STRUCTURING PROGRAMS FOR MULTIPROGRAM TIME-SHARING ON-LINE APPLICATIONS*

Kenneth Lock
California Institute of Technology
Pasadena, California

INTRODUCTION

The modern art of computation has developed from plugboard programming through the stored machine instruction programs controlled by the users on the consoles, then to problem-oriented symbolic programs computed in the batch mode, towards the on-line computing during which the users have a large amount of control over their programs. The lower cost per computation and flexibilities of a large capacity high-speed computer naturally lead us to consider the provision of on-line computing service to several users on a single high-performance machine in a time-sharing mode, rather than several smaller machines, one for each individual. To maximize the efficiency of a man-machine team working in an on-line computing mode, it is desirable to let the man choose the language—say English—for communication and to let the machine do the translation. This idealistic goal is not impossible, but is currently impractical. A good compromise is to select as the user language a formal language such as ALGOL, FORTRAN or LISP which has a set of explicit syntactical rules and a small set of basic vocabulary. The user then may extend the vocabulary by declarative statements and communicate with the machine in the extended vocabulary. Due to frequent message exchanges between the man and the machine during on-line computing, the machine representation of users' programs must be easy to modify at the source language level. The technological trend towards large random access memory suggests the retention of several users' programs in core simultaneously, hence mutual memory protection must be ensured.

This paper describes a scheme of structuring the users' ALGOL programs in accordance with the syntactical unit of a statement. The scheme enables the user to make modifications to his source language program at the statement level without recompiling the complete program. The same structure is used to provide the logic sequence of executing statements and to ensure memory protection among users. The next section describes the operating environment of on-line computing which justifies the scheme presented in this paper. The following section reviews the recursive definition of a statement in ALGOL as a syntactical unit which is used as the unit of communication from the user to the machine as well as the building block of the program structure in the machine. The next to the last section

*The study is partly supported by National Science Foundation Grant GP4264.
describes the statement-oriented program structure, and the final section shows the role played by the program structure for multiprogram time-shared on-line computations.

THE ENVIRONMENT IN ON-LINE COMPUTATIONS

There are two modes in on-line computation: constructing the program, and executing the program. Since the programmer is constructing the program on-line piece by piece, it is desirable to specify a minimum number of rules—either things the programmer is not allowed to do, or actions the programmer may take. In either case the programmer will not be burdened with remembering many rules. When executing the program on-line, the user must be able to exercise controls to start and stop the computation at will. The construction of a machine code program on an operator's console imposes a very simple rule on program modification, namely that any single instruction may be independently changed. The execution of a program on an operator's console provides complete control at the machine instruction level, namely that the program may be started or stopped at any specific instruction, or that the program may be stepped through. However, the direct use of an operator's console for on-line computing was discarded on account of the weakness in using a machine code language for program construction and the wastefulness of computer time due to human intervention. The introduction of high-level programming languages and batch operation eliminated the above shortcomings and at the same time ruled out the features of on-line computation.

From the above analysis, we can say that an acceptable on-line computing system must offer each user an input/output device. From this device, he may construct his program piece by piece in a high-level programming language, in which a statement is the building block. When executing the program, he may control the sequence by starting, stopping or stepping through his program at the statement level. For economy, the system must be time-shared among several users to minimize system idle time. From a user's point of view, he enjoys the advantages of an operator's console and a high-level programming language. In this environment, the following are taken as the design specifications:

1. The programming language must be easy to learn and powerful in expressing algorithms. The syntax of the language must allow easy extension to cope with applications such as symbol manipulations. ALGOL 60 is considered as a promising language.

2. The source language program should not be completely compiled into a single machine code program such that local changes in source program only require local modifications to its machine representation. An incremental compiler is required.

3. The communication between the user and the machine should be machine independent. For example, the user may ask for the values of variables by specifying their symbolic names, rather than the actual locations in memory.

4. Several users on-line should share the processor, and all users' programs and data should be retained in core whenever possible, to minimize swapping.

5. A statement is taken as the basic unit of processing such that the user may start or stop his program at specified statements or may execute his program one statement at a time in a "step" mode.

A STATEMENT IN ALGOL

Since the publication of the "Revised Report on the Algorithmic Language ALGOL 60," suggestions have been made to generalize it. The following generalization of the definition of a statement is introduced here to give a simpler syntax and render it more suitable for on-line computations.

```
<program> ::= <statement>
<statement> ::= <unlabeled statement> | <label> : <statement>
              <unlabeled statement> ::= <block> | <declarative statement> | <assignment statement> | <conditional statement> | <for statement> | <go to statement> | empty | <line number>
<block> ::= begin <list of statements> end
<list of statements> ::= <statement> | <list of statements> ; <list of statements> | ( <lower limit>, <upper limit> )
<declarative statement> ::= <declaration>
<conditional statement> ::= if <boolean expression> then <statement> else <statement>
```

The inclusion of <line number> as unlabeled statement and (<lower limit>, <upper limit>) as
a list of statements is to facilitate on-line program manipulation. <line number> denotes a statement already compiled and assigned a unique line number for identification by the compiler. (<lower limit>, <upper limit>) denotes the list of already compiled statements bounded by the two limits which are identification line numbers. The section on Incremental Statement Compilation gives some examples of using line numbers to modify and reshuffle statements in a program.

All other undefined metalinguistic variables in the above have the same definitions as those given in the official report.²

The important differences between the above definition of a statement and that in reference 2 are the following: (1) No distinction is made between a compound statement and a block. (2) All declarations are treated as declarative statements and are allowed to appear anywhere in the block, not necessarily at the beginning. It is conceivable that in on-line computing, the user will benefit from such freedom in making declaration in the middle of a block. (3) Only one type of conditional statement is used: the (if then) is a redundant form of the (if then else) in which the statement that follows else is empty. The statement that follows then is no longer restricted to unconditional statement.

The above definition of a statement will be used as the smallest building block during the construction of a program on-line. The user may insert, add, delete or reshuffle any number of statements at a time, and the program structure is designed so that only local modifications are invoked.

THE STATEMENT-ORIENTED PROGRAM STRUCTURE

By using a statement as the building block of a program, we can structure the ALGOL program internally in the computer memory as a collection of multiply linked elements, one for each statement. An element has a set of structure parameters in the form of pointers that link the element to other elements so that the program-imposed ordering among statements is preserved. If the element represents a statement which can be decomposed into other statements such as a block, a conditional statement, a for-statement or a procedure declaration, the structure representing its component statements are linked to the element by pointers in it. Hence the number of pointers in an element depends on the type of the statement it represents.

An element in the program structure is composed from the following quantities:

1. A type indicator to specify the type of statement it represents.
2. A set of structure pointers to connect it into the program structure.
3. The compiled machine code for the statement: All references from one statement to others are made through the system interpretation on the program structure.
4. The source statement image: The statement image is retained to provide positive documentary features.
5. An identification for referencing: This system edited identification takes the form of a line number such that the user may refer to any specific statement in making a program change or in initiating execution.
6. Internal editing numbers: All quantities in an element are packed into a contiguous block of words. The size of the block depends on the statement. Two integers are used to specify the size of the block and the field that contains the statement image.

All quantities in an element are completely relocatable except the set of structure pointers. The blocks of words, one block for each element, are dynamically allocated in the memory. Words are taken from an available storage block of unused words to form new elements. The available storage block is shared by all users. It can be reclaimed when exhausted by a scheme similar to that of the garbage collector in LISP.⁴ In reclaiming the available storage block, all blocks of words being linked in the program structure are relocated with appropriate changes of structure pointers in the element so that a single available block of consecutive words can be reclaimed.

A simple example is given in Fig. 1 to illustrate the use of structure pointers as shown in Fig. 2.

In Figs. 1 and 2 we use S to denote a nondeclarative statement, B to denote a block, D to denote a declaration, BEX to denote a Boolean expression and FCL to denote a for-clause. The complete program can be treated as a single statement in the form of a block B enclosed by the statement brackets: begin in line 1 and end in line 1.*. This block consists of a list of 4 statements, where B₃

From the collection of the Computer History Museum (www.computerhistory.org)
is a block between line 4. and 4. *, and $D_2$ is a declaration.

The element representing the block $B$ has the following set of structure pointers (Fig. 2):

1. `begin`
2. $S_1$
3. $D_2$
4. `begin`
5. $S_{31}$
6. if $BEX_{321}$
7. then for $FCL_{3221}$
8. do $S_{3222}$
9. else $S_{323}$
10. $D_{33}$
11. $S_{34}$
4. * `end`
12. $S_4$
1. * `end`

Figure 1. A simple program in which the following abbreviations are used: (1) $B$, $S$, $D$, $BEX$, and $FCL$ represent respectively the syntactical units of block, statement, declaration, boolean expression, and for-clause; (2) $X_{ijk}$ is a component in the syntactical unit $Y_{ij}$.

The element representing the statement $S_1$ has pointers $f_1, h_1$.

The element representing the conditional statement $S_{32}$ between line 6. and 9. has pointers $f_{32}, h_{32}, t_{32}, f_{32}$ where $t_{32}$ points to the element representing the statement to be executed if $BEX_{321}$ is true. $f_{32}$ points to the element representing the statement to be executed if $BEX_{321}$ is false.

The element representing the for-statement $S_{322}$ has pointers $f_{322}, h_{322}$ and $d_{322}$ where $d_{322}$ points to the element representing the statement that follows do.

The $f$ pointer of the last statement in a block is a block return $BR$. The $f$ pointer in the statement within a conditional statement points to if-return $IR$. The $f$ pointer in the statement that follows do in a for-statement is a for-statement return, $FR$. The $f$ pointer in the last element of a list of declarative statements is a declaration return, $DR$. The $f$ pointer in the element representing a procedure body is a procedure body return, $PBR$. Using the above set of pointers, the structure of the program in Fig. 1 can be constructed as shown in Fig. 2.
Figure 2. The structure of the program in Fig. 1.
Figure 3 shows an element in the program structure. It is a block of \( \gamma \) words. The various quantities in an element are listed as follows:

\[
\begin{array}{c|c|c}
\text{TYPE} & \beta & \gamma \\
\hline
f & h & \\
\hline
P_1 & P_2 & \\
\hline
\text{line number} & \\
\hline
\text{PM code} & \\
\hline
\text{statement image} & \\
\end{array}
\]

Figure 3. An element in the program structure.

**TYPE** : a type indicator from block, declaration, procedure, if-statement, for-statement, others.

\( \beta \) : an integer specifying the location of the first word in statement image field relative to the first word in the element.

\( \gamma \) : an integer that specifies the number of words in the element.

\( f \) : a pointer to the element representing the next statement.

h : a pointer to the element representing the block in which the given element is a statement.

\( p_1, p_2 \) : pointers depending on the TYPE according to the following table:

<table>
<thead>
<tr>
<th>TYPE</th>
<th>( p_1 ) points to</th>
<th>( p_2 ) points to</th>
</tr>
</thead>
<tbody>
<tr>
<td>block</td>
<td>list of declaration</td>
<td>list of statements</td>
</tr>
<tr>
<td>declaration</td>
<td>not used</td>
<td>not used</td>
</tr>
<tr>
<td>procedure</td>
<td>not used</td>
<td>procedure body</td>
</tr>
<tr>
<td>if-statement</td>
<td>statement</td>
<td>after else</td>
</tr>
<tr>
<td>for-statement</td>
<td>not used</td>
<td>statement</td>
</tr>
<tr>
<td>others</td>
<td>not used</td>
<td>after do</td>
</tr>
</tbody>
</table>

**PM code** : Pseudo-machine code for the compiled statement. The PM code differs from the absolute machine code in the following ways:

1. All references to identifiers are indirectly addressed through non-relocatable entries in the user's symbol table.
2. The last instruction in a PM code block always returns control to the execution monitor in the system which selects the next element in the program structure for execution.

**statement image** : The source statement image is retained in its symbolic form.

THE ROLE OF PROGRAM STRUCTURE IN MULTIPROGRAM TIME-SHARED ON-LINE COMPUTATIONS

In this section, we will describe how the statement-oriented program structure in the last section can be used in multiprogram, time-shared on-line computations. The first part of this section describes the list-structure-like operations on the program structure during the program statement input and modifications. The second part shows the use of program structure during execution in keeping track of the next statement to be executed. In the final part the dynamic nature of the program structure is demonstrated to be extremely desirable in applications that involve frequent man-machine interactions and dynamic data structure.

**Incremental Statement Compilation and the Statement-Oriented Program Structure**

Conventional compilers translate the source language programs into relocatable codes, and the loader converts them into absolute code. This scheme usually produces an efficient object code; however, the complete process has to be repeated if any changes, however small, are made in the source language program. An incremental compiler is characterized by its ability to compile each statement independently, so that any local change in a statement calls only for recompilation of the statement, not the complete program. When the compiled program is structured as in the preceding section, statement insertions, deletions or modifications are han-
dled by adding, removing or replacing some elements in the program structure with appropriate changes in structure pointers. The dependencies between any two statements lie only in the common set of identifiers that appear in them and their relative location within a program. The latter is encoded into the set of structure pointers in the program structure. The identifier dependency among statements is made indirect through reference entries in the symbol table. Only one reference entry is used for each distinct identifier such that all statements can be independently compiled into PM codes. The contents in the reference entries are set dynamically during execution according to the declaration on the identifiers. Figure 4 shows the indirect dependence among statements through the symbol table and the program structure.

![Diagram](image)

Figure 4. Indirect dependence among statements through symbol table and program structure.

When the program is incrementally constructed on-line, some building code must be specified. With our statement-oriented program structure and the definition of a statement given earlier, the rule becomes very simple:

Any integral number of statements in the program structure, called “out-statements,” can be replaced by any integral number of newly specified statements, called “in-statements.”

Figure 5 shows several examples that represent integral number of statements and also some examples that do not represent integral number of statements.

Replacing no out-statements amounts to inserting in-statements. Specifying no in-statements amounts to deleting the out-statements. Some method of specifying the out-statements in the program for replacement must be provided. One way is probably to display the source program on a CRT and let the user mark, by light pen, the limits that enclose the out-statements. Another method is to associate each element in the program structure with an identification line when the statement for that element is compiled and connected into the structure. The user may subsequently refer to any element in the structure by its line number. The out-statements can be specified by a pair of line numbers \((l_1, l_2)\) which represent the first and the last of the out-statements, or an insertion point in the program structure when out-statements are empty. For ease of cross referencing, successive statements are assigned line numbers in an increasing order. All statements inserted between the statements numbered \(n\) and \(n + 1\) are numbered into sub-levels \(n.1, n.2,\) etc. Syntactically the out-statements can be defined as follows:

\[
\text{<out-statements>} :: = \text{<insertion point> |}
\]

\[
(\text{<lower limit>}, \text{<upper limit>})
\]

\[
\text{<insertion point>} :: = \text{<line number>} + | \text{<line number> -}
\]

\[
\text{<lower limit>} :: = \text{<upper limit>} :: = \text{<line number>}
\]

\[
\text{<line number>} :: = \text{<unsigned integer>} | \text{<line number> <line number>}
\]

**Examples:**

\(1, 2.4\)

\(1.2. +\)

\(2.3.4. –\)

**Semantics:**

\((\text{<lower limit>}, \text{<upper limit>})\) denotes the set of statements enclosed by \(\text{<lower limit>}\) and \(\text{<upper limit>}\) inclusively, e.g., \((2., 3.)\) in Fig. 6.

\(\text{<line number> +}\) specifies the point in program structure that follows the f pointer in the element identified by \(\text{<line number>}\), e.g., \(2. +\) and \(3. +\) in Fig. 6.

\(\text{<line number> -}\) specifies the point in the program structure that precedes the element identified by \(\text{<line number>}\), e.g., \(3. -, 4. -\) and \(5. -\) in Fig. 6.

The in-statements that replace the out-statements can be syntactically defined as follows:

\[
\text{<in-statement>} :: = \text{<statement> | <line number>}
\]

\[
(\text{<lower limit>}, \text{<upper limit>}) | \text{<in-statement> ; <in-statement>}
\]
A: = B + C; \( (1, 5) \); begin C: = 0 end; 7. 1.

Semantics

\(<\text{statement}>\) can be any ALGOL statement as defined above (A Statement in ALGOL).

\(<\text{line number}>\) denotes the statement already in the program structure identified by \(<\text{line number}>\).

\((<\text{lower limit}>, <\text{upper limit}>)\) denotes the
list of statements in the program structure that are inclusively enclosed by <lower limit> and <upper limit>.

When line numbers are used in forming an in-statement, they represent the statements already in the program structure. Copies of these elements are incorporated into new locations in the program structure; they are not automatically deleted from their old locations.

A compile command that alters, builds, or manipulates the program structure takes the form:

\[
\text{<compile command>} := \text{compile}<\text{out-statement}>, \\
\text{<in-statement}> \text{EOM}
\]

EOM is an action on the input device that will interrupt the machine and cause the monitor in the system to respond to the message.

Example:
Let Fig. 7a be some program structure, then the compile command

\[
\text{compile (1., 3.), A:=B; B:=0 EOM}
\]
changes the program structure into the form in Fig. 7b.

\[
\text{compile (1., 2.), begin (1., 2.) end EOM}
\]
changes it into the form in Fig. 7c which can be transformed to Fig. 7d by

\[
\text{compile 1. +, if BEX then 6. else C:=0 EOM}
\]

By using independent statement compilations and the program structure described in the preceding section, the user may manipulate his program quite freely provided that a statement is taken to be the smallest unit for manipulation. Since the user has to be familiar with the definition of a statement in ALGOL before he can express the problem algorithms in the language, the program manipulation rules based on the concept of a statement should become very natural and easy to apply for the user. This is analogous to a user manipulating his machine code program on a console, in which case the smallest unit he may change in his program is a single machine code instruction.

On-Line Control over Program Execution

After the source language program is converted into the statement oriented program structure, interactions among statements are made indirectly through the reference entries for identifiers in the symbol table and the set of structure pointers in the program structure. The last instruction in the pseudo-machine code for a statement always returns con-
Figure 7.
control to the execution monitor which, from the pointer to the element for the statement just executed, selects the next statement for execution. Due to the recursiveness of a statement in ALGOL, a push down list called ESL for execution status list is maintained for each user. The top element in ESL points to the current statement being executed, the element next below in ESL points to the statement of which the current statement is a component. For example, in Fig. 7d, when the statement A := B is being executed, the top element in ESL points to the element numbered 2, and the element next below in ESL points to the element numbered 1. Depending on the type of the element, the last instruction in the pseudomachine code returns control to different points in the execution monitor which takes action depending on the user’s operation mode.

The user’s operation mode is set by an execution command.

Syntax:

\[
\text{<execution command> ::= < start > | <step> | <stop>}
\]

\[
\text{<start> ::= execute <execution bounds> E0M}
\]

\[
\text{<execution bounds> ::= empty | (<starting point>, <stopping point>)}
\]

\[
\text{<step> ::= step <starting point> E0M}
\]

\[
\text{<stop> ::= E0M}
\]

\[
\text{<starting point> ::= <stopping point> ::= empty | <line number>}
\]

Examples:

\[
\text{execute E0M}
\]

\[
\text{execute (1. 1., 3. 5.) E0M}
\]

\[
\text{execute ( , 4. 5. 6.) E0M}
\]

\[
\text{step E0M}
\]

\[
\text{step 5. 6. E0M}
\]

\[
\text{E0M}
\]

Semantics:

A user’s program can be either in “execute mode” or “step mode.” A <start> will set the user into the execute mode. If a nonempty <execution bound> is specified, the program will start from the <starting point> and stop at the <stopping point>. An empty <starting point> implies the top element in ESL, and an empty <stopping point> implies an infinite line number. A <stop> sets the user’s program into step mode. If the <execution bounds> is empty, the program will continue from the statement currently being pointed by the top element in ESL and will come to a halt only if <stop> is initiated from the input device or it comes to the previously specified <stopping point> or a program stop. In step mode, execution is halted after each statement. A <step> instructs one statement to be executed. If the <starting point> in <step> is empty, the element in the program structure pointed by the top element in ESL is executed. Otherwise, <starting point> is set to be the top element in ESL with appropriate pop ups and push downs in ESL to maintain the proper block level being referenced, then the element pointed to in ESL is executed. In step mode, the execution of each statement provides the user certain trace information on the on-line output device such as the value of an expression. At this point we can again see the analogy to the control a programmer can exercise on his machine code program in the computer from an operator’s console. The <start>, <step> and <stop> commands are analogous to the start-, step- and stop-push buttons. The <execution limit> is analogous to setting the instruction counter which is, in our system, generalized into a push down list ESL. The control unit in a computer that maintains the correct execution order from one machine instruction to the next is conceptually extended in our system into the “execution monitor.” However there is the difference that in our system the user communicates in a problem-oriented language.

The execution monitor’s operation is described below by using a set of ALGOL-like statements. The following terminology is employed:

\[
\text{ESL [1]: the top element in the push down list ESL.}
\]

\[
\text{t(ESL [1]): the t-pointer in the element representing a block pointed by ESL [1]. t points to the structure that represents all the nondeclarative statements in the block.}
\]

\[
\text{f(ESL [1]): the f-pointer in the element representing a statement pointed by ESL [1]. f points to the next statement, namely the statement that follows the statement separator.}
\]

\[
\text{ts(ESL [1]): the ts-pointer in the element representing a conditional statement pointed by ESL [1]. ts points to the statement that follows then.}
\]

\[
\text{fs(ESL [1]): the fs-pointer in the element representing a conditional statement pointed by ESL [1]. fs points to the statement that follows else.}
\]
ds(ESL [1]): the ds-pointer in the element representing a for-statement pointed by ESL [1].

ds points to the statement that follows do.
push down A into B: all elements in the push down list B are pushed down one level and the quantity A becomes the top element in B, i.e., B[i] := B[i-1] for i > 2 and B[1] := A.
pop up B: all elements in the push down list B are popped up one level, i.e. B[i] := B[i+1] for i > 1.

The original top element in B is lost.

return control to the user: the user's program is halted and the system is ready to receive a message from the input device.

return control to PM(ESL [1]): go to execute the pseudomachine code compiled for the statement which is represented in the program structure as an element pointed by ESL [1].

output trace information: when in the step mode, the execution of each statement provides information on the execution result such as the value of an evaluated expression, and displays the next statement to be executed upon receiving step E09M.

initiate block entry procedure: save all current machine addresses for the identifiers declared in this block and load their new local machine addresses into their reference entries. If the block is entered recursively, savings are implemented into push down lists.

initiate block exit procedure: restore the machine addresses for the identifiers declared in this block to their values in the outer block which for recursively entered blocks were the top elements in their push down lists.

set up ESL in accordance with the designational expression: transfer out of a block is allowed in which case all the top elements in ESL will be popped up until the pointer to the block in which the designated statement is a component appears as the top element in ESL, then the designated statement is pushed in ESL. Each time a pointer to the block is popped up from ESL, the block exit procedure is initiated.

set up actual parameters: save all current machine addresses for the identifiers used as formal parameters in the procedure, into push down lists when it is recursively called, and load the machine addresses of words containing the actual parameters into the reference entries of these formal parameters.

initiate procedure exit: restore the machine addresses for the identifiers used as formal parameters in this procedure to their values in the block that initiated this procedure call.

The following ALGOL-like statements describe the execution which offers the user extensive controls over his program execution on line.

return from go-to statement:

if in step mode
then output trace information
else;
set up ESL in accordance with the designational expression;
go to execute next statement;
return from block:
if in step mode
then output trace information
else;
initiate block entry procedure;
push down ts(ESL [1]) into ESL;
go to execute next statement;
return from if-statement:
if in step mode
then output trace information
else;
if the Boolean expression is true
then push down ts(ESL [1]) into ESL
else push down fs(ESL [1]) into ESL;
go to execute next statement;
return from for-statement:
if in step mode
then output trace information
else;
if all elements in the for list are serviced
then ESL [1] := f(ESL [1])
else push down ds(ESL [1]) into ESL;
go to execute next statement;
return from procedure call:
if in step mode
then output trace information
else;
set up the actual parameters;
push down pointer to the procedure body into ESL;
go to execute next statement;
return from all other statements:
if in step mode
then output trace information
else;
ESL[1] := f(ESL[1]);
go to execute next statement;
execute next statement:
  if in execution mode and ESL [1] not equal to 
    to <stopping point>
    then go to continue
  else begin enter step mode;
    return control to the user
  end;
continue:
  if ESL [1] is a program return PR
    then begin enter step mode; return control to 
       the user
    end
  else
    if ESL [1] is a block return BR
      then begin initiate block exit procedure;
        pop up ESL;
        ESL [1]:=(ESL [1]);
        go to execute next statement
      end
    else
      if ESL [1] is an if return IR
        then begin pop up ESL;
          ESL [1]:=(ESL [1]);
          go to execute next statement
        end
      else
        if ESL [1] is a for return FR
          then begin pop up ESL;
            return control to PM(ESL [1])
          end
        else
          if ESL [1] is a procedure body return PBR
            then begin initiate procedure exit;
              pop up ESL;
              return control to PM(ESL [1])
            end
        else
          ESL [1] is a pointer to an element in the pro-
          gram structure:
          return control to PM(ESL [1])

The Statement-Oriented Program Structure Used
in Time-Shared Multiprogramming and Its Com-
patibility with Dynamic Data Structures

The dynamic nature of on-line computing calls
for a dynamic data structure as well as dynamic
program structure. Nonnumerical applications such
as analytical expression manipulations on comput-
ers will increase in efficiency and effectiveness if
they can be performed on-line. ALGOL can be
easily extended to manipulate list-structure-like
data. The use of a dynamic program structure is
completely compatible with dynamic data structure.
The same dynamic memory allocator will service all
users' programs and data structures.

Figure 8 shows the configuration for the multi-
program time-shared system. Each user's activity
in the system is represented by an I/O device, its
program structure, symbol table, data structure and
operation status, all properly linked under the user's
pointer. Since storage allocations for program and
data structures and the execution of their programs
are all under the control of the multiprogram
time-shared system, memory protection against
each other is assured. The system consists of an in-
cremental compiler, an execution monitor, an avail-
able storage block manager and a system monitor
that coordinates various phases of operations. Fig.
9 shows the organization that incorporates the
self-optimization technique of adapting a set of
monitor system parameters in accordance with the
operation environment. Such system parameters
may, for instance, cause the monitor to operate in
one of several possible modes. In multiprogram
time-shared on-line computations, there is al-
ways the question of whether all users' programs
and data should be retained in core, or should only
one be in core with swap between users. Our solu-
tion is to let the operation environment dictate the
mode: if all users' programs and data can be com-
fortably accommodated in core, they will all remain
in core; otherwise they will be divided into groups
and swap among groups. The actual rules used for
adapting the monitor parameters are still subject to
experimentation. Since the multiprogram time-
shared system should be in core all the time, it
should be constructed so that read-only memory
can be used to store them.

CONCLUSION

A multiprogram time-shared system based on
the concept presented in this report has been under
implementation as an experimental project at the
California Institute of Technology. Invariably many
of its details have been modified to suit the particu-
lar hardware which consists of an IBM 7040 com-
puter, a 7288 multiplexor and several Institute-
developed typewriter consoles.
In conclusion, we believe that the use of incremental compilation, system-controlled execution, dynamically structured programs and data can offer the users the power of programming in a high-level algorithmic language and the advantage of interacting with the machine by means of an on-line console. The time-sharing mode further makes such operation economically acceptable.

Figure 8. The multiprogram time-shared system configuration.
The differences of this system from other similar systems are the following:

1. Statements in our system are compiled incrementally into directly executable codes. System interpretation is called for only between statements.
2. Several users may be accommodated simultaneously in core memory.
3. Easily extendable to cope with applications that call for dynamic data structure such as algebraic expression manipulation.

The study reported in this paper also reflects study of the computer organization for on-line time-sharing applications. Some applications are given below:

1. The incremental compilation achieved by indirectly addressing all operands through their reference entries suggests a small very-high-speed memory, functioning much like the index registers, to be used by all identifiers' reference entries.
2. The dynamic nature of multiprogram on-line computation should have a strong influence on memory organization. The algorithms trying to maximize the utilization of computer memory without sacrificing computing speed and programming flexibility should be investigated for possible direct incorporation into hardware configurations. For the same reason that arithmetic unit is used to perform arithmetics and data channels for input and output, special processors should be designed to allocate and relocate users' areas in memories possibly in parallel with the main computation.
3. The central control unit in a computer used for multiprogramming should be re-
sponsible for scheduling various programs to various processors. The organization of the central control unit must also reflect the nature of man-machine interactions and the types of control statements in the programming language.

4. The encoding of information, numeric or symbolic, into computer words should include type indication such that, for example, arithmetic operations performed on nonnumeric quantities can be detected as errors. This redundancy in information representation can be used to provide some error check during execution as well as to provide a simpler machine instruction set. For example, the same arithmetic instruction can be used for both floating point and fixed point numbers if the number representation suggests its type and whose indication is decoded accordingly in the arithmetic unit.

REFERENCES


