A computer aided instructional system for teaching formal languages

Dennis Keith Branstad
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd
Part of the Computer Sciences Commons

Recommended Citation
BRANSTAD, Dennis Keith, 1939-
A COMPUTER AIDED INSTRUCTIONAL SYSTEM FOR
TEACHING FORMAL LANGUAGES.

Iowa State University, Ph.D., 1970
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan
A COMPUTER AIDED INSTRUCTIONAL SYSTEM
FOR TEACHING FORMAL LANGUAGES

by

Dennis Keith Branstad

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
The Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

Head of Major Department

Signature was redacted for privacy.

Dean of Graduate College

Iowa State University
Ames, Iowa

1970
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>Purpose of CAI</td>
<td>2</td>
</tr>
<tr>
<td>History of CAI</td>
<td>8</td>
</tr>
<tr>
<td>REVIEW OF LITERATURE</td>
<td>13</td>
</tr>
<tr>
<td>Formal Language</td>
<td>13</td>
</tr>
<tr>
<td>Automatic Translators</td>
<td>19</td>
</tr>
<tr>
<td>Tree Structure Representation</td>
<td>26</td>
</tr>
<tr>
<td>META Translators</td>
<td>29</td>
</tr>
<tr>
<td>THE METACAI SYSTEM</td>
<td>36</td>
</tr>
<tr>
<td>Introduction</td>
<td>36</td>
</tr>
<tr>
<td>System Structure</td>
<td>40</td>
</tr>
<tr>
<td>Language Structure</td>
<td>43</td>
</tr>
<tr>
<td>User Control</td>
<td>58</td>
</tr>
<tr>
<td>Display Control</td>
<td>62</td>
</tr>
<tr>
<td>SAMPLE SESSION</td>
<td>64</td>
</tr>
<tr>
<td>Sign-on Procedure</td>
<td>64</td>
</tr>
<tr>
<td>Language Definition</td>
<td>65</td>
</tr>
<tr>
<td>Language Testing</td>
<td>85</td>
</tr>
<tr>
<td>CONCLUSIONS AND SUMMARY</td>
<td>90</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>94</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>99</td>
</tr>
<tr>
<td>APPENDIX A: METACAI SAMPLE SESSION</td>
<td>100</td>
</tr>
<tr>
<td>APPENDIX B: METACAI PROGRAM STRUCTURE</td>
<td>107</td>
</tr>
<tr>
<td>APPENDIX C: METACAI PROGRAM LISTING</td>
<td>109</td>
</tr>
</tbody>
</table>
INTRODUCTION

Computer aided instruction (CAI) has become a common term in computer science in the last decade. Several equivalent terms such as computer assisted instruction, computer based instruction, computer aided learning, and computer managed instruction have been associated with this concept. These terms have been coined by individuals interested in uniqueness, but all primarily have the same concept, that of using the electronic computer to aid learning. The effectiveness of this approach has been argued from many standpoints during this decade of research, but as yet no "giant steps for mankind" have been made in this area. It can only be said that the computer is more effective in some areas of instruction than in others.

This paper will describe some educational areas that have been investigated, the approaches used in these investigations, and some of their results. It will describe formal languages, the use of the computer in translating formal languages, and the effect of formal languages on computer science. One method of specifying a formal language will be presented in detail. A computer program which helps a student to understand the construction of a formal language translator will be outlined. Examples of simple formal languages, along with their derivation, translation, and machine implementation will be presented. The final section will
state the conclusions and results obtained in preparing and using the program.

Purpose of CAI

The purpose of computer assisted instruction (CAI) is to assist instruction; however, more can be said under this heading. Education has become one of the biggest, most expensive, and most important endeavors that an individual can undertake. Emphasis has been placed on education to such an extent that a student will undergo personal hardship to attain its rewards. The expense of a college education has doubled in the last decade and no immediate change of this trend is anticipated. A significant factor in this increase is the higher salaries that must be given qualified teachers. Since qualified teachers are scarce, search for instructional aids has spread to the area of computer applications. Hopefully, the computer can lighten the instructional burden of the instructors and the financial burden of the taxpayers.

An early use of the computer in education was in administration. The electronic computer was found to be an effective aid in record keeping, classroom assignment, grade processing, and report publication. Large universities began to obtain computers for use in research programs, both for computer research and to support work in other programs. These general-purpose machines were used for research work
during regular hours and daily administrative work during off-hours, enabling a single computer system to support several different functions concurrently. Diversification of services helped to justify the cost of the computer equipment. These services assisted education, but they did not fulfill the direct instructional assistance that is required for CAI.

The next step in CAI development began when a research group from IBM (37) used the computer as a teaching machine for binary arithmetic. The early computing machines were programmed using binary instructions and binary data entered directly into computer memory. It was therefore logical that the computer's first use as a direct instructional tool was in binary arithmetic.

Computer operating systems, the supervisory control programs which read, translate, and monitor user's programs, have undergone significant changes during the 1960's. Before this period it was customary for individual users to operate the machine themselves. The person who had written the problem program controlled the operation from the machine console. Since this method of operation was unworkable if many people wanted to use the computer, a "closed shop" approach, now called batch processing, was adopted. Professionals were hired to operate the machine. This change resulted in more efficient use of the equipment, but
the user no longer had direct communication with his operating program.

The computer was heavily used by the students during this evolutionary period. Problems now could be solved that were much too tedious for hand, slide rule, or mechanical calculation. Students in statistics, economics, and accounting used its data processing capabilities for data reduction and analysis. Students in computer science used it as a laboratory instrument in their programming courses. More student jobs could be run in a batch processing system, but rapid problem solution was not possible since the first error caused immediate program termination. Information about the program's operation in later stages was lost, requiring many computer "runs" to eliminate all the errors. Because of problems encountered in batch processing, students would often abandon an interesting problem, rather than invest the effort needed to obtain its solution. Educational benefits of the computer were lost in this environment because of the difficulty students had in communicating with the computer. Pattern recognition, error diagnosis, intuitive decisions, and other processes that are simple to a human but difficult to program had to be automatically handled in the batch system. These difficulties created a desire for new methods of computer operation and instigated new research in man-machine communications.
Time sharing

Concurrent with the increase in computer capability, the improvement of input-output equipment, and the development of sophisticated operating systems, research was proceeding in a new mode of computer operation called time-sharing. Time-sharing refers to operation of several programs in the computer system concurrently, under control of several control consoles called user terminals. Satisfactory operation of a time-sharing system required direct user communication with his program, program protection from other users, and rapid reaction to user requests. This required a complex computer, a network of terminals, and a sophisticated operating system. The time-sharing mode of operation gave the computer user a means of interacting with his program and opened the doors to successful computer assisted instruction.

Why CAI? Dr. Patrick Suppes, Stanford University, stated, "The huge information-processing capacity of computers makes it possible to use them to adapt mechanical teaching routines to the needs of the individual student" (47). Pressey, in support of his early mechanical teaching machine said, "Mechanical methods are necessary to arrange and present the contingencies of learning reinforcement. These machines can, in their systematic presentation, allow the student to learn more rapidly and effectively than in
the often unsystematic classroom environment" (35). These arguments are advanced for the computer's use as an automatic teacher. "If learning specialists and psychologists can determine the variables which represent the best input for making wise decisions, then a computer can be programmed to perform the task which a single teacher would no doubt find impossible to manage" (36). This problem must be faced in CAI, as well as in all educational methods. No general rules of learning, either for specific subjects or people, have been formulated. Individualization of instruction is a desirable goal and uniqueness of treatment should cover learning rate, motivational techniques, and instructional methods.

Many arguments, both pro and con, exist in the field of computer assisted instruction. CAI seems to be better suited to rote and mechanical areas of instruction, where simple decisions may be made by the computer program. Computation and structured information storage and retrieval are easily done in a CAI environment, and are very helpful in many types of courses. Natural language communication between man and machine is not yet available, but research is progressing in this area. Unsolicited requests or responses from the student cannot be interpreted in most CAI systems. The cost of CAI is higher than present teaching methods in similar subjects, making the method unacceptable to cost-conscious school boards. A final criticism
is that the educational control has typically been under computer program control, rather than student control.

The most successful applications of CAI have been drill and practice exercises in mathematics and foreign languages. Question presentation requiring simple answer alternatives has been implemented satisfactorily in many different systems. The branching strategy of programmed instruction has been found easy to implement in the CAI environment because of the fast random access data storage devices. Interactive computational use of the computer has met with great satisfaction in the scientific field.

An observer of a student in a CAI environment is likely to see a highly motivated individual. This motivation has been attributed to successful individualized instruction, interest in the unknown, appreciation of a reactive device, or attempts to "beat the machine". Motivation appears to be a primary argument for CAI, as motivation is a prerequisite to learning. CAI can make the necessary conditioning of drill pleasant to the student if there is an element of challenge in the presentation. Freedom from drill supervision leaves the teacher time for personal consultation.

The purposes of CAI can be summarized as:

1. Individualized instruction,
2. Motivation,
3. Record keeping and analysis,
4. Allowing the teacher more educational freedom,
5. Faster attainment of instructional goals,

History of CAI

The IBM research group began work in CAI about 1959. Their computer system was a small 650 computer and a single typewriter terminal. The entire computer facility was devoted to one student at a time. The student was charged with the entire operations cost even though he was not using the total capability of the system. This wasting of computational facilities led to research in time sharing.

An early project in time sharing took place at MIT in the early 1960's. Although this research was not initially directed towards CAI, the developed system was found to be useful in education research. Studies in natural language processing (43, 44, 45) and in artificial intelligence (51) have led to additional work in CAI student-response analysis. Project MAC, Multiple Access Computer, has had a significant effect on man-machine communication (8). This system showed the feasibility of direct computer access by many users simultaneously. The system contained one processor, capable of servicing one user instantaneously. When properly programmed, however, it appeared to service 30 users simultaneously. This multi-programmed system has greatly
influenced computer system development during the past
decade. Time slicing and program swapping algorithms
originally tested in Project MAC are being used and de­
veloped in current CAI systems.

The university and industry have both shared in the
development of CAI. Universities have been more interested
in educational objectives than in monetary returns, but both
have contributed greatly to CAI system's research. IBM and
RCA each developed complete systems for CAI applications.
Both companies have publishing subsidiaries engaged in de­
velopment of course materials. This work is difficult and
time consuming, making these materials expensive. Course
objectives, while not difficult to define, are difficult to
implement in a CAI environment. Testing and modification
of educational material must be done before its commercial
release. The CAI system must be heavily used to justify the
expense of lesson material preparation. Computer system
operation is still very expensive. Economic considerations
have minimized the commercial success of CAI systems.

Universities have had different problems implementing
successful CAI system. Under the title of research, the
economic problem can temporarily be neglected. However,
some technical problems of CAI still hamper implementation
efforts. Satisfactory communication facilities between the
student and the computer are not currently available. Type-
writers, teletypes, display devices with keyboards, slide projectors, and tape recorders have been used as computer controlled terminals. These devices each have technical disadvantages when used as student communication devices. Slow data transfer rate, lack of flexible audio and visual output, student-typed input, and high cost of communication lines have prevented wide acceptance of these devices in CAI. Computer system limitations have restricted the number of students that can use the system concurrently. Lack of theory of effective teaching methods is felt in CAI situations. The desire for individualization of instruction requires huge quantities of instructional material to be available. Students may be working on several courses simultaneously; even those in the same course may be working different lessons.

A semantic problem, misinterpretation of information passed between student and computer, is present in the CAI environment. An effectiveness problem, lack of correct instructional methods, also plagues research. This essentially is the problem of learning and measurement of that learning.

In spite of the technical and educational problems found in current CAI systems, several have found success in limited applications. Stanford University's work in elementary mathematics (47) is one of the best known. The University of Illinois' PLATO project (2) has contributed
new communication media. The Systems Development Corporation's PLANIT CAI system (11) is one of the most powerful. The student can freely mix question-answer sessions with a computational service in the language. IBM's 1500 and RCA's Educational 70 computer systems were designed for total CAI applications.

According to Dr. Zinn (56), University of Michigan, the primary methods of CAI are:

1. drill and practice: rote learning by having the student work author written and computer generated problems;

2. author-controlled tutorial: tutorial information presented by the computer to the student, followed by simple questions over the material;

3. dialogue tutorial: tutorial information presented to the student upon his request, or upon computer evaluation of past performance;

4. information handling: simple information storage and retrieval;

5. simulation and gaming: simulation of real events in computer and student interaction with the simulated event;

6. computation and display: an application in computer technology sometimes overlooked in general CAI systems;
7. Instructional management: a study of the learning process from an educational viewpoint;

8. Computer based tools: analysis of data collected from CAI experiments.

This paper will deal with the computation and display aspect of CAI. Primary emphasis will be placed on the use of the computer as a laboratory instrument in studying formal languages. Further explanation of formal language translation and interpretation will be given.
REVIEW OF LITERATURE

Formal Language

A formal language is described explicitly by its grammar, a finite set of structure rules applied to a finite set of symbols. Several definitions exist in computer literature for a formal language L, and its grammar G. The one given below will be used throughout this paper.

Let A be a finite set of characters, called the alphabet of L.

Let V be a finite set of symbols, called the vocabulary of L. These symbols are composed of concatenated members of A.

Let T be a subset of V, called the terminals of L.

Let N be the subset V-T, called the non-terminals of L. These are the syntactic types of L.

Let P be a finite, nonempty set of structure rules, called the syntax rules of L. These are the syntactic definitions of L.

Let S be one symbol of N, called the special (initial) symbol of L. S is also called the entry point of L.

A context-free grammar is defined as an ordered four-tuple \( G = (V, T, P, S) \). A context-free language L, is the set of terminal strings, called sentences, which can be produced from the special symbol S using the grammar G.
The form of $P_i$, a member of $P$, can be varied in this paper. The form used most often will be: $N_i := R_i$. $N_i$ is a member of the set $N$ of non-terminals, $:= $ is read "is defined to be", and $R_i$ is a sequence of members from $V$. $R_i$ is called the right hand side (RHS) of the syntax definition $P_i$. A single member of $N$ is always on the left hand side (LHS).

Finite sequences of symbols of $V$ are called strings (41). Let $x$ and $y$ be strings in $L$. Then $x$ directly produces $y$ ($x \Rightarrow y$) if and only if there exist strings $u$ and $v$ such that $x = uZv$ and $y = uwv$ with $Z := w$ in $P$. Generalizing, $x$ produces $y$ ($x \Rightarrow y$) if and only if either $x = y$ or there exists a sequence of nonempty strings $(w_0, w_1, ..., w_n)$ such that $x = w_0$, $y = w_n$, and $w_i \Rightarrow w_{i+1}$ ($i = 0, 1, ..., n-1$ and $n \geq 1$). Informally, a string $x$ is a sentence of the language $L$ if it can be produced from $S$ by a finite number of applications of the rules $P_i$ in the grammar $G$. A parse of a sentence $x$ is the sequence of rules $P_j$, when applied in reverse order, will produce $x$ from $S$. Thus a parse is a set of reduction rules which reduces a sentence of $L$ to $S$.

**Formal language example**

As a simple example, using a small subset of English, let

$A = \{a,b,c, ..., z\}$, the letters of the alphabet,

$V = \{\text{dog, man, bites, loves, sentence, noun, verb}\}$, the allowable
symbols or WORDS of the language L,

\[ T = \{ \text{dog, man, bites, loves} \}, \] the terminal symbols of L,

\[ N = \{ \text{sentence, noun, verb} \}, \] the non-terminals of L,

\[ P_1: \text{sentence} := \text{noun verb noun}; \] a member of V following another member of V is read "followed by a",

\[ P_2: \text{noun} := \text{man} \mid \text{dog}; \] the symbol \( \mid \) is read "or a",

\[ P_3: \text{verb} := \text{loves} \mid \text{bites}; \]

\[ S = \text{sentence}, \] the initial symbol of L.

The total language L allowed by this grammar, \( G = (V, T, P, S) \), is:

1. man loves man
2. man loves dog
3. man bites man
4. man bites dog
5. dog loves dog
6. dog loves man
7. dog bites dog
8. dog bites man

There are exactly eight sentences in this language, although there is an infinite number of finite length strings over V.

For each string in this example, the parsing sequence is \( P_2 P_3 P_2 P_1 \). Each of these eight strings is a sentence in the language.

Syntax analysis (13) is the sequential application of syntax rules \( P_i \) to a string to decide if it is a sentence.

Top-down syntax analysis begins with \( S \) contained in \( P_1 \) and continues to sequentially specified \( P_i \) until a member of T is encountered. This terminal symbol is compared with the first symbol in the string. If it matches, the symbol is "deleted" from the string, and "true" is reported for the
test. If it does not match, the alternatives in rule $P_i$ are attempted. If no alternatives match, "false" is responded for that rule. This procedure continues until either the entire string is deleted or until a "fail" is reported for the analysis. If the string is deleted, it is a sentence of $L$. Top-down analysis thus begins at the root of the "syntax tree", follows the branches to a terminal leaf, and asks, "Is this the next symbol to be deleted?" Bottom-up syntax analysis begins with the first symbol in the string, locates the syntax definition $P_j$ containing the symbol, "reduces" the sentence if possible by replacing the symbol with the $P_j$, and continues to the next symbol. This reduction process continues until either the string has been reduced to the special symbol $S$ or until no applicable reduction rule can be found. It is declared to be a sentence in the first case, a non-sentence in the second.

Bounded context grammars (17) limit the number of symbols that must be simultaneously examined for correct reduction. In the above description, reduction only can take place when all requirements of $P_j$ are met. Precedence grammars (53, 54) have precedence relations that exist between all pairs of symbols of $V$ which state explicitly when reduction can occur. If a reduction cannot occur, the next symbol of the input string is checked.

A syntax tree for the simple phrase structure grammar
of the previous example is:

root: sentence
branches: noun verb noun
nodes: leaves: man dog loves bites man dog
branches: TOP BOTTOM

The syntax tree for sentence number 1 is:

sentence

noun verb noun

man loves man

The terminal symbols are always at the leaves of the syntax tree, the special symbol S is always at the root of the tree, and the members of N, the non-terminals are at the intervening nodes. This visual representation helps to give a better understanding of the terms top-down and bottom-up.

Language semantics

The discussion this far has only covered the formal language structure, called its syntax rules. These are the rules to recognize which strings of a language are its sentences. No "meaning" of a sentence can be inferred from its structure. The meaning that is associated with each sentence is called the sentence semantics. The meaning of all the sentences of a language is called the language's semantics. The eight sentences or the previous example have
no meaning, although some may have implied meaning by their appearance in the English language. Even then, "dog loves dog" may cause semantic ambiguity.

The development of formal semantics has not kept pace with that of formal syntax. It is often necessary to express the semantics of a language in a meta-language. English descriptions of sentence meaning is an example. English is the meta-language in this example. Informal semantic description often leads to ambiguity in the interpretation of a sentence. Some attempts (12) have been made in formalizing semantics, but they have not been as successful as those in formalizing syntax.

Semantics can be associated with each sentence of a formal computer language by specifying what computer operations are to be performed. If the sentence part, A+B, of a programming language produces computer instructions to form the arithmetic sum of variables A and B, the semantics of the sentence part would be specified. This ad-hoc specification of semantics serves a useful purpose in this paper.

Specifying the syntax and semantics of a large formal language and constructing a translator for it can be a formidable task. Computer literature (5, 13, 39) describes many approaches to this task. The next section describes some of these approaches.
Automatic Translators

In its simplest form, an automatic translator can be put in the form of a block diagram:

```
Input   Program   Output
Sentences in Language 1  Translator  Sentences in Language 2
```

Linguist Noam Chomsky (6) developed the formalism of Phrase Structure Grammars during the 1950's as a tool in the study of natural languages; language 1 was natural English and language 2 was a labeled dependency tree. This tree was similar to the sentence diagrams used in English high school courses. It is well known that English is not a formally structured language, hence no automatic translator exists for it. Some approaches have been made to the problems of natural language recognition and translation (43, 44, 45, 51), but no final solution has been found.

If we restrict the discussion to automatic translators for computer programming languages, the list of papers is still formidable. In "The syntax of programming languages—a survey", Floyd (14) lists 77 bibliographic references. A general outline will be given in this section on automatic translators.
Assemblers

The first computer programs consisted of lists of numerical constants that represented machine instructions to be inserted into the computer memory and executed. To alleviate the programmer's problem of memorizing numerical instruction codes and also to handle the memory location assignment problem, translators were developed to translate from programs written with mnemonic operation codes and symbolic location names into machine language. It was found that the computer could do this quite effectively. These translators were called assemblers.

The FORTRAN compiler

Translation and interpretation of algebraic equations were attempted in the mid-fifties and shown feasible (39). The basis of this effort was the idea that computer users should be able to write programs in a familiar language. The first algebraic translator was called FORTRAN (FORMula TRANslator). The output of FORTRAN was a particular machine language. Use of this translator reduced the costs of mathematical programming, resulted in more scientific computer programming, and established the importance of user oriented languages. Little regard was given to the language's structure however, and the translation process reflected this lack of structure. Translators of this period
had to be hand tailored to the language being processed and to the target machine language.

English descriptions, flow charts, and program listings were the primary methods of describing a language and its translator. Languages were developed based on user need and existing computer specifications, not a formalized structure.

**Backus Naur form**

A major change of philosophy occurred in the ALGOL (ALGOrithmic Language) 60 Report (32, 33) when a method of formal language description was devised. This formal description covered only syntax definition; semantic definition was still handled by prose. This syntactic formalism became known as the Backus-Naur Form (BNF). \( \langle U \rangle ::= R \) is an example of a BNF statement. The elements of \( N \) were placed in brackets; elements of \( T \) were unbracketed; the symbol \( | \) is read "or"; \( R \) is the right hand side of the definition. The syntactic definition \( \langle A \rangle ::= \langle B \rangle | \langle C \rangle d \) is read "a syntactic type \( A \) is defined to be a syntactic type \( B \), or a syntactic type \( C \) followed by a terminal symbol \( d \)."

**Recursive definitions**

A syntactic definition which has the syntactic type of the LHS appearing in the RHS is said to be a direct recursive definition. A definition of the form \( \langle A \rangle ::= \langle A \rangle \langle B \rangle \) is called direct left recursive; one of the form \( \langle A \rangle ::= \langle B \rangle \langle A \rangle \) is called
direct right recursive.

In the ALGOL 60 report, the definition for an unsigned integer is \( \langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{unsigned integer} \rangle \langle \text{digit} \rangle \), read "an unsigned integer is defined to be a digit or an unsigned integer followed by a digit". This example of direct left recursion will "recognize" a string of 1 or more digits.

If a syntactic type that is used in the RHS of a definition has in its own definition the syntactic type of the LHS, the definitions are said to be indirectly recursive. Thus, the pair of definitions, \( \langle A \rangle ::= \langle B \rangle \langle C \rangle \) and \( \langle C \rangle ::= \langle E \rangle \langle A \rangle \) are indirectly recursive.

**Syntax directed analysis**

One of the earliest automatic translators for the ALGOL language was written by Irons (23), considered by many to be the father of syntax directed compiling. In his paper, he separates the processes of defining the language and translating it. In previous translators, these two phases were merged into a single program that did the entire translation. According to Iron's definition of syntax directed compiling, the syntax statements of the language are to be used directly in producing the translator. A translator can thereby be automatically produced to translate a specific language. The description language is called a
meta-language. The meta-language of ALGOL 60 was BNF. Irons' program PSYCO (Princeton SYntax CCompiler) was written for a CDC 1604 computer. It used the meta-linguistic definitions directly while translating the input string.

Table driven translators

Several different "table-driven" translators have been written (5, 50). These programs consist of a skeletal system of service routines which read syntax definitions, build syntax tables, read semantic specifications, link syntax definitions and semantic specifications, do intermediate code optimization if specified, do code selection for a specific target machine, and assemble the code into an executable object module. The number and type of tables, as well as the flexibility of input and output, vary widely among implementations. They all tend to produce a coded table for the syntax tree of the language to be translated. These translators have had commercial acceptance because only the last two phases of translation need be changed if a translator for a given language is to be executed on a different computer. The first two phases only need to be changed if a different translator is to be produced for the same computer. Language 1 of a translation process is often the set of definitions for a compiler and language 2 is a compiler.
("high-level language translator"). Thus, such translators are often called **compiler-compilers**.

**Precedence grammars**

Another group of automatic translators operates on languages described by precedence grammars. Several classes of precedence grammars have been defined. In all classes, a unique relationship exists between the members of ordered pairs of symbols of specified symbol sets. These precedence relations are written: $\prec$, $\succ$, $\equiv$, meaning "less in precedence", "greater in precedence", and "equal in precedence", respectively. The difference in precedence grammar classification rests upon which symbol sets require precedence relations. A **simple precedence grammar** has a unique relation between the members of all ordered symbol pairs of $V$.

A non-precedence grammar normally can be transformed into a precedence grammar by adding symbols until unique relations do exist between all symbols. An example by Wirth and Weber (43, 54) shows this process.

$$G = (V,T,P,S) \quad P: \quad <S>::=<H>''$$
$$V = \{S,H,1,'\} \quad <H>::='$$
$$T = \{1,'\} \quad <H>::=<H>l$$
$$\quad <H>::=<H><S>$$

The alternatives of $<H>$ are listed vertically in this example and are called **productions**.

Informally, this grammar is the definition of a string,
S, of 1's that must start and end with an apostrophe and may have nested substrings of 1's enclosed in apostrophes. A precedence matrix is a two dimensional array. Each entry in the array is the precedence relation between the symbols designating the rows and the columns of the matrix. The matrix of G appears:

\[
\begin{array}{c|cccc}
 & S & H & 1 & \text{'} \\
\hline
S & \Downarrow & \Downarrow & \Downarrow & \Downarrow \\
H & \Downarrow & \Leftarrow & \Downarrow & \Downarrow \\
1 & \Downarrow & \Downarrow & \Downarrow & \Downarrow \\
\text{'} & \Downarrow & \Downarrow & \Downarrow & \Downarrow \\
\end{array}
\]

The rules for generating this matrix are given in the reference, and will not be repeated here. There are as many rows and columns in M as there are symbols in V.

In this grammar, the problem arises that it is impossible to tell if a ' is the end of one substring or the beginning of another, causing an ambiguity in the grammar. This ambiguity is shown in the matrix by the double entry in one position. Introducing a new symbol, ", for the closing apostrophe resolves this ambiguity. The new grammar is:

\[
\begin{align*}
\bar{G} &= (V,T,P,S) \\
V &= \{S,H,1,\text{'},\text{"}\} \\
T &= \{1,\text{'},\text{"}\} \\
\bar{P} : & \quad \langle S \rangle ::= \langle H \rangle " \\
& \quad \langle H \rangle ::= \langle H \rangle 1 \\
& \quad \langle H \rangle ::= \langle H \rangle <S>
\end{align*}
\]
The new precedence matrix for $\overline{G}$ appears:

\[
\begin{array}{cccc}
M & S & H & I \\
\hline
S & \Rightarrow & \Rightarrow & \Rightarrow & \Rightarrow \\
H & \Rightarrow & \Rightarrow & \Rightarrow & \Rightarrow \\
I & \Rightarrow & \Rightarrow & \Rightarrow & \Rightarrow \\
' & \Rightarrow & \Rightarrow & \Rightarrow & \Rightarrow \\
" & \Rightarrow & \Rightarrow & \Rightarrow & \Rightarrow \\
\end{array}
\]

The reference gives further detail for precedence language translation. The method is bottom-up and deterministic, i.e. the symbols of the input string are located in the precedence matrix; a string of precedence relations among the symbols is produced; when the string of precedence relations appears as $... \leq \leq ... \geq \geq$ in a left to right scan, a reduction can occur; the syntax type of the corresponding production replaces the deleted string and the process continues. The actual implementation is discussed in the reference.

Tree structure representation

I think that I shall never see
A poem as lovely as a tree
Joyce Kilmer

A coded tree form of an expression can be used directly by an automatic translator. The entire syntax structure of a formal language may be coded as a syntax tree, or any
sentence of the language can be diagrammed as a sub-tree, i.e., only the parts of the syntax tree are used that are present in the sentence. In the expression, \( A = B \times C + D \times (E - F) \), the tree form is shown below. An expression which has the operators between the operands is in infix form. The order of evaluation of the operators is assumed to be from left to right. Parenthesized expressions, products, sums, and assignments are evaluated in that order. The parentheses are not explicitly put in the tree.

\[
\begin{align*}
\text{\( A \)} & \quad \text{\( = \)} \\
\text{\( A \)} & \quad \text{\( + \)} \\
\text{\( B \)} & \quad \text{\( * \)} \\
\text{\( C \)} & \quad \text{\( * \)} \\
\text{\( D \)} & \quad \text{\( - \)} \\
\text{\( E \)} & \quad \text{\( F \)}
\end{align*}
\]

According to Knuth (27), this tree can be "traversed" in three ways, pre-order, post-order, and end-order. To reform the original expression (minus the parentheses), the tree is traversed in post-order, defined recursively to be:

1. traverse the left subtree,
2. "visit" the root,
3. traverse the right subtree.

A parentheses-free form of this expression is called Polish postfix notation. The order of sub-expression evaluation is specified explicitly in this form. The right hand argument of any operator is the first complete expression
to its left. Both arguments precede the operator in this notation and may be evaluated easily by a pushdown stack computing machine. To obtain the expression in this form, traverse the given tree in end-order, defined recursively to be:

1. traverse the left subtree,
2. traverse the right subtree,
3. "visit" the root.

The expression may be directly read as: \( ABC*DEF-*+= \). In this example, the left hand argument of the first * is B, the right hand argument is C; for the operator +, the left hand argument is BC*, and the right hand argument is DEF-*.

A few more necessary definitions are now given. Well-formed, or un-ambiguous, formal language sentences may be recognized and translated by an algorithm. Each sentence in these languages has exactly one left-to-right parse. No ambiguity exists in their structure. Left-factorable grammars contain syntactic definitions which have identical symbols appearing at the left of two or more alternatives. These symbols may be factored to the left and written with special meta-linguistic symbols. Parentheses are used for grouping in left factoring definitions. \( <A>::=BC|BD \) will be written \( <A>::=B(C|D) \). Left factorable grammars are discussed in references 26 and 55. A similar algebraic example: \( a=b*c + b*d \) may be written \( a=b*(c + d) \).
Left factoring grammars eliminates the need for "back-up" in most cases. If the left symbol of two or more alternatives of a definition is identical, back-up will be needed to correctly translate the language. That is, the input string and the translator environment must be backed up to the point where the wrong branch of the syntax tree was taken. The referenced paper shows the conditions when back-up may be totally eliminated. A language requiring no back-up in a left to right parse is said to be deterministic.

META Translators

Top-down syntax analysis of a string can be considered an attempt to generate a language sentence using the syntax definitions of the language. A series of comparisons is then made to check if the generated sentence compares with the test string. If it does not, an alternative sentence is generated and tested. If this process is carried out in an unambiguous, deterministic grammar using left-to-right parsing, the portion of the input string already deleted need never be rescanned. This is the basis of a family of translators called META-compilers. Two members of the family, META II (42) and META PI (34) are discussed in detail in this section.

The META syntax definitions differ from BNF in the following ways:
1. All terminal symbols are enclosed in apostrophes, called "quotes" in this paper.
2. Syntactic types, i.e. non-terminals, are unquoted.
3. ::= is replaced by :=, left-factoring with grouping symbols (and ) is used, the dollar sign, $, is used as a repetition symbol, and a semi-colon follows each definition.
4. Position in the definition reflects position in the string.

Syntax definitions written this way are called META syntax form (MSF) definitions.

The MSF definition of an unsigned integer, given previously, appears: UNSGINT := DIGIT $ DIGIT:, read, "a syntactic type UNSGINT is defined to be a digit followed by 0 or more digits". The iterative operator has replaced direct recursion in the definition. The syntactic type <unsigned integer> had to be replaced by a single symbol, UNSGINT.

Semantic actions associated with the syntax may be embedded in META syntax definitions. Specifications of these actions are embedded in output statements that generate computer code. The code form differs among implementations. Either directly executable code or assembler code may be generated as the META translators recognize segments of an input string. This code, representing the semantics of the input string, may be executed immediately
or saved to be executed later.

**META II translator**

META II, described by Schorre (42), was the first published META translator. The META translator was defined in MSF. The translator output code was assembly language statements for a theoretical META machine, which was simulated on an IBM 1401 computer. The machine was designed to recognize three basic entities:

1. **Identifiers**: a letter followed by digits or letters;
2. **Strings**: any set of characters enclosed in quotes;
3. **Numbers**: a string of digits with/without a decimal point.

The META machine could recognize the basic elements of a formal grammar. Given the set of syntax rules for a language written in MSF, a translator could be produced to translate sentences of the language. Since the description of the META II translator was described in MSF, it could reproduce itself. This was actually done after the initial translator was hand coded to see if the code that was produced matched the original code when it translated "itself". The author claims that this was the case, except for some minor exclusion of redundant code that he had neglected. Using the newly generated code to retranslate the machine
definitions did, however, "reproduce itself". Disconcert-ing as this may be, it is a powerful feature of the system.

META PI translator

META PI (34) was implemented at RCA on a RCA SPECTRA 70/45 computer. This system also was based on META syntactic definition form. Both a batch mode and a time-shared, interactive version were implemented. META PI had a more powerful set of syntax checking routines than META II, but the output code was absolute-address machine language. This code was directly executable. The translator was very fast, but the entire burden of numerical machine code generation was placed on the user.

Both translators made use of recursively coded subroutines derived directly from the syntax definitions of the META language. Each subroutine was given the name of the syntactic type that was being defined. The appearance of a syntactic type in any syntax definition would generate a subroutine call to check for that syntax type. Because the subroutines were recursive, the definitions could be both directly and indirectly recursive.

Assignment statement grammar

Consider the MSF grammar for algebraic assignment statements like the example previously used, \( A = B \times C + D \times (E - F) \). The definitions of \( G \) and \( V \) can be implied and will be eliminated
in subsequent examples.

$$T = \{ \text{'A', 'B', 'C', 'D', 'E', 'F', '+', '-', '*', '(', ')', '='} \}$$

$$S = \text{ASSNSTAT}$$

$$P : \text{ASSNSTAT} :: \text{VARIABLE} = \text{EXPRESS};$$

$$\text{VARIABLE} ::= \text{'A'|'B'|'C'|'D'|'E'|'F'} ;$$

$$\text{EXPRESS} ::= \text{FACTOR} ((\text{'+'|'-'})\text{FACTOR}) ;$$

$$\text{FACTOR} ::= \text{SIMPLE} (\text{'*'}\text{SIMPLE}) ;$$

$$\text{SIMPLE} ::= \text{VARIABLE} | ('\text{EXPRESS}') ;$$

The reader should verify that the "recognizer" can analyze the sample input string, finally declaring it a sentence in the language. The recognizer (a translator which generates no output language) always begins at $$S$$, the special symbol. An appearance of a syntactic type causes transfer of the recognizer to that definition. When a terminal symbol is encountered and that terminal is the "current" one in the input string, i.e., the symbol under current consideration in a left to right scan of the input, the symbol is marked "deleted" and the symbol to its right becomes "current". Any alternative must be entirely satisfied for a "true" to be reported for that type; all alternatives of a definition must be tried before a false is reported. In words, the recognizer proceeds as "An ASSNSTAT is a VARIABLE which is an 'A' (matches A of input, delete it) followed by a '=' (matches = of input, delete it), followed by an EXPRESS
which is a FACTOR which is a SIMPLE which is a VARIABLE which is an 'A' (no) or a 'B' (matches input, delete B) followed by 0 or more groups of '*' (matches input, delete *) followed by a SIMPLE which is a VARIABLE which is an 'A' (no) or a 'B' (no) or a 'C' (matches input, delete C), check for another '*' (no, but still a FACTOR), followed by 0 or more groups of '+' (matches input, delete +) followed by a FACTOR, ..." This top down syntax recognizer will finally delete the entire string and report that it was indeed a sentence of the language. These syntax definitions only generate META code that can recognize the sentence of the language. A later example shows semantics attached to the grammar.

**META grammar**

The basic grammar of a META syntax definition can be given in MSF as follows:

\[
T = \{'.ID', '.STR', '.', '::', '|', '|', '|', '\\', ':', '(', ')', '}', 'S' \}\n\]

\[
S = \text{METADEF}
\]

\[
P : \text{METADEF} := '.ID':='METAl';'
\]

\[
\text{METAl} := \text{META2} ('| '..\text{METAl}') ;
\]

\[
\text{META2} := \text{META3} \text{ META3} ;
\]

\[
\text{META3} := '.ID'|'.STR'|'.ID'|'.STR'|'(\text{METAl}')'|'\$\text{META3} ;
\]

Letting '.ID' be the identifier recognizer, and '.STR' be the string recognizer, the reader should verify that each of the four definitions of P can be recognized by this grammar,
thus proving that they are legal META language sentences. Every symbol appearing in these definitions is either a syntactic type or a terminal symbol. The organization of the definitions is dependent upon the semantics associated with the definitions. An explanation is left until the next chapter.

The output code of a META translator is a function of the "machine" that is to execute it. META II output META assembler code that was simulated on a 1401 computer. META3 output assembler language code for the 7090 computer. This was assembled and executed by the normal operating system of the computer. META PI output pure machine code for the SPECTRA 70 computer. META 3 and META PI contained fixed basic translators. META II was a programmable "machine" containing basic instructions with which any translator could be built.
THE METACAI SYSTEM

Introduction

The META Computer Aided Instruction (METACAI) system is based on the META series of translators but has been designed in a computer-aided instructional environment. It has evolved through several approaches to the interaction problem of CAI. The remainder of this paper will describe the system's development, its operational qualifications for CAI, and some of its possible uses and extensions.

This work began as a search for a tool which would allow an instructor or a student to define and test formal computer languages. It was hoped that the resulting system could also be used to study the natural language communication problem between student and computer. This was not a design criterion however. As Lindsay (29) says, "...a procedure which is appropriate for a natural language translation program will probably be unnecessarily elaborate in processing unambiguous computer languages as required in syntax directed compilers." Primary emphasis was placed on the CAI aspect of the METACAI system. The student was given primary control in designing and testing a language. He could easily follow its development from the language specification phase to the testing of the resultant code.

The META series of translators was chosen as the basis
for the METACAI system for several reasons. The META syntax form (MSF) is very similar to the classic BNF, but the META machine grammar may be translated by its own translator because of the differences between BNF and MSF. The operation of deterministic translators is easy to follow. The trace of sentence recognition and code generation is easy to understand. Errors occurring during translation can be easily pinpointed. The basic translator structure may be modified by changing the translator's syntactic definitions. Finally, embedding semantic meaning in the syntax specifications makes language semantic definition easy.

The initial task in creating a suitable interactive META translator was to test the available META II system written for the IBM 360/65 computer by R. A. Sharpe. This system consisted of two parts; an assembler and an interpreter. Both programs were written in PL/I. The bootstrap translator was created by encoding the META II syntax definitions in a set of PL/I recursive procedures. The bootstrap translator generated the assembler program for the META II machine from the META II syntax definitions. When assembled and loaded into the interpreter, a META II machine was created. This system was abandoned because of the separate steps of compiling, assembling, and interpreting the entire set of user syntax statements, making it too slow for interactive use.
The next approach in creating the translator was to test the META PI batch processor implemented by R. A. Sharpe. Two versions had been done; one in PL/I and the other in System 360 assembler language. The PL/I version was used to "bootstrap" the assembler code that was used in the assembler version. The assembler version output was directly executable 360 machine language and was loaded directly into internal tables. This translator was extremely fast; the entire FORTRAN PI defined by O'Neil was generated in 3 seconds. A great amount of effort was spent studying the qualifications of META PI for CAI and it was found that several basic problems existed in the system. All the output code from the translator had to be basic 360 machine code. Linkage between the standard META PI primitives and user defined primitives was nonexistent, making extensions to the system impossible. The entire META PI program was written in 360 assembler language and the source program was big and difficult to handle. Finally, no interactive version of META PI was available and thus this program was not satisfactory for a CAI environment.

In search for an interactive, interpretive system required for CAI, the Conversational Programming System (CPS) was next investigated as a basis for a META translator. CPS is an interactive system using the IBM 2741 typewriter console as the user terminal. It supports two programming
languages; a PL/I subset and BASIC. A META II interactive translator was written in the former language in such a way that each META syntax definition was independent of the others. This allowed the user to selectively delete, replace, or add definitions without recompiling the entire system. However, since CPS is an interpretive programming language, and META II was an interpretive system, the resultant system was found intolerably slow. Eighteen minutes of real time were needed to translate the seven definitions of the META II grammar. The maximum amount of space that CPS allowed each user was quickly used, allowing no expansion. This effort was abandoned.

The final version of development began by rewriting the CPS version in a PL/I batch version, maintaining the ideas of statement independence, user controlled edit facilities, and openendedness of the language. A subset of the VALGOL II machine language described by Schorre (42) was implemented and tested as the first extension.

An important decision of the METACAI implementation concerned user communication with the system. The only feasible interactive devices on the existing system were the 2741 typewriter and the 2260 display terminal. The 2741 terminals were currently being used as CPS terminals and were not available for other programs. The 2260 display terminal was selected as the METACAI communication
device because of its simple communication with a PL/I pro-
gram and its rapid display of textual material. A Multiple
Terminal Monitor Task (MTMT) utility program was available
as a 2260 terminal communication sub-system. The MTMT ser-
vices include user program storage and retrieval, the capa-
bility of creating, editing, and deleting user data sets,
a communication interface between the user and his program,
and the ability to call and execute system utilities from
the display console. The MTMT program operates within one
region of the IBM Multiple programming with Variable sized
Tasks (MVT) operating system. It uses the operating sys-
tem's subtasking support to control several 2260 terminals
concurrently, all within the single region of 360 memory.
Several users may operate independent programs within MTMT,
or several terminals may use a single re-entrant program
(a read-only program in which each user has a copy of all
necessary variables).

System Structure

The interactive METACAI system consists of two types
of commands; system control commands and METACAI instructions.
The system control commands are used to load, edit, save, or
delete language grammars and translators. There are 19 sys-
tem commands of the form #Comand or #C0argument. Only the
first two letters of a command are used to recognize the
command.
The argument varies among commands. There are 76 METACAI instructions which are simulated by the METACAI interpreter. METACAI symbolic assembler instructions may be generated during the translation of an input string, and must be grouped in the form of a closed subroutine. A subroutine group of assembler instructions can be assembled (translated to METACAI machine code) and executed by the METACAI interpreter. Detailed descriptions of the system commands and the METACAI instructions will be given in the next two sections.

The METACAI system consists of one PL/I program with two primary procedure levels designed to allow user defined array sizes. The outer procedure is used to give descriptive HELP material to the user, to open required user data sets, and to allow the user to define the anticipated array size requirements for this terminal session. The METACAI program is coded in a RE-ENTRANT manner so that more than one user can operate it simultaneously. The entire program must therefore be in memory throughout the session. Each user has a private copy of all program variables.

The sizes of some arrays may be defined by the user from the terminal. These array size parameters are passed to the inner procedure where the required storage is obtained from free space in the MTMT region. The size of this region limits the number of users as well as the array sizes.
Requests for too much memory aborts the program and gives the user an error message.

The inner procedure of METACAI is the operational level of the system. It contains the unified METACAI assembler/interpreter and executes the systems commands. A detailed description of this procedure will be given in later sections.

Each system user needs a private data set to save information from one session to the next. This data set is organized as a direct access regional file with 960 character records and is kept on a permanent system disk file. This organization was chosen so that the file could be randomly referenced and because the 2260 terminal displays 960 characters at once. Due to system restrictions, only one permanent data set is used by the METACAI system for each user. This one data set is used to store several things: system "help" frames (one display screen of information), user syntax definitions, translator programs ready for execution, array size parameters, and user generated records. Primary maintenance of the data set records is done by the program. Three utility programs are available to create, save, edit, and print these data sets when they are not being used by the METACAI system.

The METACAI system uses an additional temporary output data set for each user. This data set is used for PL/I
error messages and for a permanent student record. It may be printed using an MTMT utility print program.

Language Structure

The METACAI program passed through several phases of organization, but the final one is of a unified structure. The operational section is contained in one procedure. Only one assembler and one interpreter exist for the entire system. User program protection was sacrificed for flexibility of translator definition. Any portion of the basic META translator may be modified, and the program may be reloaded if the modification leads to an irrecoverable position.

The METACAI program uses an array called MEM as the simulated METACAI machine memory. MEM contains the instructions for the current translator being used, as well as assembler-produced code. The initial translator must be loaded from the program library on the user data set, CAIFILE. This initially loaded translator may be the basic translator of the METACAI system or one that the user has produced during a previous session. Code output during successful translation is assembled by the METACAI assembler and the resulting code is loaded into MEM. Control may be passed to this new code by one of the control commands or by executing a subroutine call to the name of the subroutine. Since subroutines may be deleted or replaced during system
operation, a new translator can be produced. The unified structure of the system allows MEM to contain a compiler-compiler, a compiler, a program, or results of the program simultaneously. Previous system designs separated these program categories, but it was found desirable to be able to freely intermix them. Thus complete communication among the classes is now possible.

Each syntax definition, or program segment, must be an individual entity in the form of a closed subroutine. Communication is passed from one subroutine to another via a call to the name of the subroutine. This was done so that a single definition or segment could be modified without changing the rest of the program. The simulated METACAI machine has a relocatable machine language instruction format, making program relocation easy. Every instruction using an operand local to the subroutine uses a relative address. Instructions using a global variable or a subroutine name reference it through a symbol table. This allows relocating code in MEM by simply changing the global variable pointers in the symbol table. Memory fragmentation problems are eliminated in this way.

User syntax definitions reside in an area of the CAIFILE data set. During a student session, these definitions may be edited and translated. If a definition or program segment is successfully translated, and assembler
code is produced, this code is automatically assembled into machine language, and automatically loaded into MEM. Selective monitoring of this process can be made on the display terminal.

The METACAI assembler

Since the semantic output of a METACAI translator is assembler language, a simple assembler program had to be provided. It is a two pass assembler. The code output during a syntax statement or program segment translation is stored in INP, a 2 dimensional character array. During the first assembler pass of INP, labels of the form #NN or @NN are recognized. The two digit integer NN is used as a pointer into a label table where the current address counter is stored. These labels are called internal labels. The # labels are generated by special label generating instructions; the @ labels may be created by the user. NN may range from 01 to 99. All alpha-numeric labels are placed in a global label table. A block reserve instruction, BLK NN, reserves a block of NN memory locations for program variable storage.

During pass 2, the assembler uses a binary chopping search algorithm to find the symbolic instruction in the OPCODE table. Its position in the table yields the machine language instruction. These symbolic operation codes are
stored in the user data set and may be changed, but must be alphabetized for the binary search routine. If a symbolic operator is not in the table, the next available numerical operator code is assigned to the unlisted symbol. The symbol and its assigned numerical code are placed in an undefined operator table. If it is later encountered in the session, the same "OP" code is assigned.

The METACAI machine instruction format is $XXYYYZZ$, a seven digit integer. ZZ is the instruction (OP) code, YYY is a relative address, and XX is a character string length, if one is necessary. YYY is an offset-500 address; 523 is an address 23 locations after the current location, 477 is an address 23 locations before this location. If an instruction was 754463, the instruction is number 63 and its argument is a character string 7 long, beginning 44 locations after the current one. An OP code is always present, but the others depend on the instruction.

The assembled machine instructions output by the assembler are put into a temporary pseudo-memory called CORE. If a translation and assembly process is entirely successful, the contents of CORE are moved into MEM, the global variables and names are placed in the global symbol table, and a translation completion message is sent to the user. If the optional program trace is active, the statement number, assembler language statement, relative address, and numerical
instruction are displayed, 10 lines at a time.

The METACAI interpreter

The METACAI machine is a simulated single address computer having auxiliary numerical and character string push-down stacks. It has numeric and character handling ability, facilities for input-output to the terminal and to the CAIFILE data set, and recursive subroutine calling capability. The METACAI interpreter sequentially fetches instructions from MEM, dividing them into the instruction code, address, and string length. If the numeric stack was used during the last execution cycle, the top two elements of the stack are put into temporary variables. This action is to optimize simulation and to do indirect addressing to MEM if required. An instruction trace can be energized to display interpreter operation. If the trace is ON, the location counter, instruction, stack pointer, and top two elements of the numeric stack are output. The location counter is incremented and the instruction is "executed". Simulation continues until the main sub-program is "exited", whereby control passes to the METACAI control system.

Output during the simulation is sent to an output procedure for permanent record and for immediate viewing, if desired. All input to the program comes from the display terminal.
The following is a detailed description of the METACAI instructions. Some of them are identical to those described in META II. The instruction set has been greatly expanded, however, and some of the functions have been modified. The implementation for this interactive system is totally different from the batch META system.

The current METACAI machine has 19 instructions that are used primarily for translation. It should be noted that any of the 76 present instructions are available for execution at any time. The division of instructions into groups is purely logical, not practical.

**Input test instructions**

**A1. ID:** recognizes an identifier, a string of letters and digits with an initial letter (8 character maximum). If the input is an identifier, the SWITCH flag is SET, the identifier is "deleted" and stored in Q, a variable length character variable.

**A2. SR:** recognizes a string of characters enclosed in quotes (24 character maximum, including the quotes). If a string, the same actions as in A1 are taken.

**A3. TST 'string':** tests for the specified string in the input. If the correct string, same actions as A1.

**A4. NUM:** recognizes numbers of the form INTEGER or
INTEGER where an INTEGER is a string of digits. If a number, same actions as A1.

Copy instructions

B1. CI : copies the "deleted" input found in Q into the assembler code line, beginning in the position marked by the variable COL. COL is incremented by the length of the deleted input.

B2. CL 'string': copies the literal character string between the quotes into the code line. COL is incremented as in B1.

B3. B label: branches to the instruction at the label; the label may be an internal label (#NN or @NN) or an 8 character global label.

B4. SET: sets SWITCH to be TRUE. SWITCH is normally used to show that a test was true or false.

B5. CMS: sets SWITCH to be FALSE.

B6. BT label: branches to label if SWITCH is SET to TRUE. See B3.

B7. BF label: branches to label if SWITCH is FALSE. See B3.

B8. BE: branches to an error routine if SWITCH is not SET. Outputs the input line and an error message showing position of syntax error.
Assembler output instructions

C1. LB : sets the COL variable to 1. Column 1 of the code line is the label field for the assembler.

C2. OP : sets the COL variable to 10. Column 10 of the code line is the operator field for the assembler.

C3. ARG : sets the COL variable to 16. Column 16 of the code line is the argument field for the assembler.

C4. GN1 or GN2 : generates a local label of the form #NN. If either is executed more than once in one subroutine level, the same label is generated. Two unique labels can therefore be generated at any one level. Each label should be generated at least twice: once for the label field, and at least once for the argument field.

C5. OUT : outputs the assembler code LINE into an array called INP, clears LINE, and sets COL to 10.

C6. CLL name : executes a recursive subroutine call to name (8 character global identifier). The subroutine call stack is "pushed" by four locations; two for holding generated labels (GN1 and GN2), one for KPOS (pointer to current subroutine), and one for LOC, the current location counter. LOC is reset to the address of name, and simulation execution continues from there.

C7. R : return from a subroutine. The subroutine call
stack is "popped" by 4, LOC and KPOS are reset from the stack. If the stack is empty, the interpreter stops, and control returns to the METACAI control system.

These instructions are sufficient to do syntax checking and code generation for any META translator. See Appendix B for the METACAI program structure and Appendix C for the METACAI program listing.

**METACAI arithmetic instructions**

Many additions and refinements have been made to the interactive METACAI machine. Arithmetic operations use a numerical push-down stack mechanism. In an infix notation, arithmetic expressions may be translated into code for a stack machine by proper definition of the syntax definitions. The addresses of the operands should be entered into the stack as they are encountered in a left-to-right scan. The intervening operator must be saved and output after the operands are entered. Arithmetic expressions in post-fix form are in proper order for stack evaluation.

The METACAI numeric stack may be referenced either as an integer or a floating point vector, called MSTACK and STACK, respectively. Numerical quantities are always handled in floating point, and addresses as integers. If an arithmetic operator uses stack arguments, the address of an argument is automatically replaced by the argument before the
operation is executed. A stack pointer, ISTKP, points to the top element of the stack, TOS. The element in the stack immediately below this one is called NTOS, next to the top of the stack. A PUSH of a number means putting the number on top of the stack, pushing the stack down by 1. A POP means removing the top element of the stack.

The character push-down stack is a simple character string with an associated vector of pointers. When a character string is placed into the stack, it is merely moved into the string, beginning at the current position pointer. The length of the string, when added to the current position pointer, yields the next position for possible string entry. This value is entered into the pointer vector. The instructions operating on these stacks are described below.

**Numeric operator instructions**

D1. LD variable : PUSHes the address of the global variable on top of the stack (TOS).

D2. LDL number : PUSHes the floating point number on TOS.

The number may be of form INTEGER or INTEGER.

D3. SS0 : PUSHes a 0.0 on TOS. This is also BOOLEAN FALSE.

D4. SS1 : PUSHes a 1.0 on TOS. This is also BOOLEAN TRUE.
D5. *SST : saves the TOS in a variable, STACK1, and stores it at the address which is next to the top of the stack (NTOS). POPs 2 elements from stack.

D6. ST : stores STACK1 in address on TOS. POPs 1 element.

D7. RSR : PUSHes STACK1 on TOS.

D8. SSR : POPs the TOS into STACK1.

D9. *ADS : adds the TOS to contents of address in NTOS and puts in STACK1. Stores STACK1 in address in NTOS and POPs 2 elements.

D10. **ADD : replaces top two elements of stack with their sum.

D11. **MLT : replaces top two elements of stack with their product.

D12. **SUB : replaces top two elements of stack with their difference. The TOS is subtracted from NTOS.

D13. **DIV : replaces top two elements of stack with their quotient. The NTOS is divided by the TOS.

D14. NEG : changes the algebraic sign of TOS.

D15. WHL : truncates the TOS and replaces it in stack.

* The instruction expects a number to be in the stack. If it is an address, the number in that address replaces the address.

** The instruction expects two numbers in the stack. The same action as * occurs for each.
D16. **LEQ : If NTOS < TOS, replace them with 1.0, else a 0.0.
D17. **LES : If NTOS < TOS, replace them with 1.0, else a 0.0.
D18. **EQU : If NTOS = TOS, replace them with 1.0, else a 0.0.
D19. NOT : If TOS = 0.0, replace it with 1.0, else a 0.0.
D20. **AND : If TOS and NTOS are both not 0.0, replace them
     with a 1.0, else a 0.0.
D21. **OR : If either TOS or NTOS is not 0.0, replace them
     with a 1.0, else a 0.0.
D22. **XOR : If TOS is not equal to NTOS, replace them with
     a 1.0, else a 0.0.
D23. BE0 label : Branch to label if TOS is 0.0. See B3.
D24. BNO label : Branch to label if TOS is not 0.0. See B3.
D25. BEP label : Branch to label if TOS is 0.0. POP 1
     element. See B3.
D26. BNP label : Branch to label if TOS is not 0.0. POP 1
     element. See B3.
D27. *AIA : increment the array address in NTOS by TOS and
     replace them with the resultant address.
D28. FLP : interchange the TOS and NTOS.
D29. POP : POP 1 element from stack.
D30. RED : inputs 80 characters from the terminal input
     buffer, and "reads" TOS numbers in an F8.4 format,
     storing them in a vector of TOS locations begin­
     ning at the address in NTOS. A maximum of 10
     numbers may be read. POP 2 elements from stack.
D31. *WRT : writes TOS numbers from address in NTOS in a F8.4 format. Numbers are put in an output line beginning at the position contained in IOUTCOL. POP 2 elements.

D32. EDT 'string' : moves the string into the output line beginning at the position contained in IOUTCOL. IOUTCOL is incremented by the length of string.

D33. PNT : moves the output line to the terminal output area, clears the output line, and resets IOUTCOL to 1.

Character handling instructions

E1. SAV : saves the last deleted string contained in Q in the literal (character) stack.

E2. SAV 'string' : saves string in the literal stack.

E3. RES : restores the top element of the literal stack into Q. This stack is POPped by 1.

E4. SWP : interchanges the top two elements of the literal stack.

E5. CON : concatenates the top two elements of the literal stack into one element and leaves it on top of the literal stack. The number of literals in the stack, LITN, is decremented by 1.

E6. PLS : POP the literal stack. LITN is decremented by 1.
CAIFILE data set instructions

F1. *GET : reads record number TOS from CAIFILE into the 960 character variable, FRAME. POP 1 element.

F2. HLP : displays FRAME on the terminal screen. This may be edited and is rewritten into FRAME when ATTENTION is typed by the user.

F3. UND : Finds a free record in CAIFILE and PUSHes this number.

F4. PUT : puts FRAME in record number TOS of CAIFILE.
Caution: can rewrite any record. POPs one element.

Miscellaneous commands

G1. ERR 'string' : IF SWITCH is not SET, string is output to terminal. Execution continues in either case.

G2. MSS 'string' : string is output to terminal, beginning in position 1.

G3. LAT name : saves the input position pointer, I, and output code position pointer, INNOO, and then executes a CLL name. Sets LATCH true. This allows 1 level of back-up in a non-deterministic language.

G4. CAN : cancels the latch mechanism. Sets LATCH false.

G5. ENT : Sets IOUTCOL to TOS. POPs 1 element.
G6. RET: returns from entire METACAI system to a calling program. Lets METACAI be a sub-system.

G7. XEQ name: restarts the interpreter at name: simulates control command #EX name. Allows automatic restart of another translator or program execution.

G8. ASM: initiates the assembly of any assembler code generated during program execution. Initiates immediately the normal operation done after successful interpreter exit.

G9. BL1: turns the program trace flag ON. Simulates action of control command, #TN. Displays trace of translator subroutine level, "deletion" of input, assembler code output of translator, and assembler routine output. The control command, #TF turns the trace flag OFF.

G10. LL1: turns the instruction trace flag ON. Displays trace of interpreter action. Displays LOC, the location counter; NOP, the numerical OP code; ISTKP, the numeric stack pointer; TOS and NTOS.

G11. LL0: turns the instruction trace flag OFF.

G12. DO: immediately executes the symbolic instruction currently contained in Q. Any instruction that needs no argument can be executed. Allows execution of instructions during translation without going through assembler.
More instructions may be added to the METACAI system by the automatic undefined OP code assignment mechanism and the recompiling of the METACAI PL/I program to correctly interpret the instruction.

User Control

The user control commands act as an operating system to METACAI. The following commands are implemented in the present METACAI system. Only the first two letters are necessary.

Program commands

H1. #PUBlish: displays the number and names of the programs available for interpretation in the user library.

H2. #LO name: loads the program, name, into MEM. If name is not found, #PUBlish is automatically executed. The SYNTYP vector is filled with the syntax types and global variables in the program. The first name in SYNTYP is made the current ENTRY point for the interpreter. SYNADX corresponds to SYNTYP and contains the address pointers.

H3. #AD name: adds the program, name, onto the end of the current program in MEM. Same action as #LO but MEM is not initialized. Allows program segmentation.

H4. #SA name: saves the current contents of MEM on the
CAIFILE data set and puts name in the program directory. Use this to save programs from session to session and to create backup copies of programs during a session.

H5. #ER name : erases the program, name, from the program directory and releases its space used in CAIFILE.

MEM editing commands

I1. #LIst : Displays the syntax type names and global variables currently contained in SYNTYP.

I2. #DE namel-name2 : deletes the area of MEM from namel to name2. Namel must come before name2 in SYNTYP. MEM is compressed to recover this space and the pointers in SYNADX are modified to reflect this compression. Displays the new list of names and variables.

I3. #DE namel- : deletes the area of MEM after namel. Displays the new list of names and variables.

Syntax definition display commands

J1. #SH : shows the first frame of user definitions.

J2. #SH NN : shows frame NN of user definitions, NN=1 to 20.

J3. #FR NN : frees frame NN of user definitions, NN=1 to 20. Erases screen and lets user begin new language definition.

J4. #FOrward : displays next frame of definitions.
J5. #BAck : displays previous frame of definitions.

**Interpreter control commands**

K1. #EN name : sets the entry point to name for a future compile command. Name is the special symbol in a formal grammar.

K2. #CO NN : starts the interpreter in a compile mode beginning at the entry point. The input to the translator begins on line NN on the frame being displayed, NN=1 to 10.

K3. #EX name : starts the interpreter in an execute mode, beginning at name. Input is requested from the terminal if the user program requests it. The only difference between an execute and a compile is the initial input step in a compile.

**Translate trace commands**

L1. #TN : turns program trace ON. Displays trace of subroutine position, "deleted" input, and code generation. Normally ON for the student user.

L2. #TF : turns program trace OFF. Stops displaying trace of program activity. Output still goes to the temporary SYSPRINT file. May be executed any time during traced output. Error and completion messages are always displayed.
**System sign-off commands**

M1. `#CANCEL`: immediately cancels program and returns to MTMT.

M2. `#OFF`: stores current system environment onto CAIFILE and returns to MTMT. This is the normal sign-off command and **must** be used if the program library has been changed.

M3. `#%CANCEL`: emergency MTMT cancel command which cancels entire terminal activity. Should be used only in case of an interpreter malfunction.

**User help request commands**

N1. `#HELP`: displays the help frames describing system action. Contains concise description of system commands.

a. `#FORWARD`: displays next help frame when in help mode.

b. `#BACK`: displays previous help frame when in help mode.

c. `ATTENTION`: typing shift and enter returns to control mode.

N2. User may sign off the METACAI system and call the CAIMARK system for detailed information on formal languages and METACAI instructions. MTMT space restrictions made this necessary.
These systems commands give the user control while he is defining, editing, and testing his syntactic and semantic definitions for a simple programming language.

Display Control

The IBM 2260 display terminal consists of a 6" x 10" cathode ray tube (CRT) display and a keyboard. The terminal is attached by cable to a communication device which contains a 960 character display buffer, refreshes the display picture, updates the picture with typed input, and communicates the information to the computer. No computer time is used while information is being passed between the terminal and the display controller.

The 960 characters of the display buffer appear on the CRT as 12 lines of 80 characters. Typing position on the display screen is controlled by a non-destructive cursor. This is a pointer, controlled by special keys, which may be moved on the screen without destroying displayed information. The keyboard characters include the 26 letters of the alphabet, the digits 0 through 9 and punctuation or control characters. A total of 64 characters can be displayed. The display control characters and their keys are:

1. Non-destructive cursor: a small vertical bar | appearing below and immediately to the left of the next character position to be filled. The space and backspace keys move the cursor right and left:
the up and down keys move the cursor one line up or down. If the cursor goes off one edge of the screen, it reappears on the opposite edge.

2. New line symbol: a symbol appearing as | marks the end of the line, moving the cursor to position 1 of the next line.

3. End of message symbol: a symbol appearing as  causes an ATTENTION interrupt to the computer program. The information appearing on the display is read by the program. The ATTENTION interrupt is caused by depressing the ENTER key while holding the SHIFT key. Both the new line and end of message symbols are read as blanks and may be placed in any position that may be logically blank.
SAMPLE SESSION

Sign-on Procedure

The IBM 2260 display terminal is turned on by pulling out the round knob on the right side of the display. After a 15 second warm up, the MTMT sign-on frame (Figure 1) is presented. The user's name, account number, and key are typed on the designated lines. Whenever input is complete and program operation is to continue, the user should depress the SHIFT key and strike the ENTER key. This is called an ATTENTION command. An ATTENTION command after typing the accounting information causes an information frame about MTMT to be presented. Another ATTENTION will display the MTMT option menu (Figure 2). Entering 4D as the desired function will display the data set definition frame (Figure 3). The SYSINxx DD name should be changed to the user's METACAI data set as shown in the figure. If the operation is successful, a * will replace the A. An ATTENTION returns the user to the option menu again.

METACAI should now be entered on the option line with J placed on the "ADDITIONAL SPECIFICATIONS" line (Figure 2). The METACAI program is loaded into the MTMT region from the JOBLIB data set and the first frame of METACAI directions (Figure 4) is displayed by the program. If the data sets have not been defined as requested, typing #CA will cancel
the program returning to MTMT where this may be done. An ATTENTION command causes the program to proceed.

The CAIFILE and SYSPRINT data sets are opened and record 0 (Figure 5) is read from CAIFILE into FRAMEP. This record contains the status of the 100 records on CAIFILE. If the number in position N is 0, record N is free. If the number in position N is I, record N is found at position I of the data set. Normally, I=N. Record 1 (Figure 6) contains the symbolic OP codes for the assembler, and is read into OPCDS. Record 2 (Figure 7) contains the user array-size parameters. This frame is displayed so that the user may change the array sizes if he desires. Record 3 (Figure 8) contains the program directory including the number of available programs as well as their names and initial record pointers. The inner procedure CAISYS is entered and the variables and arrays for the METACAI program are obtained from free storage. The METACAI idle frame (Figure 9) signifies that the system is ready for user operation. If the user is unsure of the systems commands, issuing the #HELP command places him in the help mode and Figures 10 to 16 are displayed.

Language Definition

How does one design a new formal language? The first question to be answered is, "For what will the language be
PLEASE ENTER THE FOLLOWING ACCOUNTING INFORMATION AND THEN CAUSE INTERRUPT

BRANSTAD

I1234

CAIKE

*** NEW VERSION OF MTMT NOW IN USE ***

FIGURE 1. MTMT SIGN ON DISPLAY. THIS IS TERMINAL NUMBER 1.

SELECT BY NUMBER ONE OF THE OPTIONS BELOW OR SUPPLY PROGRAM NAME AS APPROPRIATE

0. LOGOFF
1. ADDITIONAL TERMINAL SERVICES
2. DISPLAY SEQUENTIAL DATA
3. EDIT SEQUENTIAL CARD IMAGES
4. DATA DEFINITION AND RESOURCE OPTIONS
5. INPUT SEQUENTIAL CARD IMAGES
6. INITIATE A BACKGROUND JOB

1 IS THE OPTION OR PROGRAM CHOSEN

-------(ADDITIONAL SPECIFICATIONS)

FIGURE 2. MTMT MENU OPTION DISPLAY. THIS MENU IS USED TWICE, OPTION 4D TO CHANGE SYSIN01; THEN METACAI WITH ADDITIONAL SPEC. J TO CALL PROGRAM.
**T#01 WELCOME TO THE META-CAI SYSTEM.**

# - : TYPE A COMMAND

This system is designed as a laboratory tool for understanding formal computer languages. You are expected to have some knowledge of the system, as it is a powerful, but complicated system. You should have changed the SYSINXX DD name to be your METACAI data set. If you did not, type #CANCEL (only #CA is necessary) to return to MTMT and use option 4D to change this. SYSOUTXX is used for PL/I error messages and must be a scratch output data set (normally MTMT2260.SYSOUTXX). If you are unfamiliar with METACAI, or at any time need help, type #HELP and directions will be given. Otherwise, hold the shift and hit the enter keys. Proceed.

---

**FIGURE 3. MTMT DATA SET DEFINITION DISPLAY. THE 'A' SIGNIFIES A CHANGE.**

**FIGURE 4. METACAI WELCOME FRAME. THE 'XX' MEANS 01 IN THIS CASE.**
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
</tr>
<tr>
<td>61</td>
<td>62</td>
<td>63</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
<td>96</td>
<td>97</td>
</tr>
</tbody>
</table>

FIGURE 5. METACAI RECORD USAGE FOR MTMT2260.METACAI DATA SET. NEVER SHOWN.

---

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD ADD ADS AIA ALL AND ARG ASM B BE BEP BEO BF BLK BNP BNO BT CAN CI CL CLL CLR CON DIV DO EDT ENT EQU ERR FLP GET GNI GN2 HLP ID LAT LB LD LOL LEQ LES LLO LL1 MLT MSS NEG NOT NUM OP OR OUT PLS PNT POP PUT R RED RES RET RSR SAV SET SLS SR SSR SST SS0 SS1 ST SUB SWP TST UND WHL WRT XEQ XOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 6. METACAI LIST OF CURRENT SYMBOLIC ASSEMBLER INSTRUCTIONS. NEVER SHOWN.
T#01 CORE REQUIREMENTS FOR THE META-CAI SYSTEM; MAY BE CHANGED IF NECESSARY.

1000: TOTAL AMOUNT OF SIMULATED MACHINE USABLE MEMORY, 300-2000 IS NORMAL
30: TOTAL NUMBER OF SYNTACTIC TYPES AND VARIABLES, 10-90.
100: MEMORY REQUIRED FOR LARGEST SYNTAX STATEMENT OR PROGRAM, 60-200.
100: MAXIMUM STATEMENTS GENERATED FOR ANY SYNTAX TYPE OR PROGRAM 70-200.
30: MAXIMUM LENGTH OF ANY GENERATED ASSEMBLER STATEMENT, 30-50.
10: MAXIMUM VARIABLES IN ANY SINGLE PROGRAM SEGMENT, 3-20.
91: FIRST FRAME NUMBER OF YOUR SYNTAX DEFINITION AREA, 71-91.
100: LAST FRAME NUMBER OF YOUR SYNTAX DEFINITION AREA, 80-100.

FIGURE 7. METACAI DYNAMIC ARRAY ALLOCATION DISPLAY. INITIAL SETTING SHOWN.

T#01 META-CAI SYSTEM RECORD 3

MYCC 49TESTPROG 52BOTHCC 56METACC 61

FIGURE 8. METACAI PROGRAM DIRECTORY RECORD. CURRENTLY CONTAINS 4 PROGRAMS.
META-CAI SYSTEM

WHEN NOTHING ELSE IS TO BE PUT ON THE SCREEN, THIS WILL BE DISPLAYED.
YOU MAY TYPE ANY OF THE LEGAL SYSTEM COMMANDS NOW.

FIGURE 9. METACAI IDLE FRAME DISPLAY. SIGNIFIES SYSTEM IS READY FOR USE.

FIGURE 10. METACAI HELP FRAME DISPLAY NUMBER 1.
OPERATION OF META-CAI IS DIVIDED INTO TWO PRIMARY AREAS: SYSTEMS COMMANDS AND META OPERATIONS. SYSTEMS COMMANDS ARE USED FOR INTERACTION WITH THE SYSTEM, LOADING, EDITING, AND SAVING PROGRAMS, AS WELL AS CONTROL OF META OPERATIONS. META OPERATIONS CONSIST OF DEFINING A COMPILER FOR A USER LANGUAGE IN META SYNTAX, PRODUCING A COMPILER, DEFINING A PROGRAM IN THE USER LANGUAGE, COMPILING THIS PROGRAM, AND EXECUTING THE RESULTING CODE. THIS WOULD BE EQUIVALENT TO DEFINING ALGOL SYNTACTICALLY, CREATING AN ALGOL COMPILER, WRITING AN ALGOL PROGRAM, COMPILING THE PROGRAM, AND EXECUTING THE RESULTING PROGRAM. TYPE ONE OF THE CONTROL STATEMENTS ON LINE 2 AND PROCEED.

FIGURE 11. METACAI HELP FRAME DISPLAY NUMBER 2.

THE FOLLOWING IS A SYNOPSIS OF THE AVAILABLE SYSTEMS COMMANDS. THEY ARE OF THE FORM #XX. THE XX IS A CODE, NORMALLY THE FIRST TWO CHARACTERS OF THE COMMAND TO BE EXECUTED, AND THE POUND SIGN, "#", SIGNIFIES A SYSTEMS COMMAND.

PROGRAM HANDLING COMMANDS
#LO XXXXXXXX: LOAD A PROGRAM CREATED AND SAVED FROM PREVIOUS SESSION.
#SA XXXXXXXX: SAVES A NEW COPY OF THE PROGRAM XXXXXXXX. NO NAME CHECKING IS DONE
#ER XXXXXXXX: ERASES THE PROGRAM XXXXXXXX AND RELEASES ITS STORAGE SPACE.
#AD XXXXXXXX: ADDS THE PROGRAM XXXXXXXX ONTO PREVIOUSLY LOADED PROGRAM.
#PUBLISH : PUBLISHES THE NAMES OF THE AVAILABLE PROGRAMS. #PU IS SUFFICIENT.

FIGURE 12. METACAI HELP FRAME DISPLAY NUMBER 3.
T#01 HELP FRAME 4
META-CAI SYSTEM

#FO YOU MAY ALSO TYPE #BA OR JUST ATTENTION.
SYNTACTIC NAME AND VARIABLE HANDLING COMMANDS

#DE XXXXXXXX-YYYYYYYY: DELETES CODE BETWEEN NAMEX AND NAMEY, NON-INCLUSIVE
#DE XXXXXXXX-: DELETES CODE BETWEEN NAMEX AND END, -BLANKS NECESSARY
#LIST: LISTS SYNTAX NAMES AND VARIABLES CURRENTLY LOADED.

FIGURE 13. METACAI HELP FRAME DISPLAY NUMBER 4.

T#01 HELP FRAME 5
META-CAI SYSTEM

#FO YOU MAY ALSO TYPE #RA OR JUST ATTENTION.
USER SYNTAX DEFINITION HANDLING COMMANDS

#SH: PRESENTS FIRST FRAME OF USER DEFINED DEFINITIONS.
#SH NNNN: PRESENTS FRAME NNNN OF USER DEFINED DEFINITIONS, RELATIVE TO 1
#FR NNNN: FREES FRAME NNNN AND CLEARS THE DISPLAY READY FOR INPUT.
#FORWARD: PRESENTS NEXT FRAME OF USER DEFINED DEFINITIONS.
#BACKWARD: PRESENTS PREVIOUS FRAME OF USER DEFINED DEFINITIONS.
#EN XXXXXXXX: ENTRY POINT FOR USERS COMPILER
#CO NNNN: STARTS Compiling SYNTAX DEFINITIONS IN THIS FRAME LINE NNNN(1-10).
#EX XXXXXXXX: BEGINS EXECUTION OF PROGRAM XXXXXXXX

FIGURE 14. METACAI HELP FRAME DISPLAY NUMBER 5.
Figure 15. Meta-Cai Help Frame Display Number 6.

Figure 16. Meta-Cai Help Frame Display Number 7.
used?" The next question is, "What capabilities are available for its implementation?" The last question is easily answered in the METACAI system. The previous sections have given the specifications of the METACAI machine. On this machine, as on actual machines, some things can be easily done, some with difficulty, and other things not at all.

Many types of problems can be solved on a general purpose computer. The machine instructions for that computer can be thought of as a set of building blocks and any program designed to operate on that machine must be constructed from these blocks. A language designer has the job of designing the language structure and writing a translator which transforms any problem program written in that language into the correct ordered set of these blocks. The complexity of the language dictates the complexity of the translator. A problem oriented language and its translator are often developed simultaneously. Thus the METACAI system user, like a system designer in the real world, must decide what things are to be included in a language and how this language will be structured. The METACAI system user has been given the set of METACAI instructions upon which he can build his own language. He has also been given a basic translator which he can use as his translator or which he can use to build his own translator. If the student is in a formal course, the language definition problem is handled by the
instructor and the student has one less thing to do.

**Language definition frames**

Figures 17-24 are the language definition frames that were stored on the author's CAIFILE data set at the writing of this paper. These frames are shown as examples of the types of experimental work for which the system was being used. A user defined number of records are used for this work space and are stored at the end of the data set. The following paragraphs give a short description of the contents of these frames.

The basic METACAI translator is called the META compiler-compiler (METACC) and its grammar is found in Figures 17 and 18. Every symbol that this translator will recognize is found as a quoted string in the definitions, thereby enabling it to translate and reproduce itself. These symbols are the set T for METACC and are the symbols that the user may use in his definitions. The semantics for the translator are found in the .OUT operations. This basic translator is called METACC in the program library and may be used by the student as his initial translator.

The bottom five lines of Figure 18 contain the grammar for a simple sentence in English. The last of the five is a formal sentence of the language.

Using the METACC translator, the definitions of MY
Compiler-Compiler (MYCC) in Figures 19 and 20 were translated. This was used as the author's working translator. MYCC is basically the same as METACC, but was used to extend the capability of METACAI. It was found that if two similar versions of a language were available, they could be used to modify each other in a ping-pong manner. The definition of the .DO operation is different in the translators. This will be used as a later example.

A simple FORTRAN like language was designed in Figures 21 and 22. Only DIMENSION and ASSIGNMENT statements were allowed in the example. PROGRAM TEST1 was the sample program translated and tested.

Figures 23 and 24 are the solution to a problem discussed in the following section. The computerized solution is presented in Appendix A.

Sample problem

A sample problem given to a student using the METACAI system was stated in the following way: "Write a program to sum the values of two variables and assign the sum to a third variable. Define the syntax of the language in a "wordy" form, i.e., one in which the structure of the language is defined in a modified English description." The following is a solution to the problem.

The simple assignment statement form was:
FIGURE 17. METACAI USER DEFINITION FRAME 1. META COMPILER-COMPILER DEFINITIONS

FIGURE 18. METACAI USER DEFINITION FRAME 2. METACC AND SENTENCE DEFINITIONS.
T#01 LINES 1-10 FOLLOWING MAY BE EDITED AND COMPILED. FRAME # 3

# - : TYPE A COMMAND

MYCC := ID .LABEL(*) ' := ' EX1 ' ; ' .OUT("R") ;
EX1 := EX2 $(' ! ) .OUT("BT",1) EX2 .LABEL(*1) ;
EX2 := (EX3 .OUT("BF",1) | OUTPUT) $ (EX3 .OUT("BE") | OUTPUT) .LABEL(*1) ;
EX3 := ID .OUT("CLL",*1) | STR .OUT("TST",*) | ID .OUT("ID") | NUM .OUT("NUM")
| STR .OUT("SR") | EX1 ' | .EMPTY .OUT("SET") | ' $' .LABEL (*1) ;
EX3 .OUT("BT",1) .OUT("SET") ;

OUTPUT := " .OUT ' ( | .LABEL ' ( | .OUT("LB") ) $OUT1 ' ) .OUT("OUT") | OUT2 | OUT3 ;
OUT2 := ' SAV ' (( ( ID | STR | NUM ) .OUT("SAV") | "DO " (( .STR ) .OUT("DO") ) ;
OUT3 := ' IGN ' (( .OUT("SET") | ' LATCH ' ( ( ID .OUT("LAT",*) ) ) | ' ERR ' ( ( .STR .OUT("ERR",*) ) ) ;

FIGURE 19. METACAI USER DEFINITION FRAME 3. MY COMPILER-COMPILER DEFINITIONS.

T#01 LINES 1-10 FOLLOWING MAY BE EDITED AND COMPILED. FRAME # 4

# - : TYPE A COMMAND

OUT1 := STR .OUT ("CL",*) | '.1' .OUT("GN1") | '.2' .OUT("GN2") |
'.3' .OUT("CI") | '.4' .OUT("ARG") | '.5' .OUT("DE") | '/' .OUT("OUT") |
'.R' .OUT("RES") | '.S' .OUT("SAV") | '.I' .OUT("PLS") | '.X' .OUT("SWP") |
'.ON' .OUT("LL") | '.C' .OUT("CAN") ;
PRINTST := 'PRINT' ( "" PRINT1 $('!', PRINT1 ')') | .EMPTY ) ; 'OUT("PNT") ;
PRINT1 := STR .OUT("EDT",*) | '<' EXPL ' >" .OUT("ENT") | VARIABLE ( "" EXP1 "" |
| .EMPTY .OUT("SS") | .OUT("WRT") ;
DUMPST := ID .LABEL(*) ' := PRINTST .OUT("R") ;
DUMPV := PRINT ("A=" ; A, 'MYCC=' ; MYCC, "<50> ; A ;
DUMPS := PRINT ("A FOR THREE", A(0)(3)) ;

FIGURE 20. METACAI USER DEFINITION FRAME 4. MYCC AND PRINT DEFINITIONS.
T#01 LINES 1-10 FOLLOWING MAY BE EDITED AND COMPILED. FRAME # 5
# - TYPE A COMMAND
FORTPROG:= "PROGRAM" .ID .LABEL(*) ;"* SUBPART "END" .OUT("R");
SUBPART:= DIMST $DIMST ASSIGNST $ ASSIGNST;
DIMST:= "DIMENSION" .OUT("B",*1) DCLST $( '"' DCLST) '" LABEL(*1);
DCLST:= .ID .LABEL(*) ( ("" NUM .OUT("BLK","*) ") "" EMPTY .OUT("BLK","1") );
ASSIGNST:= VARIABLE ASSIGNPT ";
ASSIGNPT:= '=" EXP1 (ASSIGNPT .OUT("ST") ) .EMPTY .OUT("SST")");
EXP1:= EXP2 $( '+' TERM .OUT("ADD") ) "" TERM .OUT("SUB")
EXP2:= '=" TERM .OUT("NEG") | '+' TERM | TERM;
TERM:= PRIMARY $( '*' PRIMARY .OUT("MLT") | \" PRIMARY .OUT("DIV")");
VARIABLE:= .ID .OUT("LD",*) (ARRAYPT|EMPTY);

FIGURE 21. METACAI USER DEFINITION FRAME 5. SIMPLE FORTRAN PROGRAM DEFINITIONS

T#01 LINES 1-10 FOLLOWING MAY BE EDITED AND COMPILED. FRAME # 6
# - TYPE A COMMAND
PRIMARY:= VARIABLE|( '.TRUE.' |'1') .OUT("SS1")|( '.FALSE.' |'0') .OUT("SSO") |
\" NUM .OUT("LDL",*) | '=" EXP1 ' );
ARRAYPT:= '=" EXP1 ' ) .OUT("AIA");
PROGRAM TEST1;
    DIMENSION A,B,L(2);
    DIMENSION C,D,M(2);
    A=1; B=C;
    L(1)=B*C+A;
    M(A)=3.2*2.6; M(A+A)=L(-A+A+2/(.TRUE.+1));
    .END.

FIGURE 22. METACAI USER DEFINITION FRAME 6. SIMPLE FORTRAN TEST PROGRAM.
T#01 LINES 1-10 FOLLOWING MAY BE EDITED AND COMPILED. FRAME # 7
# TYPE A COMMAND
SUMCC := 'A' .ID .LABEL(*) 'IS A' SUM1 '.' OUT('R');
SUM1 := SUM2 .OUT('BF',*1) ('$FOLLOWED BY A' SUM2 .OUT('BE')|SUMOUT .OUT('
OUT')|).LABEL(*1);
SUM2 := $.STR .OUT('TST',*) ['.TSTID' .OUT('ID') | .ID .OUT('CL',*)];
SUMOUT := $.OPER$.STR .OUT('CL',*) ['.OPARG$.STR .OUT('CL',*/'ARG') 'INPUT'.OUT('CI') | $.NAME$.OUT('LB') .ID .OUT('CL',*)];
A SUMC IS A '%' .NAME SUM FOLLOWED BY A VARIABLE FOLLOWED BY A '=' FOLLOWED BY A VARIABLE FOLLOWED BY A '+' FOLLOWED BY A VARIABLE .OPER 'ADD' .OPER 'ST'
FOLLOWED BY A '*' .OPER 'R'.
A VARIABLE IS A $.TSTID$.OPARG 'LD' INPUT.

FIGURE 23. METACAI USER DEFINITION FRAME 7. SUM COMPILER-COMPILER DEFINITIONS.

T#01 LINES 1-10 FOLLOWING MAY BE EDITED AND COMPILED. FRAME # 8
# TYPE A COMMAND
%A=B+C.

3
4
5
6
7
8
9
0

FIGURE 24. METACAI USER DEFINITION FRAME 8. TEST STATEMENT FOR SUMC.
(1). % A=B+C; where A, B, and C could be any variable. The % and ; symbols served to mark the beginning and end of the statement. The problem was worked backwards. The assembler statements needed to be generated for the METACAI machine were:

(2). SUM
    LD    A
    LD    B
    LD    C
    ADD
    SST
    R

A translator was needed to accept (1) as INPUT and generate (2). This translator only needed to be general enough to accept different variable names. The "wordy" form of the SUMC translator description was:

(3a). A SUMC IS A '%%'.NAME 'SUM' FOLLOWED BY A VARIABLE FOLLOWED BY A '=' FOLLOWED BY A VARIABLE FOLLOWED BY A '+' FOLLOWED BY A VARIABLE .OPER 'ADD' .OPER 'SST' FOLLOWED BY A '.' .OPER 'R'.

(3b). A VARIABLE IS A .TSTID .OPARG 'LD' INPUT. A translator was now needed for (3). This was called SUMCC, a sum compiler-compiler. It was defined in the basic MSF.

(4). SUMCC := 'A' .ID .LABEL (*) 'IS A' SUM1 '.' .OUT('R');
    SUM1 := SUM2 .OUT('BF',*1) $(('FOLLOWED BY A' SUM2 .OUT ('BE'))|SUMOUT .OUT('OUT')) .LABEL(*1);
    SUM2 := .STR .OUT('TST',*)| '.TSTID' .OUT('ID')| .ID .OUT('CLL',*);
These four statements define a translator for the two "sentences" in (3). The MYCC program was used to translate (4). The author's translator (MYCC), the compiler-compiler (SUMCC), the compiler (SUMC), and the program (SUM) are now all defined for the solution. The four phases of program execution to solve the problem are diagrammed below.

### INPUT | PROGRAM | OUTPUT
--- | --- | ---
PHASE I | SUMCC DEFS IN MSF | MYCC TRANSLATOR | SUMCC TRANSLATOR
PHASE II | SUMC DEFS IN SUMCC FORM | SUMCC TRANSLATOR | SUMC TRANSLATOR
PHASE III | %A=B+C; | SUMC TRANSLATOR | SUM PROGRAM
PHASE IV | NONE | SUM PROGRAM | NONE

SUMC could have been easily defined in MSF as:

(5) SUMC := '%'.LABEL('SUM') VARIABLE '=' VARIABLE '+' VARIABLE ';'.OUT('ADD'/'SST'/'R');

A complete record of the solution to this problem is contained in Appendix A.
Another example

During the evolution of this system, many iterations had to be made to get the METACAI compiler-compiler, METACC. The META II translator and its language definitions were used as the "bootstrapping" program. In order to obtain a new translator, the definitions of the basic translator were modified and translated, leaving two different subroutines with the same name in MEM. The old version would be removed by the #DELETE system command, thus creating a new translator. The old version could not be deleted before the new version was completed, however, because it was used in the translation. It often happened that a feature of the new translator was needed to do the translation of the feature. Of course, it was not available until the new version was completed. This very confusing problem resulted in several iterations being done before the desired program was available.

During the implementation of the 'DO' instruction in the system, two iterations of translator modification were necessary. The METACAI instruction DO immediately executed the symbolic instruction contained in the character variable, Q. Thus,

(1) .DO('SS1')

was supposed to PUSH a 1.0 into the numeric stack during its translation. The first version of the
definitions of OUT2 (see Figure 19) contained the alternative:

(2) `.DO' '(.STR')' .OUT('DO')

which produced

\[
\begin{align*}
\text{TST} & \quad \text{'.DO'} \\
\text{BF} & \quad \#04 \\
\text{TST} & \quad '(' \\
\text{BE} & \\
\text{SR} & \\
\text{BE} & \\
\text{TST} & \quad ')' \\
\text{BE} & \\
\text{CL} & \quad 'DO' \\
\text{OUT} & \\
\end{align*}
\]

\#04

\begin{align*}
\text{BT} & \quad \#05 \\
\end{align*}

and when (3) translated (1), the following was produced:

DO

This, of course, was one level of translation too late. DO must be an instruction in the original translator to execute SSI as soon as it is "deleted". The next try was:

(4) `.DO' '(.STR .D')' | '.D'.OUT('DO')

believing that if a .D in the definition would output a DO, this definition would work. The output of this was:

\[
\begin{align*}
\text{TST} & \quad \text{'.DO'} \\
\text{BF} & \quad \#4 \\
\text{TST} & \quad '(' \\
\text{BE} & \\
\text{SR} & \\
\end{align*}
\]

SYNTAX ERROR IN TRANSLATION. ERROR ABOVE *

\[
\begin{align*}
\text{'.DO'} '(.STR .D')' | '.D'.OUT('DO')
\end{align*}
\]
The .D was not recognized because the old translator (3) did not recognize the .D form. A feature of the new translator was attempted to be used while it was being defined. Two steps of definition were needed.

(5) '.D' .OUT('DO') in the first step defined the needed form. (2) could now be correctly recognized and (1), when translated by the second version of the program, had the correct operation. A clean version of this solution appears in the META01 definition in Figure 17.

Language Testing

The following formal grammar, written in BNF, was found in (41).

(1) *** <PROGRAM>::=<BODY><STAT>END
* <BODY>::=BEGIN|<BODY><STAT>;
*** <STAT>::=<PROGRAM>|<ASSNMENT>
    <ASSNMENT>::=<VAR>::=<EXPR>
** <EXPR>::=<SIMPEXPR>|<IFCLAUSE><SIMPEXPR>ELSE<EXPR>
* <SIMPEXPR>::=<TERM>|<SIMPEXPR>+<TERM>
* <TERM>::=<FACTOR>|<TERM>*<FACTOR>
*** <FACTOR>::=<VAR>|<NUMBER>|(<EXPR>)
    <IFCLAUSE>::=IF<RELATION>THEN
*** <RELATION>::=<SIMPEXPR>=<SIMPEXPR>
It was decided to test this language for well-formation on the METACAI system, first by "recognizing" sentences of the language, then by generating code for a sample sentence, and finally by executing the generated code. It was noted that the definitions marked * contained direct left recursion, those marked ** contained direct right recursion, and those marked *** were indirectly recursive. The language was re-written in MSF, taking advantage of the META repetition symbol, $, and some METACAI primitives.

The definitions were now written as:

(2) PROGRAM := 'BEGIN'$STAT'END';
STAT := 'BEGIN'$STAT'END' ';' | ASSNMENT;
ASSNMENT := VAR' := 'EXPR';
EXPR := IFSTATEM | SIMPEXPR;
SIMPEXPR := TERM$( '+' TERM);
TERM := FACTOR$( '*' FACTOR);
FACTOR := '(' EXPR ')' | .NUM | VAR;
IFSTATEM := 'IF' SIMPEXPR '=' SIMPEXPR 'THEN' SIMPEXPR 'ELSE' EXPR;
VAR := 'A' | 'B' | ...;

Now only STAT has direct recursion and the IFSTATEM and FACTOR definitions contain indirect recursion. Some
sentences from this language are:

(3) 1. BEGIN A := 1.3; END
2. BEGIN BEGIN A := B + C * 2.0; END; D := B * C * (A + B); END
3. BEGIN D := IF A = B * C THEN C + D ELSE 3.4 + D; END
4. BEGIN D := IF C = A THEN B ELSE IF C = B THEN A ELSE C; END
5. BEGIN BEGIN D := C; C := 2; BEGIN A := 1; B := 2 + 3; END; A := 3; END; A := 1; END

These sentences (3) from the subset of ALGOL defined by (1), containing conditional statements, arithmetic assignment statements, and nested block structure were "recognized" by the recognizer generated by the MSF definitions (2) given above.

The second part of the language test was to see if METACAI instructions could be generated which would carry out the "intent" of the language, i.e. successfully translate and execute the sentences (3) given above. The recognizer was thus converted to a translator; its grammar is defined by:

(4) PROGRAM := 'BEGIN' .LABEL('TESTC': 'LLL') $STAT 'END'
 .OUT('R');
STAT := 'BEGIN' $STAT 'END' ';' | ASSNMENT;
ASSNMENT := VAR .OUT('LD', *)' := 'EXPR'; .OUT('SST');
EXPR := IFSTATE | SIMPEXPR;
SIMPEXPR := TERM$( '+' TERM.OUT('ADD'));
TERM := FACTOR $( '*' FACTOR.OUT('MLT'));
FACTOR := '(' EXPR ')' | NUM.OUT('LDL',*) | VAR.OUT('LD',*);
IFSTATEM := 'IF' SIMPEXPR = SIMPEXPR.OUT('EQU'/'BEP', *1)' THEN'.OUT('POP') SIMPEXPR.OUT('B',*2)' ELSE' .LABEL(*1)EXPR.LABEL(*2);
VAR := 'A' | 'B' | 'C' | 'D';

This grammar correctly defined a translator which would generate the proper METACAI instructions to execute the sentences (3) of this language. Each sentence produced a subroutine named TESTC containing the instructions.

Two initial problems existed in the METACAI implementation of (1): the global variable storage was not assigned for the variables A, B, C, D and METACAI was not designed to handle nested block structure variables. In the ALGOL block structure, different copies of variables with the same identifier must exist in the different program blocks. It was found that the first problem could be solved, the second could not. The following definitions were used to define a translator for declaring the variables.

(5) DCLSTAT := 'DECLARE' DECLAR$( ',', DECLAR);$' DECLAR := VAR.LABEL(*:'BLK','l');

This translator was used to translate:

(6) DECLARE A, B, C, D;
Memory locations were thereby saved for the variables A, B, C, D. Even though only a global copy of each variable was available, the code for the sample sentences was correctly executed.
CONCLUSIONS AND SUMMARY

The METACAI system was designed as an experimental laboratory tool for studying formal languages, giving the system user an opportunity for experimentation. There is no set pattern of questions and answers as in many CAI systems, thereby allowing the student freedom to control his own learning environment. There are training aids in the system, but like a laboratory instrument in other areas, a certain amount of understanding and training is needed to effectively use the METACAI system. This training may be obtained from outside sources or from student exploration and experimentation in the interactive "try and see what happens" approach to problem solving. The interpretive nature of METACAI gives the user protection against his own mistakes.

Many general topics of computer science can be investigated using the METACAI system. Formal computer languages; their definition and translation is the most obvious area. The interdependence of language specification and user program translation is another. The similarities and differences of compiler-compilers, compilers, and problem programs is a third. The student can gain experience in computer language assemblers and interpreters since they are an integral part of the METACAI system. Simulation of one
computing machine on another has been a helpful aid in computer system design and is exemplified in this system. On-line interactive systems have become more common and more terminals are being developed to make their use easy. The METACAI system uses one of the fastest interactive devices as the student terminal. Finally, good "debugging" techniques should be instilled in the computer science student. Allowing user control of translation and execution trace facilities lets the student see when and how these facilities can effectively be used. All these general areas may be covered in varying detail within the METACAI system.

Problem oriented computer languages are continually being defined and implemented and the cost of this language design, implementation, and evaluation is normally very high. Standardization of language implementations among computer systems is necessary for compatible operation of programs. With many methods of formal language translation available, new languages defined in this manner are faster and easier to implement. Thus their study is becoming more and more important as is giving the future computer scientist an effective and economical tool for this study.

The interactive capability of METACAI is one of its primary assets. The immediate information feedback to the student during language translation and program execution lets him follow the progress of its operation. Any mistakes
are immediately diagnosed and can be immediately corrected. These mistakes are an aid to learning in an experimental situation. Program errors are either explicitly checked and diagnosed, or the PL/I error routine intercepts the errors before disastrous results can occur. The different program traces give a history of the events leading up to the error, thereby making its diagnosis easier.

The METACAI system is not tutorial in nature but can be used in a tutorial mode. Both short and long forms of presentation are possible. The GET instruction obtains display frames from CAIFILE data set and the HLP instruction displays them. The MSS and ERR instructions both output short messages, the first unconditionally and the second if there was a translation error. These instructions may be used to produce a tutorial student translator by a course instructor, or in the language of CAI, the course author. The level and complexity of the translator are completely up to the user.

The METACAI system has not undergone extensive field usage, but appears to be a useful tool in a formal computer language and compiler construction course, such as the CS 451 course given at Iowa State University. It also can be used for remedial work assigned to computer science graduate students entering a program without proper background. Given a system description and a set of problems, he should be able
to obtain the training and experience usually presented in such a course. The set of assigned problems should be designed to lead the student in the direction of material understanding, without forcing the mature student to spend an extra year obtaining the background training.

The cost of system operation has not been totally analyzed but in several 2 1/2 hour sessions of system usage, 8-12 seconds of 360/65 computer processing time was used. Since several seconds are used for system sign-on, sign-off, the efficiency of the operational portion appears to be high. About 70,000 bytes of 360 computer memory are used for the program, depending on the user-defined size of arrays. A major factor in the time used was the 10:1 increase of execution time because the memory assigned to the MTMT region was in IBM's Large Core Storage (LCS). Non-interactive programs are not assigned this slower memory, hence a 10:1 penalty is paid by the interactive user.

From the author's viewpoint, the METACAI system is a powerful and useful tool in the study of formal languages although the extent of its capability has not been entirely explored. Even after several months usage, the author still finds new ways of using the system and discovers new methods for the study of programming languages.


29. Lindsay, R. A heuristic procedure for a language learning program. ERIC ED 022 175. 1964.


44. Simmons, R. Linguistic analysis of constructed student responses in CAI. ERIC ED 031 094. 1968.


ACKNOWLEDGMENTS

The author wishes to express his appreciation to Dr. Clair Maple, Iowa State University Computation Center, for his help and guidance in the preparation of this paper, and for his obtaining financial assistance for the author.

Special acknowledgment is extended to Mr. Douglas Hogan, Dr. W. Blankinship, and Dr. H. Campagne, Department of Defense, Fort Meade, Maryland, for their help in obtaining the author's two year leave of absence.

The author wishes to express his intense gratitude to his parents for their encouragement, understanding, and financial assistance during his long educational career. Their teaching has extended far beyond what any school could offer.

A "thank you" is extended to Miss Martha Marks, Mrs. Donald Phipps, and the Iowa State University thesis staff for their respective assistance in thesis editing, typing, and final acceptance.
APPENDIX A: METACAI SAMPLE SESSION
SAMPLE SESSION ON METACAI SYSTEM. PROGRAM TRACE IS TURNED ON AND OFF.

#PU - TYPE A COMMAND
#LO MYCC - TYPE A COMMAND
#LI - TYPE A COMMAND
#SH 5 - TYPE A COMMAND
#SH 6 - TYPE A COMMAND
#FO - TYPE A COMMAND
#BO - TYPE A COMMAND
#BA - TYPE A COMMAND
#TN - TYPE A COMMAND
#CO 1 - TYPE A COMMAND

SUMCC := 'A' .ID .LABEL (*) 'IS A' SUM1 'OUT('R')';
DELETED SUMCC SUMCC
DELETED :=

TRACE TO: EX1
TRACE TO: EX2
TRACE TO: EX3
TRACE TO: EX3
TRACE TO: EX3
TRACE TO: EX3
TRACE TO: EX3
TRACE TO: OUTPUT
TRACE TO: OUTPUT
TRACE TO: OUTPUT
TRACE TO: EX3
TRACE TO: EX3
TRACE TO: EX3
TRACE TO: EX3
TRACE TO: EX3

TST 'A'
BF #01
ID
BE
LB
CI
OUT
TST 'IS A'
CII SUM1
BE
TST '
BE
DELETED .OUT
YOU SUCCESSFULLY TRANSLATED SUMCC

#TF - : TYPE A COMMAND
#CO 2 - : TYPE A COMMAND
SUM1 := SUM2 .OUT ('BF',*,1) $('*FOLLOWED BY A' SUM2 .OUT ('BE')|SUMOUT .OUT('OUT')|LABEL(*1) ;
YOU SUCCESSFULLY TRANSLATED SUM1

#CO 4 - : TYPE A COMMAND
SUM2 := .STR .OUT('TST',*) | .TSTID' .OUT('ID') | .ID .OUT('CLL',*) ;
YOU SUCCESSFULLY TRANSLATED SUM2

#C0 5 - : TYPE A COMMAND
SUMOUT := 'OPER' .STR .OUT('CL', '*') | 'OPARG' .STR .OUT('CL', '*') | 'NAME' .OUT('CL', '*') |
YOU SUCCESSFULLY TRANSLATED SUMOUT

#TN - : TYPE A COMMAND
#EN SUMCC - : TYPE A COMMAND
#C0 7 - : TYPE A COMMAND
A SUMC IS A 'SUM' NAME SUM FOLLOWED BY A VARIABLE FOLLOWED BY A '=' FOLLOWED BY A
DELETED A
DELETED SUMC SUMC
DELETED IS A

TRACE TO: SUM1
TRACE TO: SUM2
DELETED 'SUM'
TST 'SUM'
BF #01

TRACE TO: SUMOUT
DELETED 'NAME'
DELETED 'SUM'
DELETED 'OUT'

TRACE TO: SUM2
DELETED FOLLOWED BY A
DELETED VARIABLE
DELETED VARIABLE
DELETED 'OPER' 'ADD' 'OPER' 'SST'

TRACE TO: SUM2
DELETED 'SUM'
DELETED '+'
TST '+'
BF
| 1SUMC | TST  | BE   | LOC  | 1 IS | 2 IS | 3 IS | 4 IS | 5 IS | 6 IS | 7 IS | 8 IS | 9 IS | 10 IS | 11 IS | 12 IS | 13 IS | 14 IS | 15 IS | 16 IS | 17 IS | 18 IS | 19 IS | 20 IS | 21 IS | 22 IS | 23 IS |
|-------|------|------|------|------|------|------|------|------|------|------|------|------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 2     | TST  | BF   | LOC  |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 3     | BF   | #01  | LOC  |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 4     | LB   | LOC  |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 5     | CL   | SUM  | LOC  |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 6     | OUT  | LOC  |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 7     | CLL  | VARIABLE | LOC |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 8     | BE   | LOC  |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 9     | TST  | '='  | LOC  |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 10    | BE   | LOC  |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 11    | CLL  | VARIABLE | LOC |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 12    | BE   | LOC  |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 13    | TST  | '+'  | LOC  |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 14    | BE   | LOC  |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 15    | CLL  | VARIABLE | LOC |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 16    | BE   | LOC  |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 17    | CL   | 'ADD' | LOC  |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 18    | OUT  | LOC  |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 19    | CL   | 'ST'  | LOC  |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 20    | OUT  | LOC  |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 21    | TST  | '.'  | LOC  |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 22    | BE   | LOC  |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 23    | CL   | 'R'  | LOC  |      |      |      |      |      |      |      |      |      |      |       |       |       |       |       |       |       |       |       |       |       |       |       |       |

Trace to: SUMOUT
DELETED 'ADD'

TRACE TO: SUMOUT
DELETED 'OPER'
DELETED 'SST'

FOLLOWED BY A 'OPER 'R'.
DELETED FOLLOWED BY A 'OPER 'R'.

Trace to: SUM2
DELETED 'OPER'

Trace to: SUMOUT
DELETED 'OPER'

Trace to: SUMOUT
DELETED 'R'

TRACE TO: SUMOUT
DELETED 'ADD'

CL 'ADD'

CL 'SST'

CL 'R'

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .

Trace to: SUMOUT
DELETED .
YOU SUCCESSFULLY TRANSLATED SUMC

#TF - : TYPE A COMMAND
#CO 10 - : TYPE A COMMAND

A VARIABLE IS A .TSTID .OPARG 'LO' INPUT.

YOU SUCCESSFULLY TRANSLATED VARIABLE

#TN - : TYPE A COMMAND
#EN SUMC - : TYPE A COMMAND
#FO - : TYPE A COMMAND
#CO 1 - : TYPE A COMMAND
%A=B+C.

DELETED %

SUM

TRACE TO:VARIABLE

DELETED A
DELETED =

TRACE TO:VARIABLE

DELETED B
DELETED +

TRACE TO:VARIABLE

DELETED C

LD A
LD B
LD C
ADD
SST
R

DELETE.

1SUM

2 LD A LOC 1 IS 150637
3 LD B LOC 2 IS 150637
4 LD C LOC 3 IS 150637
5 ADD LOC 4 IS 1
6 SST LOC 5 IS 67
7 R LOC 6 IS 55

YOU SUCCESSFULLY TRANSLATED SUM

#EX SUM - : TYPE A COMMAND

YOU SUCCESSFULLY EXECUTED SUM

#SA TESTPROG- : TYPE A COMMAND
FIGURE 25. SOLUTION TO SUM COMPILER-COMPILER PROBLEM. ONLY PARTIALLY TRACED.
APPENDIX B: METACAI PROGRAM STRUCTURE

**Procedure Hierarchy**

I. METACAI
   - HELP
   - ERROR1
   - CAISYS
      1. ERROR2
      2. GETFREE
      3. COM2260
         a. DSP2260
      4. SKIPB
      5. GETCD
      6. LOOKUP
      7. OUTCOM

**HELP Procedure**

Reads and displays help frames from CAIFILE data set.

**ERROR1 Procedure**

Displays error message and PL/I ONCODE, ONSOURCE, and ONKEY codes. The PL/I error is written on the SYSPRINT file. Restarts at beginning.

**METACAI Procedure**

Main procedure of METACAI system. Executes user sign-on, opens data sets, reads parameters, and enters CAISYS procedure.

**CAISYS Procedure**

Inner procedure of METACAI. Executes system control commands and METACAI machine instructions.

**ERROR2 Procedure**

Displays error message and PL/I ONCODE, ONSOURCE, and ONKEY codes. The PL/I error is written on the SYSPRINT file. Returns to METACAI idle frame, waiting for user command.

**GETFREE Procedure**

Obtains the next free record in the CAIFILE data set after a parameter number and before the first syntax definition number.

**COM2260 Procedure**

Calls DSP2260 procedure, writes the control line on SYSPRINT file, checks if system control command and branches to specified command or returns to calling procedure.

**DSP2260 Procedure**

Displays a 960 character frame, waits for user ATTENTION, and reads the display contents back into the FRAME variable.
SKIPB Procedure

Skips blanks in the input string. If more information is needed, calls GETCD procedure.

GETCD Procedure

Moves 1 card image into CARD from INFRAME. CARD is sent to the OUTCOM procedure. If more input is needed, it is requested from the terminal. The input pointer, I, is reset to 1.

LOOKUP Procedure

Looks in global symbol table, SYNTYX, for the requested symbol. Searches from present position forward; then from present position back to beginning of table. If found, returns position in table; if not found, presents error message.

OUTCOM Procedure

Handles single line output communication for METACAI system. Writes the line on SYSPRINT and puts it in OUTFRAME. Outputs frame if ten lines are collected or output is forced.

ASSEMBLER Pass 1

Checks all assembler statements for labels, putting all labels into proper label tables; reserves space for BLK instructions.

ASSEMBLER Pass 2

Translates assembler statements, generates machine instructions, collects character strings, assigns relative memory locations, initializes BLK locations, and outputs assembler trace.

INTERPRETER

Simulates the METACAI machine as defined by its instructions in the text of this paper.

COMMAND INTERPRETER

Executes the system commands as defined in the text of this paper.
APPENDIX C: METACAI PROGRAM LISTING
METACAI SYSTEM. FORMAL LANGUAGE CAI BY DENNIS K. BRANSTAD. * /
COMPLETE PROGRAM LISTING WITH DOCUMENTARY STATEMENTS. * /
COMMENTS PRECEDE THE SECTION OF PROGRAM THAT THEY EXPLAIN. * /
PROGRAM IS DIVIDED INTO INTERNAL PROCEDURE HIERARCHY TO ENABLE *
SETTING SMALL ARRAYS FOR DEBUGGING THE CAN BE EXPANDED AT RUN *
TIME. SPEED IS GIVEN UP FOR FLEXIBILITY IN THIS IMPLEMENTATION. * /
-------------------------------------------------------------

(SUBRGTSTRG):

METACAI: PROCEDURE OPTIONS (MAIN);

NMEM IS NUMBER OF MEMORY LOCATIONS IN SIMULATED METACAI MEMORY */
NSYN IS NUMBER OF SYNTAX DEFINITIONS */
NCORE IS THE SIZE OF ASSEMBLER TEMPORARY CORE FOR SIMPLE STMT */
NINP IS SIZE OF INP ARRAY WHICH HOLDS ASSEMBLER STMTS */
NLEN IS THE LENGTH OF EACH ASSEMBLER STMT */
NVAR IS THE NUMBER OF VARIABLES IN EACH PROGRAM SEGMENT */
NFDEF IS THE FIRST DEFINITION FRAME ON CAIFILE */
NLDEF IS THE LAST DEFINITION FRAME ON CAIFILE */
DCL (NMEM,NSYN,NCORE,NINP,NLEN,NVAR,NFDEF,NLDEF) FIXED BIN; /*
FRAME IS THE 960 CHARACTER VARIABLE USED FOR COMMUNICATION WITH */
THE 2260 DISPLAY. MANY VARIABLES ARE DEFINED OVER FRAME. */
FRAMEL ARE THE 12 LINES OF THE FRAME. */
NEXTNO AND RECNO ARE USED FOR PROGRAM RECORD CHAINING. */
TERMINAL IS THE TERMINAL NUMBER THE USER IS WORKING ON. */
THE CONTROLS ARE USED FOR SYSTEM COMMANDS. */
DCL FRAME CHAR (960), (FRAMEL(12) CHAR (80), *
NEXTNO CHAR (4) POS (67), RECNO CHAR (4) POS (77), *
TERMINAL CHAR (2) POS (3), CONTROL1 CHAR (1) POS (81), *
CONTROL3 CHAR (3) POS (81), RESTFR CHAR (800) POS (161), *
FIRST2 CHAR (160), LASTIO CHAR (800) POS (161), *
CONNAME CHAR (8) POS (85), NAME2 CHAR (8) POS (94), *
CONLINE CHAR (80) POS (81), DASH CHAR (1) POS (93), *
CONTROL2 CHAR (2) POS (82), CONTROL4 CHAR (4) POS (85)) /*
DEFINED FRAME,
/* IFREE IS THE PROCEDURE TO OBTAIN FREE FRAME ON CAIFILE. */
IFREE FIXED BIN,

/* KEY9 IS THE RECORD ON CAIFILE THAT IS BEING FETCHED. */
KEY9 CHAR (9), FREE4 CHAR (4) POS (6) DEFINED KEY9,

/* DSP2260 IS THE COMMUNICATION PROCEDURE IN MTMT FOR THE 2260. */
DSP2260 ENTRY (CHAR(*), FIXED BIN(31,0)),

/* CAIFILE IS THE USER DIRECT ACCESS FILE FOR HIS PROGRAMS AND DEFS. */
CAIFILE DIRECT UPDATE KEYED ENV(F(960) REGIONAL (1)),

/* SYSPRINT IS THE TEMPORARY OUTPUT FILE THAT MAY BE PRINTED. */
SYSPRINT STREAM PRINT OUTPUT,

/* CONTROLS ARE THE SYSTEM CONTROL COMMANDS AND ARE RECOGNIZED BY */
/* FIRST TWO LETTERS. ONLY THESE 20 ARE DEFINED. */
CONTROLS CHAR (40) STATIC INITIAL ('COEXHFSOBALID3SADERPUCANOFBOFRFTN'),

/* FRAMEP IS THE RECORD USAGE POINTERS FOR CAIFILE. */
FRAMEP (100) FIXED BIN (15),

/* PNAME IS THE ARRAY FOR THE PROGRAM NAMES THAT MAY BE EXECUTED. */
PNAME (36) CHAR (9),

/* OPCDS ARE THE RECOGNIZED SYMBOLIC OP CODES FOR THE METACAI ASSEM. */
OPCDS (85) CHAR (3), UNDOPS CHAR (30),

/* NAME# IS THE NUMBER OF PROGRAMS IN PROGRAM DIRECTORY. PPOS IS */
/* FIRST POSITION TO RETRIEVE PROGRAM FROM CAIFILE. SEE PNAME. */
NAME# FIXED BIN, PPOS (36) FIXED BIN,

/* SYNFRAME IS THE SYNTAX FRAME NUMBER CURRENTLY BEING VIEWED. */
SYNFRAME FIXED BIN;
/* THE HELP PROCEDURE DISPLAYS THE HELP FRAMES FROM 10 TO 20. */
/* #FO AND #BA SCROLL THE HELP FRAMES FORWARD AND BACK. IF A BLANK */
/* IS WRITTEN OVER THE # SIGN, RETURNS TO THE MAIN PROGRAM. */
HELP: PROCEDURE;
  IFR=11;
HELPIN: READ FILE (CAIFFILE) INTO (FRAME) KEY (IFR);
  CONLINE='#FO YOU MAY ALSO TYPE #BA OR JUST ATTENTION.';
  CALL DSP2260(FRAME,2);
  IF CONTROL1=1 THEN RETURN;
  IF CONTROL3=#FO* & IFR<20 THEN IFR=IFR+1; ELSE
  IF CONTROL3=#BA* & IFR>11 THEN IFR=IFR-1;
  GOTO HELPIN; END HELP;

/* PL/I ERROR INTERCEPT PROCEDURE FOR THE OUTER METACAI PROCEDURE. */
/* ERROR IN THIS PART USUALLY MEANS UNDEFINED FILE. */
/* #CANCEL RETURNS TO MTMT WHERE THE FILES MAY BE DEFINED. */
ON ERROR BEGIN;
  FIRST2=* PL/I INTERCEPTED ERROR.*;
  LAST10=THE FOLLOWING ITEMS ARE THE ONCODE, ONSOURCE, AND ONKEY
  ITEMS. LOOK THEM UP IN THE PLI MANUAL. YOU WILL THEN RESTART.
  :•|ONCODE|•'•|ONSOURCE|•':•|ONKEY;
  CALL DSP2260(FRAME,2);
  IF CONTROL3=#CA* THEN STOP;
  GOTO BEGINAG; END;

/* WELCOME DISPLAY FRAME FOR THE METACAI SYSTEM. USED TO OBTAIN */
/* USERS TERMINAL SO THAT HIS DATA SETS MAY BE OPENED PROPERLY. */
BEGINAG:FIRST2=* WELCOME TO THE META-CAI SYSTEM.*;
  LAST10=* THIS SYSTEM IS DESIGNED AS A LABORATORY TOOL FOR UNDERSTANDING FORMAL COMPUTER LANGUAGES. YOU ARE EXPECTED TO HAVE SOME KNOWLEDGE OF THE SYSTEM, AS IT IS A POWERFUL, BUT COMPLICATED SYSTEM. YOU SHOULD HAVE CHANGED THE SYSinXX DD NAME TO BE YOUR METACAI DATA SET. IF YOU DID NOT, TYPE #CANCEL (ONLY #CA IS NECESSARY) TO RETURN TO MTMT AND USE OPTION 40 TO CHANGE THIS. SYsOUTXX IS USED FOR PL/I ERROR MESSAGES AND MUST BE A SCRATCH OUTPUT DATA SET (NORMALLY MTMT2260.SYsO
UTXX). IF YOU ARE UNFAMILIAR WITH METACAII, OR AT ANY TIME NEED HELP, 
TYPE #HELP AND DIRECTIONS WILL BE GIVEN. OTHERWISE, HOLD THE SHIFT AND 
HIT THE ENTER KEYS. PROCEED."
/* OUTPUTS THE FRAME, SETTING CURSOR ON LINE 2 FOR INPUT. */
CALL DSP2260(FRAME,2);
/* IF THE TYPED INPUT IS *CA, THEN CANCEL THE PROGRAM AND RETURN TO */
/* MTMT. PROBABLY MEANS THAT DATA SET DEFINITIONS WERE NOT CHANGED.*/
IF CONTROL3='#CA* THEN STOP;
/* OPENS CAIFILE AND SYSPRINT FILES. SAVES SPACE IN OUTER PROCEDURE.*/
OPEN FILE (SYSPRINT) TITLE ('SYSOUT' || TERMINAL),
FILE (CAIFILE) TITLE ('SYSIN' || TERMINAL);
/* IF COMMAND IS #HELP, CALLS THE HELP PROCEDURE. */
IF CONTROL3='#HE* THEN CALL HELP;
/* INPUTS RECORD 0 FROM CAIFILE, AND GETS THE FRAME USAGE POINTERS. */
READ FILE (CAIFILE) INTO (FRAME) KEY (O);
GET STRING (RESTFR) EDIT (FRAMEP) (100 F(8));
/* INPUTS RECORD 1 FROM CAIFILE, AND GETS NUMBER OF OPERATORS, */
/* THE DEFINED SYMBOLIC OPERATORS, AND THE UNDEFINED OPERATOR STRING.*/
READ FILE (CAIFILE) INTO (FRAME) KEY (FRAMEP (1));
GET STRING (RESTFR) EDIT (NUMOPS,OPCDS,UNDOPS)
( F( 8), X(72), 5 (17 (A3), X(1)),X(12)),A(30));
/* READS THE DYNAMIC ALLOCATION FRAME FROM CAIFILE. DISPLAY FOR */
/* USER MODIFICATION, THEN READS THE VARIABLES TO BE USED IN NEXT */
/* PROCEDURE. */
READ FILE (CAIFILE) INTO (FRAME) KEY (FRAMEP(2));
CALL DSP2260(FRAME,2);
GET STRING (RESTFR) EDIT (NMEM,NSYN,NCORE,NINP,NLEN,NVAR 
, NFDEF,NLDEF)
(10 (F(8), X(72)));
/* INPUTS FRAME 3 FROM CAIFILE AND READS THE NUMBER OF PROGRAMS, */
/* THE PROGRAM NAMES, AND THE INITIAL FRAME OF PROGRAM STORAGE. */
READ FILE (CAIFILE) INTO (FRAME) KEY (FRAMEP(3));
GET STRING (RESTFR) EDIT (NAME#, (PNAME(I), PPOS(I)
DO I=1 TO NAME#)) (F(8), X(72), 36 (A(8), F(12)));
/* ENTERS THE INNER PROCEDURE OF METACAI. */
CALL CAISYS;
/* */
THE INNER PROCEDURE OF METACAI, OBTAINS USER STORAGE, EXECUTES SYSTEM COMMANDS, CONTAINS THE ASSEMBLER ROUTINE, AND DOES THE METACAI MACHINE SIMULATION. NO PARAMETERS NEEDED.

CAISYS: PROCEDURE;
DECLARE

CARD IS THE 80 CHARACTER VARIABLE THAT IS USED FOR INPUT TO THE TRANSLATOR, I IS THE POINTER INTO CARD FOR SEQUENTIALLY OBTAINING CHARACTERS DURING TRANSLATION, SHORTCC IS THE FIRST PART OF CARD AND IS USED FOR ASSEMBLER STATEMENTS, BIGLAB IS THE BIG LABEL USED FOR SYNTAX NAMES AND FOR VARIABLES, LABEL IS FOR THE NN OF #NN AND aNN TYPE LABELS, CALLNAM WAS USED IN BATCH VERSION, OP IS THE THREE CHARACTER SYMBOLIC OP CODE, AND CHAR1 AND CHAR2 ARE USED TO GET LABEL IN ARGUMENT FIELD, ADDRS GETS BIG LABEL IN THE ARGUMENT FIELD.

CARD CHAR (80), (CHAR11 CHAR (1), SHORTCD CHAR (NLEN), BIGLAB CHAR (8), CCHAR (80) CHAR (1), LABEL CHAR (2) POS (2), CALLNAM CHAR (8) POS (5), OP CHAR (3) POS (10), CHAR1 CHAR (1) POS (16), CHAR2 CHAR (2) POS (17), ADDRS CHAR (8) POS (16)) DEFINED CARD,

INFRAME IS THE LOWER TEN LINES OF THE INPUT DISPLAY, INLINE IS AN ARRAY OVER INFRAME, OUTFRAME IS A SIMILAR OUTPUT FRAME, OUTLINE IS AN ARRAY USED TO ENTER LINES INTO THE OUTPUT FRAME.

MEM IS THE BIG METACAI SIMULATED MEMORY DEFINED AS INTEGER, CORE IS THE SMALL ASSEMBLER TEMPORARY STORAGE OUTPUT AREA, SYNADX IS THE POINTER ARRAY TO ADDRESSES OF SYNTAX DEFINITIONS IN MEM, THESE ARE MODIFIED IF MEM IS EDITED BY SYSTEM COMMANDS, FMEM IS THE FLOATING POINT REFERENCE INTO MEM, MEMCCS IS THE CHARACTER STRING REFERENCE INTO MEM, CCS IS THE CHARACTER STRING REFERENCE TO CORE.
(MEM(NMEM), CORE (NCORE), SYNADX (NSYN)) FIXED BIN (31),
FMEM (NMEM) FLOAT BIN DEFINED MEM,
MEMCCS CHAR (4*NMEM) DEFINED MEM,
CCS CHAR (4*NCORE) DEFINED CORE,

/* INP IS A TEMPORARY AREA FOR STORAGE OF ASSEMBLER STATEMENTS. */
/* LINE IS THE OUTPUT LINE FROM A TRANSLATOR THAT IS BEING PRODUCED. */
/* DELETED INPUT AS IT IS RECOGNIZED. USED IN SAVE AND DO INSTRUCS. */
/* Q IS A VARYING CHARACTER VARIABLE USED AS TEMPORARY STORAGE FOR */
(INP(NINP), LINE) CHAR (NLEN), Q CHAR (25) VAR,

/* STACK IS THE NUMERICAL STACK USED IN THE INTERPRETER. MSTACK IS */
/* DEFINED OVER STACK AND USED TO STORE ADDRESSES IN INTERPRETER. */
STACK (60) FLOAT BIN, MSTACK (60) FIXED BIN (31) DEFINED STACK,

/* SM AND SM1 ARE SIMULATED CONTENTS OF TOP OF STACK AND NEXT TO THE */
/* TOP OF STACK, RESPECTIVELY. SAVE AND STEM1 ARE TEMPORARIES, AND */
/* STACK1 IS AUXILLARY REGISTER IN STACK MECHANISM. */
(SM, SM1, SAVE, STACK1, STEM1) FLOAT BIN,

/* LITSTK IS THE LITERAL CHARACTER STACK, A CHARACTER STRING. */
/* LITSTKP IS THE POINTER ARRAY USED TO MARK BOUNDARIES OF STRINGS */
/* IN THE STACK. ARBITRARY MAXIMUM. */
LITSTK CHAR (255), LITSTKP (50) FIXED BIN,

/* VARNAM IS AN ARRAY FOR VARIABLE NAMES IN ONE PROGRAM SEGMENT. */
/* VAROFF IS THE VARIABLE OFFSETS FROM BEGINNING OF SEGMENT. */
VARNAM (NVAR) CHAR (8), VAROFF (NVAR) FIXED BIN,

/* IVAR IS THE NUMBER OF VARIABLES FOUND IN ONE PROGRAM SEGMENT. */
/* LITN IS THE NUMBER OF LITERALS CURRENTLY IN LITERAL STACK. */
/* ISTKP IS THE INTEGER STACK POINTER WHICH ALWAYS POINTS TO TOS. */
/* IOLDP IS THE OLD STACK POINTER, USED TO TELL WHEN STACK CHANGES. */
(IVAR, LITN, ISTKP, IOLDP) FIXED BIN,

/* CCENTRY IS THE COMPILER COMPILER ENTRY POINT. SPECIAL SYMBOL HERE*/
/* USENAM IS THE NAME CURRENTLY BEING USED, EITHER SYNTAX TYPE OR VAR*/
SYNTYX IS THE STORAGE FOR THE LIST OF SYNTAX TYPES AND VARIABLES.

SMCHAR AND SMICHAR ARE USED IN INSTRUCTION TRACE OF STACK.

SYNTAX TYPES IN SYNTYX ARRAY, NEXTOP IS THE NEXT OP CODE TO BE
MEMP IS THE HIGH WATER MARK OF MEM, SYNO IS CURRENT NUMBER OF
DEFINED IN CASE OF UNDEFINED OP. KPOS IS THE POINTER TO CURRENT
SUBROUTINE NAME BEING EXECUTED. KK IS A TEMPORARY.

LLF IS THE INSTRUCTION TRACE FLAG. IF ON, OUTPUTS LOC, OP CODE,
STACK POINTER, AND TOP TWO ELEMENTS OF THE STACK.
SKPF IS THE SKIP FLAG, IF ON SKIPS INITIAL BLANKS IN INPUT.
SKPF IS THE SKIP FLAG, IF ON SKIPS INITIAL BLANKS IN INPUT.
TF IS THE PROGRAM TRACE FLAG; IF ON, TRACES TRANSLATION PROCESS.
DOF IS THE DO FLAG; DURING DO INSTRUCTION, RETURN IS CHANGED TO
DIFFERENT PLACE. ONLY ONE INSTRUCTION IS 'DONE' AT A TIME.

LABS IS LABEL TABLE FOR GENERATED LABELS OF TYPE #NN, NN=1 TO 30.
GLABS IS LABEL TABLE FOR GENERAL LABELS OF TYPE @NN, NN=1 TO 99.

IS ARRAY IS AN INTEGER STACK USED FOR SUBROUTINE CALLS, 20 LEVELS
OF SUBROUTINE NESTING ARE CURRENTLY POSSIBLE.

INNOO IS THE NUMBER OF ASSEMBLER STATEMENTS GENERATED; LIT AND LOC*/
ARE POINTERS FOR THE TWO PASSES OF INP DURING ASSEMBLY; COL IS */
THE COLUMN POINTER FOR LINE OUTPUT DURING TRANSLATION; LAB IS THE */
LABEL NN IN BOTH #NN AND @NN; DECR IS THE DECREMENT IN THE RELA- */
TIVE ADDRESSES OF METACAI MACHINE; LEN IS LENGTH OF CHARACTER */
STRING IN AN INSTRUCTION; I IS POINTER TO INPUT CARD; J IS TEMP. */
IOUTC IS AN OUTPUT COLUMN POINTER IN EDIT COMMANDS.

PP, INNOO, LIT, LOC, COL, LAB, DECR, LEN, I, J, IOUTC) FIXED BIN,
/* IH, IL, IO, AND IOP ARE USED IN BINARY CHOP SEARCH ROUTINE. */
(IH, IL, IO, IOP) FIXED BIN,

/* NOP IS THE INSTRUCTION TO BE EXECUTED. NOPTR IS THE SAME. */
/* ISM IS THE ADDRESS IN TOS IF ONE THERE. ISMRND IS TOS ROUNDED. */
/* ISML IS ADDRESS IN NTCS IF ONE THERE. ISMIRND IS NTOS ROUNDED. */
/* NN AND ITEMP ARE TEMPORARIES. */
(NOP, NOPTR, ISM, ISMRND, ISML, ISMIRND, NN, ITEMP) FIXED BIN (31),

/* GETFREE IS PROCEDURE TO FIND A FREE RECORD ON CAIFILE. */
GETFREE ENTRY (FIXED BIN),

/* GETFREE IS PROCEDURE TO FIND A FREE RECORD ON CAIFILE. */
/* OUTCOM IS OUTPUT COMMUNICATION ROUTINE TO THE TERMINAL AND SYSPRINT*/
/* T FILE. PARAMETER 1 IS LINE TO OUTPUT, 2ND PARAMETER IS FORCE */
/* OUTPUT FLAG FOR IMMEDIATE OUTPUT. USUALLY MEANS ERROR. */
OUTCOM ENTRY (CHAR(*), BIT(1)),

/* OUTLN IS OUTPUT LINE NUMBER, 1 TO 10. POINTER INTO OUTFRAME. */
/* CARDOUT AND OUTCARD ARE OUTPUT LINES THAT ARE USED FROM METACAI. */
OUTLN FIXED BIN, CARDOUT CHAR (80), OUTCARD CHAR (80),

/* LATCH IS A FLAG SHOWING IF A LAT INSTRUCTION IS BEING EXECUTED. */
/* 0 IS THE OPERATION BRANCH TO EXECUTE ONE OF 85 INSTRUCTIONS. */
LATCH BIT (1), 0(85) LABEL,

/* CON IS THE 20 SYSTEM CONTROL COMMANDS OF FORM #XX. TRANSFER. */
CON (20) LABEL,

/* GENLAB IS A GENERATED LABEL NUMBER OF TYPE #NN, NN=1 TO 30. */
GENLAB CHAR (60) INITIAL ( '010203040506070809101112131415161718192021222324252627282930'),
BLANK CHAR (1) INITIAL(' ');

/*****************************/
/* INITIALIZES THE VARIABLES FOR BEGINNING OF METACAI SESSION. */
/* ** */
/* PROGRAM TRACE FLAG AND BLANK SKIP FLAG INITIALLY ON. */
TF,SKPF=*I"B;
KPOS=1; INLN,OUTLN=0;
/* CLEARS OUTPUT FRAME, SETS SYNTAX FRAME TO FIRST ONE, AND INITS. */
/* NEXTOP, THE STACKS, MEMORY POINTERS, AND SYNTAX NUMBER. */
OUTFRAME=BLANK;
SYNFRAME=NFDEF;
NEXTOP=NUMOPS;
LITSTK=BLANK; LITSTKP=1; STACK=1.0; /*DEBUG ONLY.*/
MEMP,SYNO=0;

/* INTERCEPTS PL/I ERROR INTERRUPT, PUTS MESSAGE ON SCREEN. */
ON ERROR BEGIN;
FIRST2=" PL/I INTERCEPTED ERROR.
LAST10="THE FOLLOWING ITEMS ARE THE ONCODE, ONSOURCE, AND ONKEY ITEMS. LOOK THEM UP IN THE PL/I MANUAL. THEN RESTART LAST OPERATION.
ON ERROR SYSTEM; GOTO IDLE; END;

/* SETS FIRST FRAME TO BE DISPLAYED AS THE IDLE FRAME. */
/* GENERAL CONTROL SECTION. READS AND DISPLAYS FROM CAIFILE. */
/* CALLS 2260 COMMUNICATION ROUTINE, GOES TO CONTROL COMMAND IF */
/* ONE IS SPECIFIED OR RETURNS HERE AND THEN DISPLAYS A SYNTAX DEF. */
/* FRAME. RETURN COMES TO ONE OF THESE POSITIONS FROM COMMANDS. */
DOIDLE: IDLEF=4;
PUTFR: READ FILE (CAIFILE) INTO (FRAME) KEY (FRAMEP(IDLEF));
IDLE: CONLINE="# - : TYPE A COMMAND";
   CALL CCM2260;
   GOTO SHOWF;

CON(15):
/* #EN NAME SETS NAME AS THE COMPILER-COMPILER ENTRY POINT. */
ENTRY: CCENTRY=CONNAME; GOTO IDLE;
CON(1):
/* #CO N COMPILES LINE N OR SYNTAX FRAME BEGINNING AT CCENTRY. */
COMP: INLN=CONTROL4-1; INFRAME=RESTFR; CALL GETCD;
/* REWRITES SYNTAX FRAME IN CASE USER HAS CHANGED IT. */
REWRITE_FILE (CAIFILE) FROM (FRAME) KEY (FRAMEP(SYNFRAME));
/* DOES A LOOK UP OF CCENTRY NAME IN SYNTYX ARRAY. */
CALLCC: USENAM=CCENTRY; GOTO DOLKUP;
CON(2):
/* #EX NAME COMMAND. SETS INLN SO THAT FIRST READ REQUESTS INPUT. */
EXECUTE: US[NAM=CONNAME;
INLN=10;
/* LOOKS UP NAME AND BEGINS EXECUTION AT THAT POINT. */
DOLKUP: CALL LOOKUP;

/* INITIALIZES THE INTERPRETER FOR A TRANSLATION OR PROGRAM EXECUTION*/
M2RES: LAB,P,I,LITN,LITSTKP(1),IOUTCOL=1;
INNDO=0;
ICLPD,ISTKP=2;
OUTCARD=BLANK;
/* SIMULATES A CALL TO INTERPRETER AS IF IT WERE A SUBROUTINE. */
/* Initializes subroutine stack; turns off do flag, instruction flag */
/* and the latch mechanism. Current position and location are */
/* stacked for the return from main subroutine. */
IS(P),IS(P+1)=0; DOF,LLF,LATCH='O'B;
IS(P+2)=KPCS; IS(P+3),LOC=SYNADX(KPOS);
/* OUTPUT LINE AND COLUMN POINTER ARE INITIALIZED. */
LINSET: LINE=BLANK; COL=10;

/* FETCH CYCLE; IF DO FLAG IS ON, GO TO UNDO THE DO CYCLE. */
/* INSTRUCTION IS FETCHED FROM MEM. LOC IS LOCATION COUNTER. */
/* SPLITS INSTRUCTION INTO OP CODE, LENGTH OF STRING, AND OFFSET */
/* DECREMENT FOR RELATIVE ADDRESSING. LITP IS LITERAL POINTER. */
FETCH: IF DOF THEN GOTO UNDO;
NOPTR,NOP=MEM(LCC);
LOC=LOC+1;
IF NOP>100 THEN DO; LEN=NOP/100000; NN=MOD(NOP,100000);
DECR=NN/100-501; NOP=MOD(NN,100); LITP=4*(LOC+DECR)-3; END;
/* CHECKS TO SEE IF THE STACK HAS BEEN USED DURING LAST CYCLE.  */ /* IF THE NUMBER IN THE STACK IS AN ADDRESS, DOES A LCAD INDIRECT.  */ /* SETS SM TO BE THE TOP OF STACK (TOS) FOR SIMULATION.  */ /* CHECKS IF THE NEXT TO TOP ELEMENT IS AN ADDRESS. IF IT IS,  */ /* LOADS INDIRECT FROM THE ADDRESS. NTOS IS IN ISM1.  */ IF ISTKP=.= IOLDP THEN DO;
ISM=MSTACK(ISTKP); IF ISM<UISM>NMEM THEN SM=STACK(ISTKP);
ELSE SM=FMEM(ISM);
ISM1=MSTACK(ISTKP-1); IF ISM1<UISM1>NMEM THEN
SM1=STACK(ISTKP-1); ELSE SM1=FMEM(ISM1);
/* SETS THE ROUNDED PORTION OF SM AND SM1 FOR SOME INSTRUCTIONS. */ ISMRND=SM+.5; ISM1RND=SM1+.5;
IOLDP=ISTKP; END;

/* IF INSTRUCTION TRACE FLAG LLF IS ON, THEN DUMP LOC, OP CODE,  */ /* NUMERICAL STACK POINTER, AND TOP TWO ELEMENTS OF STACK.  */ /* PUTS INSTRUCTION TRACE ON DISPLAY IF WANTED. */ IF LLP THEN DO;
IF ISM<1|ISM>NMEM THEN SMCHAR=SM; ELSE SMCHAR=ISM;
IF ISM1<1|ISM1>NMEM THEN SM1CHAR=SM1; ELSE SM1CHAR=ISM1;
PUT STRING (CARDOUT) ECIT(LOC=LOC-1,OPCODE=OPTR,
',ISTKP=ISTKP,TOS=SMCHAR,NTOS=SM1CHAR)
(A,F(4),A,F(7),A,F(2),(4)A);
CALL OUTCOM(CARDOUT,'0'B); END;

/* GOES TO SIMULATE THE CORRECT INSTRUCTION. NOP IS OP CODE. */ GOTO C(NOP);
O(01): /* ADD PRODUCES SUM OF TOS AND NTOS. PUSHES RESULT INTO STACK. */ ADD: STEMP=SM+SM1; GOTO STP;
O(02): /* ADS IS ADD TO STORAGE. STORES SUM IN MEM AND STACK1, THEN POPS 2. */ ADS: FMEM(ISM1),STACK1=SM+FMEM(ISM1); GOTO POP2;
O(03): /* AIA IS ARRAY INCREMENT ADDRESS. FORMS ADDRESS FOR TOP OF STACK. */ AIA: ITEMP=SM; ITEMP=ITEMP+ISM1; GOTO ISTP;
O(04):
/* ALL TURNS ON THE SKIP BLANK Flag so initial BLANKS auto. skipped. */
ALL: SKIPF='L*B; GOTO FETCH;

0(05):
/* AND FORMS LOGICAL AND OF TOS AND NTOS. REPLACES TOP TWO ELEMENTS. */
AND: IF SM==0&SM1==0 THEN GOTO SET1; ELSE GOTO SETO;

0(06):
/* ARG resets the COL pointer for output to argument position. */
ARG: COL=16; GOTO FETCH;

0(08):
/* B does a branch to either a local or global label. */
B: IF LEN>0 THEN GOTO BLABEL; LOC=LOC+DECR; GOTO FETCH;

0(09):
/* BE does a branch on error. IF SWITcH is on, continue; ELSE ERROR. */
BE: IF SWITCH THEN GOTO FETCH; ELSE GOTO ERROR;

0(10):
/* BEP is branch on equal to 0 and POP. */
BEP: IF ISM=0 THEN GOTO BPOP; ELSE GOTO FETCH;

0(11):
/* BEO is branch on equal to 0. continue if not zero. */
BEO: IF ISM=0 THEN GOTO B; ELSE GOTO FETCH;

0(12):
/* BF is branch is false. branches if switch not set; ELSE continue. */
BF: IF SWITCH THEN GOTO B; ELSE GOTO FETCH;

0(13):
/* BL1 is instruction to turn the program trace on. also set by #TN. */
BL1: TF='1'B; GOTO FETCH;

0(14):
/* BNP is branch on not zero and POP; ELSE continue. */
BNP: IF ISM=0 THEN GOTO FETCH;
BPOP: ISTKP=ISTKP-1; IF ISTKP<2 THEN GOTO TESTPL; ELSE GOTO B;

0(15):
/* BNO branch if tcs is not zero; ELSE continue. */
BNO: IF ISM=0 THEN GOTO FETCH; ELSE GOTO B;

0(16):
/* BT is branch on TRUE; ELSE continue. */
BT: IF SWITCH THEN GOTO B; ELSE GOTO FETCH;
/* CAN CANCELS THE LATCH FLAG. */
CAN: LATCH='0'B; GOTO FETCH;

O(18):
/* CI COPIES INPUT FROM Q INTO LINE, INCREMENTING COL BY LENGTH OF Q. */
CI: J=LENGTH(Q); SUBSTR(LINE, COL, J)=Q; COL=COL+J; GOTO FETCH;

O(19):
/* CL COPIES LITERAL FROM MEM CHARACTER STRING INTO OUTPUT LINE. */
CL: SUBSTR(LINE, COL, LEN)=SUBSTR(MEMCCS, LITP, LEN);
    COL=COL+LEN; GOTO FETCH;

O(20):
/* CLL IS CALL AND LINK TO A SUBROUTINE. STACKS FOUR ITEMS AND LOOKS */
/* UP NAME IN TABLE, OUTPUTING TRACE IF DESIRED, AND BRANCHING. */
CLL: P=P+4; IS(P), IS(P+1)=0; IS(P+2)=KPOS; IS(P+3)=LOC;
    BLABEL: USENAM=SUBSTR(MEMCCS, LITP, LEN);
    CALL LOOKUP;
    IF IOUTC>1 THEN
        CALL OUTCOM(OUTCARD, '0'B);
        OUTCARD='TRACE TO: '||USENAM;
        IOUTC=21;
        LOC=SYNADX(KPCS); GOTO FETCH;

O(21):
/* CLR AND RST ARE USED TO RESET OR CLEAR SWITCH TO OFF. */
RST: SWITCH='0'B; GOTO FETCH;

O(22):
/* CONCATENATES TWO STRINGS CONTAINED IN LITERAL STACK. */
CONCAT: LITSTKP(LITN-1)=LITSTKP(LITN); GOTO PLS;

O(23):
/* DIVIDES NTOS BY TCS, REPLACING THEM WITH RESULT. */
DIV: STEMP=SM1/SM; GOTO STP;

O(24):
/* DO IS IMMEDIATE DO OF AN INSTRUCTION FOUND IN Q. NO ARGUMENTS. */
DOINS: Q=SUBSTR(Q, 2, LENGTH(Q)-2); /*STRIPS OFF QUOTE MARKS.*/
    DO IOP=1 TO NUMOPS; IF Q=OPCDS(IOP) THEN GOTO DOOP; END;
    IF IOUTC>1 THEN CALL OUTCOM(OUTCARD, '0'B);
    OUTCARD='IN A DO INSTRUCTION, CANNOT FIND OPERATOR '||Q;
    CALL OUTCCM(OUTCARD, '0'B); GOTO SHOWF;

DOOP: DOF='1'B; GOTO O(ICP);
/* UNDO IS RETURN FROM DOING AN INSTRUCTION. CONTINUES NORMALLY. */
UNDO: DOF=*0'B; GOTO FETCH;

/* EDT IS STRING OUTPUT EDIT INSTRUCTION. PREPARES FOR OUTPUT. */
EDT: SUBSTR(OUTCARD,IOUTCOL,LEN)=SUBSTR(MEMCCS,LITP,LEN);
     IOUTCOL=IOUTCOL+LEN; GOTO FETCH;

/* ENT INSTRUCTION SETS THE IOUTCOL PCINTER FOR STRING OUTPUT. */
ENTFFF: IOUTCOL=ISMRND; GOTO POP;

/* EQU TESTS IF TOS IS EQUAL TO NTOS AND REPLACES THEM WITH 1.0 IF SO*/
EQU: IF S^=SM1 THEN GOTO SET1; ELSE GOTO SETO;

/* ERR OUTPUTS STRING MESSAGE IF SWITCH IS NOT SET. */
ERRFFF: IF SWITCH THEN GOTO FETCH;
     OUTCARD=SUBSTR(MEMCCS,LITP,LEN);
     CALL OUTCOM(OUTCARD,'1'B); GOTO FETCH;

/* FLP FLIPS THE TOP TWO ELEMENTS IN NUMERIC STACK. */
FLP: MSTACK(ISTKP-N)=ISM; MSTACK(ISTKP)=ISM1; IOLDP=0;
     GOTO FETRET;

/* GET READS A RECORD FROM CAIFILE INTO FRAME FOR POSSIBLE DISPLAY. */
GETFFF: READ FILE (CAIFILE) INTO (FRAME) KEY(FRAMEP(ISMRND));
     GO TO POP;

/* GN1 GENERATES A UNIQUE LOCAL LABEL IN A SUBROUTINE LEVEL. */
GN1: PP=P; GOTO CKGN;

/* GN2 GENERATES A SECOND UNIQUE LABEL IN A SUBROUTINE LEVEL. */
GN2: PP=P+1;
CKGN: IF IS(PP)=0 THEN DO; IS(PP)=LAB; LAB=LAB+2; END;
     SUBSTR(LINE,COL+3)=*#'*|SUBSTR(GENLAB,IS(PP),2);
     COL=COL+3; GOTO FETCH;

/* HLP OUTPUTS HELP FRAME CONTAINED IN FRAME. USER MAY MAY REPLY. */
HLPFFF: CALL COM2260; GO TO FETCH;
0(34):
/* ID TESTS FOR AN IDENTIFIER; A LETTER FOLLOWED BY LETTERS OR DIGITS*/
/* IF AN IDENTIFIER IS FOUND, DELETES IT INTO Q, AND SETS SWITCH. */
ID:
  IF SKPF THEN CALL SKPB;
  IF C(I)<'A'|C(I)>'Z' THEN GOTO RST;
  DO J=I+1 TO 80 BY 1 WHILE (C(J)>'A'); END;
COMB1:
  Q=SUBSTR(CARD,I,J-I);
  IF IOUTC>21 THEN CALL OUTCOM(OUTCARD,'0'B);
  SUBSTR(OUTCARD,21)=*DELETED '*|Q;
  IOUTC=41;
  I=J;
  IF I>80 THEN CALL GETCD;
  GOTO SET;

0(35):
/* LAT DOES A POSSIBLE BACKUP CALL TO A SUBROUTINE. SEE ERROR. */
LAT:
  LATCH='1'B; ISAVEINN=INNOO; ISAVEI=I; GOTO CLL;

0(36):
/* LB SETS THE COLUMN POINT TO THE LABEL FIELD FOR AN ASSEMBLER */
/* STATEMENT. A LABEL WILL BE PUT HERE. */
LB:
  COL=1; GOTO FETCH;

0(37):
/* LOADS THE ADDRESS OF A VARIABLE INTO THE STACK. MUST BE GLOBAL. */
LD:
  USENAM=SUBSTR(MEMCCS,LITP,LEN);
  CALL LOOKUP; ITEMP=SYNADX(KPOS); GOTO IPUSH;

0(38):
/* LDL LOADS LITERAL NUMBER INTO STACK. EITHER INTEGER OR REAL. */
LDL:
  STEMP=SUBSTR(MEMCCS,LITP,LEN); GOTO PUSH;

0(39):
/* LEQ REPLACES TOP TWO ELEMENTS WITH A 1.0 IF NTOS<=TOS; ELSE 0.0 */
LEQ:
  IF SM1<=SM THEN GOTO SET1; ELSE GOTO SETO;

0(40):
/* LES REPLACES TOP TWO ELEMENTS WITH 1.0 IF NTOS<TOS, ELSE ZERO. */
LES:
  IF SM1<SM THEN GOTO SET1; ELSE GOTO SETO;

0(41):
/* LLO SETS INSTRUCTION TRACE OFF. */
LLO:
  LLF='0'B; GOTO FETCH;

0(42):
/* LL1 TURN INSTRUCTION TRACE ON. */
LL1:  LLE='1'B; GOTO FETCH;
O(42):
/* MLT REPLACES TOP TWO ELEMENTS WITH THEIR PRODUCT. */
MLT:  STEMP=SM*SM1; GOTO STP;
O(44):
/* MSS OUTPUTS MESSAGE STRING TO TERMINAL. */
MSSFFF: OUTCARD=SUBSTR(MEMCCS,LITP,LEN); GOTO PNT;
O(45):
/* NEG REPLACES TOS WITH ITS NUMERICAL COMPLEMENT. */
NEG:  STEMP=-SM; GOTO STG;
O(46):
/* NOT SETS STACK TO 1.0 IF 0.0, ELSE TO ZERO. */
NOT:  IF SM=0 THEN STEMP=1; ELSE STEMP=0; GOTO STG;
O(47):
/* NUM IS A TEST FOR A NUMBER IN INPUT STRING; MAY CONTAIN DECIMALS. */
NUM:  IF SKPF THEN CALL SKIPB;
     IF C(I)'<O' THEN GOTO RST;
     DO J=I+1 BY 1 WHILE (C(J)'><O'|C(J)='.'); END; GOTO COMB1;
O(48):
/* OP SETS LINE OUTPUT POINTER TO OPERATOR COLUMN. */
OPCOL: COL=10; GOTO FETCH;
O(49):
/* OR REPLACES TOP TWO ELEMENTS OF STACK WITH A 1.0 IF BOTH NON-0. */
OR:   IF SM=0|SM1=0 THEN GOTO SET1; ELSE GOTO SETO;
O(50):
/* OUT MOVES THE ASSEMBLER LINE INTO INP ARRAY AND INCREASES INNOO. */
OUT:  SUBSTR(OUTCARD,41)=LINE;
     CALL OUTCOM(OUTCARD,'0'B);
     INNOO=INNOO+1;
     INP(INNOO)=LINE;
     GO TO LINSET;
O(51):
/* PLS POPS THE LITERAL STACK. NO CHECK ON THIS FOR UNDERFLOW. */
PLS:  LITN=LITN-1; GOTO FETCH;
O(52):
/* PNT PRINTS THE LINE THAT EOT AND WRT GENERATES. PUT ON TERMINAL. */
PNT: CALL OUTCOM(OUTCARD,'O'B); GOTO FETCH;
0(53):
/* POP REMOVES TOP ELEMENT FROM THE NUMERIC STACK. THIS STACK CHECKD*/
POP: ISTKP=ISTKP-1;
TESTPL: IF ISTKP>1 THEN GOTO FETCH;
OUTCARD='THE STACK IS EMPTY. ERROR IN EXECUTION.';
CALL OUTCOM(OUTCARD,'1'B); GOTO FETCH;
POP2: ISTKP=ISTKP-2; GOTO TESTPL;
0(54):
/* PUT WRITES THE CONTENTS OF FRAME ONTO CAIFILE. USED WITH GET. */
PUTFFF: FRAMEP(ISMRND)=ISMRND;
REWRI TE FILE (CAIFILE) FROM (FRAME) KEY (ISMRND);
GOTO POP;
0(55):
/* R RETURNS FROM A SUBROUTINE. IF LAST SUBROUTINE, LEAVES INTERP. */
R: KPOS=IS(P+2); LOC=IS(P+3);
P=P-4; IF P>0 THEN GOTO CAN; ELSE GOTO DONE INT;
0(56):
/* RED READS NUMBERS FROM A CARD IMAGE GOTTEN FROM TERMINAL. */
RED: CALL GETCD; GET STRING (CARD) EDIT ((FMEM(L) DO L = ISM1 TO ISM1 +ISMRND-1)) (F(10,4));
GOTO POP2;
0(57):
/* RES RESTORES THE Q VARIABLE FROM THE LITERAL STACK. */
RES: Q=SUBSTR(LITSTK,LITSTKP(LITN-1),LITSTKP(LITN)-LITSTKP(LITN-1));
GOTO PLS;
0(58):
/* RET RETURNS FROM THE METACAI SYSTEM. IT MAY BE A BIG SUBROUTINE. */
RET: RETURN;
0(59):
/* RSR RESTORES REGISTER STACK1 INTO THE STACK. */
RSR: STEMP=STACK1; GOTO PUSH;
0(60):
/* SAV CAN EITHER SAVE A STRING OR CONTENTS OF Q IN THE LITERAL STACK*/
SAV: IF LEN>0 THEN Q=SUBSTR(MEMCCS,LITP,LEN);
LITPT=LITSTKP(LITN);LEN=LENGTH(Q); LITN=LITN+1;
LITSTKP(LITN)=LITPT+LEN;
0(61):  
/* SET TURNS THE SWITCH ON. USED MOSTLY FOR TRANSLATION TESTS. */
SET: SWITCH='1'; GOTO FETCH;

0(62):  
/* SLS TURNS OFF THE SKIP BLANK FLAG. BLANKS ARE NOW A SPECIAL CHARACTER. */
SLS: SKIPF='0'; GOTO FETCH;

0(63):  
/* SR TESTS FOR A STRING CONTAINED IN QUOTES. IF IT IS, PUTS INTO Q. */
SR: IF SKPF THEN CALL SKIP8;
    IF C(I)='*'* THEN GOTO RST;
    DO J=I+2 TO 80 BY 1 WHILE (C(J)='*'*);
    END;
    J=J+1; GOTO COMB1;

0(64):  
/* SSR SAVES THE STACK REGISTER TOS INTO REGISTER STACK1. */
SSR: STACK1=SM; GOTO POP;

0(65):  
/* SST SAVES AND STORES TOS IN STACK1 AND MEM. POPS 2 ELEMENTS. */
SST: STACK1,FMEM(ISM1)=SM; GOTO POP2;

0(66):  
/* SSO PUTS A ZERO ON TOP OF STACK. */
SSO: STEMP=0; GOTO PUSH;

0(67):  
/* SSI PUTS A 1.0 ON TOP OF STACK. ALSO A BOOLEAN TRUE FOR METACAI. */
SSI: STEMP=1; GOTO PUSH;

0(68):  
/* ST STORES THE STACK REGISTER INTO MEMORY. POPS 1 ELEMENT. */
ST: FMEM(ISM)=STACK1; GOTO POP;

0(69):  
/* SUB SUBTRACTS TOS FROM NTOS AND REPLACES THEM WITH DIFFERENCE. */
SUB: STEMP=SM1-SM; GOTO STP;

0(70):  
/* SWP INTERCHANGES THE TOP TWO STRINGS IN THE LITERAL STACK. */
SWAP: SUBSTR(LITSTK,LITSTK(LITN-2),LITSTK(LITN)-LITSTK(LITN-2))=
    SUBSTR(LITSTK,LITSTK(LITN-1),LITSTK(LITN)-LITSTK(LITN-1));
    SUBSTR(LITSTK,LITSTK(LITN-2),LITSTK(LITN-1)-LITSTK(LITN-2));
LITSTKP(LITN-1) = LITSTKP(LITN)-LITSTKP(LITN-1)+LITSTKP(LITN-2);
GOTO FETCH;

O(71):
/* TST TESTS IF INPUT MATCHES DESIGNATED STRING. YES, DELETE & SET. */
TST:  IF SKPF THEN CALL SKIPB;
IF I+LEN>81 THEN GOTO RST;
IF SUBSTR(MEMCCS,LITP,LEN)=SUBSTR(CARD,I,LEN) THEN
  GO TO RST; J=I+LEN; GOTO COMB1;

O(72):
/* UND FINDS A FREE FRAME AFTER 30 AND PUTS ITS NUMBER IN STACK. */
UNDERT: CALL GETFREE(30); STEMP=IFREE; GOTO PUSH;

O(73):
/* WHL REPLACES TOS WITH TRUNCATED TOS. */
WHL:  ITEMP=SM; GOTO ISTG;

O(74):
/* WRT OUTPUTS ARRAY TO TERMINAL, UP TO 8 MAXIMUM. */
WRT:  PUT STRING (CARDOUT) EDIT ((FMEM(L) DO L=ISM1 TO ISM1
+ISMRND-1)) (F(10,4));
ISM=ISMRND*10;
SUBSTR(OUTCARD,IOUTCOL,ISM) = SUBSTR (CARDOUT, 1, ISM);
IOUTCOL=ICUTCOL+ISM;
GOTO POP2;

O(75):
/* XEQ EXECUTES A TRANSLATOR, SIMULATES # EX NAME. */
XEQFFF: USENAM=SUBSTR(MEMCCS,LITP,LEN); GOTO DOLKUP;

O(76):
/* XOR REPLACES TCP TWC ELEMENTS WITH A 1.0 IF THEY ARE DIFFERENT. */
XOR:  IF SM=SM1 THEN GOTO SET1; ELSE GOTO SET0;
IPUSH: ISTKP=ISTKP+1;
ISTG:  MSTACK(ISTKP)=ITEMP; GOTO FETRET;
ISTP:  MSTACK(ISTKP-1)=ITEMP; GOTO POP;
PUSH:  ISTKP=ISTKP+1;
STG:   STACK(ISTKP)=STEMP;
FETRET: GOTO FETCH;
STP:   STACK(ISTKP-1)=STEMP; GOTO POP;
SET0:  STEMP=0; GOTC STP;
SET1:  STEMP=1; GOTO STP;
ERROR ROUTINE FOR METACAI. IF LATCH FLAG IS ON, RESET INPUT AND OUTPUT POINTERS, OUTPUT MESSAGE, AND EXIT FROM SUBROUTINE.

IF IT ISN'T A LATCH, OUTPUT A THREE LINE MESSAGE, MARKING THE POINT OF THE SYNTAX ERROR. CHECKS TO BE SURE ALL THREE LINES CAN FIT ON ONE DISPLAY FRAME. STOPS EXECUTION.

ERROR:

IF LATCH THEN DO:
  IF IOUTC>1 THEN CALL OUTCCM(OUTCARD,'O'B);
  CARDOUT='LATCH RETURN FROM SUBROUTINE.';
  CALL OUTCOM(CARDOUT,'O'B);
  INN00=ISAVEINN; I=ISAVEI; G0TC R; END;
  TF='1'B; IF OUTLN>7 THEN DO:
    OUTCARD='SYNTAX ERROR WILL APPEAR ON NEXT FRAME.';
    CALL OUTCOM(OUTCARD,'1'B); END;
    OUTCARD='SYNTAX ERROR IN TRANSLATION. ERROR ABOVE *';
    CALL OUTCOM(OUTCARD,'0'B);
    CALL OUTCOM(CARD,'0'B);
    SUBSTR(OUTCARD,1,1)='*';
    CALL OUTCOM(OUTCARD,'1'B);
  GO TO SHOWF;

DONE INT:

DONE WITH INTERPRET PHASE. CHECK IF ANY ASSEMBLER STATEMENTS HAVE BEEN GENERATED; IF SO, ASSEMBLE THEM; IF NOT, OUTPUT MESSAGE.

ASSEMBLER ROUTINE OF MKTACAI SYSTEM. MAY BE CALLED BY ASM INST.

DONE INT:

IF IOUTC=0 THEN GOTO ASM;
  IF IOUTC>1 THEN CALL OUTCOM(OUTCARD,'C'B);
  OUTCARD='YOU HAVE FINISHED EXECUTION OF '||CONNAME;
  CALL OUTCOM(OUTCARD,'l'B); GOTO SHOWF;

0(07):

SETS PASS 1, PASS 2, AND VARIABLE POINTERS TO ZERO.

ASM:

LIT,LOC,IVAR=0;
DO I=1 TO INN0O;
SHORTCD=INP(I);

MOVES GLOBAL LABELS INTO TEMPORARY GLOBAL LABEL TABLE.
IF CHAR11="A" THEN DO;
IVAR=IVAR+1; VARNAM(IVAR)=BIGLAB; VAROFF(IVAR)=LIT; END;
IF OP=":" THEN LIT=LIT+1;

/* BUILDS A GENERATED LOCAL LABEL TABLE. ONLY #01 TO #30 ARE OK. */
IF CHAR11="#" THEN LABS(LABEL)=LIT; ELSE
/* BUILDS A USER GENERATED LOCAL LABEL TABLE. ONLY #01 TO #99 ARE OK. */
IF CHAR11="#" THEN GLABS(LABEL)=LIT;

/* HANDLES BLCK RESERVE INSTRUCTION. */
IF OP=":BLK" THEN LIT=LIT+AD0RS-1;
END;

/* PASS 2 OF ASPMBL. OUTPUTS INSTRUCTIONS INTO TEMPORARY CORE. */
PASST2: DO I=1 TO INN00;
SHORTCD=INP(I);
IF OP=":BLK" THEN DO;
NN=LOC+ADDRS;
/* INITIALIZES BLCK RESERVE VARIABLE STORAGE. */
DO KK=LOC+1 TO NN; CORE(KK)=9999999; END;
LOC=NN; GOTO ISBLK; END;
IF OP=":" THEN DO;
LOC=LOC+1;
/* DOES A BINARY CHOP SEARCH OF THE INSTRUCTION SYMBOLIC TABLE. */
/* IF THE SYMBOLIC INSTRUCTION NOT FOUND, OUTPUTS MESSAGE AND ASSIGNS*/
/* THE NEXT AVAILABLE OP CODE. THEN THE SYMBOL IS PUT INTO UNDOPS. */
IO,IL=1; IH=NUMOPS; IOP=(IL+IH)/2;
DO WHILE (IOP=IO) ;
IF OP>OPCDS(IOP) THEN IL=IOP; ELSE
IF OP<OPCDS(IOP) THEN IH=IOP; ELSE GOTO FOUNDI;
IO=IOP; IOP=(IL+IH)/2;
END;
IOP=(INDEX(UNDCPS,CP)+2)/3+NUMOPS;
IF IOP=NUMOPS THEN DO; IOP,NEXTOP=NEXTOP+1;
OUTCARD=UNDEFINED OP CODE: I0| | I0| | ASSIGNED OP CODE | |NEXTOP;
CALL OUTCOM(OUTCARD,*0'B);
SUBSTR(UNDOPS,3*(NEXTOP-NUMOPS)-2,3)=CP; END;
/* BUILDS PROPER INSTRUCTION AND STORES INTO CORE LOCATION. */
FOUND I: CORE(LOC)=IOP;
    IF CHAR1=BLANK THEN DO;
    LEN=0;
    IF CHAR1='#' THEN DECR=LABS(CHAR2); ELSE IF CHAR1=';' THEN DECR=GLABS(CHAR2); ELSE DO;
    DO J=17 BY 1 WHILE(C(J)='*'); END;
    Q=SUBSTR(SHORTCD,17,J-17); END;
    ELSE DO;
    DO J=17 BY 1 WHILE(C(J)='); END;
    Q=SUBSTR(CARD,16,J-16); END;
    DECR=LIT; LEN=LENGTH(Q);
    SUBSTR(CCS,LIT*4+1,LEN)=Q; LIT=LIT+(LEN+3)/4; END;
    CORE(LOC)=CORE(LOC)+LEN*100000+(501+DECR-LOC)*100; END;
/* OUTPUTS THE ASSEMBLER STATEMENT AND GENERATED CODE. */
ISBLK: OUTCARD=I||SHORTCD'|| LOC '||LOC'|| IS '||CORE(LOC);
    CALL OUTCOM(OUTCARD,'0'B);
    END;
    ELSE DO; OUTCAPD=I||BIGLAB; CALL OUTCOM(OUTCARD,'0'B);
    END;
/* OUTPUTS MESSAGE IF SUCCESSFULLY TRANSLATED. */
OUTCARD='YOU SUCCESSFULLY TRANSLATED '||VARNAM(1);
/* MOVES THE TEMPORARY CORE INTO MAIN MEM. STRINGS ARE LAST. */
    IF VARNAM(1)=SYNTAX(SYN0) THEN DO; MEMP=SYNAOX(SYN0)-1;
    SYNO=SYNO-1; END;
    DO KK=1 TO IVAR; SYN0=SYNO+1;
    SYNTAX(SYN0)=VARNAM(KK); SYNAOX(SYN0)=MEMP+1+VAROFF(KK); END;
    SUBSTR(MEMCCS,4*MEMP+1,4*LIT)=SUBSTR(CCS,1,4*LIT);
    MEMP=MEMP+LIT;
    CALL OUTCCM(OUTCARD,'1'B); IDLEF=SYNFRME; GOTO PUTFR;
CON(3):
/* #HELP CALLS HELP PROCEDURE AND SHOWS HELP FRAMES. */
HELPCALL: CALL HELP; GOTO DOIDLE;
CON(4):
/* #SH N DISPLAYS SYNTAX FRAME N OR FIRST FRAME IF NO N PRESENT. */
SHOW: IF CONTROL4="*" THEN SYNFRAME=NDEF-1+CONTROL4;
    ELSE SYNFRAME=NDEF;
SHOWF: READ FILE (CAIFILE) INTO (FRAME) KEY (FRAMEP(SYNFRAME));
PUTSHOW:FIRST2=" LINES 1-10 FOLLOWING MAY BE EDITED AND COMPILED. FRAME #" ||SYNFRAME-NDEF+1;
        GOTO IDLE;
CON(5):
    /* #F0 SCROLLS THE SYNTAX FRAMES FORWARD. CANNOT GO PAST NDEF. */
    FORWARD: IF SYNFRAME<NLDEF THEN SYNFRAME=SYNFRAME+1;
            GOTO SHOWF;
CON(6):
    /* #BA SCROLLS THE SYNTAX FRAMES BACKWARDS. CANNOT GO PAST NDEF. */
    BACK: IF SYNFRAME>NFDEF THEN SYNFRAME=SYNFRAME-1; GOTO SHOWF;
CON(7):
    /* #LIST OUTPUTS LISTING OF SYNTAX NAMES AND PROGRAM VARIABLES. */
    LIST: FIRST2=" LISTING OF SYNTAX TYPES AND VARIABLES."
       PUT STRING (LASTIO) EDIT ((SYNTYX (I DO I=1 TC SYNO))
       (100 A(8)));
        GOTO IDLE;
CON(8):
    /* #DE NAME- OR #DE NAME1 - NAME2 DELETES FROM NAME TO END CR */
    DELETE: USENAM=CONNAME;
       CALL LOOKUP;
       KSTART=KPOS; USENAM=NAME2;
       IF DASH="-" THEN DO;
       LAST10="THE DELETE COMMAND MUST BE OF FORM #DE AAAAAAAA- OR
       #DE AAAAAAAA-BBBBBBBB WHERE THE AS AND BS ARE SYNTAX TYPES. PLEASE
       RETYPE AND TRY AGAIN."; GOTO IDLE; END;
       IF NAME2=BLANK THEN DO;
       SYNO=KPOS-1; MEMP=SYNADX(KPOS)-1; GOTO LIST; END;
       KPOS=KPOS+1;
       CALL LOOKUP;
       IDIFF=SYNADX(KPOS)-SYNADX(KSTART);
       IF IDIFF<1 THEN GOTO HELPCALL;
       DO KK=SYNADX(KPOS) TO MEMP;
       MEM(KK-IDIFF)=MEM(KK); END;
MEMP=MEMP-IDIFF; IDIST=KPOS-KSTART;
DO KK=KPOS TO SYNO;
SYNTYX(KK-IDIST)=SYNTYX(KK);
SYNAOX(KK-IDIST)=SYNAOX(KK)-IDIF;
END;
SYNO=SYNO-IDIST; GOTO LIST;

CON(9):
/* #LO NAME INITIALIZES MEM AND LOADS PROGRAM NAME FROM CAIFILE. */
LOADP: MEMP,SYNO=0; MEM=0;

CON(11):
/* #AO NAME ADDS THE PROGRAM NAME FROM CAIFILE TO WHAT IS IN MEM. */
ADDP: DO IDUM=1 TO NAME#
IF PNAME(IDUM)=CONNAME THEN GOTO LOADUP; END;

CON(13):
/* #PUBLISH OUTPUTS THE PROGRAM DIRECTORY ON THE TERMINAL. */
LISTP: FRAME=' LISTING OF PROGRAM LIBRARY';
PUT STRING (RESTFP) EDIT (NAME#, (PNAME(IDUM) DO IDUM=1 TO NAME#
1) (F(8), X(72), 36 (A(8), X(12))));
GOTO IDLE;

/* #LO OR #AD NAME DOES A CHAINED RECORD LOAD FROM CAIFILE. */
LOADUP: READ FILE (CAIFILE) INTO (FRAME) KEY (FRAMEP(PPOS(I CUM)));
GET STRING (FIRST2) EDIT (IVAR,LIT) (X(20),2 F(10));
GET STRING (LAST10) EDIT ((SYNTYX(SYNO+I),CORE(I))
DO I=1 TO IVAR) (A(8), F(4));
DO I=1 TO IVAR; SYNAOX(SYNO+I)=CORE(I)+MEMP; END;
SYNO=SYNO+IVAR;
DO I=0 TO LIT/200;
K=MIN(LIT,200);
READ FILE (CAIFILE) INTO (FRAME) KEY (FRAMEP(NEXTNO));
SUBSTR(MEM#CCS,4*MEMP+1,4*K)=LAST10;
LIT=LIT-K;
MEMP=MEMP+K; END;
CCENTRY=SYNTYX(1);
GOTO LIST;

CON(10):
/* #SA NAME WRITES CURRENT CONTENTS OF MEM ONTO CAIFILE WITH CHAINED */
/* RECORDS, UPDATES THE PROGRAM DIRECTORY, AND UPDATES RECORD USAGE. */
SAVEP: NAME#=NAME#+1;
PNAME(NAME#)=CONNAME;
CALL GETFREE(40);
PPOS(NAME#)=IFREE;
PUT STRING (FIRST?) EDIT (CONNAME, SYNO, MEMP)
(X(10), A(8), X(2), 2 F(10));
PUT STRING (LAST?) EDIT ((SYNTYX(I), SYNADX(I)
DO I=1 TO SYNO))(A(8), F(4));
MEMPT=0;
RECN0=FREE4;
DO I=0 TO MEMP/200;
K=MIN (MEMP-MEMPT,200);
CALL GETFREE(40);
NEXTNO=FREE4;
REWRITE FILE (CAIFILE) FROM (FRAME) KEY (RECN0);
RECN0=NEXTNO;
LAST0=SUBSTR(MEMCCS,4*MEMP+1,4*K);
MEMPT=MEMPT+K;
END;
NEXTNO=00000;
REWRITE FILE (CAIFILE) FROMM (FRAME) KEY (RECN0);
GOT() LISTP;

CCN(12):
/* #ER NAME ERASE NAME FROM PROGRAM DIRECTORY AND FREES ITS RECORDS. */
ERASE: DO IDUM=1 TO NAME#;
IF PNAME(IDUM)=CCNNAME THEN GOTO ERASEP; END;
GOTO LISTP;

ERASEP: I=PPOS(IDUM);
FREEP: READ FILE (CAIFILE) INTO (FRAME) KEY (FRAMEP(I));
FRAMEP(I)=0;
I=NEXTNO;
IF I=0 THEN GOTO FREEP;
DO I=IDUM TO NAME#-1;
PNAME(I)=PNAME(I+1);
PPOS(I)=PPOS(I+1); END;
NAME#=NAME#-1;
GOTO LISTP;
CON(16):
/* #OFF Rewrites the program directory and record usage on CAIFILE. */
OFF: READ FILE (CAIFILE) INTO (FRAME) KEY (FRAMEP(3));
   PUT STRING (RESTFR) EDIT (NAME#, (PNAME(I), PPOS(I))
   DO I=1 TO NAME#) (F(8), X(72), 36 (A(8), F(12)));
   REWRITE FILE (CAIFILE) FROM (FRAME) KEY (FRAMEP(3));
   READ FILE (CAIFILE) INTO (FRAME) KEY (0);
   PUT STRING (RESTFR) EDIT (FRAMEP) (100 F(8));
   REWRITE FILE (CAIFILE) FROM (FRAME) KEY (0);

CON(14):
/* #CANCEL immediately closes data sets and returns TO MTMT. */
CANCEL: CLOSE FILE (CAIFILE), FILE (SYSPRINT); STOP;

CON(17):
/* #BO was the bootstrap routine. Has been removed. */
   GOTO DOIDLE;

CON(18):
/* #FR N sets syntax frame to N and clears display. */
   FREEON: SYNFRAME=NFDEF+CTRL4-1; RESTFR=BLANK;
   GOTO PUTFOR;
/* GetFree procedure finds a free frame on CAIFILE if possible. */
GETFREE: PROCEDURE (IPARAM);
   DO IFREE=IPARAM TO NFDEF;
   IF FRAMEP(IFREE)=0 THEN GOTO GOTF; END;
   CARDOUT='SORRY, NO FREE FRAMES IN YOUR FILE. ERASE SOMETHING.';
   CALL OUTCOM(CARDOUT,'1'8); GOTO SHOWF;
GOTF: KEY9=IFREE; FRAMEP(IFREE)=IFREE;
   END;
/* CCM2260 procedure handles all communications with terminal. */
/* writes input control line on SYSPRINT, checks for system command, */
/* checks for trace on or trace off, and branches or returns as nec. */
CCM2260: PROCEDURE;
   CALL DSP2260(FRAME,2);
   PUT FILE (SYSPRINT) EDIT (FRAME(2)) (COL(1), A(80));
   IF CONTROL1='#' THEN RETURN;
   ICON=(INDEX(CONTROLS, CONTROL2)+1)/2;
IF ICON>18 THEN DO;
IF ICON=19 THEN TF='0'B; ELSE TF='1'B; RETURN; END;
IF ICON=-0 THEN GOTO CON(ICON);
CALL HELP;
END COM2260;

/* SKIPB PROCEDURE SKIPS LEADING BLANKS IN INPUT STREAM. */
SKIPB: PROCEDURE;
MSKIP: DO I=1 TO 80 WHILE (C(I)=' '); END;
IF I>80 THEN DO; CALL GETCD; GOTO MSKIP; END;
END SKIPB;

/* GETCD PROCEDURE READS A CARD IMAGE INTO CARD VARIABLE. GETS */
/* INPUT FROM TERMINAL INFRAME NOW. */
GETCD: PROCEDURE;
IF INLN=10 THEN DO;
FIRST2='•' MORE INPUT REQUESTED. YOU MUST START INPUT ON LINE
3 AND TYPE ATTENTION TO CONTINUE. •;
LAST10=BLANK;
CALL CCM2260; INLN=0; INFRAME=RESTFR; END;
INLN=INLN+1; CARDOUT,CARO=INLINE(INLN);
CALL OUTCOM(CARDOUT,'0'B);
I=1; END GETCD;

/* LOOKUP PROCEDURE FINDS A SYNTAX NAME OR GLOBAL VARIABLE. */
/* THE POSITION OF THE DESIRED NAME IS PUT INTO KPOS. */
LOOKUP: PROCEDURE;
DO KK=KPOS TO SYNO;
IF SYNTYX(KK)=USENAM THEN GOTO GGTNAME; END;
DO KK=KPOS-1 TO 0 BY -1;
IF SYNTYX(KK)=USENAM THEN GOTO GGTNAME; END;
CARDOUT='•MISSING SYNTAX TYPE OR VARIABLE:'''|USENAM|;
CALL OUTCOM(CARDOUT,'1'B); GOTO DOIDIE;
GCTNAME=KPOS=KK; END LOOKUP;

/* OUTCOM PROCEDURE HANDLES ALL OUTPUT COMMUNICATION. WRITES */
/* ALL OUTPUT ON SYSPRINT FILE, PUTS INTO OUTFRAME IF DESIRED BY */
/* USER, AND OUTPUTS TO TERMINAL VIA COM2260 PROCEDURE. FIRST */
/* PARAMETER IS CARD IMAGE TO OUTPUT, SECOND IS FLAG TO FORCE OUTPUT. */
OUTCOM: PROCEDURE(CARDNAM, FLAG);
    DCL CARDNAM CHAR (*), FLAG BIT(1);
    PUT FILE (SYSPRINT) EDIT (CARDNAM) (COL(1), A);
    IF ~FLAG & ~TF THEN GO TO RESETC;
    OUTLN=OUTLN+1;
    OUTLINE(OUTLN)=CARDNAM;
    IF OUTLN=10 & FLAG THEN DC;
    OUTLN=0; RESTFR=OUTFRAME;
    FIRST2=' OUTPUT FROM PROGRAM. JUST TYPE ATTENTION TO CONTINUE.'; CALL COM226C; OUTFRAME=BLANK; END;
RESETC: CARDNAM=BLANK; IOUTC=1;
    END OUTCOM;
    END CAISYS;
    END METACAI;