system should also be able to analyse i to When "see" is looked up in the dictionary the only suitable (transitive) entry gives:

SELF TRANSMIT BIRMINGHAM EYE SELF

THROUGH SELF PERCEIVE BIRMINGHAM.

The next verb ("fly") is' looked up in the dictionary but it is not clear what the subject is. In such cases the last actor is taken, in this case "Birmingham". However, the system cannot make Birmingham fly because Birmingham is a place and fly requires an animal as subject (more specifically FLY requires BIRD,PLANE,INSECT,PILOT or PASSENGER as subject, but Birmingham is none of these). The choice of actor is therefore rejected and the next rule is tried which selects, the next actor which is SELF. Selecting this actor meant crossing back over the main verb and another rule states that if this happens then the conception being generated is a time conception that modifies the original conception. This actor choice succeeds and assuming SELF is a PASSENGER and filling in the destination ("to place") gives:

SELF TRANSMIT BIRMINGHAM EYE SELF

THROUGH SELF PERCEIVE BIRMINGHAM,
WHILE SELF TRANSFER SELF A PLACE EDINBURGH
THROUGH A PLANE TRANSFER SELF A PLACE EDINBURGH.

3) "The old man's glasses were filled with sherry." This example is given in order to show how the knowledge possessed by the system can be used by the analyser to resolve semantic ambiguity.

The sentential analysis of this sentence will find it is passive and so the preceding actor will be made the object. If "by" is found then the following noun group forms the subject but in this case there is no subject.

The word "glasses" is looked up in the dictionary and found to be ambiguous between "drinking vessel" (GLASS) and "spectacles" (SPECTACLE). Further, the ambiguity is not resolved by its qualifier ("old man's"). As the definite article is used the system tries to resolve the "glasses" referred to by searching the memory. If no such information is found a choice could be made immediately that the most likely choice is SPECTACLE because they are more often associated with old men than GLASS. Alternatively the choice can be left until more information is available from the analysis of the rest of the sentence. The main verb "fill" requires a LIQUID as object of a TRANSFER act and a CONTAINER as the destination. This enables the first definition of "glass" to be chosen and the final statement becomes:

A PERSON TRANSFER SOME SHERRY A CONTAINER SOME

GLASS <BELONG JOHN>.

where JOHN is the entity concept corresponding to the singular description "the old man".

4.3 The Synthesis of English

The synthesis of English can vary from the random generation of semantically meaningless but syntactically correct sentences by trivial algorithms, through slot-and-filler techniques such as that used by Winograd's system, to the complete generation of semantically correct sentences that convey sensible information in the context of a conversation. In terms of PIDGIN the problem divides into two parts, the generation of a statement that conveys the required information and the translation of that statement into English. 4.3A. Generating Statements A statement may be "generated" simply by retrieving a matching statement from memory. This is the way in which an answer to a user's question is generated, the question forms the basis of the answer. Thus it is the user's ability to generate statements that is being utilised by PIDGIN. Because PIDGIN may reply elliptically the answer may bear little resemblance to the question at the English level, for example:

Who went to the park?

John.

but at the level of the deep structure involved the two are equivalent. A second point at which statements must be generated is when the system wishes to inform the user that some problem has arisen, such as an unresolvable reference, an ambiguity, an inconsistency or an unknown word. In some of these cases the statement comes from parts of the user's sentence and in others from the memory, these parts are then put together to form a statement which is output to the user. This system is a type of slot-and-filler algorithm but at the level of the deep structure. The basic forms of the possible statements the system might need to make to the user ("error messages") are stored in the memory and when used they have the appropriate parts filled in with the particular relevant structures.

The third type of generation arises in the mode of working in which the system automatically generates new statements from old and then asks the user if they are true by outputting them as questions. Another type of generation is discussed by one of Schank's associates (Goldman 1975). In this the basis of the output is past input which has been translated into the deep structure in memory. The point of this exercise is to illustrate that the system has "understood" the input because the output is a sensible paraphrase of the input. Because PIDGIN always outputs from its deep structures this always occurs. Finally, it would be easy to randomly generate statements using the concepts and conceptual knowledge in the memory. This could per haps be done by randomly choosing a conception structure from the verb-act dictionary, filling in a number of the actors with concepts of the correct type, quantifying and qualifying these actors by choosing quantifiers, attributes and specifiers that may modify the concept chosen and finally choosing a selection of statement modifiers. The output of such a procedure would be a semantically meaningful sequence of unconnected sentences.

4.3B Translating Statements into English

The synthesis of statements into English is an interesting problem as a good synthesiser should not output any more information than is necessary for the user to understand the statement. It should be fluent for example in its use of pronouns and idioms and it should be able to translate deep structure statements into the most natural phrasing and word order. The details of the formation of a syntactically correct sentence using the analysis rules and the dictionary in reverse can be found in Schank (1969b) and Goldman (1975). The ability to use pronouns and omit information known by the user is one of the good features of Winograd's outputting system. In PIDGIN the translation of deep structures into English is divided into three steps. The first, most complex, step is called synthesis and consists of searching the dictionary for the English words that are equivalent to the various parts of the deep structure. These words are then assembled into the shallow-structure buffer using the language dependent rules of analysis in reverse. The second step, called amalgamation, substitutes idioms and generates the surface structure which the final step, called actualisation, translates into a character stream suitable for output to the user.

4.4 Conversation

Conversation contains many features and is associated with many problems that have not been discussed. A conversation is usually about some topic, that is, people usually talk for a reason. This fact is made use of in conversation by leaving certain things unsaid because they are obvious to both parties. However, when one party is a computer this raises problems concerned with how much the system needs to know before it can understand typical typewritten communication. In order to answer this question a lot more needs to be known about the form of such conversations. Although AI workers generally assume that typical typewritten communication reads like children's story books there is evidence that this is just not the case. Work done by Chapanis (1975) indicates that when two people communicate to solve a problem the language used is a long way from the English that most of us believe we use. The results of Chapanis are so relevant to the problems faced by conversational computer systems that they are worth describing in detail with respect to PIDGIN. Chapanis set out to investigate how two people communicate to solve a simple problem when various combinations of communication channel are available, for example, voice only, voice and video, voice and typewriting, typewriting only and so on. He discovered some interesting relationships between the time taken to solve the problem and the channel available but the interesting result from the point of view of PIDGIN is a transcription of a typical conversation between two people using only a typewriter to communicate and trying to solve a simple problem. The problem was for one person to assemble a piece of equipment guided only by instructions from the other person who had the equipment

assembly manual but could not see the equipment. The following is part of the communication:

> **<u>goahead</u>doyouknowhowto put this togteher**
>
> **<u>ill try</u>its a trash toter ill type you the directions ok**
> **put axle thru 38th holes from outside**
> **<u>38th holes/??</u> yes**
> **put 1 handlebar on back of each outer frame line up bol t holes**
> **<u>what does outer frame look like?</u> its like a (W)**
> **put bottom frame to outer frame on front+rear of outer frames .**
> **<u>ok</u>use 1+12 bolts**
> **are your parts lab led by lettrs???**
> **<u>no</u> ok the thing looks like a cart with room for 2 trash cans the part that looks like this(XX)goes on the bottom +the 2(W)parts goon the sids**
> **put male ends? into female ends**
> **<u>what does that mean?</u> i dont no**
> **it looks like 3(U)s**
> **<u>what?</u> 2(U)s go into each other then theyare put on other u+put on W put top frame to front of outer fr.+to handlbar 2 1/4 bolts put center support fro inside topfr. use 2 1/4 boo thru center of top fro put 2 1/12 bolts thrub center of side fr., bottomfr. 2 bottom of center support fro**
> **<u>ok</u> put on wheels 3 spoks on outside put on hubcap with hammer put on handgrips DO ALL THESE STEPS FOR BOTH SIDES ok?????**

The above conversation is obviously between two inexperienced typists but it was found that inexperienced typists took only about 3% longer than experienced typists. This was because only about one third of the time was spent communicating but it does illustrate that little was lost because of typing errors. The above conversation illustrates this, not once is a spelling mistake, ambiguity or abbreviation questioned. The only mistake that may have been due to typing problems that was questioned was the omission of the oblique in "3/8" in the third line. The other person queried this and surprisingly the person confirmed the mistake and yet the conversation was continued without further discussion. In terms of a PIDGIN type of system a number of points arise:

i) Typing mistakes are common, such as missing spaces, misspelling of words, typing the wrong character.

ii) Spelling mistakes are common, this includes not only mistakes such as spelling "together" as "togteher" but also spelling "know" as "no".

iii) Punctuation is omitted or used to convey specific information. The only use of the full stop in the above example is to terminate an abbreviation and this is not done consistently.

iv) Abbreviations are invented and used as required. Thus one person abbreviates "frame" to "fr." without explanation.

v) The typewriter is fully utilised to convey information, for example the shape of the letters is used to convey pictorial information and although the first time this is done it is enclosed in brackets "(W)" this convention is not maintained consistently. An interesting combination of pictorial with linguistic information is the construction "3(U)s".

How can a computer system cope with these types of real conversational problem. They can be simplified to:

i) Any syntax driven system will not be suitable. English as it is used seems to avoid every rule at some time. Hopefully, a system, such as PIDGIN, which only uses syntax to guide it in its generation of deep structures will be able to cope with this.

ii) The system must cope with spelling mistakes, words run together, abbreviations, and words misused. This requires a much more careful analysis of the input at the character level making full use of conceptual information. It means that the itemisation of words cannot be separated from the complete analysis.

iii) Analysis must proceed largely from left to right as it cannot be assumed that sentences will be terminated in any way.

iv) Pictorial information must be capable of being picked out from within the English input.

All the above points show that conversational problem solving systems have a long way to go before they can cope with the full horrors of the real world.

So far no overall controlling program has been described at the English level corresponding to the PIDGIN driver at the PIDGIN level. This is because the role of such a program has intentionally been kept to a minimum so that as much as possible of the control is retained by the statements and rules in the memory. In this way the overall control remains flexible and capable of being altered and extended by the user. If the control program were incorporated at the ABC level it could not be altered by conversation at the English level. However, most of the detailed control is below the level that can be specified in PIDGIN (and thus at the English level). The way in which this is partially overcome is by means of special concepts that are recognised by certain commands (see Division 2.3B) and cause special action to be taken, for example:

SELF TRANSMIT A THOUGHT USER HERE.

will call the Analyser to read and translate a sentence and then bind the concept THOUGHT to that sentence. And:

SELF TRANSMIT A THOUGHT HERE USER.

will call the Synthesiser to translate the PIDGIN statement which is the reference of THOUGHT and then output it to the user. Both of these examples demonstrate how a built-in ABC program can be called from the PIDGIN level, hut obviously these programs are fixed and cannot be amended using PIDGIN. If this is compared with the way that people appear to handle the same problem then an interesting possible future extension to the system can be imagined. For example, consider how an adult learns a foreign language. Initially the vocabulary and syntax rules are learnt and are internally run through every time a sentence is constructed but slowly the process seems to become more automatic as if the rules were built-in and being used at a level below that of ratiocination. It is as if people can use a complex English description of what is required but as it is used it becomes translated into a lower level unverbalised but equivalent form. In PIDGIN terms it is as if a complex set of PIDGIN rules (whose evaluation corresponds to ratiocination) could" be translated into an equivalent ABC program. Further, people seem to be able to examine themselves carrying out these low level programs (for example, problem solving) and generate approximately equivalent English level programs that can be conveyed to others. It is interesting that these two abilities do not themselves seem to be capable of examination suggesting that there is an even-: lower level of program. The investigation of how these abilities might be defined in terms of PIDGIN and ABC would seem to be most fruitful.

CHAPTER 5 SUMMARY

PIDGIN is a programming language for developing conversational problem-solving systems. It has a syntax based on an extended and rationalized version of Schank's conceptualizations and a semantics derived from PLANNER and CONNIVER. A system developed using PIDGIN is thus better suited than a system based on Schank's static notation because the dynamic interpretation of PIDGIN has been fully incorporated into the language. It is also better than a system, such as that of Winograd, that uses PLANNER as its deep structure because PLANNER was designed as a programming language for people, not as a deep structure for natural language. The structure of PIDGIN enables assertions, rules and heuristics to be incorporated in the system to form a powerful deductive system. PIDGIN can also use 'this information to form schemes and plans necessary for the solution of complex problems. It can at the same time keep track of what it is doing, and as all the information is in the form of PIDGIN statements it can be used to answer questions about how and why it has carried out any particular action. It has also been shown how PIDGIN can be used to generate hypotheses from the knowledge in its memory and carryon a-conversation in which such hypotheses are checked with the user. The limitations of PIDGIN have been mentioned all through the thesis. The first and major requirement is an implementation of the complete PIDGIN system so that problems raised can be investigated and used to improve PIDGIN. The following list briefly describes some of the main areas that such a system could be used to investigate:

i) The translator briefly described in this thesis could be implemented in order to investigate the possibility of translating the type of real human communication described at the end of the last chapter.

ii) PIDGIN could be extended to investigate the type of belief structures described by Colby (1973) and Abelson (1973).

iii) The problem solving ability of PIDGIN could be extended by studying the ways in which a forward tree search and static evaluation function could be derived from the linguistic deep structure.

iv) PIDGIN's learning ability could be extended to cover a scheme involving regulated weights as well as an investigation of the more difficult problem of generating new pattern-matching rules.

v) The ability to generate extended discourse on a single topic could be studied working from a PIDGIN base. In this context a topic seems to be similar to the type of scheme generated during problem solving. Perhaps there is a close analogy between a conversation concerned with solving a problem and the problem solving process itself.

vi) There are many other types of deep structure that could profit ably be combined with the PIDGIN linguistic deep structure. For example, pattern Matching structures of the type described in connection with the "view", arithmetic and mathematical deep structures as simply exemplified by the quantifier expressions of PIDGIN, and deep structures corresponding to the notation of logic. The important thing with all of these is to find some smooth transition between them and the linguistic deep structure which should form the common link between them all (although a better but more difficult approach would be to regard them all as variations on some more basic structure, such as ABC).

vii) As was pointed out at the end of the last chapter the translation between PIDGIN and ABC is an important step in extending the system to a point where all further extensions to the system can be performed at the English level.

An investigation of the use of PIDGIN in a practical problem area would be a good way to bring together the diverse and vague requirements of such a system as well as forming a basis for further extensions. Such a problem area needs to be some area that provides simple, ill defined problems that could usefully benefit from being put into a rigid algorithmic framework.

APPENDICES

APPENDIX I IMPLEMENTATION

A. ABC Implementation

ABC (Associative Backtrack Computer) was designed and developed in order to simplify and rationalize the implementation of PIDGIN.

A brief description of ABC is given in Section 2.1.1 where it is clear that PIDGIN is an extension of ABC. Because ABC forms the core of PIDGIN and because ABC contains some features which are of interest in their own right this Appendix describes ABC in some detail. However, this Appendix is not a reference manual for ABC nor does it describe every feature in the language.

ABC is currently implemented in the programming language POP-2 (Burstall197l). It could be more efficiently implemented in a system programming language but it would take about a year to implement and would not be as flexible. The basic ABC system was implemented in a few weeks using POP-2. It should be noted, that even if implemented in an assembler language it would be very inefficient running on present day computers because it is a recursive hierarchy language (Stansfield 1972) not a simple recursive language and because it requires an associative memory in order to implement its data-base search efficiently. ABL (Associative Backtrack Language) is the assembler language of ABC. Its basic syntax is described in Section 2.1.1. In order to program in ABL a computer terminal is required that has at least the' twenty six letters of the alphabet, the ten digits and characters for the opening and closing application, band and class brackets, the orderer, the negator, the decimal point and sub-ten. This form of ABL is called Strict-ABL or SABL and it has a syntax which can be simply described as words, numbers and expressions enclosed in any of the three types of brackets (application, band and class) to form an expression. However, SABL is not a convenient language in which to program because of the large number of nested brackets usually required. Therefore, an extended version of SABL called Meta-ABL or MABL was developed. MABL requires extra characters to represent a terminator and combiner (described later) and it is designed to make us of any further characters that are available in order to allow expressions to be abbreviated.

1. The ABC Primitives

There is a special type of expression called an instruction, it is an application whose first expression is a primitive (one of 42 special concepts) and whose other expressions are called arguments. The 42 primitives currently defined are listed below with their required arguments and a brief description of what they do. In the description of the arguments "expr" stands for "expression", "aspect" is a concept being used in a special way, alternative arguments are enclosed in round brackets and optional arguments are followed by "=" and any default value :

1 ASPECT concept aspect=REF

returns the expression associated with the concept-aspect pair. Always succeeds.

2 ASSEMBLE

returns the SABL expression corresponding to the next MABL expression from the current input stream. Fails if an assembler error is detected.

3 ASSOC expr (concept band) aspect=REF

The expression is associated with the concept-aspect pair (or with every such pair if the second argument is a band of concepts). Always succeeds.

4 BAND expr (concept band) aspect=REF

like ASSOC except that the expression is added to the band which is currently associated with the concept-aspect pair. If the pair is not associated with a band then a band is created consisting of the first expression and the current association. Always succeeds.

5 BYE expr= [ABC RESTARTED]

saves the complete current system work space on a disc file and returns to the operating system level. When restarted the expression is printed. Always succeeds.

6 CLASS expr (concept band) aspect=REF

same as BAND but a class is formed.

7 CLEAR expr aspect=REF

every concept in the expression is associated with its concept aspect pair.

8 CLEARABC

re-initia1ises the ABC system and prints:

[ABC CLEARED.]

no aspect associations are altered.

9 COMBINE expr1 expr2, aspect=SUB

the two expressions are compared (see JOIN) and if this succeeds then if a sub-expression of expression1 is substitutab1e for the corresponding concept in expression2 then it is associated with the concept-aspect pair. Succeeds and fails as for JOIN. It is the basis of binding in PIDGIN.

10 DIV (number concept) (number concept)

the result is the first number divided by the second. If either argument is a concept then the reference of the concept is used. Will fail if either argument is not a number (or concept whose reference is a number).

11 DUMP level:::3 (number expr) =9999 outfile=CUOUT

prints details of the current associations (amount of detail is specified by level) either from the complete data-base (when number specifies the number of associations to be printed) or from the specified expression to a specified disc file (outfile) or to the current output file (CUOUT, see TO) if omitted.

12 ERROR expr number=100

equivalent to <RISE number expr FAIL> (see RUN and RISE).

13 FAIL expr=

evaluates the optional expression, if present, and fails.

14 FAILIF expr

evaluates the expression and fails if the expression succeeds, succeeds if the expression fails.

15 FREEZE (concept band) aspect=REF

fixes the current expression associated with the aspect of the concepts so that future attempts to alter the association are ignored (see THAW). Always succeeds.

16 FROM (infi1e band) outfi1e=

makes the current input stream the file specified by the concept infile. If infi1e is either USER or HERE then input is taken from the terminal. If the first argument is an ordered band then this is regarded as the current input stream. If the optional outfi1e argument is specified then everything read from the input stream is printed on the output file (see TO). The current input stream is made the reference of the concept CUIN.

17 GET (number concept) expr

the number (or reference of the concept) is used to index the expression. For example, if the number is n then the value is the nth sub-expression of the expression. The evaluation will fail if the first argument is not a number or is a number out of range.

18 GT (number concept) (number concept)

Succeeds if the first number (or reference of the concept) is greater than the second, otherwise it fails.

19 JOIN expr1 expr2= aspect=SUB

The first argument is called the question and the second the candidate. This primitive succeeds if the question matches the candidate using the specified aspect. The following are the rules of matching: if the question is the same as the candidate it succeeds. if the question, candidate or aspect is the concept

NIL then it fails.

if the question is one of the following concepts and the candidate the specified expression then it succeeds:

EXPR expression (always succeeds)

APPL application

BAND band (ordered or unordered)

CLASS class (ordered or unordered)

CHOICE band or class

ORDERED ordered band or class

CONCEPT concept

SYMBOL special character

GROUP concept whose reference is not ENTITY

ENTITY concept whose reference is ENTITY

FROZEN frozen concept (see FREEZE)

NUMBER number

INTEGER integer number

REAL non-integer number

PRIMITIVE concept which is one of the 42 primitives

ITEM concept or number

if the question is a concept then it succeeds if

    the aspect of the question matches the candidate

    using the TRANS aspect of the aspect.

if both are applications then it succeeds if each

    sub-expression of the question matches the corresponding

    sub-expression of the candidate.

if both are bands or both are classes then it

    succeeds if corresponding (if both are ordered)

    or any (if either or both are unordered)

    sub-expressions match.

if the question is a class then it succeeds if any

    sub-expression of the question succeeds.

if the candidate is a band or class then it succeeds if all (band) or at least one (class)

sub-expression matches the question.

if none of the above are true it fails.

If the candidate expression is omitted then this primitive will match the question with every expression currently associated with a concept-aspect pair.

20 LT (number concept) (number concept)

the inverse of GT

21 MAKE expr aspect=REF

the result is an expression which is the same as the first argument except that every concept is replaced by its aspect (unless it is frozen). Always succeeds.

22 MAX band

the result is the number or reference of a concept that is the largest in the band.

23 MIN band

inverse of MAX.

24 MONITOR outfile

sets up an output monitor stream to which all monitor information (see TRACE) will be written. Outfile is the output file name, or USER or HERE to have the information written to the terminal.

25 OK expr=

evaluates the optional expression, if present, and then succeeds.

26 POP concept aspect=REF

if the aspect of the concept is an ordered band or class then the first sub-expression is returned as value and the rest of the band or class is associated with the aspect of the concept. If the band or class is unordered then a pseudo-randomly, selected sub-expression is returned as value and the aspect is not altered. If the aspect is anything else then that is returned as value and the aspect is not altered. The primitive always succeeds.

27 PROCESS expr

The expression is evaluated and the primitive succeeds or fails as the expression succeeds or fails.

28 PROD band

the result is the product of all the numbers (which may be the . reference of a concept) in the band, it always succeeds.

29 QUOTE expr

the expression is returned as value. Always succeeds.

30 READ

reads and returns as value the next expression from the current input stream.

31 READABL

reads and returns as value the next MABL unit from, the current input stream (used by ASSEMBLE).

32 READS

reads a sequence of expressions from the current input stream terminated by the character ".", "!" or "?" and returns all the expressions as an ordered band.

33 REPEAT expr

repeatedly evaluates the expression until it fails, when the primitive succeeds.

34 RESUME

an instruction with this as primitive is created by the RUN and RISE primitives. It should not be used directly by the user as its arguments are related to the internal state of the ABC processor. If a RESUME instruction is evaluated then the system will return to the state it was in immediately after the instruction was created.

35 RISE leve1=1 expr=NIL (OK FAIL) =OK (concept band) =

For details of the theory behind RUN and RISE see the report by J Stansfield (1972). In some programming languages the facility these provide is called "backtrack programming". RISE saves the current state of the ABC system as a RESUME instruction and then associates this with the reference of the concept CONTINUE. It then returns to the specified level, or a higher level, and exits with the specified value (the second argument), either succeeding (if the third argument is OK) or failing (if FAIL).

The fourth argument specifies one or more concepts whose references are to be reset to their current value if the RESUME instruction is evaluated.

36 RUN expr level
evaluates the expression at the specified level. If an expression is evaluated at level n then if a RISE is evaluated at level m while evaluating that expression then it will exit from the RUN if m ~ n, otherwise the RISE will continue on past the RUN. That is, the higher the RUN level the more levels of RISE it will stop. In practice this usually means that the more "important" a program the higher will be the level at which it runs other programs. The ABC driver (see later) will stop all RISEs and so can be regarded as RUNning the user's program at an infinite level. User's errors (using the ERROR primitive) usually RISE at level 100 and ABC system errors at levels between 200 and 300 depending on the severity. Therefore, by running a program at level 300 the user will be able to trap all errors.

37 SIZE expr

its value is the number of top level sub-expressions in its argument. Always succeeds.

38 SUB (number concept) (number concept) its value is the second argument subtracted from the first. Always succeeds.

39 THAW (concept band) aspect=REF

unfreezes the expression currently associated with the aspect of the one or more concepts specified so that it may again be altered. Always succeeds.

40 TO outfile
alters the current output stream so that output is directed to the specified file. If outfile is USER or HERE then output will be directed to the terminal. The current output stream will be made the reference of the concept CUOUT. Always succeeds.

41 TRACE primitive

causes the specified primitive to print out a trace print whenever it is used. If TRACE is applied to the same primitive a second time the trace is switched off and so on. The details printed by the trace mechanism depend on the trace level. The trace level is an integer between zero and ten that is the reference of the concept TRACE. Trace printing varies between none at all (level zero) and all primitives evaluated with their arguments plus full details every time a RESUME instruction is created plus full details of all association changes, resetting of associations on failure, new concepts read, reordering of unordered bands and classes and machine utilisation statistics (level ten).

42 WRITE expr=

writes the expression to the current output file. If the expression is omitted then a newline character is written. Always succeeds.

As well as the above primitives there are a number of concepts whose reference is se t by the sys tem:

CONTINUE	the last RESUME instruction created by RISE.
ARGS	the arguments of the application currently being evaluated formed into an ordered band.
CUIN	the current input stream.
CUOUT	the current output stream.
VALUE	the result of the last evaluation.

2. The ABC Driver

When ABC is started it interacts with the user according to an algorithm called the driver. The driver first calls CLEARABC then prints the message :

[MABL STARTED.]

and then calls ASSEMBLE. If this fails because of an assembler error then :

[MABL ERROR.] value

is printed and the driver is restarted. Otherwise, the value of the assembly is processed by calling PROCESS. If the evaluation fails then:

[TOP FAIL.] level value

is printed and the driver restarted, otherwise the ASSEMBLE primitive is recalled. The above description of the driver corresponds to the following ABL program:

<ASSOC

```
        [<REPEAT
              [<CLEARABC> .
              <WRITE [MABL STARTED.]>.
              <REPEAT
              [( <ASSEMBLE><FAIL <WRITE [MABL ERRORJ>> )
              ( <PROCESS VALUE> <FAIL <WRITE [TOP FAIL.]>>)J>
   ]] DRIVER> .
```

When ABC is running it is continually re-ordering all the unordered bands and classes according to the results of the processing. The processor orders unordered bands so that the member that has failed most often in the past is processed first, the second most often next and so on. Classes are ordered so that the member processed first is that which has succeeded most often in the past. Further, when a band or class is searched for a matching member then it will also be automatically re-ordered to speed up future processing. This process of learning from the results of evaluating will improve the efficiency of ABC when it is simulated on a single processor computer assuming that there is a relationship between separate evaluations or searches of the same band or class.

3. The MABL Assembler

The MABL assembler language is described because all the following examples are written in MABL. It was developed in order to reduce the number of nested bracketed expressions required by Strict-ABL and so improve the clarity of programs written for ABC. The syntax of a MABL statement is the same as that of a SABL expression except that the following construction is also allowed:

op1 expr1 op2 expr2 , expr3 ... stopper

where op1 is an optional monadic operator, op2 is an optional dyadic operator and stopper is either the orderer (".") or the terminator (";"). If the dyadic operator and the monadic operator are present then the monadic operator is applied to expr1 and then the dyadic operator is applied to the result and expr2. The optional combiner (",") and its expression (expr3) may be followed by zero or more other combiners plus expression. The combiners are used to specify additional arguments for the dyadic operator, or if this is omitted for the monadic operator. The following are the initial operators :

```
" QUOTE          - ASSOC

! POP            * BAND
$ WRITE          + CLASS
/ ASPECT         = JOIN
^ RISE           ? COMBINE
\ RUN            : MAKE
```

The following examples make the syntax clearer:

```
MABL                    SABL

JOHN*NAME.              <BAND JOHN NAME>
$/NAME.                 <WRITE <ASPECT NAME>>
$/x/!y;                 <ASPECT <WRITE <ASPECT X>>
                                       <POP Y>>
MALE*PERSON,SUB.        <BAND MALE PERSON SUB>
4 ^ [NOT FOUND.] ,FAIL<RISE 4 [NOT FOUND] FAIL>
[<READ> -ITEM.          <ASSOC [ <ASSOC <READ> ITEM>
     /ITEM = !ARGS] -RD     <COMBINE <ASPECT ITEM>
                                <POP ARGS>>]
                                RD>
```

Note that the number of arguments taken by the operator is determined solely by the context of its use therefore the same operator symbol can be used in one place as a monadic operator and in another as a dyadic operator. A new operator symbol may be defined or an old one altered by associating the required primitive with the reference of the symbol. In fact the operators do not need to be symbols they can be any concept, including the primitive themselves. For example WRITE can be defined as an operator by:

WRITE-WRITE.

After this WRITE can be used as an operator:

MABL	**SABL**
WRITE [I KNOW.].	**<WRITE [I KNOW]>**

In MABL the ordered and unordered version of the band and class are distinguished not by different types of brackets (because of limitations in the character set of most terminals) but by the use of the orderer (".") symbol. If any sub-expression of a band or class is terminated by the orderer then that band or class is ordered otherwise it is unordered.

The escape symbol ("%") must precede any symbol or operator that is being mentioned to stop it being used.

B. The PIDGIN Implementation

PIDGIN is implemented in MABL (see last division) but some of the more basic PIDGIN routines have been rewritten in POP-2 and added to the ABC system as primitives in order to improve the efficiency of the PIDGIN system.

The following is a list of the major areas of the system that have been implemented: i) The complete ABC system, including the MABL assembler.

ii) The PIDGIN interpreter.

iii) The PIDGIN assembler (Input to Strict PIDGIN).

iv) The PIDGIN disassembler (Strict to Input PIDGIN).

v) The PIDGIN resolver (matcher, binder and deductive apparatus).

vi) The PIDGIN driver.

The following is a list of the major areas of the systems that have not, or have not fully, been implemented:

i) Translation between PIDGIN and English.

ii) The VIEW.

iii) Scheming and Planning.

The following MABL programs describe part of the PIDGIN system, although much of the logic shown is now implemented in POP-2:

```
[<REPEAT

      ([<READTHOUGHT> -THOUGHT.
              ([/THOUGHT? MEMORY/CONNECTOR. $ [I KNOW.]]
              [/THOUGHT * MEMORY, CONNECTOR. $ OK]) ]
      [<READABL>-INSTR. <INSTR> ]
              <ERROR [ILLEGAL PIDGIN STATEMENT.]>>>
      $ [CHAT TERMINATED.] ]-CHAT.

[<READACTOR> -SUBJECT.
<OK READMODIFIER> -MODIFIER. <<READACT>. "BE) -ACT.
<OK READACTOR> -OBJECT.
<OK READACTOR> -SOURCE. <OK READACTOR>-DESTINATION.
:" <ACT [SUBJECT OBJECT SOURCE .DESTINATION]
MODIFIER>*CONCEPTION.
[<OK READMODIFIER> -THMOD.
      <OK READCONNECTOR> -CONNECTOR.
      ([/CONNECTOR? /TERMINATORS. !CONCEPTION-THOUGHTJ
      [ <READTHOUGHT>
              ! CON CE PTI ON -CON.
              :" <CONNECTOR THMOD CON THOUGHT> -THOUGHT.]
      <ERROR [ILLEGAL PIDGIN THOUGHT. ]» ]
]-READTHOUGHT.
```

```
[%. %1] -TERMINATORS.

[MOVE PASS TRANSFER TRANSMIT] -TRANS ,SUB.

[COGITATE IDENTIFY PERCEIVE] - TROW,SUB

[BE BECOME DO TRANS TROW] -ACT,SUB.

[<READ> 1 ACT/SUB] -READACT.

[<OK READQUANTIFIER> -QUANTIFIER.

<OK READATTRIBUTE> -ATTRIBUTE.
    <<READNOMINAL> -NOMINAL.
        [<<FAILIF. /QUANTIFIER=UNDEF>.
            <FAILIF./ATTRIBUTE=UNDEF>>
        <ERROR [ILLEGAL PIDGIN NOMINAL]>] ).
    <REPEAT READSPECIFIER*SPECIFIER>.
    :" <NOMINAL QUANTIFIER ATTRIBUTE SPECIFIER>
] - READACTOR.

[<READ> =% [ .
    <REPEAT ( <READSPECIFIER> * SPEC. <READMOD> * MOD ».
    <<READ> = %J. <ERROR [MISSING %J.]>.)
    :"<MOD SPEC>
]-READMODIFIER.

[<READ> = % [.
<REPEAT [<READ> -ITEM <FAILIF ITEM= %]>.
ITEM*ATTRIBUTE]>. :ATTRIBUTE.
]-READATTRIBUTE.

[THE A AN SOME MOST EVERY ANY ALL] -QSPECIALS.

[( [<READ> = %<.
    <READ> 1 [MORE LESS EQUAL ABOUT]. VALUE-COMPARATOR.
    <READACTOR> -QUANT.
    <READ> = %>.]
    [<READ> = % =.
        ([ <READ> =NUMBER. VALUE-QUANT.
            EQUAL-COMPARATOR.]
        <ERROR [MISSING NUMBER]> )].
[<READ> ? /QSPECIALS. VALUE-QSPEC.
: [QSPEC COMPARATOR QUANT.] ? [[THE EQUAL 1.][A EQUAL 1.]
                              [AN EQUAL 1.]
                              [SOME MORE 0.]
                              [MOST MORE [DIV ALL 2.].]
                              [EVERY EQUAL ALL.]
                              [ANY EQUAL ALL.]
                              [ALL EQUAL ALL.]]]
[<READ> -NUMBER. VALUE-QUANT. EQUAL-COMPARATOR.])
:"<COMPARATOR QUANT>
]-READQUANTIFIER.

[<READ> = %<.
    ([ <READ> = %<.
        ([ <READ> -RELATION.
            <READ> 1 [MAX MIN MEAN]. VALUE-RELMOD.
            <READ> = %>.
```

```
                              : [RELATION RELMON.] -RELMOD.].
                              <ERROR [ILLEGAL RELATION MODIFIER.]>>]

[<READ> -RELATION.
<READACTOR> -ACTOR. <READ> = %>.])
:" <RELATION ACTOR>
]-READSPECIFIER.
```

APPENDIX II THE KNOWLEDGE BASE

A. Primary Knowledge

The primary knowledge consists of those PIDGIN statements read in at priority 100 that determine the allowed form of all future statements. There are a number of predefined concepts which can be substituted for a variety of structures, for example, any nominal can be substituted for by the concept THING, any PIDGIN statement by STATEMENT, any action by ACTION, state by STATE and so on.

```
ALL FORCE <SUB A THING>.

ALL OWNER <SUB A THING>.
ALL PLACE <SUB A THING>.
ALL OBJECT <SUB A THING>.
ALL BEING <SUB A THING>.
ALL THINKER <SUB A THING>.
ALL LIFE <SUB A BEING>.

ALL BODYPART <SUB AN OBJECT>.
ALL BODYP ART <PART A LIFE>.
ALL MIND <SUB A PLACE>.
ALL MIND <PART A THINKER>.
MEMORY <SUB A MIND>.
HERE <SUB A MIND>.
ALL USER <SUB A MIND>.
ALL VIEW <SUB A MIND>.
SELF <SUB A USER>.

ANY THING BE.
ANY THING BECOME SELF.
ANY BEING TROW ANY THING.
ANY THINKER COGITATE ANY THOUGHT.
ANY LIFE IDENTIFY ANY PATTEPN.
ANY LIFE PERCEIVE ANY OBJECT.
ANY LIFE PERCEIVE ANY OBJECT.
ANY LIFE DO ANY THOUGHT.
(ANY BEING ANY FORCE) TRANS ANY THING ANY THING2 ANY
THING3.
ANY LIFE MOVE ANY BODYPART ANY PLACE} ANY PLACE1.
ANY THINKER PASS ANY OBJECT ANY OWNER1 ANY OWNER2.
ANY FORCE TRANSFER ANY OBJECT ANY PLACE ANY PLACE .
ANY THINKER TRANSMIT ANY THOUGHT ANY MIND1 ANY MIND2.
ANY STATE SUGGEST ANY STATE.
ANY STATE ENABLE ANY ACTION.
ANY ACTION PRODUCE ANY STATE.
ANY ACTION CAUSE ANY ACTION.
ANY ACTION CAUSE ANY ACTION, THEREFORE ANY ACTION.
ANY ACTION THROUGH ANY ACTION.
ANY ACTION WHILE ANY ACTION.
```

ANY CONCEPTION IF ANY RULE.

B. General Knowledge

The following statements form part of the general knowledge base of a typical use of PIDGIN.

```
ALL ANIMAL <SUB AN OBJECT>.
ALL ANIMAL <SUB A FORCE>.
ALL ANIMAL <SUB A LIFE>.
ALL PERSON <SUB AN ANIMAL>.
ALL PERSON <SUB AN OWNER>.
ALL PERSON <SUB A THINKER>.
ALL MAN <SUB A PERSON>.
ALL WOMAN <SUB A PERSON>.
BILL <SUB A MAN>.
JOHN <SUB A MAN>.
MARY <SUB A WOI1AN>.
JILL <SUB A WOMAN>. ALL HAND <SUB A BODYPART>.
ALL FOOT <SUB A BODYPART>.
ALL SENSOR <SUB A MIND>.
ALL EYE <SUB A SENSOR>.
ALL EAR <SUB A SENSOR>. ALL DOG <SUB AN ANIMAL>.
ALL CAT. <SUB AN ANIMAL>.
ALL BOOK <SUB AN OBJECT>. ALL ALIVE <SUB AN ATTRIBUTE>.
ALL DEAD <SUB AN ATTRIBUTE>.
ALL HAPPY <SUB AN ATTRIBUTE>.
ALL SAD <SUB AN ATTRIBUTE>.
ALL COLOUR <SUB AN ATTRIBUTE>.
ALL RED <SUB A COLOUR>.
ALL BLUE <SUB A COLOUR>. ALL BIRTHDAY <SUB AN EVENT>.
ALL HOUSE <SUB AN OBJECT>.
ALL PARK <SUB A PLACE>.
ALL INCH <SUB AN OBJECT>.
ALL YEAR <SUB A TIME>. ALL BELONG <INVERSE A POSS>.
ALL HEIGHT <SUB A MEASURE>.
ALL AGE <SUB A MEASURE>.
ALL OLD <INVERSE AN AGE>. =N INCH <HEIGHT AN OBJECT>.
=N YEAR <OLD A LIFE>.
A LIFE <AGE =N YEAR>. ALL [COLOUR] OBJECT.
ALL [ALIVE] ANIMAL. ALL [DEAD] ANIMAL.
ALL [HAPPY] THINKER. ALL [SAD] THINKER.

<MULT M N> OBJECT1 <PART AN OBJECT3>

     IF =M OBJECT1 <PART AN OBJECT2>
     AND =N OBJECT2 <PART AN OBJECT >.

A PERSON <POSS AN OBJECT>
     ENABLE THE PERSON PASS THE OBJECT SELF A PERSONZ.

A PERSON PASS AN OBJECT SELF A PERSON
```

```
        PRODUCE THE PERSON <POSS THE OBJECT> [NOT]
                AND THE PERSON2 <POSS THE OBJECT>. #

A PERSON DO AN ACTION
        PRODUCE A [HAPPY PERSON2
SUGGEST
        THE PERSON DO AN ACTION
                PRODUCE THE [HAPPY] PERSON.

A PERSON DO AN ACTION PRODUCE A [HAPPY] PERSON
AND THE PERSON DO AN ACTION2 PRODUCE A [SAD] PERSON3
        SUGGEST
        THE PERSON2 DO AN ACTION3 PRODUCE A [SAD] PERSON3.
```

c. Specialist Knowledge

The level of knowledge referred to as specialist will be indicated by showing some of the PIDGIN statements necessary to set up the EIKASIA chess system described in Chapter 3.

Chapter 3 already shows a number of statements necessary for setting up EIKASIA and these all form part of the specialist knowledge concerned with playing the two kings, one pawn endgame. The following statements give some of the overall positive and negative states and a few simple rules for playing:

```
A WIN <SUB AN ATTRIBUTE <FEEL 100 GOOD».

A DRAW <SUB AN ATTRIBUTE <FEEL 0 GOOD».
A LOST <SUB AN ATTRIBUTE <FEEL -100 GOOD>>.

A [WIN] GAME IF A PAWN <ABOVE A7>.
A [DRAW] GAME IF WPAWN <LOC BOD.

AN [ADVANCE] PAWN ENABLE PAWNIDVE.
A [SELFBLK ROOKPAWN] PAWN ENABLE LETPASS.
A [ROOKPAWN] PAWN ENABLE WGOPAWN.
SELF IDENTIFY * <WKING WPAWN BKING>
        CAUSE PAWNMOVE .
SELF IDENTIFY * <<A SQUARE BKING>
                  </{ SQUARE WPAWN>
                  <WKING 2 A SQUARE »
        CAUSE SELF TRANSFER WKING A SQdARE4 A SQUARE3'

A[CANRUN]PAWN ENABLE RUN.
```

In the above few statements PAWNMOVE, LETPASS, WGOPAWN and RUN are labels (concepts whose reference is used) that have previously been defined as the appropriate actions. Next a few of the more complex predicates and actions are described:

i) CANRUN

This predicate is the implementation of the "rule of the square". This rule says that if the black king lies within a square formed with the pawn in the bottom corner farthest from the king then the king will be able to reach the pawn before the pawn can be queened. The rule assumes that white has the move and that if the pawn is on the second rank then it is assumed to be on the third rank.

```
A PAWN <WC 3 RANK> IF THE PAWN <LOC 2 RANK>. A [CANRUNJ
PAWN IF

THE PAWN <LOC =R RANK> <LOC =F FILE>
AND BKING <ABOVE =R RANK>
AND BKING <LEFT <SUB F <SUB 8 R»FILE
OR BKING <RIGHT <ADD F <SUB 8 R»FILE.
```

ii) NEEDSUP This predicate is the "rule of the square" for the white king, that is, it determines if the white king is in the promotion square of the white pawn and can thus reach the pawn before the pawn can reach rank 8. The predicate is the same as CANRUN with WKING substituted for BKING. For a description of the predicates ADVANCE, CAPTURE, SELFBLK and MATCH1 see Chapter 3.

iii) WGOPAWN

This action moves the white king to the square nearest to the white pawn:

SELF TRANSFER WKING A SQUARE1 A SQUARE2

<BETWEEN [THE SQUARE,_WPAWNJ>

iv) SUPPORT

This action moves the white king to support the pawn. This is done by moving the king to the square which is between them (WGOPAWN), if this places the king next to the pawn then the action fails (automatically restoring the original position of the king) and the pawn is advanced instead (PAWNSTEP). WGOPAWN AND A SQUARE <BE1~EN [WKING WPAWNJ~

OR PAWNSTEP: SUPPORT.

v) MANOEUV4

This action is one of the set manoeuvres, it moves the white king towards square G7:-

SELF TRANSFER WKING A SQUARE!

A SQUARE2 <BETWEEN [THE SQUARE! G7J>

: MANOEUV 4.

MANOEUV5 is the same as MANOEUV4 but with BKING substituted for WKING.

D. The Dictionary

The following are some typical dictionary entries. The verbs are given first as they form the foundation of the translation scheme. In order to form a valid PIDGIN statement from the lines below each should start with "$<DICT" and be terminated by">.". By convention the sentential subject is marked in the deep structure by the subscript 1, and any objects by subscripts 2 and 3. ADVISE TRANSITIVE

```
A PERSON TRANSMIT A THOUGHT SELF A PERSON2
BE TRANSITIVE
A THING1 <SUB A THING2>
BE TRANSITIVE
A THING1 <SUB AN ATTRIBUTE2>
BE TRANSITIVE
A THING1 <SUB A RELATION2>
BELIEVE STATE
A PERSONL [CAN] TRANSMIT A THOUGHT2 SELF MEMORY
BELIEVE TRANSITIVE
A PERSON TRANSMIT A THOUGHT SELF A PERSON
CAUSE2THE THOUGHT BECOME SELF<LOC[PERSON1]MEMORY>
BELIEVE TRANSITIVE
A PERSON2 TRANSMIT A THOUGHT SELF A PERSON1
CAUSE THE PERSON1 COGITATE THE THOUGHT
AND THE PERSON1 TRANSMIT THE THOUGHT
SELF MEMORY
COMFORT TRANSITIVE
A PERSON DO AN ACTION
CAUSE AN [UPSET] PERSON BECOME [COMFORTABLE] SELF
EAT TRANSITIVE
AN ANIMAL TRANSFER A FOOD2 A PLACE STOMACH
FILL TRANSITIVE
AN ANIMAL1 TRANSFER A LIQUID A PLACE A CONTAINER2
FLY INDIRECT
A FLYER] TRANSFER SELF A PLACE4 A PLACES
FLY TRANSITIVE
A PILOT DO A PLAN
CAUSE A PLANE2 TRANSFER SELF A PLACE4 A PLACES
FLY INDIRECT
A PASSENGER1 TRANSFER SELF A PLACE4 A PLACES
THROUGH A PLANE TRANSFER SELF A PLACE4 A PLACES
GET TRANSITIVE
A PERSONI PASS A THING2 A PERSON A PERSON!
GET TRANSITIVE
A PERSON1 [CAN] TRANSMIT A THOUGHT A PERSON2 SELF
GIVE DOUBLE
A PERSON1 PASS A THING3 SELF A PERSON2 GO INDIRECT
AN ANIMAL1 TRANSFER SELF A PLACE4 A PLACES
GROW STATE
A PERSON DO A PLAN
[INTEND] CAUSE A PLANT BECOME [BETTER] SELF
HAVE TRANSITIVE

A PERSON1 TRANSMIT A THOUGHT2 MEMORY SELF
HAVE TRANSITIVE
A PERSON4 PASS AN OBJECT2 SELF A PERSON1
HAVE TRANSITIVE
```

```
AN ANIMAL1 <HAS A SICKNESS2>
HIT TRANSITIVE
A PERSON1 [<DEGREE VIOLENT> ]TRANSFER AN OBJECT
SELF A PERSON2
KILL TRANSITIVE
A PERSON DO A PLAN
CAUSE AN ANIMALZ BECOME [DEAD] SELF
LEARN INDIRECT
A PERSONL TRANSMIT A THOUGHTZ SELF MEMORY
LIKE TRANSITIVE .
A PERSONZ DO AN ACTION
CAUSE A PERSON! BECOME [PLEASED] SELF
LOVE TRANSITIVE
A PERSON COGITATE A PERSON.
CAUSE A PERSON! BECOMEZ[LOVE] SELF
MOVE TRANSITIVE
A FORCE1 TRANSFER AN OBJECT2 A PLACE4 A PLACE1)
PONDER TRANSITIVE
A THINKER1 COGITATE A THOUGHT
PUNCH TRANSITIVE . .
. A PERSON1[ <DEGREE VIOLENT> ] MOVE A FIST SELF AN
OBJECT2
RECOGNISE TRANSITIVE
A PERSON1 IDENTIFY AN OBJECTZ
REMEMBER TRANSITIVE
A PERSON1 TRANSMIT A THOUGHTZ MEMORY SELF
SEE INTRANSITIVE
A PERSON1 [CAN] TRANSMIT A THOUGHT SELF MEMORY
SEE TRANSITIVE
AN ANIMAL1 PERCEIVE AN OBJECTZ
TELL TRANSITIVE
A PERSON1 TRANSMIT A THOUGHT SELF A PERSON2
THREATEN TRANSITIVE
A PERSON TRANSMIT
<A PERSON2 DO A THOUGHT CAUSE A PERSON1 DO
AN ACTION [INTEND] CAUSE THE PERSONZ
BECOME [HURT] SELF>
SELF A PERSON
THROW TRANSITIVE 2
AN ANIMAL TRANSFER AN OBJECTZ SELF A PLACE4
THROUGH THE ANIMAL! MOVE HAND A PLACES A PLACE6
WALK INDIRECT
AN ANIMAL TRANSFER SELF A PLACE4 A PLACES
THROUGH THE ANIMAL1 MOVE 2 FOOT A PLACE6 A PLACE]
WANT STATE
A LIFE DO AN ACTION2
CAUSE A PERSON1 BECOME [PLEASED] SELF
WANT TRANSITIVE
A PERSON PASS AN OBJECT A PERSON A PERSON
CAUSE A PERSON1 BECOME [PLEASED] SELP
WANT TRANSITIVE
A PERSON2 TRANSFER SELF A PLACE4 A PLACES<LOC A PERSON1>
CAUSE A PERSON1 BECOME [PLEASED] SELF
WONDER INDIRECT
A PERSON1 COGITATE A THOUGHT2

THE   DETERMINER       A THING
A     DETERMINER       A THING
SOME        DETERMINER        SOME THING
```

ALL	**DETERMINER**	**ALL THING**
PAWN	**NOUN**	**A PAWN**
KING	**NOUN**	**A KING**
TREE	**NOUN**	**A TREE**
ON	**PREPOSITION**	**A THING <ABOVE A THING2>**
BY	**PREPOSITION**	**A THING <LOC A THING2>**
WHITE	**ADJECTIVE**	**A [WHITE] THING**
HAPPY	**ADJECTIVE**	**A [HAPPY] PERSON**
WHERE	**QUESTOR**	**A THING <LOC A THING2>**
WHY	**QUESTOR**	**A THINKER COGITATE A SCHEME**
HOW	**QUESTOR**	**A THINKER COGITATE A PLAN**
WHEN	**QUESTOR**	**A THINKER DO A PLAN**

APPENDIX III BIBLIOGRAPHY

The following abbreviations are used in this bibliography:

ACM Association for Computing Machinery

AI Artificial Intelligence

CACM Communications of the ACM

DMIP Department of Machine Intelligence and Perception

IBM International Business Machines

IJCAI International Joint Conference on AI

JCC Joint Computer Conference HI Machine Intelligence

MIT Massachusetts Institute of Technology Proc. Proceedings.

The date given for books is the date of the first publication, this is not always the edition or publishers described.

Abelson, R.P. (1973): The Structure of Belief Systems Computer Models of Thought and Language. Schank and Colby, San Francisco: W.H. Freeman.

Ambler, A.P., Burstall, R.M. (1969): Question Answering and Syntax Analysis. DMIP Experimental programming Report No. 18.Edinburgh University.

Anderson, D.B. (1972): Lib Pico-Planner. POP-2 Program Library, School of AI, Edinburgh University.

Averbakh, Y. (1958): Chess Endings: Essential Knowledge Pergamon Press.

Becker, J.D. (1973): A Model for the Encoding of Experiential Information. Computer Models of Thought and Language. Schank and Colby, San Francisco: W.H. Freeman.

Bernstein, A. (1958): A Chess-playing Program for the IBM 704 Computer, Proc. of the Western JCC, pp 157-159.

Black, F. (1964): A Deductive Question-Answering System, Ph.D. Thesis Harvard University. Also in Semantic Information Processing, Minsky.

Bobrow, D.G. (1964): Natural Language Input for a Computer Problem Solving System. Semantic Information Processing, Minsky.

Bratley, P., Dakin, D.J. (1968): A Limited Dictionary for Syntactic Analysis, MI 2, Edinburgh University Press.

Burstall, R.M., Collins, J.S., Popplestone, R.J. (1971): Programming in POP-2, Edinburgh University Press.

Carroll, J.B. (1964): Language and Thought. Prentice Hall.

Chapanis, A. (1975): Interactive Human Communication, Scientific American, Vol. 226, No.4, pp 76-83.

Chomsky, N. (1968): Language and Mind. Harcourt, Brace and World.

Colby, K.M., Tesler, L., Enea, H. (1969a): Experiments with a Search Algorithm on the Data-base of a Human Belief Structure, Stanford AI Memo No. 94. Stanford University, California.

----Smith, D.C. (1969b): Dialogues Between Humans and an Artificial Belief System. Stanford AI Memo No. 97.

----(1973): Simulations of Belief Systems. Computer Models of Thought and Language, Schank and Colby, San Francisco: W.H. Freeman.

Coles, L.S. (1968): An On-line Question-Answering System with Natural Language and Pictorial Input. Proc. of the National ACM Conference, pp 157-167.

--- (1968): Syntax Directed Interpretation of Natural Language. Representation and Meaning, Simon and Siklossy, Prentice Hall.

Craig, J.A., Berezner, S.C., Carney, H.C., Longyear, C.R. (1966): DEACON: Direct English Access and Control. Proc. of the Fall JCC, pp 365-380.

Cresswell, M.J. (1973): Logics and Languages. Methuen and Co. Darlington, J. (1963): Translating Ordinary Language into Symbolic Logic. Project MAC, Memo MAC-M-149. MIT. Davies, D.J.M. (1971): POPLER: A POP-2 PLANNER. DMIP Report MIP-R-89, Edinburgh University. _Isard, S.D. (1972): Utterances as Programs, MI 7, Edinburgh University Press. Dewar, H., Bratley, P., Thorne, J.P. (1969): A Program for the Syntactic Analysis of English Sentences. CACM Vol. 12, No.8, pp 476-479.

Feigenbaum, E.A., Feldman, J. (1963): Computers and Thought, McGrawHill.

Fine, R. (1941): Basic Chess Endings, Philadelphia.

Fodor, J.A., Katz, J.J. (1964): The Structure of Language, Prentice Hall.

Friedberg, R.M. (1968): A Learning Machine: Part 1, IBM Journal, January 1968, pp 2-13.

Geschwind, N. (1973): Language and the Brain, Scientific American, Vol. 226, No.4, pp 76-83.

Goldman, N.M. (1975): Sentence Paraphrasing from a Conceptualase, CACM, Vol. 18, No.2, February 1975, pp 96-107.

Good, I.J. (1968): A Five Year Plan for Automatic Chess. MI 2. Edinburgh University Press.

Green, B.F., Wolf, A.K., Chomsky, C., Laugherty, K. (1961): Baseball: An Automatic Question-Answerer, Proc. of the Western JCC, May 1961. Also in Computers and Thought, Feigenbaum, McGraw Hill.

Green, C.C., Raphael, B. (1969): Research on Intelligent Question-Answering Systems. Proc. ACM 23rd National Conference, Princeton. Brandon systems.

Halliday, M.A.K. (1970): Functional Diversity in Language as Seen from a Consideration of Modality and Mood in English. Foundations of Language. Reidel, Vol. 6, pp 322-361.

Hebb, D.O. (1949): The Organisation of Behaviour: A Neuropsychological Theory. John Wiley and Sons, New York.

Hewitt, C. (1971): Procedural Embedding of Knowledge in PLANNER. Proc. of the Second IJCAI, September 1971, pp 167-182.

---(1972): Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot. MIT AI Laboratory, Ph.D. Thesis.

Huberman, B. (1968): A Program to Play Chess Endgames. Stanford AI Memo No. 65

Hunt, E. (1973): The Memory We Must Have, Computer Models of Thought and Language, Schank and Colby, San Francisco: W.H. Freeman.

Jinich, A. (1971): Some Suggestions for a Question-Answering System Using Types. DMIP Diploma thesis, Edinburgh University.

Lamb, S.M. (1966): Outline of Stratificational Grammar, Georgetown University Press.

Landin, P.J. (1965): A Generalisation of Jumps and Labels. Univac Systems Programming Research Report.

Levien, R.E., Maron, M.E. (1967): A Computer System for Inference Execution and Data Retrieval. CACM, Vol. 10. No. 11, November 1967, pp 715-721.

Lighthill, J. (1973): Artificial Intelligence: A Paper Symposium. Science Research Council.

Lindsay, R.K. (1963): Inferential Memory as the Basis of Machines Which Understand Natural Language. Computers and Thought, Feigenbaum, McGraw Hill.

Longuet-Higgins, H.C. (1972): The Algorithmic Description of Natural Language. Proc. of the Royal Society 182, pp 255-276.

McCarthy, J. (1959): Programs with Common Sense. Proc. of the Symposium on the Mechanisation of Thought Processes. H.M.S.O.

----Hayes, P. (1973): Some Philosophical Problems from the Standpoint of Artificial Intelligence. Stanford AI Memo. Also in MI 4,1969, pp 463-502, Edinburgh University Press.

Michie, D. Machine Intelligence 1 - 8, Edinburgh University Press.

----(1972): Programmer's Gambit, New Scientist, August 1972, pp 329-332.

Minsky, M. (1968): Semantic Information Processing, MIT Press.

----(1968): Matter, Mind and Models, Semantic Information Processing, Minsky. Montague, R. (1969): On the Nature of Certain Philosophical Entities. The Monist. Vol. 35, pp 159-194.

----(1970a): English as a Formal Language. Linguaggi nella Societa e nella Technica Milan, pp 189-224.

----(1970b): Pragmatics and Intensional Logic. Synthese 22 pp 68-94.

----(1970c): Universal Grammar. Theoria (a Swedish Journal of Philosophy), Vol. 36, pp 373-397.

----(1973): The Proper Treatment of Quantification in Ordinary English. Approaches to Natural Language, Hintikka, Reidel.

Mott, D.H. (1973): A Computer Program that Learns to Model Its Environment by Experimentation and Communication via Natural Language, unpublished research paper.

Muir, J. (1972): A Modern Approach to English Grammar: An Introduction to Systemic Grammar, Batsford.

Newell, A., Shaw, J.C., Simon, H.A. (1957): Empirical Explorations of the Logic Theory Machine: A Case Study in Heuristic. Western Computer Proc. 1957, pp 218-230.

----Shaw, J.C., Simon, H.A. (1958): Chess-playing Programs and the Problems of Complexity, IBM Journal of Research and Development, vol. 2, No.4, pp 320-335. Also in Computers and Thought, Feigenbaum.

----Shaw, J.C., Simon, H.A. (196la): Report on a General Problem-Solving Program. Engineering Summer Conferences.

----Simon H.A. (196lb): GPS, a Program that Simulates :Human Thought, Lernende Automaten, Billing, H. Munich pp 109-124. Also in Computers and Thought, Feigenbaum.

----et.al. (1973): Speech Understanding Systems, North Holland! American Elsevier.

Ogden, C.K. (1933): Basic English: An Introduction with Rules and Grammar. 4th Edition, London: Kegan Paul, Trench, Trubner and Co.

----(1968): Basic English International Second Language, Harcourt, Brace and World. Piaget, J. (1929): The Child's Conception of the World, Routledge & Kegan Paul.

Phillips, A.V. (1960): A Question-Answering Machine, MIT AI Memo No. 16.

Polya, G. (1945): How to Solve It, Princeton University Press.

Pople, H.E. (1973): A Goal-Oriented Language for the Computer, Representation and Meaning, Simon, Prentice Hall.

Quillian, M.R. (1968): Semantic Memory, Semantic Information Processing, Minsky.

----(1969): The Teachable Language Comprehender, CACM Vol. 12, No.8, August 1969, pp 459-475.

Quine, W.V.O. (1960): Word and Object. MIT Press.

Raphael, B. (1968): SIR: A Computer Program for Semantic Information Retrieval, Semantic Information Processing, Minsky.

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 210-229. Also in Computers and Thought, Feigenbaum.

Schank, R.C. (1968): A Notion of Linguistic Concept, A Prelude to Mechanical Translation, Stanford AI Memo No. 75, Stanford University, California.

---- Tesler, L.G. (1969a): A Conceptual Parser for Natural Language, Stanford AI Memo No. 76.

----(1969b): A Conceptual Dependency Representation for a Computer Oriented Semantics. PhD. Thesis, University of Texas. Also Stanford AI Memo No. 83.

----Tesler, L. ,Weber, S. (l970a): Spinoza II: Conceptual Case-based Natural Language Analysis, Stanford AI Memo No. 109.

----(1970b): "Semantics" in Conceptual Analysis, Stanford AI Memo No. 122.

----(1971): Finding the Conceptual Content and Intention in an Utterance in Natural Language Conversation. Proc. of the Second IJCAI, September 1971.

----(1972): Conceptual Dependency, Cognitive Psychology, I Vol. 3, No.4.

----(1973a): The Fourteen Primitive Actions and Their Inferences, Stanford AI Memo No. 183.

----(1973b) Identifications of Conceptualisations Underlying Natural Language, Computer Models of Thought and Language. Schank and Colby, San Francisco: W.R. Freeman. ----Goldman, N., Rieger, C., Riesbeck, C. (1973c): MARGIE: Memory, Analysis, Response Generation and Inferences on English. Proc. of the Third ICAI.

Schwarcz R.M., Burger, J.F., Simmons, R.F. (1970): A Deductive Question-Answerer for Natural Language Inference. CACM, Vol. 13, No.3, pp 167-173.

Shannon, C.E. (1950): Programming a Computer for Playing Chess, Philosophical Magazine, Vol. 7, No. 41, pp 256-275.

Siklossy, L. Simon, H.A. (1972a): Some Semantic Methods for Language Processing, Representation and Meaning, Simon, Prentice Hall.

----(1972b): Natural Language Learning by Computer, Representation and Meaning., Simon, Prentice Hall. Simon, H.A.,

Siklossy, L. (1972a)} Representation and Meaning: Experiments with Information Processing Systems. Prentice Hall.

----(1972b): On Reasoning About Actions, Representation and Meaning, Prentice Hall.

Simmons, R.F. (1962): Answering English Questions by Computer: A Survey, CACM, Vol. 8, pp 53-70.

----(1969): Natural Language Question-Answering Systems: 1969, CACM, Vol. 13, No.1, pp 15-30.

----(1972): Generating English Discourse, CACM, Vol. 15, No. 10.

Slagle, J. (1965): Experiments with a Deductive Question-Answering Program, CACM, Vol. 8, pp 792-798.

Slobin, D.I. (1971): Psycholinguistics, Scott, Foresman & Co. Spencer-Brown, G. (1969): Laws of Form, Lowe and Brydone.

Stansfield, J.L. (1972): PROCESS 1: A Generalisation of Recursive Programming Languages, Bionics Research Report No.8, School of AI, Edinburgh University.

Sussman G.J., Winograd, T. (1970): Micro-Planner Reference Manual, MIT AI Memo No. 203.

. ----McDermott, D.V. (1972): Why Conniving is Better Than Planning MIT AI. Memo No. 255A

Tan, S.T. (1972): Representation of Knowledge for Very Simple Pawn Endings in Chess, School of AI Memo MIP-R-98, Edinburgh University.

Thompson, F.B. (1966): English for the Computer. Proc. AFIPS 1966 Fall JCC Vol 29, Spartan Books, New York.

Thorne, J., Bratley, P., Dewar, H. (1968): The Syntactic Analysis of English by Machine, MI 3, Michie, Edinburgh University Press.

Turing, A.M. (1950): Computing Machinery and Intelligence, Mind, October, 1950, Vol. 59, pp 433-460.

Vigor, D.B., Urquhart, D., Wilkinson, A. (1969): PROSE - Parsing Recogniser Outputting Sentences in English, MI 4, Michie, Edinburgh University Press.

Vygotsky, L.S. (1962): Thought and Language, MIT Press.

Weizenbaum, J. (1966): ELIZA - A Computer Program for the Study of Natural Language Communications Between Man and Machine, CACM, Vol. 9, No.1, pp 36-45.

----(1967): Contextual Understanding by Computer, CACM, Vol. 10, No.8, pp 474-480.

Winograd, T. (1969): PRO GRAMMAR: A Language for Writing Grammars, MIT AI Memo No. 181.

----(1971): Procedures as a Representation for Data in a Computer Program for Understanding Natural Language, MIT Ph.D. Thesis.

----(1973): A Procedural Model of. Language Understanding, Computer Models of Thought and Language, Schank and Colby, W.H. Freeman.

Wittgenstein, L. (1922): Tractatus Logico-Phi1osophicus, Routledge & Kegan Paul.

Woods, W.A. (1967): Semantics for Question-Answering Systems, Aiken Computation Laboratory Report, Harvard University.

----(1968): Procedural Semantics for a Question-Answerer Machine, Fall JCC 1968.

----(1970): Transition Network Grammars for Natural Language Analysis, CACM, Vol. 13, No. 10, pp 591-603.

Zobrist, A.L.,Car1son, F.R. (1973): An Advice-Taking Chess Computer, Scientific American, Vol. 228, No.6, pp 92-105.