C O B O L L A N G U A G E

I M P L E M E N T A T I O N

V I A E N G L I S H

by

O w e n D. K e s s e l l

A thesis presented in partial fulfilment

of the requirement for the degree of

Master of Science in Computer Science

at

Massey University

December 1976.

ABSTRACT

In this thesis, synthesis of COBOL programs is discussed. The programs are generated with the aid of an interactive Natural Language dialogue.

Reasons for and against the use of English as a general programming aid are discussed, also the use of English in program synthesis is discussed.

The major portion of this thesis describes the design of the system known as CLIVE. The discussion illustrates the relative ease in which COBOL programs can be generated by using ordinary English responses.

ACKNOWLEDGEMENTS

In presenting this thesis I would like to take this opportunity to express my thanks to the following people:

First and foremost to my supervisors, Brian Carpenter and Lindsay Groves, whose continual guidance and criticizm has been most helpful.

To the various members of the Computer Science Department and Computer Unit at Massey University who, through assorted discussions, assisted in the gathering of information contained in this thesis; especially to Neil James, for his assistance with details regarding the implementation of CLIVE.

Finally to my wife Margaret, for her continual understanding and patience.

Massey University — Owen D. Kessell

December 1976

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1.

INTRODUCTION.

"Where shall I begin, please your Majesty?" he asked. "Begin at the beginning", the King said gravely, "and go on till you come to the end: then stop."

Lewis Carroll.
"Through the Looking Glass".

CLIVE, which is an acronym for COBOL Language Implementation Via English, is a system that generates COBOL source programs from a Natural Language discourse.
CLIVE is used in a interactive manner; the users responses forming the basis upon which the desired program is generated.
Chapter Two discusses the justification for the use of Natural Language as an input medium. It is not the authors intention to advocate using English as a programming language but to advocate the use of it as a programming aid.
Chapter Three discusses the approach taken towards program synthesis. A description of a sample dialogue is given to illustrate the types of responses that the user and CLIVE may give, and the chapter concludes with a description of CLIVE's overall construction.

Chapters Four, Five and Six discuss the major components of CLIVE: How they generate the appropriate section of COBOL code and how the user may respond to convey his information.

Chapter Seven discusses briefly some modules that aren't strictly neccessary but are added to improve CLIVE's performance.

Chapter Eight is a brief conclusion about CLIVE and the overall philosophy behind it.

The appendices contain a sample session and results, using the current implementation of CLIVE which was done on the B 6700 at Massey University.

CHAPTER 2.

The Reasons For Such A System.

"What ever sceptic could
inquire for;
for every why he had a
wherefore."

Samuel Butler.
"Hubridas", Pt 1.

2.1 Marginal and Economic Reasons.

Why should we want a system that will convert Natural Language input into a COBOL program, or indeed into any programming language? The computer fraternity already know how to write computer programs using the various programming languages that are available, so it might appear to be a spurious exercise to design a system that would generate COBOL source code by some other method. Whilst this is possibly an understandable criticizm to make, there are strong reasons as to why such types of systems should be considered.

If Natural Language systems are not to be considered, then to make use of a computer, a user would be required to know a programming language. However the use of the computer is becoming very wide spread amongst most sectors of the community and so the above mentioned restriction would mean that all of those new users would need to learn a programming language.

There are over 120 known programming languages (Sammet 1969) that have been used or are still in use. Obviously some of the languages are similar, but the majority were written for specific purposes; purposes that the "common" languages could not adaquetely fulfill. For example, a complex problem that made extensive use of strings could be programmed in LISP or SNOBOL without a great deal of difficulty. However to solve the same problem in a non-string characteristic language, like FORTRAN, would be extremely difficult.

The new user would choose the particular programming language that could best be applied to the task that was to be solved. However, the majority of these new users generally only require the computer to perform well and are usually not interested in the mechanics of the programs that produce the results. That is, the new user wants to utilize the computer in the simplest possible way.

The cost of computer hardware has been falling at quite a sharp rate for several years (Mc Cusker 1976), and a state now exists in which the cost of producing a consistent amount of programs is increasing whilst the cost of the computer hardware is decreasing.

If computers could be programmed with the aid of Natural Language, the traditional programmer could be bypassed, thus alleviating a long standing bottleneck and consequently allowing more people access.

2.2 Natural Reasons.

Apart from the economic practicalities, what could be more natural than to use our own language as an aid in program development? At the present time, when a program has to be written, the problem is firstly formulated in our mind. There, the decisions as to what is required to solve the problem are made. Then the problem of implementation occurs as the problem must be translated from our own language into a machine language equivalent. Quite often this implementation (program writing) is tedious and can result in lots of "bugs".
If we could omit part of, or all of the translation step, then the task of programming would become very much easier. Whilst there would still be a problem with self expression, once a suitable system was implemented, the user would be able to develop programs in the way that he conceives them, i.e. in a natural fashion.

2.3 Error Reduction.

Groves (Groves 1975) has classified errors in programs into two categories.

1) Errors that occur before or during execution: i.e. syntax, semantic and system errors.

2) Errors in the output, after execution of the program.

The program generated from a Natural Language input system would generally be free of errors of type 1. This is because when the system is designed and implemented, the implementer

actually defines the source code that will be used in the final generated program.

Errors of type 2 are usually due to faulty logic. With an interactive system the frequency of this type of error occuring can be lessened. Before the program is compiled, the user can check the code that has been generated and make any alterations that may be neccessary.

Thus programs that are generated from Natural Language input systems will usually be terminating algorithms, but whether the correct solution can be obtained regularly will depend upon how well the user/computer interface works.

2.4 Conclusion.

As computer technology grows, computers will become smaller, faster and cheaper. As a consequence, it is very likely that more and more people will turn to the computer for use as a tool and for pleasure.

By using Natural Language input systems, the computer user will be able to "converse" much more easily with the computer than he can at present.

CHAPTER 3.

OVERVIEW.

"When you are describing
a shape or sound or hint,
don't state the matter plainly
but put it in a tint
and learn to look at all things
with a sort of mental squint".

Lewis Carroll.
"Poeta Fit
non Nascitur".

3.1 Program Synthesis.

At the present time, when a computer is needed to solve a problem, it is neccessary for a programmer to write the required program. With a system that synthesizes programs the task that faces the problem solver is made much easier. The user presents his problem to the synthesizer and a program is produced. Hopefully it is the correct one.
Manna (Manna 1970) states that in his opinion

> "A program synthesizer is a system that takes a relational description and tries to produce a program that is guaranteed to satisfy the relationship, and therefore does not require debugging or verification."

Much of the current work done on program synthesis follows along the lines of work done by Manna and Waldinger (Manna 1975). This is the thereom proving approach using predicate

calculus. Manna has restricted his research to the area that deals with recursive and iterative programs that operate on natural numbers, lists and trees.

Little work has been done in the field of program synthesis using Natural Language as an aid. CLIVE attempts to synthesize a COBOL program with the aid of a Natural Language interactive dialogue.

3.2 COBOL Programs.

A COBOL program consists of four divisions: Identification Division, Environment Division, Data Division and Procedure Division.

The Identification Division and the Environment Division are minor in that their basic function is to 'label' the program. The other two divisions, the Data Division and the Procedure Division form the major part of a COBOL program, and they can both be divided up into smaller, logical sections.

The Data Division can be divided into sections such that each section is a description of a single data item. These single sections, when grouped together, can form file descriptions or record descriptions. The file or record descriptions, when grouped together can form major sub-sections of the Data Division. These major sub-sections jointly make up the Data Division of a COBOL program.

The Procedure Division can be similarly broken down into smaller sub-sections. The smallest single component being

a simple statement or command. Single statements can be grouped together to form paragraphs; paragraphs may perform more complex functions than those of single statements (e.g. tests, loops, etc).
The paragraphs can be grouped together into 'larger' paragraphs which together form the Procedure Division of a COBOL program.

The problem of synthesizing a COBOL program is made easier because of the natural sub-division of a COBOL programs components. If the synthesis is carried out in a top-down fashion, the top goal (program synthesis) can be subdivided into lesser goals (construction of the 4 Divisions). These goals can now be subdivided (e.g. construction of file descriptions, paragraphs, etc). This subdivision of goals can continue until the lowest goals are reached (e.g. construction of single statements, single data definitions, etc).
If this process of program synthesis is considered as a tree of goals, then once the lowest set of goals is reached, a steady progression up the tree will result in the final goal being reached.
This approach to program synthesis is ideally suited to COBOL.

3.3 Question and Answer Dialogue.

Throughout the use of CLIVE there is a continuous dialogue occuring between the user and CLIVE. At the beginning of the session CLIVE will ask the user for his or her name. This name is then included into some of CLIVE's responses.

e.g. "NOW OWEN, DO YOU WANT A TITLE?"

There is no danger of CLIVE appearing patronizing as the users name is inserted at a random rate into CLIVE's remarks, as is more likely between the conversation of two people.

There are four general categories into which a users response may fit.

1) A "yes" or "no" type response.

2) A textual or variable name input response.

3) A "not sure" response.

4) An "alterations are requested" response.

Type 1.

When a "yes" or "no" response is required, a heuristic is used to ascertain what the users reply is.
At the simplest level the heuristic will look for a single YES or a single NO; at a more complex level the heuristic will handle such answers as

"WELL I SUPPOSE I MAY AS WELL".

"I GUESS SO".

Type 2.

When a text is typed in it always has a positional variable associated with it. This variable indicates where the text is to appear on the output listing.

For variable names, only common applicable words are deleted (the, and, etc). If the user types in a variable name THE STOCK ON HAND, CLIVE will assume that the variable required is 'STOCK-ON-HAND'. If the user placed a hyphen between the words THE and STOCK, CLIVE would then call the variable 'THE-STOCK-ON-HAND'.

Type 3.

A "not sure" response could be

"I'M NOT SURE WHAT YOU MEAN"

or

"I DON'T KNOW".

CLIVE handles these types of responses by simply rephrasing the question that it asked.
CLIVE has a group of basic questions.

e.g. "DO YOU REQUIRE ANY HEADINGS?"

and if a "Don't know" response occurs, then CLIVE will elaborate slightly.

e.g. "DO YOU WANT ANY HEADINGS PRINTED OUT WITH YOUR OUTPUT LISTING?"

It could perhaps be suggested that the latter type of question should be used all of the time, but such bulky "explanation" questions, in the authors opinion, should be avoided where possible.

Type 4.

The responses for alterations usually occur near the conclusion of the session, prior to CLIVE executing the program it has generated. The format of these responses is reasonably unrestricted and is described below.

The reader may at this point wonder how CLIVE can distinguish between the different types of responses. When CLIVE asks a question, it expects an answer to the question and it expects a relevant answer.
Because CLIVE knows what type of response it wants (it has just asked the question) it can therefore assume details about the input that the user supplies. If the input follows a expected pattern, then CLIVE will know that it is a correct response and it can then take suitable action to understand it.
If the input does not fit the expected pattern, CLIVE will assume that it is a "I DON'T KNOW" response and will respond accordingly.
The user can at any time ask CLIVE about the assumptions that it has made. This is a special case of the input and with the keyword assumption contained in the input, is easily understood.

Obviously there is no real limit upon the type of response a user can make, but there must be a practical limit upon the number of responses that CLIVE can recognize and understand. CLIVE is designed to be used by Data Processing orientated people, so the number of types of responses users

are liable to make is reduced; as compared to a system that was designed to handle completely general responses.

Once the program is completed, it is compiled and executed. The user now has several options open to him.

1) He can end the session, as his job is completed.

2) He can get CLIVE to change the input/output options. (He may want a listing on the lineprinter, or he may want future runs to be from cards, disk, etc. CLIVE will recompile the new program, execute it if neccessary and/or save it for future use.)

3) He can get CLIVE to solve another problem.

4) He can make additions to the current program.

Option 1 and 2 are quite simple, the user can type in something like

GOODBYE CLIVE

or

CHANGE REMOTE TO CARDS, AND SAVE IT.

The user is restricted in that the sentences need to contain certain words:GOODBYE, or CHANGE or SAVE, etc.
This restriction makes it easier for CLIVE to understand what the user wants.

Option 3 means that the user wants to start another session with CLIVE. CLIVE will ask the user, at the completion of the program, whether he has any more problems to solve, if

the answer is affirmative, then CLIVE will initiate another session.

Option 4 involves two choices.

a) The user may want to make some minor alterations to the generated program. As the program is segmented and well documented as to the function of that segment, the user should be able to understand to some degree, what that segment of code does.
If the user is not satisfied with the code produced he can indicate to CLIVE what the change is.

e.g. I WANT YOU TO CHANGE LINE 100 FROM KOUNT=0
TO KOUNT=1.

b) If the output is incorrect, the user must work with CLIVE to try and correct the mistake. (By use of option 3 or 4a).

3.4 A Sample Dialogue.

An example of the type of dialogue that could arise out of a typical session with CLIVE is described below. (U indicates the users responses and C indicates CLIVEs responses).

C. HELLO, WELCOME TO THIS SESSION. MY NAME IS CLIVE, WHAT'S YOURS?

U. MY NAME IS OWEN.

C. OKAY OWEN, IF YOU'LL TYPE IN YOUR PROBLEM I'LL SEE IF I CAN SOLVE IT FOR YOU.

U. I WANT A LISTING OF ALL MY MARRIED MALE EMPLOYEES WHO HAVE A WEEKLY INCOME OF LESS THAN $100.

C. NOW FIRST OF ALL WE'LL CONSIDER THE INPUT.
WHAT IS GOING TO BE THE INPUT?

U. WHAT DO YOU MEAN?

C. WHAT ARE THE NAMES OF THE DATA ITEMS THAT YOU ARE GOING TO USE FOR THE INPUT?

U. THEY ARE THE EMPLOYEES NAME, THE HOURS WORKED, HOURLY RATE, SEX, MARRIAGE STATUS AND THEIR AGE.

C. IS THE DATA FOR THIS PROBLEM ALREADY ON A TAPE OR DISK FILE?

U. YES, IT'S ON A DISK FILE.

C. WHAT IS THE NAME OF THIS DISK FILE OWEN?

U. IT'S AK/3908/PERS/DATA.

C. OKAY OWEN I'VE FOUND IT.

NOW WE COME TO THE OUTPUT.

DO YOU WANT A TITLE?

U. OKAY.

C. WHAT IS THE TEXT OF THE TITLE?

U. THE NUMBER OF POOR MEN.

C. AND WHEREABOUTS ON THE PAGE DO YOU WANT IT POSITIONED?

U. 20 SPACES IN FROM THE MARGIN.

C. WHAT ABOUT HEADINGS, DO YOU WANT ANY?

U. YES.

C. TYPE THEM IN.

U. NAME SEX MAR-STAT INCOME.

C. OWEN, STARTING FROM THE MARGIN OF THE PAGE, PLEASE INDICATE WHERE THE HEADINGS ARE TO BE POSITIONED.

U. 5, 13, 4, 5.

C. I HAVE ASSUMED THAT YOUR HEADINGS ARE SYNONYMS FOR THE VARIABLES THAT YOU WANT OUTPUT.

DOES MAR-STAT MEAN MARRIAGE-STATUS?

U. YES IT DOES.

C. OKAY OWEN THAT SEEMS TO BE ABOUT ALL I NEED TO KNOW TO GENERATE THE PROGRAM: ANY PROBLEMS SO FAR?

U. WHAT ASSUMPTIONS HAVE YOU MADE?

C. I HAVE MADE THE FOLLOWING ASSUMPTIONS.

C. **OUTPUT-FILE**

NAME : EMPLOYEES-NAME.

SEX : SEX EQUAL TO 0 IS A MALE

SEX EQUAL TO 1 IS A FEMALE.

MAR-STAT : MARRIAGE-STATUS : 0 IS FOR SINGLE

1 IS FOR MARRIED.

INCOME : WEEKLY-SALARY.

PROCEDURE DIVISION

WEEKLY-SALARY = HOURS-WORKED * HOURLY-RATE.

C. DO YOU AGRRE?

U. YES, IT SEEMS OKAY.

C. DO YOU WISH TO VIEW THE GENERATED PROGRAM?

U. NO THANKS.

C. OKAY THEN OWEN, I'M COMPILING THE PROGRAM NOW.

C. THE OUTPUT I'VE GOT IS AS FOLLOWS.

THE NUMBER OF POOR MEN

NAME	SEX	MAR-STAT	INCOME
SMITH	0	1	$98.95
JONESY	0	1	$99.98
YOUNG	0	1	$6.09

C. DO YOU AGRRE WITH THE RESULTS?

U. THEY LOOK GOOD TO ME.

C. OWEN, DO YOU HAVE ANY FURTHER PROBLEMS?

U. NO, NOT AT THE MOMENT.

C. IF YOU WANT THIS PROGRAM SAVED, I'LL SAVE IT WITH A NAME OF OWEN/38K/POOR/MEN.

U. OKAY YOU CAN SAVE IT.

C. GOODBYE OWEN.

3.5 Structure of the system.

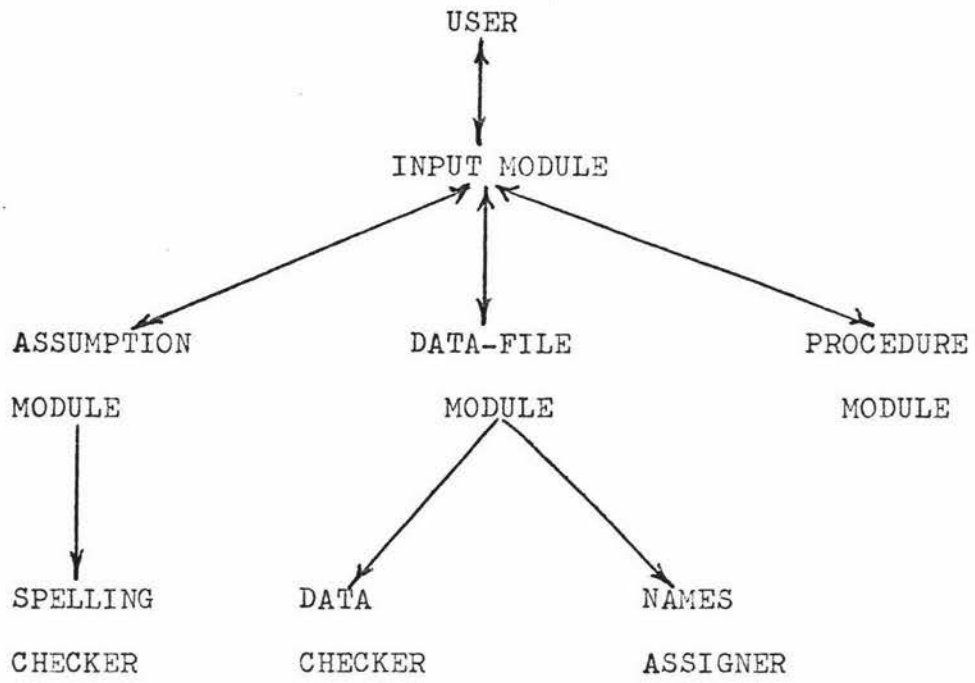

Figure 1: Structured outline of CLIVE.

CLIVE can effectively be broken up into three major modules. The Input Module, the Data-file module and the Procedure Module.

The Input Module handles all of the input that the user supplies. The main task of this module is to analyse the problem that the user wishes to be solved.

The Data-file Module makes up the Identification Division, the Environment Division and the Data Division.

The Procedure Module makes up the neccessary Procedure Division statements that will be needed to solve the problem. Information for the construction of the Procedure Division initially will come from the question that the user asks.

The other modules in figure 1 are ancillary ones and are discussed in chapter 7.

3.6 Readability.

When writing a computer program, it is usually a good idea to try and make the program as "readable" as is possible. That is, a programmer should try to use identifiers, labels, etc, so that their names give a guide as to what they do. The programmer should make extensive use of comments throughout the program to explain what different parts of the program do. The following example tries to illustrate this point.

Two segments of a COBOL program are given, both do the same thing; however in the authors opinion, one segment is much easier to understand than the other.

Segment 1.

```
...
...
A.
    IF T IS LESS THAN 41.0 GO TO B.
    COMPUTE OP = (T - 40.0) * OR.
B.
...
...
```

Segment 2.

```
...
*   THIS PARAGRAPH CHECKS TO SEE IF THERE IS ANY
*   OVERTIME PAY, IF THERE IS SOME, THEN THE
*   AMOUNT IS WORKED OUT.
*
 OVERTIME-PAY-SECTION.
     IF TOTAL-HOURS-WORKED IS LESS THAN 41.0
     GO TO NO-OVERTIME-WORKED.
*
*   OVERTIME PAY IS TO BE CALCULATED
*
 COMPUTE OVERTIME-PAY = (TOTAL-HOURS-WORKED - 40.0)
                        * OVERTIME-RATE.
*
 NO-OVERTIME-WORKED.
...
```

One of the aims of CLIVE is to produce source code similar to that of segment 2, i.e. well named identifiers and well documented code.

Whilst CLIVE may be used by people who have little programming knowledge, this factor does not neccessarily mean that they will not be interested in viewing the final source code that is generated. If the final code is well documented and well defined as to the functions of each section; then it is possible that the user will gain some insight into the programming logic involved.

3.7 Other COBOL Source Code Generators.

The COBOL programming language is a widely used one. It is not surprising therefore that some attempts have been made to help a programmer in the development of a COBOL program. These "assistants" have usually been in the form of macro-preprocessors. No attempt at Natural Language input is considered by these macros and their input format is quite restrictive. The aim of these processors is really to aid the COBOL programmer as opposed to aiding the non COBOL programer.

Two such systems are breifly described.

3.7.1 SURGE.

SURGE (Peterson 1976) is the name of a COBOL pre-processor. The main capabilities of which are file sorting, selective retrieval and tabular report preparation. SURGE has a system of notation which is intended to simplify and speed up the users retrieval of information by requiring only that the user describes what he wants. From this description, the SURGE pre-processor generates a program, built from pre-tested program logic. SURGE however is not intended to be used for file maintenance nor can it be used to detect invalid data; that is the users responsibility.

To use SURGE, certain steps must be taken.

1) Using SURGE notation and SURGE coding forms, the user describes what he wants.

2) These specifications are then key-punched into machine readable specification cards.

3) These cards are then read by the SURGE pre-processor.

4) A COBOL program is produced.

The notation used on the coding forms is reasonably complex, different types of entries are identified by a special code. However, according to Peterson, once the notation has been learnt, the user can quickly get a program written, thus SURGE could be used as a short cut to generate skeletal COBOL programs.

3.7.2 Computer Aided Program Production.

C.A.P. (Hamilton 1973) generates COBOL programs by making use of the following.

1) A small set of logic skeletons (COBOL macros).

2) A system whereby record and file descriptions can be coded up as macros.

3) A system allowing decision tables to be expressed as processing requirements.

To date, four logic skeletons have been produced. They are for validation, serial file updating, analysing serial files, and generating sorts with COBOL own-coding.
The following is an example of a file updating program.

```
*PROGRAM.  FRED SMITH.
*REMARKS. THIS IS AN UPDATING PROGRAM.
```

```
*COMPUTER.
*LEVEL-MERGE-2 WITH 3 AS NUMBER OF LEVELS
                    KEY SIZES 3,5,5
*STOCKFILE  POINTMASTER.
*STOCKCHANGES  POINTUPDATE.
*STOCKFILE.
*PRFILE  CHANGELIST  "SC-XSTCKLIST"
         AS STATIONERY 55 AS LINES.
*W-S
...
   working storage, provided by the user
...
*PROC.
```

There are also 23 modules which the user must provide, i.e. ensure are available upon disk.
The asterisked figures represent macros.

e.g. *COMPUTER will generate the Environment Division.
*LEVEL-MERGE-2 calls in a two file merge logic skeleton.
*PRFILE defines a print file, with 55 lines per page.
etc.

The user must know what each macro consists of, and how the input is formatted before he can make use of this system.

3.8 CLIVE Re-visited.

The two above systems have different qualities. SURGE allows the user to define Procedure Division elements, i.e. calculations, etc; but to use SURGE the user must learn the SURGE notation.
The other system described, C.A.P. has no facilities for more complex procedures, but it has a more presentable format.

CLIVE will be able to do much more than these two systems can, and in a much easier way. The user can use English input, there are no macros to learn about, in fact no programming knowledge is needed to make effective use of CLIVE.

CHAPTER 4.

The Input Module.

"The challenge of programming a computer to use language is really the challenge of producing intelligence."

T. Winograd, 1971.

4.1 Introduction.

Chapter four explains how the question that is input to CLIVE is handled. The chapter begins with a discussion of some of the problems that are involved with the use of Natural Language in programming. Then a discussion of some of the techniques that have been used in Natural Language understanding systems follows. This discussion is included because in the authors opinion, by discussing the different methodologies of these systems, the reader will perhaps be able to see more clearly why the author chose the approach that he did.

The main part of this chapter will explain how CLIVE handles the original problem that is input. This in an important part of CLIVE because the success or failure of the system is dependant upon how well CLIVE can handle the question that the user asks.

4.2 Ambiguity.

The English language abounds with ambiguities and many examples of these have appeared from time to time to illustrate why Natural Language is not a good enough medium to be used as a programming aid. Hill(Hill 1972) quotes as an example:

> "During the present fuel shortage, please take advantage of your secretary between the hours of 12 and 2."

This is clearly an ambiguous sentence, but is it likely that anyone would want to make a statement like that to a computer?

The sentence,

> "John shot the girl with long hair",

is also ambiguous, however most people are capable of seeing through the ambiguity because they have some extra knowledge; they have more information about the sentence than that which the sentence explicitly contains.

We know from past experience, that the word "shot" implies that some sort of weapon was used, we also know that long hair is not normally used as a projectile; hence it is quite easy for us to conclude that the girl had long hair and that she was shot by John. The ability of human beings to clarify these types of ambiguities is dependant upon what we have learnt about the world around us. The task of "teaching" a computer about the world around it is quite a tremendous one and at the present time, impracticable.

Ambiguities can arise in specific areas, and therefore are

understood by only a specific amount of people.
The part of a sentence,

"... after a photo, Lord Jim came first."

is quite meaningful to a horse racing fan; Lord Jim is the name of a race horse and a photograph of the finish of the race was taken to decide who was the winner.
However a person with no horse racing knowledge may arrive at a misleading conclusion.

Bearing the problem of ambiguities in mind, it would not be unreasonable to pre-define the class of English that might be used in a interactive dialogue: i.e. have a restricted subject area. If the dialogue was going to involve some discussion about race horses, then the above type of sentence is liable to be encountered, so provision could be made to ensure that the required meaning was obtained.
Therefore the risk of getting ambiguities could be lessened by assuming that certain subjects would not be encountered, whilst also assuming that other subjects would be encountered: Thereby making a system that is to be used for a specific application.

4.3 Natural Language Understanding Systems.

Most research into Natural Language systems appears to be divided into two broad categories.

1) formal methodology: with a general outlook or goal.

2) informal methodology: with a problem specific goal.

The latter category usually has more of a practical quality

about it, whereas the former category tends to be rather theoretical.

The informal approach generally is aimed at the specific user who has a specific problem to solve and generally works quite satisfactorily. Winograd (Winograd 1975) has suggested that a problem specific system that made use of informal methods could perhaps be a first step in the creation of a "General-All Purpose" problem solver.

Systems like STUDENT (Bobrow 1968), SIR (Raphael 1968), ELIZA (Weizenbaum 1966), PARRY (Colby 1974) all use some sort of pattern matching to obtain the relevant information from a sentence. These systems do not do any syntactical analysis of the input as they make the assumption that only the information that fits their "patterns" is of importance to them. This approach has been very successful and because the input is to some extent pre-defined (i.e. STUDENT has algebra type problems, PARRY has a doctor-patient interview) special heuristics can be added to improve the system and to help it in "sticky" areas.

The major drawback with these type of systems is that they are not flexible enough to be used outside of the scope for which they are intended. They are too specifically orientated towards a particular task; even HELP (Shapiro 1971), which is derived from Weizenbaums ELIZA has a limited repertoire, but adaptations like HELP show the versatility of the specific approach.

Systems using English text as a basis for storing information as opposed to using patterns to store the information have

had a limited amount of success. These systems, such as PROTOSYNTHEX (Simmons et al 1966), Semantic Memory (Quillian 1968) are generally known as "Text Based" systems (Winograd 1971), and they use some type of indexing scheme to store groups of text. In PROTOSYNTHEX, an English sentence is interpreted as a request to retrieve a relevant sentence or group of sentences from the text.
Quillian, with his Semantic Memory used a slightly different technique. He stored a processed version of some English dictionary definitions in which multiple meaning words were eliminated by having the user indicate which was the correct interpretation. The responses to these types of systems often depended upon the exact way that the text and questions were stated rather than any reliance upon the meaning of the question.

Green (Green 1968) felt that the problems of Natural Language understanding might be solved by the use of predicate calculus. If a question could be expressed in some form of mathematical notation, then by expressing the problem as a thereom to be proved, the "thereom prover" would actually deduce the information neccessary to answer the question.
A drawback with this approach is that first-order logic is a declaritive rather than an imperitive language and as a consequence, the task of formulating a problem is somewhat arduous. Green (Green 1968) has had reasonable success though by improving his system with the addition of special heuristics.

The use of grammars to obtain and aid in the understanding of Natural Language has been boosted by the work of Woods. Woods (Woods 1970) makes use of an augmented transition net-

work grammar, the power of which is that they can make

> "changes in the contents of a set of registers associated with the network, and whose transitions can be conditional on the contents of those registers." (Woods 1970).

A disadvantage with ATNG's, as with other similar parsers, is that in general they cannot reliably operate in a real-time situation. As ATNG's examine every word in a sentence, every word must convey some meaning and this word-by-word analysis can become cumbersome. There appears to be no neglecting mechanism with these parsers and so in a real-time situation they can prove too fragile to work satisfactorily.

Winograd has designed a system SHRDLU (Winograd 1971) which uses a "systemic grammar" instead of an ATNG. However Winograd states (Winograd 1971) that his "systemic grammar" is virtually equivalent to an ATNG.

SHRDLU consists of various procedual blocks, the important ones, from a Natural Language understanding viewpoint, are PLANNER, PROGRAMMAR, and GRAMMAR.

GRAMMAR is the main coordinator of the language understanding process. It consists of a few large programs written in PROGRAMMAR to handle the basic units of the English language. PLANNER is the deductive system used by the program at all stages of the analysis, both to direct the parsing process and to deduce facts about the block world in which SHRDLU operates. The system uses Micro-Planner, an abbreviation of Carl Hewitts original PLANNER language (Hewitt 1971). A very powerful feature of PLANNER is that it need not start a problem

from "scratch" but can make use of knowledge stored from previous problems and thus start the problem solving process at a much higher level than other systems can.
SHRDLU is capable of answering questions about itself, it can make decisions and is capable of remembering and understanding the contents of conversations.

4.4 The Specific Approach.

Two systems that make efficient use of the specific approach are now described.

4.4.1 LUNAR.

Woods (Woods 1973) has implemented a system called LUNAR which was designed to retrieve geological information about lunar rock samples. LUNAR permits a scientist to ask questions about the chemical composition of rock samples in straight forward Natural English.
Therefore any geologist can use the system successfully as he needs no knowledge about programming to obtain his answers. However it was found that when a non-geologist (in fact a Psychology student) used the system, LUNAR failed to respond in the correct fashion. This failure highlights the fact that LUNAR was designed for a specific purpose and consequently to operate successfully would need to be used by a specific type of person; namely a geologist.

4.4.2 GENIE.

Berkeley (Berkeley et al 1973) have implemented a program synthesizer called GENIE. GENIE will produce a BASIC or FORTRAN program for a specific problem from a Natural Language type of input.

Berkeleys reasoning for his system was that ...

"If an ordinary clerk can understand an ordinary manager, then a computer program ought to be able to do as well."

GENIE relies upon the user asking specific questions and the way it works is as follows.

Firstly, the user tells Berkeley exactly what problems are to be solved and exactly how the problems are going to be presented. Secondly, a suite of programs (known collectively as GENIE) are written and these are given to the user.
In essence, GENIE is a set of macros that have a Natural Language front end. Each "version" of GENIE is designed for a specific task, a sort of specific specific problem solver.

GENIE will work quite satisfactorily as long as the correct type of question is asked, however the type of problems that can be solved are restricted to simple routine calculations.

e.g. (Berkeley 1973)

"Take the rate of interest in coloumn 4 of the register, express it as a decimal to three decimal places, and put it in column 5 of the worksheet."

In fact the commands given to GENIE are virtually identical to the commands a manager might give to his clerk.

4.5 The Problem To Be Solved.

What is required from the question that the user inputs is the logic of the problem. Sometimes the logic will be very trivial and sometimes the logic will be complex.
There are basically two possible situations that can arise.

1) Trivial logic in the question.

   e.g. ...PAYROLL CHEQUES FOR ALL OF MY EMPLOYEES.

   The question contains no explicit arithmetic operations and thus questions of this nature can be handled by pre-defined logic elements. This would be done by the Data-file module (Chapter 5).

2) Programming logic is contained explicitly in the question.

   e.g. ...STOCK WITH A REORDER QUANTITY OF MORE THAN ...
   ...INCOME OF BETWEEN $100 AND $200...

   In this case the arithmetic logic must be understood and therefore the extraction of this logic forms a major feature of the Input module.

As can be seen from the two above examples, sometimes the question will contain relevant information (i.e. programming logic) and other times it may contain comparatively little. In either case, not all of the input is required. Because formal methods of Natural language understanding would be far too cumbersome and as the problems to be solved are specifically orientated ones, it was decided that a less formal approach would yeild the best results.

The method of Natural Language understanding used in CLIVE is similar to that used by Colby in PARRY (Colby 1974). The Natural Language input to PARRY is transformed until a pattern is obtained which matches a stored pattern. This pattern match may be a complete one or only a partial one but Colby has had adequete success with this approach.
The transformation of the input occurs in four steps:

1) Certain words in the question are recognized and converted into internal synonyms.

2) The input is broken into segments at certain words.

3) Each segment is matched (independantly) with a stored pattern.

4) The resulting group of recognized segments is matched with a set of stored complex patterns.

Each of the steps, except the segmenting, throws away what it cannot understand.

CLIVE adopts a very similar approach to that of PARRY.
The input to CLIVE is first segmented at certain words. This segmenting infact checks to see if there is more than one question involved in the input.
Then suffixes, prefixes and common words are removed and idioms and synonyms are converted into internal synonyms.
Once the input is in this state, the Procedure module then can produce the required Procedure Division logic.

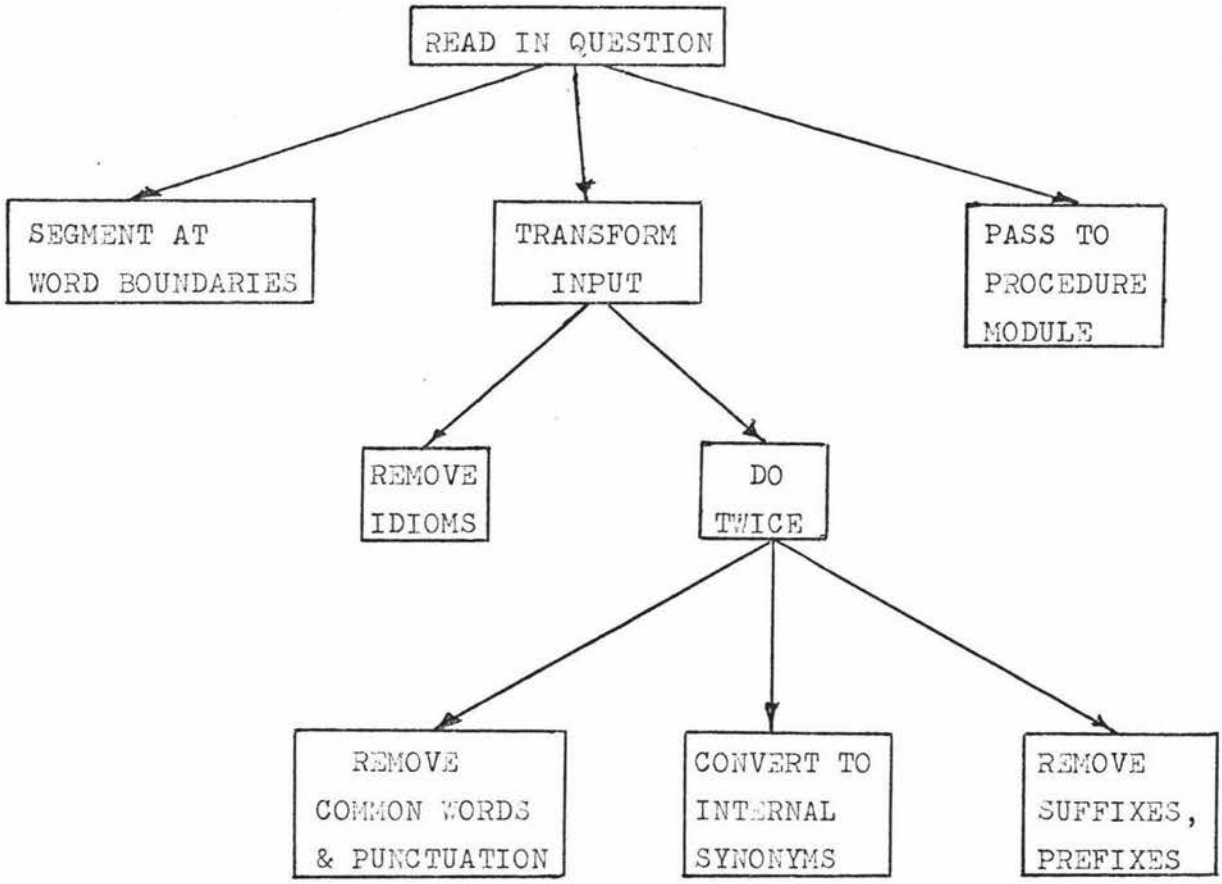

FIGURE 2: Structure Diagram Of The Input Analysis.

4.5.1 Question Boundaries.

The problem that is going to be input will generally take the form of one or more sentences. One sentence can contain two different questions, so when CLIVE receives the input, it segments the question into single problems.
Punctuation delimiters are easily used to aid in the segmenting, e.g. : , ; . ? etc, but the main criteria for segmenting hinges upon keywords and the association they have with other words around them.

An example:
A simple question like

"I WANT A LISTING OF ALL EMPLOYEES WHO EARN OVER $300 PER WEEK AND ARE OVER 30 YEARS OLD."

differs little from a question like

"I WANT A LISTING OF ALL EMPLOYEES WHO EARN OVER $300 PER WEEK AND ALL THOSE WHO ARE OVER 30 YEARS OLD."

In the first question, there is only one problem; the AND being a boolean operator.

i.e. (PAY GTR 300 AND AGE GTR 30).

The second question however consists of two problems. The addition of the words ALL THOSE WHO changes the meaning of this question from that of the first question. The AND in the second question is simply a conjunction and thus serves only as a delimiter.

The dividing function CLIVE uses is actually a set of heuristics. The heuristics look for words that are normally

used as conjunctions and then examines the surrounding words. Boolean operators are ignored and if the word or words do act as a delimiter, then a special character is inserted into the question in place of the delimiting word or words.

An example of some of the delimiting words that CLIVE looks for follows.

... AND OF ...

... AND HOW ...

... AND THEN ...

... AND AFTER ...

... I THEN ...

... THEN ...

If the user delimits the problems himself, that is by the use of commas, full stops, etc; then the heuristics for the dividing function have a relatively easy job.

The heuristics must also be aware that whilst some punctuation still acts as a delimiter, it is not the sort of delimiter that segments questions or problems.

e.g. ... EARN OVER $50, BUT LESS THAN $200 ...

(In terms of correct English usuage, the comma in this instance is optional.)

A point to note here is that the segmenting must be the first operation that is done upon the input. If the common words were removed from the input first, then the dividing function would not work.

If we consider the two examples above, we see that both sentences are nearly identical but the second sentence has the extra words ALL THOSE WHO. A common word deleter is likely to remove these words, thus both sentences would appear to be equivalent when in fact they are not.

4.5.2 Idioms.

The types of idioms CLIVE is likely to come across fall into the redundant information category. Most of the problems that are input to CLIVE will start off with an introductory clause.

For example "I WANT TO KNOW ..."
"I WISH TO FIND ..."
"COULD YOU WORK OUT ..."
"WOULD YOU TELL ME ..."

As the user wants CLIVE to solve a problem or problems, it is only natural for the user to actually ask CLIVE to solve it. Knowing this fact it is logical to consider these idioms redundant and remove them from the sentence. Similarly polite phrases are also removed from the input.

E.g. "PLEASE TELL ME ..."
"THANK YOU ..."
"... CLIVE ..."

Each segment is scanned for these phrases, usually they only appear in the first segment but they may appear in others

as well.

A small table of common idioms is used to further reduce the input.

For example, date of birth is replaced by birth-date.

However the major function of the idiom procedure is to remove all the polite and redundant non-information bearing words.

4.5.3 Common Words and Punctuation.

According to Meadows (Meadows 1970) there are approximately 70 different words in conventional English that will normally acount for about half of the text in any given sentence or sentences. These words are generally known as noise words and they carry no information in themselves but they make the entire sentence appear more attractive and flowing. The symbols that identify objects are generally referred to as key-words and a hierarchical organisation of key-words should be able to identify the key data relationships regardless of the types of questions that may be asked.

This is because the English sentence usually has a regular nested structure and so it is possible for key-words to be grouped with their near bye associates with regard to their context sensitivity. The removal of non-meaningful words can facilitate this end.

CLIVE has a table of common words that are in fact a subset of all the common words that exist. CLIVE's common words are related to Data Processing concepts and to the way that people would express them. The segment of text that was

input is scanned and all common words are deleted from that segment.

Some of CLIVE's common words are as follows:

SOME, THE, THESE, IN, WHAT, WAS, A,
IT, WOULD, YOU.

As most punctuation in the input is used to segment the input, there usually is little punctuation left, however a common punctuation symbol that is left is the dollar sign ($).
When this symbol is found in the input it is deleted and CLIVE makes a "note" of the fact that it appeared, and where it appeared. This "note" serves as a heuristic aid in the development of the Procedure Division of the COBOL program.
For example, if a sentence contains:

" ... EARNS OVER $300 ... "

then CLIVE will make the assumption that the word EARNS can be replaced by the word PAY, which in turn may be replaced by the word WEEKLY-SALARY which in turn may be obtained from the calculation of HOURS-WORKED * HOURLY-RATE and so on.
Of course this assumption would only be made if the word WEEK came after the $300, (as oppossed to YEAR or MONTH etc).

4.5.4 Synonyms.

Synonyms and near synonyms are stored in a dictionary and each segment is scanned for these words. Any synonyms found are then converted to a root or common word.

e.g. TYPE IN, READ, INPUT, PUT IN

can all be replaced by the word INPUT.

The word TYPE can be used in two ways, the meaning of which is dependant upon the words surrounding it.

e.g. TYPE IN or TYPE OUT

In the sentence

I WANT YOU TO TYPE THE RESULTS FOR ME,

the correct meaning for the word TYPE can be obtained from a previous word, YOU.

Similarly, sentences such as:

"... WHO EARN OVER $300 PER WEEK."

"... WHOSE SALARY IS MORE THAN $300 A WEEK."

"... WHOSE INCOME IS GREATER THAN 300 DOLLARS PER WEEK."

"... WHO RECEIVES MORE THAN 300 DOLLARS PER WEEK."

can be replaced by:

"...PAY OVER 300."

Where EARN, INCOME, SALARY, RECEIVES (with a $ sign following) are all considered to be synonyms for PAY,
and where GREATER THAN, MORE THAN are considered to be synonyms for OVER.
X DOLLARS becomes $X which in turn sends a "note" to CLIVE and becomes X.

The importance of synonyms is easily seen from the above examples as they reduce the amount of dictionary storage that is required to solve a particular problem. As the input to CLIVE will be of a predictable nature, the table of synonyms

need only have those entries that are likely to be applicable.

4.5.5 Suffixes And Prefixes.

As CLIVE uses key-words to obtain the meaning from a sentence, it is important that CLIVE gets all the key-words. The word LIST may be replaced by the word OUTPUT; however the words LISTS and LISTINGS are not recognized by the synonym replacer. Therefore all suffixes and prefixes must be removed. Obviously not all words that end with an "s" can be shortened. e.g. HAS to HA.
So CLIVE has a dictionary of important words, (key-words are a subset of this list) and the input words are compared against the entries in the list. If a match is nearly correct, (e.g. LIST and LISTS), then the input word is altered accordingly.
The "near correctness" of a match is only true when the difference between the words is either a suffix or a prefix.

E.g. a-, con-, sub-, anti-

or

-ing, -ed, -s, -ent, etc.

The Input module must examine the input segments several times and in a cyclic fashion (in fact it must be done twice).
The reason for this is as follows.
If a word in the input segment contains a common word that has a suffix or prefix added to it, then when the common word deleter is used, the word will not match any of the common words in the common word dictionary. The suffix and prefix remover will then remove the suffix or prefix. The input

string now has a common word contained in it, which must be removed. Hence the need for two complete checks by the Input module.

4.5.6 Analysis Finished.

By this time the input has been segmented into separate questions and each segment has been reduced to a set of recognizable (to CLIVE) key-words. These segments are then passed to the Procedure module which will make up the neccessary source code that is required to answer the questions.

CHAPTER 5.

THE DATA-FILE MODULE.

"Hark, the numbers soft and
clear,
gently steal upon the ear."

Pope.
"Ode on St. Cecilias
Day".

5.1 Introduction.

This chapter describes the Data-file module. The Data-file module makes up the Identification Division, the Environment Division and also the Data Division.
The Identification Division and the Environment Division of a COBOL program are relatively minor. They are usually small and easily constructed because there is a limited amount of information that can be contained in them.
The Data Division is used to define the data that will be used throughout the program, the size of this division being dependant upon the amount of data that has to be used.
All data that is used in any programming language must be suitably defined. That is, what type of data it is and how much storage location the data will need.
In COBOL formats are defined by the use of the word PICTURE. How CLIVE obtains these data formats is explained at the end of the chapter.

5.2 The Identification Division.

The Identification Division of a COBOL program is used to identify the program. During the generation of the program, CLIVE uses this division to write information about the program. The users name is obtained, the time and date are also obtained (hardware functions) and all are written in the appropriate position of the division. CLIVE also writes a brief description about the program, in fact a precis of the question that the user inputs. Thus the generated program is well documented.

When the program has been compiled and executed, the user may wish to have the code file saved for future use. CLIVE will tell the user that it has stored the code file under some variable name; e.g. FREDS/PAYROLL/33, and for the user (or anyone) to reference this code file later, the appropriate file name must be used.

CLIVE will automatically place this variable name into the Identification Division at the beginning of the program generation. This feature ensures that all of CLIVE's prodgeny are identified.

5.3 The Environment Division.

The Environment Division of a COBOL program is used to assign peripheral devices to the files that the COBOL program will be using and also to reference the computer that the program is to be run on.

CLIVE will ask the user if any disk or tape files are going to be used. If the files are for input, their name(s) are

obtained; if the files are for output, then the user may supply a name, otherwise CLIVE will supply one.

In most sessions with CLIVE, the user will want the input and output to appear on the remote terminal he is at; once a program has been generated (interactively), the user may, for example, tell CLIVE that in future, all input files will come from a card reader and that all output files will be sent to a line printer. CLIVE will alter the peripheral device(s) of the program, recompile the program and then store it for future use.
So a user can develop a program, and then use it later in a conventional batch manner. Obviously any suitable combination of I/O devices can be used, thus allowing a wide range of operations to be done.

5.4 The Data Division.

The Data Division of a COBOL program is used to describe the data that is going to be input, manipulated, created or produced as output. The Data Division consists of several sections.

The FILE-SECTION is perhaps the most important section of the Data Division; it is certainly the most commonly used section. The file section describes the format of the files that the program will be using, e.g. input and output files.

The WORKING-STORAGE-SECTION describes the structure of the records that need temporary storage during the course of the programs execution.

The CONSTANT-SECTION contains literals and figuratives that are to be assigned a specific value.

e.g.
```
77   ONE    PICTURE 9 COMP  VALUE 1.
01   STAR   PICTURE X  VALUE "*".
```

The LOCAL-STORAGE-SECTION is used to describe parameters when externally compiled procedures are to be bound into the main source program.

The REPORT-SECTION is used by the report writer: a special purpose subset of the COBOL language which is used to produce reports.

All of the above sections are optional. Generally almost all COBOL programs have a File-section and a Working-storage section, but may not have any other sections.

5.4.1 The Input File.

The Input File describes the input that is going to be used by the program. The variable names used in this file will be assigned some values when the file is read. CLIVE asks the user to supply the names of the input variables. The user will know what data is going to be used by the program and so the names of the input variables will correspond to the names of the data. Identifiers are extracted from the input so that the generated program talks about the problem in the same terms as the user, rather than using any other form of generated symbols.

The format the user can use for typing in the identifiers is reasonably unrestricted,

e.g. STOCK ON HAND, THE REORDER POINT AND THE
REORDER QUANTITY.

The actual input file is then easily generated, i.e.

```
01  INPUT-FILE.
    02  STOCK-ON-HAND      PIC ...
    02  FILLER  ...
    02  REORDER-POINT      PIC ...
    02  FILLER  ...
    02  REORDER-QUANTITY   PIC ...
    02  FILLER  ...
```

The value of the filler would be chosen by CLIVE and would be dependant upon how data items were in the input file and the size of each data item.

The input is not restricted to 02 level items, an input like:

... THE DATE WHICH CONSISTS OF THE DAY, MONTH AND THE
YEAR ...

would cause the generation of a file segment like:

```
02  DATE.
    03  DAY ...
    03  MONTH ...
    03  YEAR ...
```

The input string is looked at by certain sections of the Input Module. This facilitates removal of redundant input, for example, 'AND', 'THE', etc and also indicates when items are to be subdivided (02, 03, 04 levels etc).

Some restrictions upon the input are neccessary however, as all items must be 'partitioned' by a delimiter, such as a comma, the word 'AND', etc.
This restriction is brought about by the fact that CLIVE cannot use blanks as delimiters. COBOL allows more than one word per variable name but seperates each word by a hyphen. An input like:

HOURS WORKED HOURLY RATE

could be considered as being 1, 2, 3, or 4 identifiers, depending upon where the hyphens were to be placed. Therefore the user must either use hyphens in the input when neccessary, or use some type of delimiter. In the authors opinion, the latter is the more desirable alternative.

5.4.2 The Data To Be Used.

The data that CLIVE uses can be supplied in two ways.

1) The user can have the data already on a tape or on a disk file, (perhaps from a previous session with CLIVE). When the COBOL program requires the data, the tape or disk file will be supplied.

2) The user can input his own data (from a terminal or a card reader). CLIVE will then make up a disk file of this data and the COBOL program will use this as is required.

The data that is used is input in free format. CLIVE knows exactly what type each data item should be, how long each data item can be and how many items there should be in a

group. CLIVE gets all this information from the Input File (5.4.1).

Several heuristics aid the Data Checker module (7.4) in ascertaining that the data is of the correct type and is also valid. A heuristic has values that are 'likely' to be given to certain commonly used variables. Thus a data entry such as,

HOURS-WORKED = 2.0 HOURLY-RATE = 40.0

would be flagged as being wrong, possibly due to a transposition of numbers and consequently changed.

If the data is input from a remote terminal, CLIVE will prompt the user by positioning the cursor on the screen so that blocks of data are aligned. By doing this, the user is more liable to see any mistakes that he might make.

5.4.3 The Output File.

The Output file is made up in a similar manner to the Input file. That is, the user types in the names of the items that he requires to be output, the input is analysed, and the appropriate file description is then generated.
Whereas in the creation of the Input file, the distance between variables (the filler) was dependant upon CLIVE only, in the Output file, the distance between output variables is user dependant.
Obviously the user wants his output to appear in a pleasant form, i.e. spaced out across the page. The way that the user can determine the output spacing is by the use of headings. If headings are to be used, the user will determine their

position across the page. (Headings are explained in 5.5.2). Therefore the distance between headings and the distance between the output variables must correspond. That is, all of the output will be aligned underneath the appropriate headings.

In some instances the user may not have to input the variable names that are to be output. The Assumption Module (7.5) may make up the output file without the direct aid of the user. CLIVE assumes that headings are normally synonyms for the output variables. If the heading names are found in a dictionary of common data processing variable names, then CLIVE will assume that the output variables and the heading names are equivalent.

Consider the following example:

The user inputs headings of NAME, AGE, INCOME and CODE. Once these names have been found in the dictionary, CLIVE will make several checks.

1) Do these names or their synonyms appear in the Input file?

2) Do these names or their synonyms appear in the original problem to be solved?

The variable EMPLOYEES-NAME may be found in the Input file, along with the variables AGE and CODE.

The original problem may have the variable INCOME mentioned in it and this could be equivalent to the variable WEEKLY-SALARY or MONTHLY-SALARY, depending upon the question that was asked.

Therefore CLIVE would make up the Output file with the variable names,

EMPLOYEES-NAME, AGE, WEEKLY-SALARY, and CODE.

5.5 The Working-Storage Section.

The Working-storage section of a COBOL program is that area in which data that is not associated with an input or output file is declared.

There are two types of entries that may appear in the Working-storage section; they are non-contiguous entries (i.e. single unrelated variables) and group record descriptions.

The non-contiguous entries are usually referenced as 77 level entries although 66 and 88 level entries exist also.

The record description entries are in the conventional level by level manner that is generally associated with COBOL.

The decision as to what variables need to go into the non-contiguous section is made when the calculation portion of the Procedure Division is generated (6.4). Variables that are to be used in intermediate calculations are usually assigned to non-contiguous sections.

The record description entries are usually titles, headings, tables or total/count lines. Sometimes there may be all four types of record entries present and sometimes there may be none. CLIVE will ask the user if he requires any titles, etc, and will then make up the appropriate record description.

5.5.1 Titles.

CLIVE will ask the user if he requires a title. If the users response is affirmative, then CLIVE will ask the user to type in the text of the title. After the title has been input, CLIVE will then ask the user where he wants the title positioned.

CLIVE responds to certain keywords to obtain the positioning of the title. The user could type in, for example

AT THE MARGIN (Margin being a keyword) or

IN THE CENTRE (Centre being a keyword) or

5 SPACES IN FROM THE MARGIN

etc.

It is a relatively easy task for CLIVE to work out the required positioning for the title once two things are known.

1) The number of blanks that are to preceed the title.

2) The actual text of the title.

From this information, the record description is easily generated.

```
01  TITLE-LINE.
    02  FILLER  PIC X(number of blanks)
        VALUE IS SPACES.
    02  FILLER  PIC X(17)  VALUE IS
        "THIS IS THE TITLE".
    02  FILLER PIC X(number of blanks left)
        VALUE IS SPACES.
```

5.5.2 Headings.

Record descriptions for headings are handled in a similar way to that of titles. CLIVE considers headings as groups of titles. The user may indicate spacings between the text by simply using numbers. For example, the user may input

15 NAME 13 AGE 5 INCOME 3 CODE.

The record description is thus easily made up. As explained in the Output file (5.4.3), the layout of the headings has a direct relationship upon the layout of the Output file and upon the variables in that file.

5.5.3 Tables.

Tables in a COBOL program are actually arrays or lists and they may have up to three dimensions.
The types of tables that appear in the Working-Storage section are generally ones whose variables have predefined values. e.g. income tax groupings, pay rates, insurance tables, etc. For example,

Pay rate is 1 : Value is 2.00
Pay rate is 2 : Value is 2.46
Pay rate is 3 : Value is 2.93
Pay rate is 4 : Value is 3.06

The tables can be read in from a disk file, or from a tape file or the user may insert the tables contents himself either via a card reader or a remote terminal.
If CLIVE knows the name(s) of the variable(s) involved the dimension of the table can be ascertained.

In the above example the table is one dimensional and the variable involved with the table is PAY-RATE.

CLIVE will assist the user (when he is using a terminal) by prompting him. CLIVE will output the variable name or names as headings, then the screen cursor will be aligned under the appropriate heading. Thus the user can see exactly where each value in the table is placed.

It is quite likely that most tables will be stored on magnetic tape. This is due to the fact that the contents of most tables remain reasonably constant and are generally reasonably large, making it more convenient to have these tables stored on a tape file.

5.5.4 Counts and Total Lines.

It is almost a standard practice in programming circles that when a list of figures is to be printed out, there is also a summary or count of the various figures that are involved in the listing.

For example: Consider a listing of the exam marks for a class of students. Typically the output could consist of the students name and the mark he or she received for each of the exams that the student sat. It is natural to have at the conclusion of the students figures a summary of the overall marks, for example, what the mean mark and standard deviation for each exam was, the mean and standard deviation for the overall marks (the aggregate), the upper and lower quartiles, etc. This type of information is usually found to be of reasonable importance to those people involved in the output list-

ing as much individual information is gained from the overall picture that the listing forms a part of. It is quite common for D.P. problems to involve totals, etc., so CLIVE has a set of heuristics to handle them if they are required.

CLIVE will ask the user to type in the names of the variables required. Common names like MEAN, MEDIAN, TOTAL, SUM, are recognized and handled by the heuristics. Unrecognized words are passed to the Assumption Module where they are considered as being synonyms for the above mentioned variables. The required source code appears in the Working-Storage section and in the Procedure Division.

The code in the Working-Storage section contains the default or user supplied text and the output variable. The code to print out the MEAN would be as follows.

```
01  MEAN-IS-REQUIRED.
    02  FILLER     PIC X(10)      VALUE IS SPACES.
    02  FILLER     PIC X(12)      VALUE IS
        "THE MEAN IS ".
    02  MEAN       PIC ZZ9V99.
    02  FILLER     PIC X(45)      VALUE IS SPACES.
```

The code in the Procedure Division describes the logic to perform the calculations that are required. These 'logic' elements are added to the generated program whenever they are required (6.2).

5.6 Pictures.

The type and size of a data item in COBOL is specified by the PICTURE clause. The two most commonly used (and the most elementary) options are:

1) A sequence of n 9's. This represents a numeric value of length n. A single V may be inserted somewhere in the string of 9's to indicate the position of the decimal point.

   E.g. 99V99 would be a suitable picture for the variable HOURS-WORKED.

2) A sequence of n X's, n A's, or 9's which must contain at least one X. This represents an alphanumeric string of length n, and it doesn't matter whether a alphabetic part of the string occupies a 9 slot since the COBOL language defines the word alphanumeric in quite general terms.

   E.g. A(20), i.e. 20 A's, would be a suitable picture for the data item EMPLOYEES-NAME if only alphabetic characters were used.

   X(5) or XXXXX would be a suitable picture for the variable STOCK-NAME, where the name was of the form AB123, EX025, etc.

The above type of PICTURE definitions are usually confined to the input data, whereas the data to be output usually appears in an altered form (i.e. edited), requiring the addition of dollar signs and asterisks, the removal of leading

zeros, etc. Although the basic fixed pictures suffice for input and output, normally use is made of COBOL's editing features for all output.

CLIVE has a dictionary of variable names and their synonyms. With each name is a pointer to a storage location that contains an appropriate picture definition. As many variables can have the same picture definition, this storage area is relatively small in comparison to the size of the dictionary itself.

When the Input file is generated, each data item is given a picture from this storage area. The output data also receives a picture definition, but it is edited.

CLIVE has default options for edited output, suppression of leading zeros and addition of dollar signs when the output is money (the $ flag has been set), etc.

The user will have an opportunity to see the PICTURE's that CLIVE has allocated to each data item by asking the Assumption Module to display the relevant information.

If, as is possible, CLIVE cannot allocate a suitable picture to a variable, then the user is asked to supply one and CLIVE will assist the user in obtaining one.

For example CLIVE may ask the user:

> IS THE VARIABLE (NAME) GOING TO BE ASSIGNED ALPHABETIC VALUES OR NUMERIC VALUES?
>
> HOW MANY LETTERS DO YOU THINK IT WILL HAVE?

CLIVE goes to these lengths in questioning because it is always possible that the user knows no COBOL and hence would know nothing about PICTURES.

CHAPTER 6.

THE PROCEDURE MODULE.

> "What was that?" inquired Alice.
> "Reeling and Writhing, of course,
> to begin with", the Mock Turtle
> replied, "and then the different
> branches of Arithmetic-
> Ambition, Distraction,
> Uglification and Derision."
>
> Lewis Carroll.
> "Alice in Wonderland".

6.1 Introduction.

This chapter explains how CLIVE generates the Procedure Division of the COBOL program. Whereas the Data Division describes the data to be used in the program, the Procedure Division describes what is to be done with the data. The generation of statements that are common to the majority of COBOL programs is discussed first, then the analysis of the question that the user submits to CLIVE is considered. Finally the chapter considers the generation of code for any calculations that may be required in the program.

6.2 Standard Logic.

In the Procedure Division of a COBOL program, some statements are normally always present.

e.g. for reading in data,

for opening and closing files.

i.e.
```
OPEN INPUT INWARDS.
OPEN OUTPUT OUTWARDS.
CLOSE INWARDS OUTWARDS.
```

Other statements will always appear if there are certain types of Data Division entries.

For example. If a record description for a title appears in the Data Division, then the form that the statements to output the title must take are predefined.

Namely,

```
MOVE TITLE-NAME TO THE-OUTPUT-FILE.
WRITE THE-OUTPUT-FILE AFTER ADVANCING 1 LINES.
MOVE SPACES TO THE-OUTPUT-FILE.
WRITE THE-OUTPUT-FILE AFTER ADVANCING 2 LINES.
```

CLIVE has several procedures that generate source code for the above types of statements. Thus every time a file needs to be opened or a title needs to be written, the neccessary COBOL source code is easily generated.

Each of the procedures has some parameters associated with it. In the above example, there are three parameters involved.

1) A variable indicating how many titles are to be printed. (Text that is underlined is considered as 2 titles, one title of text and the other title of underlining).

2) The name of the title (TITLE-NAME in the above example).

3) The name of the output file (Obtained from the file-namer module).

When these procedures are used, source code is generated using the relevant identifiers as parameters.

Whilst the above mentioned standard logic modules are relatively straight forward, there are several more modules that CLIVE uses that are more complex. These modules are for generating source code to calculate means, standard deviations, sums of numbers, and so on.

To calculate the mean of a set of numbers, the sum of the numbers must be divided by the number of items used. This can not be done in one step but has to be done in stages during the running of the program. Therefore CLIVE must:

1) Keep a count of the number of items used.

2) Keep a running total of all these items.

3) Work out the mean when all of the items have been used.

These operations are reasonably straight forward, and although they are being done with other operations, CLIVE handles them efficiently.

Parameters are also used by this procedure to indicate information, the most important parameter being the one that indicates which set of figures or data to use in the calculation.

The procedure used to obtain the standard deviation is a good deal more complex. Firstly CLIVE must calculate the mean of the numbers, and then use this mean in further calculations.

CLIVE makes use of these standard logic modules whenever they are required, to generate the neccessary sections of the Procedure Division. Because CLIVE puts all of the input data into a disk file, the data can then be referenced whenever it is required (unlike cards in a card reader), therefore allowing CLIVE to do multiple tasks on the same data quite easily.

6.3 The Question.

The original problem that was input to CLIVE at the beginning of the session is 'rearranged' by the Input Module (Ch 4) and then passed to the Procedure Module, where the appropriate source code is generated.

CLIVE makes use of keywords to generate the code, and uses the knowledge that if a sentence is to make sense, then certain keywords must have certain variables surrounding them. This will always be true providing the problem is phrased in a logical manner.

For example, in a D.P. context, the words GREATER THAN must always be followed by some value. The sentence would be meaningless otherwise.

We will now consider a problem that the user might input and illustrate the steps required for the production of the relevant source code.

The user inputs the following problem:

I WANT A LISTING OF ALL MY MARRIED MALE EMPLOYEES WHO ARE OVER 30 YEARS OF AGE AND WHO EARN LESS THAN $100 A WEEK, AND ALSO A LISTING OF ALL THE WOMEN WHO EARN LESS THAN $50 A WEEK.

The Input Module will 'rearrange' the problem and 'send' to the Procedure module, the following:

A flag which indicates that <u>two</u> listings are required.
A flag indicating that the word EARN means money/week, and the rearranged input;

MARRIED MEN AGE OVER 30 AND PAY UNDER 100 **BRK**
WOMEN PAY UNDER 50.

Certain keywords can then be replaced by internal synonyms.

MARRIED becomes MAR-STATUS EQUAL TO 1

where 1 is for married and 0 is for single.

MEN becomes SEX EQUAL TO 0.

The input string is now:

MAR-STATUS EQUAL TO 1 SEX EQUAL TO 0 AGE OVER 30 AND PAY UNDER 100.
SEX EQUAL TO 1 AND PAY UNDER 50.

At this point, the only unknown variable is PAY. A flag has been set to indicate that the word EARN means money/week and therefore the word EARN was replaced by the word PAY.
CLIVE will look through the entries in the input file to try and find a suitable synonym for PAY, e.g. WEEKLY-SALARY or WEEKLY-PAY, etc. If a synonym is found, then it is used in place of the word PAY.

If a synonym is not found, CLIVE will assume that the word PAY or a suitable synonym will be defined when the code for the calculations is generated (6.4). If the word PAY is still not defined after the calculations have been defined, CLIVE will ask the user for the correct synonym or word.
Once a synonym has been found, the final source code is generated.

i.e.

IF MAR-STATUS IS EQUAL TO 1 AND SEX IS EQUAL TO 0 AND AGE IS GREATER THAN 30 AND WEEKLY-SALARY IS LESS THAN 100 PERFORM PRINT-IT-OUT.

IF SEX IS EQUAL TO 1 AND WEEKLY-SALARY IS LESS THAN 50 PERFORM PRINT-IT-OUT.

There is of course 'padding' to go around the IF...PERFORM... statements. As stated in 6.2, the data can be read in many times, therefore many different operations can be done. CLIVE incorporates suitable logic to do this. Once one IF...PERFORM... statement has been completed, the data needs to be read in again, for the next IF...PERFORM... statements, etc.

If a user requires an and situation in a sentence,

e.g.

...IS MARRIED AND IS A MALE...

then he will usually omit the conjunction.

i.e.

...MARRIED MALE...

The meaning of each sentence is still the same.

However an or situation,

e.g.

...IS MARRIED OR IS A MALE...

cannot usually be shortened if the sense of the statement is to remain the same.

CLIVE therefore assumes that input of the form

...NAME CONDITION VARIABLE NAME CONDITION VARIABLE...

must be joined by an and and so will insert one in the correct place.

6.4 Calculations.

If a variable is used in the output file description, then it must be assigned a value before it can be output.

The variable can receive a value from

1) the input file.

2) a calculation.

CLIVE checks that every output variable receives a value.

If no value is received from the input file (e.g. read in a persons name and print the same name out), then some calculations must be needed.

CLIVE has a table of calculations that are frequently used in data processing,

e.g.

WEEKLY-SALARY = HOURS-WORKED * HOURLY-RATE.

OVERTIME-WORKED = HOURS-WORKED - 40.0.

When a calculation is needed, CLIVE will look at this table to see if the relevant calculation can be found. If neccessary synonyms will be used to establish a match. Once a

match is found, the neccessary source code is generated. As the majority of COBOL programs have only small amounts of calculations,(McCracken 1970), the method of having a table of calculations is quite satisfactory, providing of course that the table is of a reasonable size.

If however, no entry is found in the table to correspond to a particular variable, then the user will input his own calculation or calculations,

e.g. CLIVE may ask the user

HOW DO I OBTAIN A VALUE FOR THE VARIABLE 'STOCK'?

and the user may type in

STOCK = ON-HAND + ON-ORDER + IN-TRANSIT.

The required COBOL source code for that calculation can then be generated.

There is a problem involved with the user putting in his own calculations, if the calculations are complex ones. Simple calculations, as in the above example, are easily handled, but complex ones would be much more difficult as a scanner and a parser would be required. Fortunately complex calculations are seldomn used in a COBOL program and so CLIVE can be justifiably restricted to relatively simple ones.

An important feature of the procedure that generates the code for the calculations is its regard for order. When the code for the calculations is generated, a check is made to ensure that the calculation of any variable preceeds its use. For example, if the user inputs

CUBE = SQUARE* NUMBER

SQUARE = NUMBER **2

then CLIVE will transpose the input to generate the correct source code;

i.e. COMPUTE SQUARE = NUMBER ** 2.

COMPUTE CUBE=SQUARE * NUMBER.

All of the calculations are treated in this fashion, thus lessening the chance of any semantic errors occuring during the running of CLIVE.

6.5 Program Completed.

When all of the Procedure Division code has been generated, the program is ready to be compiled and executed.
Before this is done the user may wish to view the generated source code and/or may wish to make some alterations to it. After any changes have been made, the generated program is run.

CHAPTER 7.

ANCILLARY MODULES.

"Unimportant, of course,
I meant", the King hastily
said, and went on to him-
self in an undertone,
"important - unimportant
- unimportant - important"
as if he were trying which
word sounded best.

Lewis Carroll.
"Through the
Looking Glass".

7.1 Introduction.

There are several modules that aren't strictly neccessary for the overall operation of CLIVE. They are however important if CLIVE is going to operate in as natural a manner as is possible. This chapter describes these modules.

7.2 Spelling Mistakes.

There is a module that attempts to fix any spelling mistakes that may occur.
Spelling mistakes usually occur in one of two forms (Colby 1974).

1) A spelling mistake may arise from the typing of a wrong letter.

For example, "YUS" instead of "YES"

"THANL" instead of "THANK".

There are three common mistakes that typists are liable to make.

1) No shift key for an apostrophe,

e.g. "DON8T" instead of "DON'T".

2) Hitting a nearby key,

e.g. "WHAY" instead of "WHAT".

3) Transposing two letters,

e.g. "INSTAED" instead of "INSTEAD".

These types of errors are quite easily checked. Colby (Colby 1974) has found a group of letters that are more liable to be transposed; namely

(TY) (OP) (AS) (QW) (IO) (GH) (NM)

2) The other form mistakes take are <u>actual</u> spelling mistakes as opposed to typing errors. Two common types of mistakes in this category are the doubled letter and the extraneous letter.

CLIVE attempts to correct both types of mistakes.
Errors of type 1 are handled in a straight forward manner using simply pattern matching techniques.
Errors of type 2 are handled in a slightly different way.

A dictionary of <u>common</u> data processing words is used as a guide for possible errors of type 2

If, for example, the user makes use of an identifier called "EMPLOYEES-NAME" in one part of the program, and uses the same identifier in some other part of the program but calls it "EMPLOYES-NAME", then a spelling mistake has occured. CLIVE will use the dictionary of common words to see which identifier is the correct one and will correct the faulty one. This correction is brought about by the assumption module (Ch 7.5). The user can ask CLIVE what assumptions it has made at any time during the session.

7.3 Descriptive Names.

It is a common practice when writing COBOL programs for the programmer to use file description names that explain what the file is to be used for.
An input file that reads in payroll figures would probably be called

```
FD  PAYROLL-FIGURES.
```

An output file that prints out an overtime listing would probably be called

```
FD  OVERTIME-LISTING.
```

Using this type of naming makes it easier for the programmer, and anyone else to understand the program.
CLIVE will therefore associate file descriptors with the problem that is to be solved.
On the surface this feature of CLIVE may appear to be simply a further aid to source code readability, however the implications of it are not so trivial.

File descriptor names appear in many places throughout a COBOL program. They appear in the File Section, the Data Division and many times in the Procedure Division.
An easier implementation could be given if standard pre-defined names were used to describe the files. For example,

```
FD  CARD-INPUT
```

but it is in the authors opinion that the "readability" aspect of the program is of higher priority than ease of implementation.

7.4 Data Checking.

Regardless of what type of system a program is run under, (i.e. batch, on-line, etc.,) the correctness of the data to be used by the program is very important. Consequently data checking facilities are usually considered an important part of any program that may have to handle large volumes of data. The amount of data that CLIVE will use may vary, depending upon the problem to be solved. If however CLIVE is to be considered as a practical data processing tool, then large quantities of data will be used. CLIVE therefore incorprates several data checks for incorrect data.

The data to be used by the generated program must correspond to the formats defined in the input file description. Any irregularities (e.g. numeric items in alphabetic fields, numbers or names too long, etc.,) are flagged by CLIVE and the user can then correct them.

It is the users responsibility however, to check that run

time errors arising out of incorrect data do not occur. CLIVE will help with this problem to some extent by checking for some hardware errors, e.g. integer overflow, division by zero, but if faults occur in the program due to faulty data then the user must fix them.

An additional module that CLIVE could have would be one that did test runs on selected data items. In other words a test run of the program could be made using a small random sample of data. This would be a very good facility to have if very large amounts of data are going to be used in the program.

7.5 The Assumption Module.

The premise upon which the Assumption Module is based is that it is more desirable to "assume" certain things at certain times, than it is to be constantly seeking confirmation of these facts from the user.

Humans in their daily conversation, assume details, using past experiences and knowledge as a guide. If they still cannot understand a particular situation, they will then ask about it. To make CLIVE operate in a similar fashion, the Assumption Module has been added.

This module is perhaps mis-named as it is really a synonym matcher. A fact is assumed if a suitable synonym match is found.

For example, the output file variables can be made up from the heading description, simply because they are basically labels for the same things. Because it is reasonable for a human to assume that the headings depict what is to be out-

put, then CLIVE can assume the same thing.

This similarly occurs with words used in the question that the user inputs. For example, if the word SALARY appears in the question and the variable WAGES appears in the output file description, CLIVE will check in a table of words that are synonyms to see if the words are in fact synonyms. If SALARY and WAGES mean the same thing, CLIVE will change the word SALARY to WAGES. This "synonym" match may not always happen, if WAGES meant money/week and SALARY meant money/year then the two words are not synonyms.

During the construction of the Procedure Division, CLIVE will compare the variable WAGES with a table of calculation synonyms. Thus CLIVE may be able to obtain the neccessary calculations without asking the user to supply them.

CLIVE keeps track of <u>all</u> the assumptions it has made and the user can view them at any time and make changes if he so desires.

CHAPTER 8.

CONCLUSION.

"Let us hear the conclusion
of the whole matter."

New Testament:
Ecclesiastes, XII, 13.

In the realms of Science Fiction one often finds computers that converse intelligently with human beings. These computers are portrayed as having the human qualities of speech, intelligence and understanding and they can solve problems mearly by listening to a question.
It is a rather large jump back to the actual reality of current computer operation. The higher level languages at present in use do not give a programmer very much flexibility in the way that he can solve his problem.
The gap between computer fantasy and computer reality is very large. CLIVE is a very small attempt at trying to bring the "current state of the art" a little closer to the ultimate.

CLIVE operates by conversing with the user in English and the user responds in a similar manner, thus the dialogue generally appears to flow in a natural fashion.
The small success (in this thesis) of using Natural Language as an aid to Program synthesis is probably due to the specific nature of CLIVE. CLIVE can only generate COBOL programs and rather than try to get a general synthesizer, the author

feels that a better approach would be to design systems like CLIVE, for use with other problems, and other languages. CLIVE is restricted to Data Processing problems, other systems could be likewise restricted, but in other fields.

APPENDIX I.

A Listing from a session with the current implementation of CLIVE. (C is for CLIVE's response, U is for the users'.)

C. HELLO, MY NAME IS CLIVE, WHAT"S YOUR NAME?

U. OWEN.

C. WELL OWEN WELCOME TO THIS SESSION.
BETWEEN NOW AND UNTIL THE END OF THIS SESSION, YOU MUST END ALL OF YOUR CONVERSATION WITH A FULL STOP.
OKAY THEN, WE"RE READY TO ROLL; IF YOU WILL NOW TYPE IN YOUR PROBLEM, I"LL SEE IF I CAN SOLVE IT FOR YOU.

U. I WANT A LISTING OF ALL THOSE MEN WHO ARE OVER 30 YEARS OF AGE AND EARN MORE THAN $100 A WEEK.

C. DO YOU REQUIRE A TITLE?

U. NO THANKS.

C. OWEN, DO YOU WANT ANY HEADINGS?

U. YES I DO.

C. PLEASE TYPE IN HOW YOU WISH YOUR HEADING(S) TO APPEAR ON YOUR LISTING.

U. NAME AGE SEX INCOME.

C. I NOW WANT TO FIND OUT WHAT THE NAMES OF THE INPUT VARIABLES ARE. SO TYPE THEM IN USING FREE FORMAT, JUST MAKE SURE OWEN THAT YOU SEPARATE EACH VARIABLE NAME BY A COMMA.

U. NAME, SEX, AGE, HOURLY RATE, HOURS WORKED.

C. DO YOU WANT THE CURRENT DATA FILE SAVED?

U. NO THANKS.

C. I NOW WANT YOU TO TYPE IN THE DATA THAT YOU HAVE FOR THIS PROBLEM. YOU MAY FIND IT HELPFUL IF YOU USE SEPARATE LINES FOR EACH GROUP OF DATA; HOWEVER, THAT IS UP TO YOU, SO LONG AS YOU SEPARATE EACH DATA ITEM BY BLANK SPACES OR A COMMA.

U. SMITH 0 25 2.0 40
JONES 1 39 3.75 27.8
HOGNORT 1 23 4 20
BADEN-SMIT 0 46 3.50 10
IGOR 1 38 1.50 10
DAGG 0 100 2.0 51 KESSELL 0 85 5.0 40.

C. NOW OWEN I WANT YOU TO TYPE IN THE NAMES OF THE OUTPUT VARIABLES. DO IT THE SAME WAY THAT YOU DID THE INPUT FILE. I.E. SEPARATE EACH VARIABLE NAME BY A COMMA.

U. NAME, AGE, SEX, INCOME.

C. DO ANY CALCULATIONS NEED TO BE DONE?

U. YES THERE IS ONLY ONE.

C. OKAY THEN OWEN, TYPE IN THE CALCULATIONS THAT I WILL NEED TO SOLVE THE PROBLEM.
I'M AFRAID THAT I MUST RESTRICT YOU TO ONE CALCULATION PER LINE.

U. INCOME=HOURS WORKED*HOURLY RATE.

C. I"M NOW GOING TO EXECUTE THE PROGRAM,
IT SHOULDN'T TAKE VERY LONG.

APPENDIX II.

A Listing of the results that were produced from the session of Appendix I.

C. OWEN, YOUR RESULTS ARE AS FOLLOWS.

NAME	AGE	SEX	INCOME
DAGG	100	0	102
KESSELL	85	0	200

APPENDIX III.

A Listing of the program that was produced from the session of Appendix I.

```
1    * THIS SEGMENT EXPLAINS WHAT THIS PROGRAM IS ABOUT,
2    * WHO WROTE IT AND WHEN, ETC.
3    *
4     IDENTIFICATION DIVISION.
5     PROGRAM-ID.     THIS PROGRAM GIVES A LISTING OF ALL
6                     THOSE MEN WHO ARE OVER 30 YEARS OF AGE
7                     AND EARN MORE THAN $100 A WEEK.
8     AUTHOR.  WRITTEN FOR OWEN BY C L I V E.
9     DATE-WRITTEN.    THURSDAY,  11/12/76, 01:40 PM.
10    DATE-COMPILED.  THURSDAY,  11/12/76, 01:40 PM.
11   *
12   *
13   * THIS SEGMENT SIMPLY INDICATES WHICH COMPUTER THIS
14   * PROGRAM IS BEING RUN ON.
15   *
16    ENVIRONMENT DIVISION.
17    CONFIGURATION SECTION.
18    SOURCE-COMPUTER.   B-6700.
19    OBJECT-COMPUTER.   B-6700.
20   *
21   *
22   * THIS SEGMENT SHOWS THE NAMES OF THE FILES THAT ARE
23   * USED AND WHAT PERIPHRAL DEVICES THESE FILES WILL BE
24   * USING.
25   *
26    INPUT-OUTPUT SECTION.
27    FILE-CONTROL.
28        SELECT IN-DATA      ASSIGN TO DISK.
29        SELECT LISTING      ASSIGN TO REMOTE.
30   *
31   *
```

```
32   * THIS IS THE START OF THE DATA DIVISION, WHERE ALL
33   * OF THE DATA USED IN THIS PROGRAM IS DEFINED.
34   *
35    DATA DIVISION.
36    FILE SECTION.
37   *
38   * THIS IS THE DESCRIPTION FOR THE INPUT FILE
39   * CLIVES/DATA IS THE NAME OF THE DISK FILE THAT THE
40   * DATA FOR THIS PROGRAM IS STORED ON.
41   *
42    FD  IN-DATA
43        VALUE OF ID IS "CLIVES"/"DATA"
44        DATA RECORD IS DATAIN.
45    01  DATAIN.
46        02  FILLER    PIC X(5).
47        02  IN-NAME    PIC X(15).
48        02  FILLER    PIC X(5).
49        02  IN-SEX    PIC 9.
50        02  FILLER    PIC X(5).
51        02  IN-AGE    PIC 999.
52        02  FILLER    PIC X(5).
53        02  IN-HOURLY-RATE    PIC 99999.
54        02  FILLER    PIC X(5).
55        02  IN-HOURS-WORKED    PIC 9999.
56        02  FILLER    PIC X(20).
57   *
58   * THIS IS THE DESCRIPTION OF THE OUTPUT FILE
59   * IT TELLS YOU WHAT IS GOING TO BE PRINTED OUT
60   *
61    FD  LISTING    DATA RECORD IS DATAOUT.
62    01  DATAOUT.
63        02  FILLER    PIC X(03).
64        02  OUT-NAME    PIC X(15).
65        02  FILLER    PIC X(01).
66        02  OUT-AGE    PIC 999.
67        02  FILLER    PIC X(06).
68        02  OUT-SEX    PIC 9.
69        02  FILLER    PIC X(06).
```

```
70          02  OUT-INCOME   PIC ZZ999.
71          02  FILLER    PIC X(33).
72      *
73      * THIS IS THE WORKING STORAGE SECTION WHERE TEMP-
74      *ORARY STORAGE FOR VARIABLES IS MADE
75      *
76       WORKING-STORAGE SECTION.
77       77  A   PIC 99    USAGE IS COMP.
78       77  COUNTA    PIC 9    USAGE IS COMP.
79       77  IN-INCOME    PIC 99999
80           USAGE IS COMP.
81      *
82      * WE NOW HAVE THE DESCRIPTION FOR THE HEADINGS THAT WE
83      * WANT PRINTED OUT.
84      *
85       01  HEADING-LINE-1.
86           02  FILLER    PIC X(03)     VALUE IS SPACES.
87           02  FILLER    PIC X(04)     VALUE IS "NAME".
88           02  FILLER    PIC X(12)     VALUE IS SPACES.
89           02  FILLER    PIC X(03)     VALUE IS "AGE".
90           02  FILLER    PIC X(04)     VALUE IS SPACES.
91           02  FILLER    PIC X(03)     VALUE IS "SEX".
92           02  FILLER    PIC X(05)     VALUE IS SPACES.
93           02  FILLER    PIC X(06)     VALUE IS "INCOME".
94           02  FILLER    PIC X(32)     VALUE IS SPACES.
95      *
96       01  HEADING-LINE-2.
97           02  FILLER    PIC X(03)     VALUE IS SPACES.
98           02  FILLER    PIC X(04)     VALUE IS "____".
99           02  FILLER    PIC X(12)     VALUE IS SPACES.
100          02  FILLER    PIC X(03)     VALUE IS "___".
101          02  FILLER    PIC X(04)     VALUE IS SPACES.
102          02  FILLER    PIC X(03)     VALUE IS "___".
103          02  FILLER    PIC X(05)     VALUE IS SPACES.
104          02  FILLER    PIC X(06)     VALUE IS "______".
105          02  FILLER    PIC X(32)     VALUE IS SPACES.
106     *
107     *
```

```
108  * WE NOW COME TO THE PROCEDURE DIVISION.
109  * IN THIS DIVISION ALL OF THE CALCULATIONS AND OTHER
110  * COMMANDS ARE DONE.
111  *
112   PROCEDURE DIVISION.
113  *
114  * THIS PART OF THE PROGRAM OPENS THE I/O FILES AND
115  * PRINTS ANY TITLES AND/OR HEADINGS THAT ARE REQUIRED.
116  *
117   BEGIN-THE-PROG.
118       SET  IN-DATA (FILETYPE) TO 7.
119       OPEN INPUT IN-DATA.
120       OPEN OUTPUT LISTING.
121       PERFORM PAGE-HEADING.
122  *
123  * THIS IS THE "MAIN" PART OF THE PROGRAM AS THE LOGIC
124  * OF THE PROBLEM APPEARS HERE.
125  *
126  * I.E.  THIS IS WERE THE CALCULATIONS AND TESTS ARE DONE
127  *
128   MAINLINE.
129       READ IN-DATA;  AT END GO TO EOJ.
130       COMPUTE IN-INCOME = IN-HOURS-WORKED * IN-HOURLY-RATE.
131       IF IN-SEX IS EQUAL TO 0 AND
132       IN-AGE IS GREATER THAN 30
133       AND IN-INCOME IS GREATER THAN 100
134       PERFORM PRINT-OUT.
135       GO TO MAINLINE.
136  *
137  * THIS PART OF THE PROGRAM MOVES DATA INTO THE OUT-
138  * PUT FILE SO THAT IT CAN BE PRINTED OUT.
139  *
140   PRINT-OUT.
141       MOVE IN-NAME TO OUT-NAME.
142       MOVE IN-SEX TO OUT-SEX.
143       MOVE IN-AGE TO OUT-AGE.
144       MOVE IN-INCOME TO OUT-INCOME.
145       WRITE DATAOUT AFTER ADVANCING 1 LINES.
```

```
146  * THIS IS THE PART THAT IS NECCESSARY FOR THE HEADING
147  * TO BE PRINTED OUT.
148  *
149   PAGE-HEADING.
150       MOVE HEADING-LINE-1 TO DATAOUT.
151       WRITE DATAOUT AFTER ADVANCING 1 LINES.
152       MOVE HEADING-LINE-2 TO DATAOUT.
153       WRITE DATAOUT AFTER ADVANCING 1 LINES.
154       MOVE SPACES TO DATAOUT.
155       WRITE DATAOUT AFTER ADVANCING 2 LINES.
156  *
157  * THIS PART OF THE PROGRAM CLOSES ALL THE FILES THAT
158  * WERE OPENED AND USED IN THE PROGRAM.
159  *
160   EOJ.
161       CLOSE  IN-DATA   LISTING.
162  *
163  * THIS IS THE END OF THIS PROGRAM
164  *
165       STOP RUN.
```

References and Bibliography.

ADAMS, J.M., and HADEN, D.H. (1973)

Computers: Appreciation, Application, Implications.

Wiley, Canada.

BANERJI, R.B. (1969)

Theory of Problem Solving: An approach to Art. Intell.

Elsevier, New York.

BARRON, D.W. (1971)

Programming in Wonderland.

The Computer Bulletin 15, 4 (April 1971), p.153.

BERKELEY, E.C., LANGER, A., and OTTEN, C. (1973)

Computer programming using Natural Language -Part 3.

Computers and Automation 22, 8 (August 1973), pp. 28-35.

BOBROW, D.G. (1968)

Natural Language input for a computer problem solving system. pp. 135-215. In Semantic Information Processing. Minsky, M. (ed). The MIT press, (473 pgs).

BORGIDA, A.T. (1974)

Topics in the understanding of English sentences by computer. Thesis, M.Sc., University of Toronto.

BUCHANAN, J.R., and LUCKHAM, D.C. (1974)

On automating the construction of programs.

Stanford: Stanford Art. Intell., AIM-236.

CHOMSKY, N. (1959)

On certain formal properties of grammers.

I.C. vol 2, pp. 135-167.

COLBY, K., PARKINSON, R. and FAUGHT, B. (1974).

Pattern matching rules for the recognition of Natural Language dialogue expressions.

Stanford: Stanford Art. Intell., AIM-234.

DAHL, O.J., DIJKSTRA, E.W. and HOARE, C.A.R. (1972)

Structured Programming.

Academic Press, London.

DARLINGTON, J.L. (1965)

Machine methods for proving logical arguements expressed in English. Mech. Trans. Comput. Linguist. 8 (July 1965), pp. 41-47.

DORF, R.C. (1974)

Computers and Man.

Boyd and Fraser Pub. Co. (467 pgs).

ERMAN, L.D. (1976)

Overview of the Hearsay speech understanding system.

pp. 9-15. SIGART No 56, (Feb.1976).

EVANS, T.G. and DARLEY, D.L. (1966)

On-Line debugging techniques: A survey.

Proc. AFIPS 1966, FJCC, 29, pp. 37-50.

FEIGENBAUM, E. and FELDMAN, J. (1963)

Computers and Thought.

Mc Graw-Hill, New York.

FLOYD, R.W. (1967)

Assigning meanings to programs.

Proc. Symp. Appl. Maths. 19, pp. 19-32.

GELB, J.P. (1971)

Experiments with a Natural Language problem solving system. IJCAI, pp. 445-462.

GREEN, C. and RAPHAEL, B. (1968)

The use of thereom proving techniques in question-answering systems.

Proc. ACM., Thompson Book Co., Washington DC.

GROVES, L. (1974)

The provision of debugging facilities for high level languages. Thesis, M.Sc., Massey University, (167 pgs).

HAMILTON, J., FINLAYSON, E. and HEYWOOD-JONES, A. (1973)

Computer aided program production.

Datafair 73, Vol 1, pp. 191-196.

British Computer Society.

HAYS, D.G. (1966)

Readings in automatic language processing.

American Elsevier Publishing Co., (199 pgs).

HAYS, D.G. (1967)

Introduction to Computational Linguistics.

Macdonald & Co., (225 pgs).

HEWITT, C. (1969)

PLANNER: A language for proving thereoms in Robots.

Proc. IJCAI. Mitre Corp., pp. 259-301.

HILL, I.D. (1972)

Wouldn't it be nice if we could write computer programs in ordinary English-or would it.

Computer Bulletin, June 1972, pp. 306-312.

JACKSON, P.C. Jnr. (1974)

Introduction to Artificial Intelligence.

Petrocelli Books, N.Y. (447 pgs).

KANEFF, S. (1969)

Picture Language Machines. (editor).

Academic Press, (424 pgs).

LONDON, R.L. (1970)

Bibliography on Proving the correctness of programs.

Machine Intelligence 5, pp. 569-580.

MANNA, Z. (1969)

The correctness of programs.

J. Comp. Sys. Sci., vol. 3, no. 2, pp. 119-127.

MANNA, Z., and WALDINGER, R.J. (1970)

Towards automatic program synthesis.

AIM-127. Stanford:Stanford Art. Intell. Proj.

MANNA, Z., and WALDINGER, R.J. (1975)

Knowledge and Reasoning in program synthesis.

In Artificial Intelligence 6, pp. 175-208.

MANNA, Z., NESS, S., and VUILLEMIN, J. (1971)

Inductive methods for proving properties of programs.

AIM-154. Stanford:Stanford Art. Intell. Proj.

McCRACKEN, D.D., and GARBASSI, U. (1970)

A guide to COBOL programming.

Wiley-Interscience, N.Y. (209 pgs.).

McCUSKER, T. (1976)

The industry in '76.

Datamation, vol 22, no. 1, Jan. 1976, pp. 66-68.

MEADOWS, C.T. (1970)

Man-Machine Communications.

Wiley-Interscience, N.Y. (422 pgs.).

MOORE, J., and NEWELL, A. (1973)

How can Merlin understand.

Dept. of C.S., Carnegie-Mellon University, (56 pgs.).

MINSKY, M. (1968)

Semantic Information Processing. (editor).

The MIT Press. (437 pgs.).

NEWELL, A., et al. (1973)

Speech Understanding systems: First report of a study group. North Holland Publishing Co., (137 pgs.).

NILSSON, N.J. (1971)

Problem solving methods in A.I.

McGraw-Hill, N.Y.

ORR, W.D. (1972)

Conversational Computers. (editor).

J Wiley & Sons, N.Y.

PETERSON, N.D. (1976)

COBOL generation of source programs and reports.

Software-Practice and Experience, vol. 6, pp. 117-131.

QUILLIAN, M.R. (1966)

Semantic Memory.

In Semantic Information Processing, pp. 227-270.

RALSTON, A. (1971)

Introduction to programming and computer science.

McGraw-Hill, N.Y. (513 pgs.).

RAPHAEL, B. (1964)

SIR: A computer program for Semantic Information Retrieval. In Semantic Information Processing, pp. 33-145.

RUBINOFF, M., and MARSHALL, C.Y. (1975)

Advances in computers. (editors).

Volume 13, Academic Press, N.Y.

SAMMET, J.E. (1969)

Programming Languages: History and Fundamentals.

Prentice-Hall Inc., Englewood Cliffs, New Jersey.

SARKAR, P.A. (1970)

Natural Language and Artificial Intelligence.

In The Robots are coming, GEORGE, F., and HUMPHRIES, J. (editors)., NCC publications, pp. 1-186.

SCHANK, R.C., and COLBY, K.M. (1973)

Computer models of thought and language. (editors).

W.H. Freeman & Co., (455 pgs.).

SCHANK, R.C., and TESLER, L.G. (1969)

A conceptual parser for Natural Language.

IJCAI, pp. 569-578.

SCHANK, R.C. (1971)

Intention, memory and computer understanding.

AIM-140, Stanford:Stanford Art. Intell. Proj.

SCHWARCZ, R.M. (1969)

Towards a computational formalization of Natural Language semantics. Proc. Conf. on Computational Linguistics.

Stockholm, September 1-4.

SHAPIRO, S.C., and KWASNY, S.C. (1975)

Interactive consulting via Natural Language.

Com. of the ACM. August 1975, vol. 18, no. 8.

SIMMONS, R.F. et al. (1966)

An approach towards answering English questions from text.

Proc. of the Fall Joint Computer Conference. pp. 357-364.

TEITELMAN, W. (1972)

Do what I mean: The programmers assistant.

Computers and Automation, April 1972, pp. 8-11.

THOMPSON, F.B., and THOMPSON, B.H. (1975)

Practical Natural Language processing: The REL system as a prototype. In Advances in computers. pp. 110-167.

WALDINGER, R.J., and LEE, R.C.T. (1969)

PROW: A step toward automatic program writing.

IJCAI. pp. 241-252.

WATERMAN, D.A. (1968)

Machine learning of heuristics.

AI Memo 74. Stanford: Stanford Art. Intell. Proj.

WEINBERG, G.M., and GRESSER, G.L. (1963)

An experiment in Automatic program verification.

Com. of the ACM. Oct. 1963, vol 6, no. 10, pp. 610-613.

WEINBERG, G.M. (1971)

The psychology of computer programming.

Van Nostrand Reinhold, N.Y.

WEIZENBAUM, J. (1966)

ELIZA: A computer program for the study of Natural Language communication between man and machine.

Com. of the ACM. Jan. 1966, vol. 9, no. 16, pp. 36-45.

WILKS, Y. (1971)

One small head: Some remarks on the use of "model" in linguistics. CS-247. Stanford: Comp. Sci. Dept., Stanford University.

WINOGRAD, T. (1972)

Understanding Natural Language.

In Cognitive Psychology. vol. 3, (1972), pp. 1-191.

WINOGRAD, T. (1974)

Five lectures on Artificial Intelligence.

AIM-246, Stanford: Comp. Sci. Dept., Stanford University.

WONG, H.K.T. (1975)

Generating English sentences from semantic structures.

Thesis, M.Sc., University of Toronto., Dept. of Comp. Sci.

WOODS, W.A. (1970)

Transition network grammars for Natural Language analysis.

Com. of the ACM. vol. 13, pp. 591-602.

WOODS, W.A. (1973)

Progress in Natural Language understanding- An application to lunar geology.

Conf. Proc. AFIPS., vol. 42, pp. 441-450.