Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022

A general purpose macrogenerator

By C. Strachey*

The macrogenerator described in this paper is a symbol string processor, both its input and its output being strings of symbols. It operates by a form of substitution which is completely general in its application in that it is allowed anywhere. The result is a powerful system including such features as recursive functions and conditional expressions which can be implemented with very few instructions.

Part 1 describes the operation of the macrogenerator and gives some indication of how it has been used at Cambridge. Part 2 contains a sufficiently detailed account of its implementation to make it a relatively simple task to transfer it to any suitable computer.

Part 2 describes in some detail an implementation of the general purpose macrogenerator described in Part 1. The implementation is based on the use of a single stack and is described both as a series of transformations on the state of the stack, and by a CPL program. Various error checking features are described which greatly simplify the discovery of errors in macro programs.

Part 1

1. Introduction

A *macrogenerator* is usually found in close association with a symbolic assembly routine. In most cases this association is so close that it is very hard to distinguish between them, and the system becomes known as a *macro-assembler*. In this form, and particularly so when a card input system is used, it is a program which allows the use of macro-instructions which it replaces by an appropriate sequence of machine instructions after substituting the actual parameters provided in the macro-call for the formal parameters which are used in the schemata defining the macro-instructions. Thus the macro-assembler is often considered as extending the instruction repertory of the computer, and for many purposes this is the most convenient way of looking at a combined macro-assembler system. However, such a system is in fact a combination of two distinct programs, and any overall view of the process conceals the separate effects of the component parts, which may perhaps repay a closer investigation.

We are not concerned here with the second part of the combined system, as this is merely an ordinary symbolic assembly routine. The first part, which might be called the macro-generator, is in effect a processor whose input and output are both sequences of symbolic instructions which generally occupy one line of text (or one card) each. Its operation, apart from the straightforward copying of input to output, is to replace one line of input by one or more lines of appropriately defined output, after performing some substitutions of actual parameters taken from parts of the input line for the formal parameters used in the definition. When the operation of the macro-generator is thus isolated it is obvious that it suffers from certain limitations, the most important of which follows from the fact that the result of any macro-generation must be one or more complete instructions or lines of text. This means that it is not possible to have macro-definitions that specify only parts of instructions (such as an address), so that one cannot use a macro-call whose actual parameters are other macro-calls.

We can regard a macro-call as the application of a function (defined by one of the macro-definitions) to its arguments (the actual parameters in the macro-call). In these terms the limitation on the generality of the macro-generator described above is that the domain and range of these macro-functions do not overlap; the domain, being actual parameters, is parts of instructions, while the range is one or more complete instructions.

The macro-generator described in this paper is both "general" in the sense that it does not suffer from this asymmetry of domain and range, and "general-purpose" in the sense that it is not associated with a particular symbolic assembly program but is a separate program in its own right.

2. Informal description

The General Purpose Macro-generator (GPM) is a symbol-stream processor. It takes as its input a stream of characters and produces as its output another stream of characters which is produced from the input by direct copying except in the case of *macro-calls* in the input stream, which are "evaluated" in a way which is described below before they are put into the output stream.

2.1 *Macro-call*

A macro-call consists of a macro-name and a list of the actual parameters, each separated by a comma.

* *Project MAC, Massachusetts Institute of Technology, Cambridge* 39, *Mass. Formerly of University Mathematical Laboratory, Corn Exchange St., Cambridge, England.*

The name is preceded by a section sign (§) and the last parameter followed by a semicolon, e.g.

§*RCVF*,305, 1;

Here *RCVF* is the name of a macro which uses two parameters. In this call the actual parameters supplied are 305 and 1.

A macro which uses no parameters would be called by preceding the name by a section sign and following it by a semicolon, e.g.

§*LINK*;

2.2 *Evaluation*

Before this macro-call can be evaluated, the macro must have been defined by associating its name with a symbol string. This string may contain the special symbols ~1, ~2, etc., which stand for the first, second, etc., formal parameters; the symbol ~0 stands for the name of the macro being evaluated. When the macro-call is evaluated, these symbols are replaced by a direct copy of the actual parameters supplied in the call.

Thus, for example, if the macro-name *ABC* has been associated with the string

AB~1*C*~2*AB*

the macro-call

§*ABC*,*XY*,*PQ*;

will produce the output

ABXYCPQAB

The system is completely general and it is possible to use a macro-call in place of or in conjunction with a symbol string anywhere. In particular macro-calls are allowed in the actual parameters of other macro-calls (including the name) and also in the defining string. The following examples demonstrate this point:

Suppose we have defined the following macro-names

Name	*Associated string*
A	*A*~1*A*
B	*B*§*A*,*X*~1*X*;*B*
APA	*P*~1~1*P*

we should then get the following results:—

Macro-call	*Result of evaluation*
§*A*,*C*;	*ACA*
§*A*,*ACA*; §*A*,§*A*,*C*;;	*AACAA*
§*A*,*XDX*;	*AXDXA*
§*B*,*D*;	*BAXDXAB*
§*A*,*P*;	*APA*
§*APA*,*Y*; §§*A*,*P*;,*Y*;	*PYYP*

2.3 *String quotes*

Enclosing any string in the string quotes ⟨. . .⟩ has the effect of preventing evaluation of any macro-calls inside; in place of an evaluation, however, one "layer" of string quotes is removed.

Examples (using the macros defined above):

Input String	*Result*
Q⟨§*A*,*C*;⟩*R*	*Q*§*A*,*C*;*R*
§*A*,⟨§*A*,*X*;⟩;	*A*§*A*,*X*;*A*
§*B*,⟨§*A*,*X*;⟩;	*BAX*§*A*,*X*;*XAB*
Q⟨§⟩*R*⟨;⟩	*Q*§*R*;
Q⟨⟨§*A*,*C*;⟩⟩*R*	*Q*⟨§*A*,*C*;⟩*R*

The use of string quotes makes it possible to include any symbol in the output stream except an unmatched opening or closing string quote.

2.4 *Definitions* (*Part* 1)

A macro is defined by associating its name with a specific symbol string. This is performed by a special macro *DEF* which has been written in machine code and included in the system. *DEF* takes two arguments: the name of the macro to be defined, and the defining symbol string. It is usual to enclose the symbol string in string quotes in order to prevent any macro-calls or uses of formal parameters from being effective during the process of definition (i.e. when the symbol string is being treated as an argument for *DEF*). These quotes will be removed by the normal process of evaluating the arguments for *DEF*.

Examples:

The macros *A*,*B* and *APA* used above would be defined by the following macro-calls.

§*DEF*,*A*,⟨*A*~1*A*⟩;
§*DEF*,*B*,⟨*B*§*A*,*X*~1*X*;*B*⟩;
§*DEF*,*APA*,⟨*P*~1~1*P*⟩;

As definition is performed by an ordinary macro-call (and not a special process) the system ensures that it is possible to carry out a definition anywhere it is possible to use a macro-call. In particular a definition can be included in an actual parameter for a macro-call and hence in the symbol string defining a macro (as this is merely part of an actual parameter for a call of the macro *DEF*).

In general the actual parameter list of a macro-call is lost when the call has been completed, and this applies also to definitions which are part of the list. Definitions of this sort are therefore temporary and their scope is confined to this particular macro-call. If a macro name which has already been defined is defined again by a call of *DEF*, the latest definition supersedes the earlier one, though without destroying it. If the later definition is a temporary one, the earlier one will become effective again at the end of the later ones' scope. This means that the names used for temporary macro-definitions are local and can be chosen without considering any larger context. A fuller account of the effect of using definitions inside other macro-calls is given in Section 2.6.

Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022

2.5 *Sequence of operations*

The input stream is scanned from left to right and copied to the output until a macro-call is encountered. This is then evaluated and the result copied to the output. The evaluation of a macro-call is performed in the following stages.

(1) The macro-name and the actual parameters are evaluated in sequence from left to right. This process involves in its turn evaluating any macro-calls which occur so that the whole process of evaluation is a recursive one.

(2) When the argument list is complete—i.e. when the semicolon terminating the macro-call is encountered—the current list of definitions (known as the *environment chain*) is scanned in search of the name of the macro now being evaluated. The environment chain consists of name-value pairs which have been established by calls of the macro *DEF*, and it is scanned in reverse chronological order—i.e. from the most recent additions towards the oldest entry (which is, in point of fact, the definition of *DEF* itself). The scanning stops at the first entry with the correct name, so that the most recent definition is used. If no corresponding entry is found, there is an error exit from the program.

(3) The symbol string corresponding to the macro-name is now scanned in the same way as the original input stream (so that any macro-calls inside it will be evaluated) except that occurrences, if any, of the symbols $\sim r$ where $r = 0, 1, 2 \ldots$ are replaced by exact copies of the corresponding actual parameter, which has by now been evaluated. ~ 0 corresponds to the macro-name, ~ 1 to the first actual parameter, etc.; if r is greater than the number of actual parameters supplied, there is an error exit from the program. Note that in this operation the symbol string comprising the actual parameter is copied directly without any further evaluating processes being performed on it. The result of the macro-call is the output produced by this scan.

(4) On reaching the end of the defining string the argument list (i.e. the macro-name and actual parameters) are lost and any definitions which may have been added to the environment chain in the course of its evaluation are deleted.

(5) Scanning of the input stream is then resumed at the point where it was interrupted by the final semicolon of the macro-call.

It is worth noting that all scanning is strictly from left to right, that each macro is applied (i.e. its defining string is scanned) immediately the terminating semicolon of its call is encountered, and that supplying a macro with more parameters than it needs produces no ill effects.

2.6 *Definitions* (*Part* 2)

We can now describe the effect of the macro *DEF* rather more precisely; from this it should be possible to predict the result of evaluating any expression containing calls for *DEF*.

The macro *DEF* has an ordinary result which is the name-value pair, and a side effect of attaching this result to the environment chain. The actual form of the result can be represented by the triplet $(\epsilon_0{:}\sim 1{:}\sim 2\varpi)$ where ~ 1, ~ 2 have their usual meanings—i.e. the evaluated actual parameters of the call for *DEF*, ϵ_0 is a pointer to the current head of the environment chain, and ϖ is an internal termination symbol. The side effect is to install the ϵ_0 in this result as being the new head of the environment chain. In general neither ϵ_0 nor ϖ correspond to any external character, and any attempt to output them, or strings containing them, may lead to errors.

When the argument list of any macro-call is lost, the environment chain is modified in such a way that any name-value pairs which are both in the environment chain and in the argument list are removed from the environment chain. This chain is suitably adjusted so that no other definitions (either previous or subsequent to these) are lost.

Examples:

$$\S A,X,U,\S DEF,A,\langle\sim 1\sim 2\sim 1\rangle;;$$

has the effect of using a local definition of A in its own call, so that the result is XUX. This form gives the same sort of effect as the CPL* expression

$$f[\ldots]\,\textbf{\textit{where}}\,f[x] = \ldots$$

An elegant example of its use is the macro *Suc* defined by

$$\S DEF,Suc,\langle\S 1,2,3,4,5,6,7,8,9,10,\S DEF,1,\langle\sim\rangle\sim 1;;\rangle;$$

The effect of a call, say, $\S Suc,3;$ is to make a temporary definition of a macro with name 1 and defining string ~ 3, and then immediately to call it with arguments 2,3, . . . 10. The result is 4 and in fact $\S Suc,r; = r + 1$ for $r = 0, 1, \ldots 9$ so that *Suc* produces the successor of a digit. The temporary macro has the name 1 so that $\S Suc,0;$ which will produce the defining string ~ 0, will give the correct result.

The effect of a conditional expression can be obtained by making use of the fact that only the later of two definitions of the same macro is used. Thus the expression

$$\alpha = \beta \rightarrow \gamma, \delta$$

where α, β, γ and δ are strings, perhaps containing macro-calls, can be translated

$$\S\alpha, \S DEF,\alpha,\langle\delta\rangle;\S DEF, \beta, \langle\gamma\rangle;;$$

* See (Barron *et al.*, 1963); also *CPL Elementary Programming Manual*, Edition 2, 1965.

Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022

This defines α to be δ and then β to be γ, and then immediately evaluates §α;. If $\alpha = \beta$, the second definition will be used, if not, the first.

An example of this technique, making use of the macro *Suc* defined above, is

§*DEF*,*Successor*,⟨§~2, §*DEF*,~2,~1⟨,§*Suc*,⟩~2⟨;⟩;
§*DEF*,9,⟨§*Suc*,⟩~1⟨;,0⟩;;⟩;

which gives the successor of a two-digit number. Thus §*Successor*, α,β; first defines β by the string

α,§*Suc*,β;

then defines 9 by the string

§*Suc*,α;,0

and then evaluates §β;.

Thus if $\beta \neq 9$ the result is α, $\beta + 1$, while if $\beta = 9$ the result is $\alpha + 1$,0.

These examples also show the use of string quotes, particularly in the second parameter of *DEF*, to control the stage at which various evaluations are carried out, and to incorporate free variables into the definitions of temporary macros.

The next example shows the use of *DEF* in a context in which it is not an actual parameter.

§*DEF*,*FNPROD*,⟨§*DEF*,~1~2,⟨§⟩~1⟨,§⟩~2⟨,~1;;⟩;⟩;

The macro *FNPROD* effectively defines the functional product of its two arguments so that, for example, the macro call

§*FNPROD*,*Log*,*Sin*;

has exactly the same effect as the definition

§*DEF*,*LogSin*,⟨§*Log*,§*Sin*,~1;;⟩;

which defines *LogSin* to be the functional product of *Log* and *Sin*.

The definitions added to the environment chain as the result of a call for *FNPROD* are not part of any argument list. They form the result of the macro call and so are not lost when the call is completed. A special arrangement ensures that these definitions always remain in the machine on the environment chain, even when they are produced in circumstances in which other ordinary results would be immediately output.

The last example shows the use of an auxiliary macro definition which is recursive.

§*DEF*,*Sum*,⟨§*S*,~1,~2,0,§*DEF*,*S*,
⟨§~3,§*DEF*,~3,⟨§*S*,⟩§*Successor*,~1,~2;
⟨,⟩§*Suc*,~3;⟨;⟩;
§*DEF*,⟩~3⟨,~1⟨,⟩~2;;⟩;;⟩;

The effect of a call such as §*Sum*,α,β,γ; is to add the two-digit number α,β to the single digit γ, making use of the macros *Suc* and *Successor* defined above.

The first step in the call of §*Sum*,3,4,2; will be to call §*S*,3,4,0; after defining the macro *S* to be the string

§~3,§*DEF*,~3,⟨§*S*,⟩§*Successor*,~1,~2;⟨,⟩§*Suc*,~3;⟨;⟩;
§*DEF*,2,~1⟨,⟩~2;;

This in turn will call §0; after defining the macros 0 and 2 to be the strings §*S*,3,5,1; and 3,4 respectively. The resulting call for §*S*,3,5,1; will use the same definition for *S* and so will call §1; after defining the macros 1 and 2. The last step will result from the call of §*S*,3,6,2; which will define the macro 2 to be first §*S*,3,7,3; and then 3,6 and call §2;. Thus the final result will be 3,6.

2.7 *Other basic macros*

The basic macro *DEF* forms a name-value pair and puts it in the environment chain, but the only use we can so far make of this is to treat the value as an input string—i.e. to apply the macro. The basic code macro *VAL* allows us to obtain the value associated with a name without scanning it. Thus it copies the value string on to the stack as its result, without any evaluation, as if it were an argument called in by the symbol ~.

Note that §*DEF*,*X*,⟨α⟩; followed by §*VAL*,*X*; produces the result α whatever the string α contains (apart from unmatched string quotes), and that §*DEF*,*X*,⟨⟨α⟩⟩; followed by §*X*; produces the same result. Thus in most cases it is not strictly necessary to use *VAL* at all. However, its use corresponds rather closely to the important concept of obtaining the value associated with a name, while the symbols §. . .; can be considered as standing for "Apply the value of".

The basic macro *UPDATE*, which takes two arguments, has the same sort of effect as *DEF*, except that instead of establishing a new name-value pair on the environment chain, it alters the value associated with its first argument to be its second argument. There is a limitation on the use of *UPDATE* as the space available for the value is fixed by the first definition. The new string may be of equal length or shorter, but any attempt to update with a longer string will cause an error exit.

It is often useful to be able to perform simple arithmetic operations in the course of a macro call. These are made available by the basic machine code macros *BIN* and *DEC* which do decimal-binary and binary-decimal conversion, and the macro *BAR* which performs binary arithmetic. From these it is easy to define a set of macros which perform decimal arithmetic on digit string arguments. The details are given in Part II, Section 7.4.

3. Uses

The GPM was originally devised for the very practical purpose of helping to write a compiler for CPL. For reasons which are irrelevant here, a considerable section of this compiler consists of manipulations on a tree structure which are carried out with the aid of a stack. The technique we decided to use was to write the compiler originally in CPL itself, as being the best language we knew, both for ease of writing and for subsequently

Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022

understanding the operation of the program, and afterwards to hand-translate this into machine code.

However, as the mode of operation was to use a stack, the natural way to do this translation was not to go directly to machine code, but to go via series of "stack operations" each one of which performed a simple well-defined operation on the stack. Examples of these "control items" as they were called are

"Load bound variable number 3 onto the top of the stack"
"Apply the function on the top of the stack to the actual parameters on the stack immediately behind it"
"Replace the top two items on the stack by their sum."

Each of these control items can be coded in a few machine orders, possibly incorporating one or more parameters, and it was the desire to avoid writing (and punching) long strings of these which led to the development of the GPM. The method we have adopted is to define a number of the control items as macros and to translate the original CPL version of the compiler entirely in terms of these. There are now about 70 of these basic macros used for stack operations, and in effect they specify the operation code of a hypothetical computer organized around a stack.

We started on this course primarily for reasons of laziness, reinforced by the rationalization that reducing the amount of repetitious writing and punching would reduce the number of clerical errors and slips in the program. (Like so many rationalizations, this one happens to be perfectly valid while remaining a rationalization.) However, we soon found we had a number of other unexpected bonuses, and what started as a method of saving trouble very soon became a matter of policy. Initially we had intended to mix machine code and calls for macros, and the GPM was designed to make this possible. We later decided that, for the CPL compiler at any rate, all machine code sections would be incorporated as macro calls even if it involved defining a special-purpose macro which would only be called once in the entire compiler program.

The compiler at this stage consists of the following components:

(*a*) The original version written on paper in CPL. This is the basic document from which the others are derived.
(*b*) A hand-translated version of (*a*) written in the form of macro calls for control items. This version is punched and later read by the GPM.
(*c*) A set of definitions of the basic control items. This consists of a set of macro calls for *DEF* defining each of these in terms of a sequence of machine code instructions written in the assembly language of the machine.
(*d*) A copy of the GPM written in machine code.

We can now use the machine to produce an assembly language version of the compiler by installing (*d*) and then reading in (*c*) and (*b*) which produces the assembly language program as an output. This process, of course, has only to be done once—in theory. In practice it is repeated many times as errors in the compiler are discovered or improvements made.

This arrangement has a number of useful features:

(1) The macro form (*b*) is relatively easy to follow, and the translation from (*a*) to (*b*) very simple and straightforward.
(2) The macro form (*b*) is completely machine independent.
(3) The translation from (*b*) to machine code can be run on any machine for which the GPM is available—it is not dependent on having the target machine or its assembler working.

In some ways this is a sort of "bootstrapping" technique, but it has the advantage of not imposing any limits on the sophistication of the final program. The use of human translation from (*a*) to (*b*) allows a considerably more efficient compiler to be produced than any simple bootstrapping scheme could manage, as not only is the process under the programmer's control, but, if necessary, he is entirely free to introduce new macros to deal with tricky or unusual situations efficiently.

There are other advantages which flow from the delayed approach to machine code and the relative ease of altering the macro definitions. The control items can exist in at least two forms:

(1) A closed form, in which they appear as calls for short closed subroutines, perhaps associated with one or more parameters.
(2) An open form, in which they appear as short sections of machine code in which the parameters have been incorporated in some way.

The first form is extremely compact but rather slow, and the second much longer but also faster. We propose to use both forms in the CPL compiler—the fast form for the inner loops and the compact form for the less frequently used parts of the program. Moreover, by using another form of closed call, we can count the number of times each particular section of program is entered, and thus determine by experiment which parts of the program form the inner loops. With a deeply recursive program such as this, it is often very difficult to be certain how much the various parts of the program are used, so an automatic method like this is a considerable convenience.

Having decided which parts are to be macrogenerated as open and which as closed, we simply enclose the latter in the string quotes ⟨ ⟩ and macrogenerate twice. The first time we use the open definitions of the macros, when the closed macro calls being enclosed in quotes are merely copied without the enclosing quotes. The second macrogeneration is done with the closed definitions of the macros on the stack, and the resulting program combines the advantages of speed and compactness.

It is also possible to use the macrogenerator itself to do some stages of economization of the program. A

Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022

single example will show the general idea—others, some of a quite sophisticated nature, are being used, but their description belongs to a paper on the compiler rather than the GPM.

The compile-time stack, on which the control items operate, can be in one of two states: its top element may be in the accumulator (A-state) or in the main store (S-state). Transfers from one state to the other are very simple and only involve one or two orders. The control items also naturally start in one of these states and finish in one of them—not necessarily the same—so that they can be characterized as being $A \rightarrow A$, $A \rightarrow S$, $S \rightarrow A$ or $S \rightarrow S$. We would clearly like to be able to use these minimum or "natural" forms, particularly in the open form, and only insert the stack transformations when necessary.

This can easily be done by making the GPM keep track of the state of the stack at the end of the last control item—say by defining a macro *STACK* whose value is either A or S. Then a control item such as X which is of the form $S \rightarrow A$ would have the following macrodefinition:

§*DEF*,*X*,⟨§§*VAL*,*STACK*;*S*;§*UPDATE*,*STACK*,*A*;
(machine code for *X*)⟩;

and the macros *AS SA* would be defined for transforming the stack state. The macros *AA* and *SS* would be defined to be null as no transformation is required.

The effect now of a call for X is to call the macros *AS* or *SS* depending on the current value of *STACK*, to update *STACK* to the value A and then to output the relevant piece of machine code for the control item.

4. General comments

The GPM is of course a programming language of a sort. It has an extremely limited and rebarbative syntax which uses the symbols § and ; as brackets; in some ways it would be better-looking if these were replaced by [and] so that a macro call took the form

[*MACRONAME*, *ARGUMENTS*]

The symbols actually used have the advantage that they mean nothing in the normal assembly language of the Titan (Atlas 2) computer on which the GPM was first implemented, so that macro calls can be mixed with machine orders.

The actual symbols used are relatively unimportant; the facilities they provide are much more interesting, and one of the remarkable features of the GPM is its great power in spite of using so little apparatus. Conditional expressions, recursive definitions and similar advanced features all appeared as it were in the wash—albeit at the cost of a very considerable obscurity of the written programs.

It has been our experience that the GPM, while a very powerful tool in the hands of a ruthless programmer, is something of a trap for the sophisticated one. It contains in itself all the undesirable features of every possible machine code—in the sense of inviting endless tricks and time-wasting though fascinating exercises in ingenuity—without any of the irritating *ad hoc* features of real machines. It can also be almost impenetrably opaque, and even very experienced programmers indeed tend to spend hours simulating its action when one of their macro definitions goes wrong. Furthermore, it is remarkably good at using up machine time—fortunately the programs written for it are usually rather short.

One of its peculiar features is the existence of two sets of brackets which do not nest—these are ⟨ ⟩ and § ; . An examination of the examples in Section 2 shows that the two sorts operate more or less independently. Moreover as § and ; can be the results of a macro call, it is not even necessary that they should match—though, of course, in practice they always do.

Another unusual feature—at least as far as programming languages go—is the fact that definitions are produced as the result of a macro call. This is rather like having an ALGOL type procedure which produces a result which is another procedure declaration.

The GPM has a generic resemblance to the machine described by Dijkstra (1962) in his paper on Substitution Processes, though it was developed independently and, I think, from a different point of view. Dijkstra regards his machine as a theoretical exercise: the GPM was produced to meet a pressing practical need.

Indeed, one of the most attractive features of the GPM is the simplicity of its implementation; the first version for Titan only uses about 250 orders. The second section of this paper describes this implementation in sufficient detail to make it possible for a reasonably experienced programmer to reproduce it on any suitable machine in about a week.

Part 2

5. General outline

The implementation described below is based on the use of a single stack (or push-down list) and forms a good example of the great convenience and generality of this sort of organization.

The basic item on the stack is a character, as the GPM is a character stream processor. In addition to characters, it is necessary to store a number of pointers on the stack, and each of these, in general, will be the index number of a cell in the stack considered as a vector. (An alternative description would be the address of a cell in the stack relative to the start of the stack.) These pointers will often need more space than a single character and, indeed, there is an internal terminating symbol (written as ϖ) which is not an external character at all and so may be expected to need more space than a single character.

Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022

It would be possible to use a stack whose items were only just large enough to contain all the symbols required, and to arrange to use several consecutive cells to hold a pointer when this was required. In the interests of simplicity and ease of programming, however, the stack described below has cells which are large enough to hold a pointer—i.e. a full address. This means that considerably more space is required for the stack than is strictly necessary, and on Titan (Atlas 2), for example, where the first implementation of the GPM was run, this makes the stack about three times as large as it would be if the characters were tightly packed. However, as the initial use of the GPM was to assist in writing system programs, and as it is not required to be in the store while any other program is being run, the advantages of simplicity overweighed the disadvantages of using more space.

5.1 *Stack organization*

In the diagrams below the stack will be represented horizontally with its free end on the right. The pointer S always points to the next available cell. Each cell contains one character or pointer, and vertical lines are used to separate the cells (they are sometimes omitted).

The basic objects handled by the GPM are strings of characters; these may be of any length, and in the external format they are separated by commas or other special symbols. Internally they are represented by preceding them by a cell containing the total number of stack cells they occupy. Thus the string, ABC, would be represented on the stack by the four cells

$$|4|A|B|C|$$

The pointer H indicates the length-cell of the incomplete string at the top of the stack. While the string is being assembled this cell holds the number of extraneous cells between H and S; this is set to zero when a new string is started.

When the pointer H is zero (as opposed to the cell it points to) the string being assembled is output character by character as it is found.

5.2 *The main scan*

The operation of the GPM is to scan characters sequentially from left to right and to take certain actions, described below, on encountering one of the warning characters $\langle\ \rangle$ § , ; and $\sim$. The source of the characters scanned is determined by a pointer C. If $C = 0$ the source is the input stream; if not, C points to the stack cell which contains the next character to be scanned.

If the character scanned is not one of the warning characters it is copied to a destination determined by the pointer H. If $H = 0$ the destination is the output stream; if not, it is the top of the stack indicated by the pointer S which is then advanced.

In the CPL descriptions which follow, these two operations are performed by the routines *NextCh* and *Load*, respectively, which make use of a common working register A.

5.3 *String quotes*

The characters $\langle$ and $\rangle$ are always recognized by the scan. A count q is kept which starts at 1 and is increased by 1 for each $\langle$ and decreased by 1 for each $\rangle$ encountered.

When $q \geqslant 2$ the input is regarded as being inside string quotes, and no other warning characters are recognized. Further string quotes, either opening or closing, increment or decrement q as appropriate, and are also copied if the altered q is also $\geqslant 2$.

When $q = 1$ the effect of the character $\langle$ is to set $q = 2$ without copying. This has the effect of stripping off one layer of string quotes each time a string is scanned until $q = 1$ which corresponds to the unquoted input string. The effect of the character $\rangle$ with $q = 1$ is arbitrary—it has been chosen to terminate the scanning operation and leave the GPM.

When $q = 1$ the scan also recognizes the other warning characters and initiates the actions described in Section 6.

5.4 *The pointers F and P*

The warning character § indicates the start of a new macro call. The macro is not entered, however, until the occurrence of the matching warning character ; , which may be separated from the § by an unlimited number of other, possibly nesting, macro calls. This means that in a typical situation we may have a number of macro calls which have been initiated by a § but not yet entered, and at the same time be inside a number of macro calls which have been entered.

We therefore have two chains on the stack; one, whose start is indicated by F, gives the macro calls started by a § but not yet entered, the other, indicated by P, gives the macro calls already entered but not yet completed. When the scan encounters a § a new member of the F-chain is created; when it encounters a ; , the top member of the F-chain is removed and added to the P-chain (after suitable modifications). When the end of a string defining a macro is encountered (i.e. when a macro call is completed) the top member of the P-chain is removed together with its associated argument list, and the results (which comprise the rest of the stack above the argument list) are copied back over the abandoned P-chain entry and argument list.

The entries in the P-chain are in effect the links which specify how the scanning is to be resumed at the completion of the macro call. They also contain a pointer to the next entry in the P-chain, and a note of the total length of the link and argument list to simplify the exit procedure. They thus require three stack cells and take the form

$$\overset{\downarrow P}{\ldots\ |\phi|P_0|C_0|} \xleftarrow{\quad} \text{argument list} \xrightarrow{\quad} |\varpi|$$

$$\underleftrightarrow{\qquad\qquad\qquad\phi\qquad\qquad\qquad}$$

where P_0 and C_0 are the values of P and C when the ; was scanned, and ϕ is length of the argument list including a final terminator ϖ.

Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022

The terminator ϖ is added to allow a simple dynamic check that non-existent arguments are not called for; strictly speaking either ϕ or ϖ is redundant, but the inclusion of both considerably simplifies the programming.

As the F-chain entries have to be converted into P-chain entries when the macro call is entered, they also must consist of three cells. Only two of these are used: one points to the next member of the F-chain; the other holds the value of H when the § was encountered. This is restored when the macro is entered and the F-entry converted to a P-entry. The third cell is set to zero.

$$\ldots |H_0|\overset{\downarrow F}{F_0}|0|$$

Example

The macro X is defined by the string $X\sim 1$ and the macro Y by the string

$$Y\S X,\S X,Y\sim 1;;$$

The input string is $\S X,\S Y,Z;;$

Fig. 1 shows the state of the stack when the scan reaches the $\sim$1 in the string defining X for the first time.

C_0 and C_1 point to the last character in the input string and the last character in the string defining Y, respectively. If the input string is not itself another macro definition, C_0 will be zero. H_0, F_0 and P_0 are the initial values of H, F and P.

5.5 *The E-chain*

The macros already defined in the system are represented by a name-value pair which is attached to a chain starting with the pointer E. The entry corresponding to the macro X defined in the last example will take the form

$$\ldots |E_0|\underbrace{2|X}_{\text{name}}|\underbrace{4|X|\sim|1|\varpi}_{\text{value}}| \quad \leftarrow E\text{-chain}$$

Additions to the E-chain are made by using the special machine code macro *DEF*. Alterations to the value associated with a name can be made by using the machine code macro *UPDATE*.

E-chain entries are treated specially at the completion of any macro call which has generated them either as part of its argument list or as part of its result. E-chain entries in the argument list are detached from the E-chain and lost in the same way as other arguments. E-chain entries in the results are always copied back onto the stack (even if $H = 0$) and their pointers correspondingly adjusted.

5.6 *CPL programs**

The detailed description of the operation of the warning characters is given in Section 6 as a combination of diagrams representing the stack before and after, and CPL programs using the following conventions.

The stack is a vector ST of type **index** and the stack pointers S,E,H,P, F and C are all of type **index.** A and W are working variables.

The common subroutines which are used in various places in the detailed CPL programs are described in Section 5.7.

The basic scanning cycle is the following.

Start: *NextCh*
  if A = '⟨' **do** § $q := q + 1$; **go to** *Q2* §
  go to A = '§' → *Fn*,
    A = ',' → *NextItem*,
    A = ';' → *Apply*,
    A = '$\sim$' → *LoadArg*,
    A = *Marker* → *EndFn*,
    A = '⟩' → *Exit*,
    Copy

Copy: *Load*
Scan: **if** $q = 1$ **go to** *Start*
Q2: *NextCh*
  if A = '⟨' **do** § $q := q + 1$; **go to** *Copy* §
  if $A \neq$ '⟩' **go to** *Copy*
  $q := q - 1$
  go to $q = 1$ → *Start*, *Copy*

At the start of the program the initial entry is to *Start* with H,P,F and C all zero and $q = 1$. S is initially set to the first free cell above the machine code definitions, and E is set to the start of the chain of their name-value pairs.

5.7 *Common subroutines*

Input. The routine *NextCh* reads the next character from the current stream into A. If $C = 0$ the current stream is input, otherwise it will be the defining string of some previously defined macro. After finding the

* Readers unfamiliar with CPL will find some notational assistance in Appendix 1.

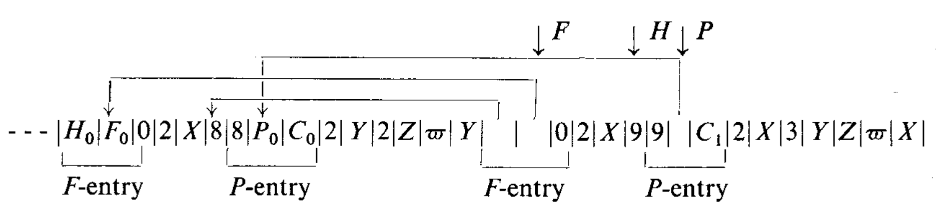


Fig. 1

Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022

next character, C is advanced if appropriate.

```
routine NextCh is
    § test C = 0
        then do  ReadSymbol [A]
        or do A, C := ST[C], C + 1
      return §
```

Output. The routine *Load* disposes of the character in A. If $H = 0$ the character is output directly, otherwise it is loaded onto the top of the stack—i.e. at $ST[S]$—and S is advanced.

```
routine Load is
    § test H = 0
        then do WriteSymbol [A]
        or do   ST[S], S := A, S + 1
      return §
```

Argument numbers

The function $Number[A]$ takes the single character in A and finds the equivalent binary integer. This function is necessary as the internal representation of the decimal digits is not always the corresponding binary integer. The use of this function may make it possible to refer to arguments with serial number 10 or more by using an appropriate single non-numerical character.

Marker

This is some recognizable integer which is not the internal representation of any external character. In the Titan implementation a one in the sign bit with zeros elsewhere is used. It is represented in the CPL programs as ϖ or *Marker*.

Machine language macros

It is necessary to introduce a few macros which are primitive—i.e. cannot be defined by a character string. These have to be written in machine code (or some other suitable language). They are distinguished from the ordinary macros in the environment chain by having the first cell of their value (which in the case of an ordinary macro would hold the length of the defining string) marked with a recognizable quantity. Then either it or the next cell will contain the address of the start of the machine code program. In the Titan implementation the marking is done by making the sign bit of this cell a one, the remaining bits holding the starting address.

The routine $JumpIfMarked[x]$ tests if x is marked in this way, and if so jumps to the indicated address. The machine code macros must finish with a special form of *EndFn* to copy back their results and the jump to *Start*.

6. Effect of warning characters

6.1 *The warning character* §

```
                                  ↓S
Initial State:  F0,H0,   -----|
                                  ↓F    ↓H ↓S
Final State:             -----|H0|F0| 0 | 0 |
```

```
Fn: H,S,F,ST[S], ST[S+1], ST[S+2], ST[S+3] :=
        S+3, S+4, S+1, H,F,0,0
    go to Start
```

This initiates a new function call by adding it to the F-chain, and starts a new item for the argument list, saving the old value of H. The empty cell between F and H will be used later when the function is applied (i.e. when the matching ; is reached).

6.2 *The warning character* ,

```
                      ↓H0             ↓S0
Initial State:  -----|θ0|---------|
                                      ↓H ↓S
Final State:    -----|θ0'|-------- |0|
```

where $\theta_0' = S_0 - H_0 - \theta_0$

θ_0 is the number of extraneous cells between H and S (i.e. cells containing housekeeping information which is not a part of the item starting at θ_0). θ_0' therefore contains the true length of the item starting there.

If $H = 0$ initially the characters being scanned are output at once, so the effect is merely to copy the comma.

```
NextItem: if H = 0 go to Copy
          H,S,ST[H], ST[S] :=
             S, S+1,S−H−ST
            [H],0
          go to Start
```

6.3 *The warning character* ;

If $H = 0$ initially, the semicolon is merely copied. If $H \neq 0$ we have

Initial State:

```
                   ↓F                         ↓H     ↓S
P0,C0----|θ0|----|H0|F0|0|(~0)|(~1)|--- |θ1|——|
```

Final State:

```
↓H           ↓P                             ↓S
|θ0'|------|φ|P0|C0|——|——|---|θ1'|——|ϖ|
             <——————————φ——————————>
```

where $\theta_0' = \theta_0 + \phi$

and $\theta_1' = S - H - \theta_1$ = true length of last argument.

ϕ is the number of fresh extraneous cells introduced by this macro call—i.e. the arguments together with the three initial and one final housekeeping cells.

If $H_0 = 0$ there is no θ_0 and so no θ_0', but the effect is otherwise the same.

The next step is to search the environment chain for the name of the macro. This is done by the routine call $Find[P+2]$. If the value corresponding to this is marked, the macro is a machine code one and is entered directly. If the value is not marked, C is set to its first character and scanning resumed. If the name does not appear in the environment chain there is an error exit with suitable monitor printing.

Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022

The effect of the routine *Find*[*w*] is to put the value corresponding to the name *w* into *W* as follows.

```
                    ↓w
Initial State: ---| l |  (name)  |----

                E-chain
Final State:   ↖ ↙            ↓W
----------| | l |  (name)  | |  (value)  |------
```

routine *Find*[*w*] **is**
  §1 $A,W := E,w$
    §2 **for** $r = 0$ **to** $ST[W]-1$ **do**
      if $ST[W+r] \neq ST[A+r+1]$ **go to** *Next*
      $W := A+1+ST[W]$
      return
  Next : $A := ST[A]$ §2
  repeat until $A < 0$
  go to *Monitor7* §1

Apply: **if** $H = 0$ **go to** *Copy*
  $F,P,H,S,ST[H],ST[S],ST[F-1],ST[F],ST[F+1] :=$
  $ST[F],F,ST[F-1],S+1,S-H,$ *Marker*, $S-F+2,P,C$
  unless H=0 **do** $ST[H] := ST[H]+ST[P-1]$
  Find $[P+2]$
  Jumpifmarked$[ST[W]]$
  $C := W+1$
  go to *Start*

6.4 *The warning character* ~

If P=0 the warning character is outside any macro call and is merely copied. Otherwise the next (single) character is read and the corresponding argument copied without alteration to the top of the stack (or output if H=0). If the argument list has too few arguments, there is an error exit.

Initial State:

```
      ↓P                                        ↓S
------|φ|P0|C0| (~0) || (~1) ||-----||——|ϖ|——|
```

Load Arg: **if** $P = 0$ **go to** *Copy*
  NextCh
  $W,W1 := P+2,$ *Number*$[A]$
  until $W1 = 0$ **do**
    § **if** $ST[W] =$ *Marker* **go to** *Monitor8*
      $W,W1 := W+ST[W],W1-1$ §
    if $ST[W] =$ *Marker* **go to** *Monitor8*
    for $r = 1$ **to** $ST[W]-1$ **do**
    § $A := ST[W+r]$
      Load §
    go to *Start*

6.5 *The warning character* ϖ

This character is an internal terminator and never occurs in the input stream. It is inserted automatically at the end of argument lists (by the warning character ;) and at the end of definition strings (by the macro *DEF*). It should never be output.

When it is encountered as a warning character it indicates the end of the defining string of the macro currently being scanned. The effect required, therefore, is to terminate this call, and to resume the scanning of the string it interrupted—i.e. approximately:

Initial State:

```
      ↓P                                          ↓S
-----|φ|P0|C0| (Argument list) |ϖ| (Results) |
      <----------------------->
```

Final State:

```
                 ↓S
----|  (Results)  |
```

The situation, however, is complicated by the need to deal with the environment chain, and by various alternative cases introduced by the several possible values of H. The first step is to remove any definitions in the argument list from the environment chain, and to adjust any definitions in the results so that the chain will be correct after removing the ϕ extraneous cells and copying back the results. This is done as shown in **Fig. 2.**

The members of the environment chain in the results are all reduced by ϕ, except for the oldest which is altered to point to the first member of the environment chain which is before the argument list. Any definitions in the argument list are ignored.

EndFn: $ST[S],A := E,S$
  while $ST[A] > P-1+ST[P-1]$ **do**
    $ST[A],A := ST[A]-ST[P-1],ST[A]$
  $W := ST[A]$
  while $W>P-1$ **do**
    $W := ST[W]$
  $ST[A] := W$
  $E := ST[S]$

E is first stored in $ST[S]$ in order to deal in a uniform manner with the case where there is only one definition in the results.

After this step there are three cases:

(i) $H = 0$

The results can only be definitions as all others will already have been output by *Load*.

Initial State:

```
             ↓P                               ↓S
H = 0 ----|φ|P0|C0|————————|ϖ|  ——|  ——|
          <------------------->     Defs only
                   φ
```

Final State:

```
                               ↓S
C0, H=0 -----|————|————|
```

(ii) $H>P$

H must be reduced by ϕ

Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022

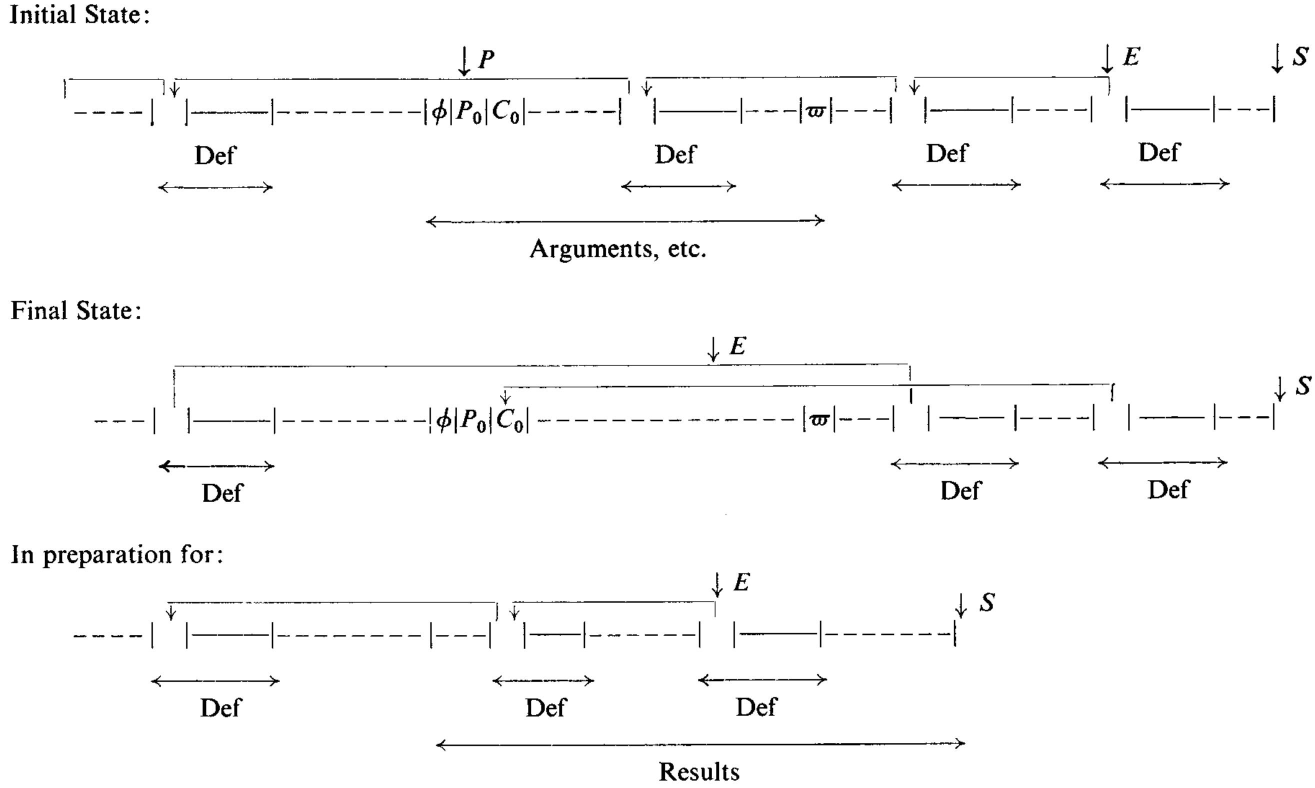
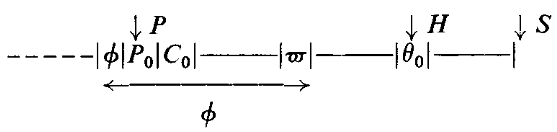
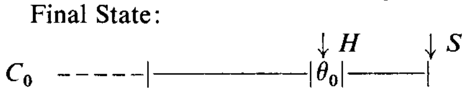


Fig. 2.

Initial State:



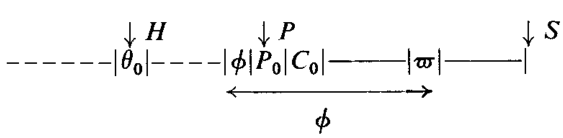
Final State:



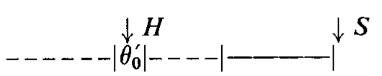
(iii) $H<P$

ST[*H*] which holds the number of extraneous cells must be reduced by ϕ

Initial State:



Final State:



where $\theta_0' = \theta_0 - \phi$

unless $H=0$ **do**
  test $H>P$
   then do $H := H-ST[P-1]$
   or do $ST[H] := ST[H]-ST[P-1]$
  $P,C,S,A,W := ST[P],ST[P+1],$
   $S-ST[P-1],P-1,P-1+ST[P-1]$
  until $A=S$ **do**
   $ST[A],A,W := ST[W],A+1,W+1$
  go to *Start*

6.6 *The warning character* $\rangle$

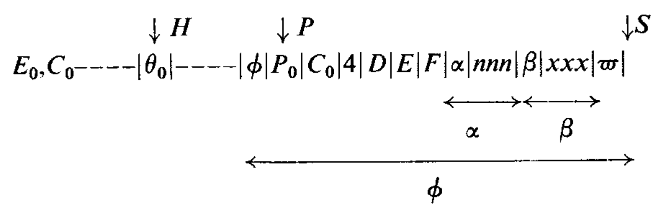
This is only a Warning Character if it occurs at the outermost level—i.e. unmatched by any opening string quote. In this case it terminates the program.

Exit: **finish**

7. Basic machine code macros

7.1 *DEF*

The state on entering the machine code after a call §*DEF,nnn,xxx*; will be



with C still equal to C_0.

Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022

At the end of the machine code we want the state to be

$$C_0,P_0 - - - - |\overset{\downarrow H}{\theta_0'}| - - - - - |\overset{\downarrow E}{E_0}|\underbrace{\alpha|nnn|}_{\alpha}\underbrace{\beta|xxx|}_{\beta}\varpi|^{\downarrow S}$$

where $\theta_0' = \theta_0 - \phi$.

If $H = 0$ the only difference is that $ST[H]$ is not relevant. Note that unlike the ordinary *EndFn* case the result of *DEF* is always left on the stack and never output even if $H = 0$.

DEF: **unless** $H=0$ **do** $ST[H] := ST[H]-ST[P-1]$
$P,E,W,ST[P-1] := ST[P],P-1,P,E$
until $W+6=S$ **do**
$ST[W],W := ST[W+6],W+1$
$S := W$
go to *Start*

7.2 *VAL*

The state on entering the machine code after a call §*VAL,nnn*; will be

$$- - - - - -|\ \overset{\nwarrow\swarrow E\text{-chain}}{}|\alpha|nnn|\beta|xxx|\varpi| - - - - - - |\overset{\downarrow H}{\theta_0}| - - - -$$

$$- - - -|\phi|\overset{\downarrow P}{P_0}|C_0|4|V|A|L|\alpha|nnn|\varpi|^{\downarrow S}$$

We want to load the string starting after β until (but not including) the first ϖ. If $H = 0$ this will be output, otherwise loaded onto the stack as the result of *VAL*. We can then use the ordinary *EndFn* sequence to remove the call for *VAL* and its argument list, and to copy back the result if any. We load the value until the first terminator instead of the whole length β, as it is possible that a call for *UPDATE* has replaced the original string by a shorter one. (See *UPDATE*.)

VAL: *Find*$[P+6]$
until $ST[W+1]=Marker$ **do**
§ $A,W := ST[W+1],W+1$
Load §
go to *EndFn*

7.3 *UPDATE*

The state on entering the machine code after a call for §*UPDATE,nnn,yy*; will be

$$- - - -|\ \overset{\nwarrow\swarrow E\text{-chain}}{}|\alpha|nnn|\underbrace{\beta|xxxx|}_{\beta}\varpi| - - - - -$$

$$- - - -|\phi|\overset{\downarrow P}{P_0}|C_0|7|U|P|D|A|T|E|\alpha|nnn|\underbrace{\gamma|yy|}_{\gamma}\varpi|^{\downarrow S}$$

We want to check that γ is not greater than β (and Monitor if it is) and then copy the string $yy\varpi$ (but not the γ) into the value string after β. We leave β undisturbed so that we can verify that no later *UPDATE* uses a string longer than the space available. We copy back the terminator so that subsequent uses of *VAL* will only produce the latest value. The final state is therefore

$$- - - -|\ \overset{\nwarrow\swarrow E\text{-chain}}{}|\alpha|nnn|\beta|yy|\varpi|x|\varpi| - - - -$$

$$- - - -|\phi|\overset{\downarrow P}{P_0}|C_0|7|U|P|D|A|T|E|\alpha|nnn|\gamma|yy|\varpi|^{\downarrow S}$$

and we return to *EndFn*.

UPDATE: *Find*$[P+9]$
$A := P+9+ST[P+9]$
if $ST[A]>ST[W]$ **go to** *Monitor*9
for $r=1$ **to** $ST[A]$ **do**
$ST[W+r] := ST[A+r]$
go to *EndFn*

7.4 *Arithmetic operations*

Integer arithmetic is provided with the aid of three machine code macros: *BIN* converts a digit string, possibly preceded by a sign, into a signed binary integer. (Note that the largest integer allowed is determined by the stack cell size; the result of *BIN* is a single cell.) *DEC* is the inverse operation—it converts a signed binary integer into a decimal digit string of characters, preceded by a minus if necessary. The argument of *DEC* is a binary number (not a character string) so that it can normally only be used immediately after a macro which produces one. *BAR* takes three arguments, the first being the character $+$, $-$, $\times$, / or *R*. The other two being binary numbers. It performs the indicated operation on these. §*BAR*, *R*, *x*, *y*; gives the remainder when *x* is divided by *y*.

The decimal arithmetic operations are defined as follows.

§*DEF*,+,⟨§*DEC*,§*BAR*,+,§*BIN*, ~1; , §*BIN*, ~2;;;⟩;
§*DEF*,−,⟨§*DEC*,§*BAR*,−,§*BIN*, ~1; , §*BIN*, ~2;;;⟩;
§*DEF*,×,⟨§*DEC*,§*BAR*,×,§*BIN*, ~1; , §*BIN*, ~2;;;⟩;
§*DEF*,*Quot*,⟨§*DEC*,§*BAR*,/,§*BIN*, ~1; , §*BIN*, ~2;;;⟩;
§*DEF*,*Rem*,⟨§*DEC*,§*BAR*,*R*,§*BIN*, ~1; , §*BIN*, ~2;;;⟩;

Thus the calls §+,*x*,*y*; , §−,*x*,*y*; etc., give the sum, difference, etc., of the decimal digit strings *x* and *y*. It would be perfectly possible to treat the macros +, −, ×, *Quot* and *Rem* as machine code macros. This would give faster operation but probably more machine code.

BIN

Initial State

$$- - - - -|\phi|\overset{\downarrow P}{P_0}|C_0|4|B|I|N|\alpha|dddd|\varpi|^{\downarrow S}$$

We want to load the binary equivalent of the string *dddd*

Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022

```
BIN: W,A := 0,ST[P+7] = '+'→P+8,
              ST[P+7] = '−'→P+8,
              P+7
     until ST[A] = Marker do
        W, A := 10W+Number[ST[A]], A+1
     S,ST[S] := S+1, ST[P+7] = '−'→ − W,W
     go to  EndFn
```

DEC

Initial State

$$\begin{array}{l} \qquad\quad\ \downarrow P \qquad\qquad\qquad\ \ \downarrow S \\ -----|\phi|P_0|C_0|4|D|E|C|2|x|\varpi| \end{array}$$

x is the binary number to be converted.

```
DEC: if W<0 do §W,A := −W,'−'
                   Load §
       W1 := 1
       until 10W1> W do W1 := 10W1
         § W,A,W1 := Quot[W,W1],Rem[W,W1],
                                     W1/10
           Load §
       repeat until W1<1
       go to EndFn
```

BAR

$$\begin{array}{l} \qquad\qquad\qquad\quad\ \downarrow P \qquad\qquad\qquad\qquad\qquad\ \downarrow S \\ \text{Initial State}------|\phi|P_0|C_0|4|B|A|R|2|f|2|x|2|y|\varpi| \end{array}$$

```
BAR: W,A := ST[P+9],ST[P+11]
       A := ST[P+7]='+'→W+A,
            ST[P+7]='−'→W−A,
            ST[P+7]='×'→W×A,
            ST[P+7]='/'→Quot[W,A],
                  Rem[W,A]
     Load
     go to EndFn
```

8. Error detection

The syntactic forms used by the GPM are simple but also confusing, and misprints and slips are very easy to make. In the simplest implementation, errors of input cause the program to run wild, and to produce curious and unpredictable results which are difficult or impossible to follow. The error monitor system described in this section does much to avoid this problem and makes it considerably easier to discover the slips and mistakes in the input stream. The cost, however, is to increase the size of the program considerably, and while it is probably essential to include some form of checking such as this in any implementation in general use, it would not be necessary to do so on a new machine where only tested inputs were to be used.

8.1 *Possible states*

There are a number of possible states of the macrogenerator, and in each one certain characters only are allowed.

(1) $q>1$

Inside string quotes any character can be scanned except a terminator ϖ which can only appear in the input stream in this state by a machine error.

(2) $q=1$

This is the main operative state and all the warning characters may be encountered. There are three states

(A) Direct input to output. This has $C=H=P=F=0$.

(B) After § and before the corresponding ; —i.e. while assembling an argument list. This has $H\neq 0$, $F\neq 0$ and $F>P$.

(C) After ; and before the ϖ of the macrodefinition—i.e. while scanning a macro definition string. This has $C\neq 0$, $P\neq 0$ and $P>F$.

8.2 *Effect of warning characters*

WARNING CHARACTER	A	B	C
§	*Fn* (→B)	*Fn* (→B)	*Fn* (→B)
,	*Copy*	*Next Item*	$H=0$ *Copy* $H\neq 0$ *NextItem*
;	*Copy*	*Apply* (→C)	*Monitor*1
~	*Copy*	$P=0$ *Monitor*2 $P>0$ *LoadArg*	*LoadArg*
ϖ	*Monitor*5	*Monitor*6	*EndFn* (→A, B or C)
> ($q=1$)	*Finish*	*Monitor*8	*Monitor*8

There are also checks which are applied to ensure that an argument called for exists (Monitors 3 and 4) and that a macro name applied (or used as an argument for *VAL*) has been defined (Monitor 7), that *UPDATE* arguments are not too long (Monitor 9), and that the argument of *BIN* contains only decimal digits (after a possible initial sign) (Monitor 10).

8.3 *Monitor output*

The monitor printing produced by any of these error entries consists of some indication of the cause of the error together, perhaps, with the name of the macro on the stack immediately in front of the P-pointer or the F-pointer. If the error is one of those from which recovery is possible (assuming that the cause has been correctly diagnosed) the ordinary action of the GPM is resumed after a suitable comment. If it is impossible to guess the intention of the user, a general monitor (Monitor 11) is given which prints the name and all the arguments of the current macro and the names of the macros in the P and F chains. It then loads a query on the stack (as an argument or result) and returns either to *Start* (states A and B) or to *EndFn* (state C). The effect of this is to make it possible to find further errors in the same input stream without requiring another run.

Experience has shown that it is advisable to limit the total number of characters output after any monitor, particularly if macrogeneration is always resumed after an error in this way.

The details of the monitor printing can best be ascertained from the CPL program of the entire GPM which is given in Appendix 2.

Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022

Appendix 1

Note on CPL

The CPL programs included in this paper should be comprehensible to anyone familiar with ALGOL without much difficulty. The following are the chief notational features which may be unfamiliar.

CPL	*Nearest ALGOL equivalent*
§ . . . . . . § §1 . . . . . . §1	**begin** . . . . **end**
$B \rightarrow x, y$ (expression)	**if** *B* **then** *x* **else** *y*
if *B* **do** *C*	**if** *B* **then** *C*;
test *B* **then do** *C*1 **or do** *C*2	**if** *B* **then** *C*1 **else** *C*2;
routine	**procedure**
return	signifies the dynamic end of a procedure body
[ ]	used for argument lists as well as array indices
a,b,c := *p,q,r*	Simultaneous assignment. This has no exact ALGOL equivalent. The three assignments *a* := *p*; *b* := *q*; *c* := *r*; are taken to happen simultaneously
for *r* = *a* **to** *b*	**for** *r* := *a* **step** 1 **until** *b*
while *B* **do** *C*	*L*: **if** *B* **then** **begin** *C*; **go to** *L* **end**;
C **repeat while** *B*	*L*: *C*; **if** *B* **then go to** *L*;

The negated CPL conditions also exist.

unless *B* **do** *C*	**if** ¬ *B* **then** *C*
until *B* **do** *C*	etc.
C **repeat until** *B*	

Comments in CPL are indicated by a double vertical bar on the left and extend to the end of the line.

Inside the string quotes '. . . .'
**n* stands for *newline*
**s* stands for *space*
**t* stands for *tab*.

Appendix 2

```
||CPL program for GPM

  let routine GPM[index n] be

||  n is the stack size allowed.  This should be as large as
||possible—say 10,000.

§1 prefer index

    let A, W all be index
    and H, P, F, C all = 0
    and S, E, q, Marker = 39, 33, 1, −2 ↑ 20
    and ST = Newarray[index, (0,n)]
    and MachineMacro = Formarray [label, (1,6)][DEF,
          VAL, UPDATE, BIN, DEC, BAR]
    and MST = Formarray[logical, (0,38)][−1,4, 'D',
                          'E', 'F', −1,
                           0,4, 'V', 'A', 'L', −2,
                           6,7, 'U', 'P', 'D', 'A', 'T', 'E', −3,
                          12,4, 'B', 'I', 'N', −4,
                          21,4, 'D', 'E', 'C', −5,
                          27,4, 'B', 'A', 'R', −6]
    for k = 0 to 38 do ST[k] := MST[k]

||  The name-value pairs for the six machine code macros
||are first assembled in the vector MST and then copied
||into the base of the stack.

§2  let routine Load be
        § test H = 0
              then do WriteSymbol[A]
              or do ST[S], S := A, S+1
        return §
  and routine NextCh be
        § test C = 0
              then do ReadSymbol[A]
              or do A, C := ST[C], C+1
        return §
  and routine Find [x] be
        §2.1      A, W := E, x
              §2.2 for r = 0 to ST[W]−1   do
                     if ST[W+r] ≠ ST[A+r+1] go to Next
                     W := A+1+ST[W]
                     return
           Next:A := ST[A] §2.2
              repeat until A < 0
              go to Monitor7   §2.1
  and routine JumpIfMarked[x] be
        § if x < 0 go to MachineMacro[−x]
        return §

||  This routine depends on the method of marking machine
||code macros.  The method adopted here (which is
||different from that described in the paper or used in the
```


Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022

Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022


```
||actual Titan program) is to make the value a negative
||index integer which is used to index the label vector
||MachineMacro, whose entries are the labels of the
||corresponding programs.

 and Number [x] = x − 16
 and Char[x] = x + 16

|| These are implementation-dependent functions. They
||convert the index equivalent of a decimal digit read in
||with ReadSymbol to the corresponding number (also of
||type index) and vice versa.

||Main cycle
Start: NextCh
       if A = '<' do § q := q+1; go to Q2 §
       go to A = '§' → Fn,
             A = ',' → NextItem,
             A = ';' → Apply,
             A = '~' → LoadArg,
             A = Marker → EndFn,
             A = '>' → Exit,
             Copy

Copy:  Load
Scan:  if q = 1 go to Start
Q2:    NextCh
       if A = '<' do § q := q + 1; go to Copy §
       if A ≠ '>' go to Copy
       q := q−1
       go to q = 1 → Start, Copy

||Warning Character Actions
Fn:       H,S,F,ST[S],ST[S+1],ST[S+2],ST[S+3] :=
              S+3, S+4, S+1, H, F, 0, 0
          go to Start
NextItem: if H = 0 go to Copy
          H,S,ST[H],ST[S] := S,S+1,S−H−ST[H],0
          go to Start
Apply:    ifP>F go to Monitor1
 if H=0 go to Copy
 F,P,H,S,ST[H],ST[S],ST[F−1],ST[F],ST[F+1] :=
   ST[F],F,ST[F−1],S+1,S−H, Marker,S−F+2,P,C
 unless H=0 do
   ST[H] := ST[H]+ST[P−1]
 Find[P+2]
 JumpIfMarked[ST[W]]
 C := W+1
 go to Start
LoadArg: if P=0 go to H=0 → Copy,Monitor2
         NextCh
         W := P+2
         if Number[A]<0 go to Monitor3
         for r=0 to Number[A]−1 do
                § W := W+ST[W]
                  if ST[W]=Marker go to Monitor4§
         for r = 1 to ST[W]−1 do
                § A := ST[W+r]
                  Load §
         go to Start

EndFn:   if F>P go to Monitor5
         ST[S],A := E,S
         while ST[A]⩾P−1+ST[P−1] do
                ST[A],A := ST[A]−ST[P−1],ST[A]
         W := ST[A]
         while W>P−1 do
                W := ST[W]
         ST[A] := W
         E := ST[S]
         unless H=0 do
              test H>P
                   then do H := H−ST[P−1]
                   or do ST[H] := ST[H]−ST[P−1]
         P,C,S,A,W := ST[P],ST[P+1],S−ST[P−1],
                               P−1,P−1+ST[P−1]
         until A=S do
                ST[A],A,W := ST[W],A+1,W+1
         go to Start
Exit:    unless C=H=0 go to Monitor8
         Finish

||Machine Code Macros
DEF: unless H=0 do ST[H] := ST[H]−ST[P−1]+6
       ST[P−1],ST[P+5], E := 6,E,P+5
       go to EndFn

|| This version of DEF is shorter than that given in
||Section 7 as it leaves EndFn to copy back the definition.
VAL:     Find[P+6]
         until ST[W+1] = Marker do
                § A,W := ST[W+1],W+1
                  Load §
         go to EndFn
UPDATE:  Find[P+9]
         A := P+9+ST[P+9]
         if ST[A]>ST[W] go to Monitor9
         for r=1 to ST[A] do
                ST[W+r] := ST[A+r]
         go to EndFn
BIN:     W,A := 0, ST[P+7]='+' → P+8,
                   ST[P+7]='−' → P+8,
                   P+7
         until ST[A] = Marker do
                §  let x = Number[ST[A]]
                   unless 0⩽x⩽9 go to Monitor10
                   W,A := 10W+x, A+1      §
         S,ST[S] := S+1, ST[P+7]='−' → −WW,
         go to EndFn
DEC:     W := ST[P+7]
         if W<0  do
                § W,A := −W, '−'
                  Load  §
         §2.1  let W1 = 1
               until 10W1>W do W1 := 10W1
               §2.2  A,W,W1 := Char[Quot[W,W1]],
                               Rem[W,W1],W1/10
                     Load  §2.2
               repeat until W1<1   §2.1
         go to EndFn
```

```
BAR:     W,A := ST[P+9],ST[P+11]
         A := ST[P+7]='+' → W+A,
              ST[P+7]='−' → W−A,
              ST[P+7]='×' → W×A,
              ST[P+7]='/' → Quot[W,A],
              Rem[W, A]
         Load
         go to EndFn

||Monitor for errors

§3 let routine Item[x] be
  §3.1   let a,h = A,H
         H := 0
         for k=1 to ST[x]=0 → S−x−1, ST[x]—1 do
            §     A := ST[x+k]
                  Load  §
         if ST[x]=0 do
                  Write['. . .*t(Incomplete)']
         A,H := a,h
         return    §3.1
||  This routine outputs the item on the stack starting at
||ST[x].  If the item is not complete, printing stops at
||ST[S−1] and is followed by '. . . (Incomplete)'.

||Monitor Entries and Effects

Monitor1:     ||Unmatched ; in definition string.  Treated
              ||   as ⟨;⟩
         Write['*nMONITOR: Unmatched semicolon
                 in definition of ']
         Item[P+2]
         Write['*nIf this had been quoted the result
                 would be *n']
         go to Copy

Monitor2:     ||Unquoted ∼ in argument list in input
              ||   stream.  Treated as ⟨∼⟩
         Write['*nMONITOR: Unquoted tilde in argu-
                 ment list of ']
         Item[F+2]
         Write['*nIf this had been quoted the result
                 would be *n']
         go to Copy

Monitor3:     ||Impossible character (negative) as argu-
              ||   ment number.  Terminate.
         Write['*nMONITOR: Impossible argument
                 number in definition of ']
         Item[P+2]
         go to Monitor11

Monitor4:     ||Not enough arguments supplied in call.
              ||   Terminate.

         Write['*nMONITOR: No argument ']
         H := 0
         Load
         Write['in call for ']
         Item[P+2]
         go to Monitor11

Monitor5:     ||Terminator in impossible place; if C=0,
              ||this is the input stream.  Probably
              ||machine error: Terminate.  If C≠0, this
              ||is an argument list.  Probably due to a
              ||missing semicolon:  Final semicolon in-
              ||serted.
         Write['*nMONITOR: Terminator in ']
         if C=0 do
         §  Write['input stream.  Probably
               machine error.']
               go to Monitor11   §
         Write['argument list for ']
         Item[F+2]
         Write['*nProbably due to a semicolon missing
                 from the definition of ']
         Item[P+2]
         Write['*nIf a final semicolon is added the
                 result is *n']
         C := C−1
         go to Apply
Monitor7:     ||Undefined macro name: Terminate.
         Write['*nMONITOR: Undefined name ']
         Item[W]
         go to Monitor11
Monitor8:     ||Wrong exit (not C=H=0).  Machine
              ||   error: Terminate.
         Write['*nMONITOR: Unmatched ⟩.  Prob-
                 ably machine error.']
         go to Monitor11
Monitor9:     ||Update string too long: Terminate.
         Write['*nMONITOR: Update argument too
                 long for ']
         Item[P+9]
         go to Monitor11
Monitor10:    ||Non-digit in argument for BIN.  Ter-
              ||   minate.
         Write['*nMONITOR: Non-digit in number']
         go to Monitor11
Monitor11:    ||General monitor after irremediable
              ||   errors.
         W := 20
         Write['*nCurrent macros are']
         until P=F=0 do
           §4 let W1 be index
              test P>F
                 then do § W1,P := P+2,ST[P]
                           Write['*nAlready entered ']  §
                 or do § W1,F := F+2,ST[F]
                           Write['*nNot yet entered ']  §
              for r = 1 to W do
                 §4.1 Item[W1]
                      if ST[W1]=0 do break
                      W1 := W1+ST[W1]
                      if ST[W1]=Marker do break
                      unless W=1 do
                        Write['*nArg ',
                        r, '*t']   §4.1
              W := 1   §4
```


Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022

Downloaded from https://academic.oup.com/comjnl/article/8/3/225/336044 by guest on 25 August 2022


```
    Write['*nEnd of monitor printing']
    A := 'Q'
    Load
    go to P>F → EndFn,Start        §1
||End of CPL program for GPM.
```

Acknowledgements

The symbol string manipulator or macrogenerator described in this paper has been in use in the Mathematical Laboratory since early in 1964. During this period it has been developed somewhat, and the present version, which is described here, is both more general and simpler than the original one. Many people have contributed to this improvement both by using the program and discovering its difficulties, and by direct suggestion and discussion. I should particularly like to thank Mr. P. Frost and Mr. J. S. Rayner for the work they did in getting the two versions of the GPM actually running on Titan.

References

DIJKSTRA, E. W. (1962). "An Attempt to Unify the Constituent Concepts of Serial Program Execution," in *Symbolic Languages in Data Processing*, Gordon and Breach: London.

BARRON, D. W., *et al.* (1963). "The main features of CPL," *The Computer Journal*, Vol. 6, p. 134.

CPL Elementary Programming Manual, Edition II (Cambridge), Cambridge University Mathematical Laboratory, May 1965.

Book Review

Numerical Methods and FORTRAN Programming, by D. D. McCracken and W. S. Dorn, 1964; 457 pages. (London and New York: *John Wiley and Sons Ltd.*, 57s.)

This textbook combines instruction in computer programming and numerical techniques suitable for undergraduates in science and engineering, and, to a lesser extent, for those in mathematics. Teachers will find it a useful guide to the selection of material for a practical course in computational methods.

The authors adopt an interesting form of presentation in which numerical methods and programming instruction in FORTRAN II are developed in parallel, so that, at all stages, suitable applications are available to illustrate in detail the techniques described for both topics; the arrangement is therefore more attractive and powerful than a straight combination of a programming manual and a tract on numerical methods. There are, however, disadvantages; for example, roughly two hundred pages pass before the DO statement is introduced, but this is offset by the excellent balance and relevance of topics discussed earlier.

Three chapters on FORTRAN II are introduced at different stages throughout the book, which derive from McCracken's well known *Guide to FORTRAN Programming* (*Wiley*, 1961); the material is rearranged and slightly extended, but otherwise is substantially the same. A useful chapter on Program Development is added, in which three of the case studies discussed in the earlier work are substantially developed to illustrate various practices involved in preparing programs for computer operation.

The numerical methods discussed fall into the following chapters: evaluation of functions (power series, Chebyshev, continued fractions); roots of equations (Newton-Raphson, complex roots); evaluation of integrals (trapezoidal, Simpson, Gauss); simultaneous linear equations (Gauss elimination; Gauss-Seidel iteration); ordinary differential equations (Taylor series, Runge-Kutta, predictor-corrector methods); partial differential equations of the linear, second-order type (difference methods, Liebmann). This is a standard selection for a limited course on numerical methods, but the interplay between requirements of mathematical formulation, style of programming, and the study of error analysis provides a distinctive approach to the subject. The authors regard the application of numerical methods as a form of preparation for further studies in numerical analysis, and emphasise that a student completing the text will not be "a finished numerical analyst." Nevertheless from Chapter 2 (on "Errors") onwards, disciplines of error analysis are practised with a meticulous attention to detail that should impress upon the reader the need for careful analysis in all computer applications. The generation and propagation of various forms of numerical errors is studied systematically, and analytical difficulties arising from ill-conditioning, or convergence problems, are illustrated in appropriate contexts. Careful comparisons are also made of the merits of alternative procedures, and the case studies in each chapter, though generally elementary, are subjected to similar standards of criticism.

This book appears at a time when new efforts are being made to construct realistic courses for computer education. Its objective and successful elementary approach could persuade those who insist that access to a computer should follow a thorough grounding in numerical analysis, of the value of practical experience (on a small machine!) at all stages of computer education, provided sound counsel is at hand. Above all it may direct early enthusiasms to the currently less fashionable, but in the long run more beneficial, fields of design and application in science and engineering; present trends towards more esoteric applications may retard developments in these fields, in which computers are the most natural and valuable tools.

An illogical feature of the book is the price. McCracken's *Guide to FORTRAN Programming* costs 23/– (paper back only), and is accommodated in roughly 120 pages of the present volume. A paper back version of the new book is due shortly at 35/–; the additional 300 pages, with all the instruction in numerical methods and programming arts can be bought for 12/– more, and at that price are remarkably good value.

H. H. GREENWOOD