## Macro memories, 1964–2013

David Walden

### Contents



### Introduction

In the summer of 2013, I was looking at a 1973 listing of the ARPANET IMP (Interface Message Processor) program[1] which makes extensive use of macros. This caused me to muse about the various macro processors I have used over the past 50 years which, in turn, led to this note.

This note is not a thorough study, extensive tutorial, or comprehensive bibliography about macro processors. Such descriptions have already been provided by, for instance, Peter Brown, Martin Campbell-Kelly, John Metzner, and Peter Wegner in their longer presentations of the topic.[2,3,4,5] Instead, the macro technology thread I follow herein is guided by the order in which I used or studied the various macro processors. I hope this is usefully representative of the scope of macro processor technology.

I have three reasons for writing this note. (1) I haven't seen much new written about macro processors in recent years (other than what is on the web); thus, it is perhaps time for a new paper on this sometimes under-appreciated topic. (2) Computer professionals and computing historians who have come to their fields only in the last decade or two may not know much about macros, and this is a chance to share my fondness for and perspective on macros. The citations in the endnotes also may be a useful starting point for further study of macros, and maybe these notes will rekindle memories for other long-time computing people like me about some of their own experiences. (3) For (LA)TEX users who may not be computer programmers or familiar with other macro processor systems and who accomplish impressive things using TEX macros, this note sketches the long technical history of which TEX macros are a part.

I assume most readers know what macros are, but just in case: Typically one gives a name to a string of text, e.g., "`\define\Macroname{Textstring}`"; then each time "`\Macroname`" is found ("called") in the input text stream, it is replaced by "`Textstring`".

The macro definition can involve the additional substitution of text specified when the macro definition is called. For example,

```
\define\Name#1{His name is #1}
```

defines a macro named "`Name`" where "`#1`" indicates a parameter to be substituted for when the macro is called. The macro might be called with "`John`" as the substitution text, as in the following

```
\Name{John}
```

which would result in

```
His name is John
```

In addition to the "`#1`" indicating where a substitution is to take place in the macro definition when the macro is called, in this example the "`#1`" immediately after the macro name indicates there is one such substitutable parameter. If there were more than one such parameter, a list such as "`#1#2#3`" would appear after the macro name in the definition, specifying three such parameters.

Donald Knuth in the index to Volume 1 of *The Art of Computer Programming* gives this succinct definition relating to macros: "Macro instruction: Specification of a pattern of instructions and/or pseudo-operators that may be used repeatedly within a program."

### 1 McIlroy's 1960 ACM paper

I'm pretty sure that while I was still in college at San Francisco State (1962–64) and using an IBM 1620 computer, I had no concept of macros. The IBM 7094 at MIT Lincoln Laboratory, my first employer after college (starting in June 1964), may have had a macro assembly capability, but I don't think I ever used it.

Probably my first contact with the concept of macros was when an older colleague at Lincoln Lab gave me his back issues of the *Communications of the ACM* (and I joined the ACM myself to get future issues of the CACM). In one of these back issues, I read the article by Doug McIlroy on macros for compiler languages.[6]

McIlroy has the following example of defining a macro (although I am using an equal sign where he used an identity symbol):

```
ADD, A, B, C = FETCH, A
               ADD, B
               STORE, C
```

where `ADD`[7] is the macro name, and `A`, `B`, and `C` are the names of macro arguments to be filled in at the

time of the macro call, and the three lines of code are what is substituted. Thus, McIlroy shows this macro being called with the sequence

```
ADD, X, Y, Z
```

resulting in the following:

```
FETCH, X
ADD, Y
STORE, Z
```

This style of macro definition uses symbolic names for the substitutable parameters, which can be useful in remembering what one is doing with long macro definitions. However, it is also a bit more complicated to implement such symbolic macro parameter names compared with using special codes such as "`#1`".

McIlroy's 1960 paper goes on to show examples of macros in an ALGOL-like language and to explain the benefits of various features of macro processors. For instance,

```
macro exchange(x,y;z) :=
    begin
        begin integer x,y,z;
              z:=y;
              y:=x;
              x:=z;
        end exchange x and y
    end exchange
```

defines a macro which, if called with

```
exchange(r1,ss3)
```

results in

```
begin integer r1,ss3,.gen001
      .gen001:=r1;
      ss3:=r1;
      r1:=.gen001;
end exchange r1 and ss3
```

Note that a special temporary register, `.gen001`, was created to replace `z` which was defined following the semicolon in the parameter list.

McIlroy's paper also has a summary list of "salient features" [the comments below in square brackets are my notes on McIlroy's list]:

1. definitions may contain macro calls
2. parenthetical notation for compounding calls [e.g., so arguments to macro calls can include multiple items separated by commas]
3. conditional assembly
4. created symbols [e.g., so labels or local variable names in the body of a macro definition are unique for each call of the macro]
5. definitions may contain definition schemata
6. repetition over a list [see the example in the discussion of Midas in section 4]

Apparently Bell Labs was a particular hotbed of macro activity in those early days. In a memorial note for Douglas Eastwood,[8] Doug McIlroy recounts:

> On joining the Bell Labs math department, I was given an office next to Doug Eastwood's. Soon after, George Mealy ... suggested to a small group of us that a macro-instruction facility be added to our assembler ... This idea caught the fancy of us two Dougs, and set the course of our research for some time to come. We split the job in half: Eastwood took care of defining macros; McIlroy handled the expansion of macro calls.
>
> The macro system we built enabled truly astonishing applications. Macros took hold in the Labs' most important project, electronic switching systems, in an elaborated form that served as their primary programming language for a couple of decades.
>
> Once macros had been incorporated, the assembler was processing code written wholesale by machine (i.e., by the assembler itself) rather than retail by people. This stressed the assembler in ways that had never been seen before. The size of its vocabulary jumped from about 100 different instructions to that plus an unlimited number of newly defined ones. The real size of programs jumped because one human-written line of code was often shorthand for many, many machine-written lines. And the number of symbolic names in a program jumped, because macros could invent new names like crazy.

By the way, Rosen's book[7] also had a paper (pp. 535–559) by George Mealy that touched on macros: "A Generalized Assembly System (Excerpts)".

## 2 Some prior history of macros

McIlroy's paper also hints at some of the history of macro processors including half a dozen references[9] to prior macro processors; they were becoming fairly widespread by the early 1960s. Lots of people were thinking about macros and macro processing by 1960. In section 6 of his paper, McIlroy says,

> ... Conditional macros were devised independently by several persons beside the author in the past year. In particular, A. Perlis pointed out that algorithms for algebraic translation could be expressed in terms of conditional macros. Some uses of nested definitions were discovered by the author; their first implementation was by J. Benett also of Bell Telephone Laboratories. Repetition over a list is

> due to V. Vyssotsky. Perlis also noted that macro compiling may be done by routines to a large degree independent of ground language. One existing macro compiler, MICA (Haigh), though working in only one ground language is physically separated from its ground-language compiler. An analyzer of variable-style source languages exists in the SHADOW routine of M. Barnett, but lacks an associated mechanism for incorporating extensions. Created symbols and parenthetical notation are obvious loans from the well-known art of algebraic translation.

Donald Knuth and Luis Trabb Pardo also touch on the history of macros in their paper "The Early Development of Programming Languages".[10] Early in the paper,[11] they note that Turing's 1936 paper on a universal computing machine used a notation for programs which amounted to being "macroexpansions or open subroutines". Later in the paper,[12] they say that Grace Hopper in 1951 came up with the "idea that pseudocodes need not be interpreted; pseudocodes could also be expanded out into direct machine language instructions." Later on the page they note, "M.V. Wilkes came up with a very similar idea and called it the method of 'synthetic orders'; we now call this macroexpansion."[13]

## 3 Strachey's General Purpose Macrogenerator

At the time I joined the ACM to receive the CACM, I also subscribed to *The Computer Journal*. In this I studied Christopher Strachey's GPM (General Purpose Macrogenerator).[14] The paper presents Strachey's macro processor and its possible uses. Then the paper explains how it is implemented. Finally, it has the code for the CPL language (sort of ALGOL-like) which can be transliterated to implement GPM in any other computing language.

GPM was a change in the way macro definitions and calls were formatted from the series of macro processors originally developed at Bell Labs in what I will call the McIlroy style. These early assemblers and the macro processors tended to be shown in columns with keywords (`DEFINE`, a defined macro name, `IRP`, etc.) being recognized by the processor as a special symbol and the other parts of the definition or call being detected by their separation with spaces or commas, or perhaps bracketing parentheses. In fact, many of the early macro processors were embedded parts of an assembler or language compiler. GPM indicated its macro definitions and calls and their arguments with unusual characters, and it was independent of any particular language — a possible preprocessor for any other language or as a stand-alone string processor.

Here is a simple definition in GPM:

```
§DEF,REFORMATNAME,<LAST=~2, FIRST=~1>;
```

It could be called like this:

```
§REFORMATNAME,David,Walden;
```

which would produce the output:

```
LAST=Walden, FIRST=David
```

Note that the GPM approach to macro definitions does not specify how many substitutable parameters there are. Note also that bracketing macro definitions and macro calls with special symbols ("`§`", "`;`") makes it a bit simpler for definitions and calls to occur anywhere in the input stream.

Strachey's paper is a wonderful and now classic article. By introducing the macro processor and its uses, describing its implementation, and then providing the code for its implementation, Strachey's paper is a superb model for presenting a programming language. This was perhaps possible because Strachey, purportedly a genius programmer, had managed a very general and beautiful implementation. The CPL code was only two double-column journal pages long; and, according to the history of the `m4` macro processor,[15] fit into 250 words of machine memory.

The just-mentioned `m4` history also touches on the influence of GPM on later macro processors. Also, GPM was used by later authors as an illustrative example of a macro processor.[2,16]

Some readers by this time may be asking, "But how is a macro processor implemented?" One can sketch this intuitively. Text in the input stream to the macro processor that is not a macro definition or macro call is just passed on to the output stream. Macro definitions in the input stream are saved in a software data structure with their names and associated definitions. When a macro call is spotted in the input stream, the definition is pulled out of storage to replace the macro call in the output stream with the call parameters being substituted at the proper places in the definition. If all this is done using a first-in-last-out stack in the proper way, definitions within definitions, recursive calls, and so forth are possible. For a detailed description of a macro processor implementation, access Strachey's GPM paper in its journal archive or find a used or library copy of Peter Wegner's book[4] (pp. 134–144).

## 4 Midas macro processor

The next macro processor I came across (and the first I actually used) was the macro processor that was part of the Midas assembler for the PDP-1. PDP-1

Midas had its origins in MIT's TX-0 computer, all the way back to MACRO on the TX-0.

MACRO was an assembler with a macro processor capability written by Jack Dennis for the TX-0. I don't know of a manual for TX-0 macros; however, MACRO was later released by DEC with its PDP-1 computer, and Jack Dennis states[17] that he wrote the manual for that.[18]

When I asked Jack Dennis about predecessor technology to MACRO, he mentioned McIlroy's paper (which was published after MACRO was available on the TX-0 in 1959, so perhaps Jack saw a draft or preprint). Of his MACRO, Jack said,[19] "Doug's macro processor was of the string substitution sort ... Mine was different: it permitted a user to give a name to a sequence of assembly instructions, with integer parameters that would be added to instructions to create modified addresses. (Thus the essential mechanism was one's complement binary arithmetic instead of string concatenation.)"

Next on the TX-0 came Midas, which was derived by Robert Saunders from TX-0 MACRO. Then, TX-0 Midas was moved to the PDP-1.[20] The TX-0 Midas memo[21] is dated November 1962 which suggests that the PDP-1 Midas was up and running sometime in 1963, as the Midas manual for the PDP-1 says it was ported from the TX-0 where it had been running for about a year.

In any case, the PDP-1 editing, assembling, and debugging set of programs was probably the best set of interactive program development and debugging tools that were available for a mini-computer in the mid-1960s. Therefore, four of us using a Univac 1219 computer at Lincoln Lab decided to reimplement these PDP-1 tools for our 1219.[22]

Midas for the PDP-1 and our version for the Univac 1219 had macro processor definition and call formats that were similar to those in the tutorial part of McIlroy's paper, e.g., "`MACRO NAME ARG1, ARG2 (string)`" to define a macro and "`NAME X,Y`" to call it with `X` and `Y` to be substituted for `ARG1` and `ARG2` in the macro definition. For example,

```
MACRO MOVE X,Y
      (ENTAL X
      STRAL Y)
```

The above when called with

```
MOVE K,L
```

resulted in

```
ENTAL K
STRAL L
```

Midas also had what I now think of as map commands, i.e., apply some function over a list of arguments — the Midas commands `IRP` (indefinite repeat over a list of arguments) and `IRPC` (indefinite repeat over a string of characters). In our version of Midas, the `IRP` command might have been used as follows:

```
IRP A, (W1, W2, W3)
(ADD   A
)
```

expanding to

```
ADD  W1
ADD  W2
ADD  W3
```

and in another example

```
IRP  X,Y,(A,Q,B,R,C,T)
(CLA X
 STO Y
)
```

expanding to

```
CLA  A
STO  Q
CLA  B
STO  R
CLA  C
STO  T
```

The same thing could have been accomplished using the command to repeat for each character in a string of characters:

```
IRPC  X,Y,(AQBRCT)
(CLA X
 STO Y
)
```

I'm pretty sure that the following also worked in our version of Midas:[23]

```
MACRO ADDTHEM X,Y,Z
      (ENTAL X
       IRP W,(Z)
      (ADD W
      )
       STRAL Y
      )
```

when called with `ADDTHEM A,B,(C,D,E)` resulted in

```
ENTAL A
ADD C
ADD D
ADD E
STRAL B
```

All in all, this macro effort was a significant piece of computing and programming education for me.[24]

## 5 More study and use of macro processors (and language extension capabilities)

In September 1967, I moved to Bolt Beranek and Newman Inc. in Cambridge, MA, where I had access to the company's PDP-1d time sharing system. I

immediately began extensive use of the macro facility built into the editing program TECO.[25]

TECO[22] was an early, very powerful, text editor with a macro capability using the same type of keystrokes one used for editing. One could type a list of keystrokes to do some complex editing function but delay evaluation and instead save the sequence of keystrokes (i.e., defining a macro) and then later give a few keystrokes to execute the saved string of keystrokes (i.e., calling or executing the macro). TECO macros typically looked very cryptic. People also played games with what complicated things they could do with TECO macros, e.g., calculating digits of pi, implementing Lisp, etc.[26]

Also early on at BBN, as a weekend hack, I transcribed the Algol-like listing of Strachey's GPM system from his 1965 paper into PDP-1 Midas assembly language and made it run. This was easy to do given Strachey's complete description of the system.

I also investigated the TRAC language.[27,28] TRAC was presented by Calvin Mooers as a text processing language; but to my mind, TRAC was not so different from a macro processor in the way it defines and manipulates strings.[29]

The basic TRAC operation is a call to a built-in function introduced by a pound sign, e.g.,

```
#(function-name,arg1,arg2,...)
```

Two of the built-in functions are define string (`ds`) and call (`cl`), as follows:

```
#(ds,greeting,Hello World)
```

and

```
#(cl,greeting)
```

resulting in replacement of the call by the string "`Hello World`".

Another built-in TRAC function, `ss`, specifies the strings for which a substitution is to be made at call time. Thus,

```
#(ds,greeting,Hello, name)
#(ss,greeting,name)
```

creates a macro with one call-time argument, such that

```
#(cl,greeting,Hello Dave)
```

results in the text string "Hello, Dave".

There are other built-in functions for string comparison, and so on.

By this time, I was pretty fascinated by programming languages and macros, in particular the idea of extensions to programming languages.

Macros have often been used as a form of language extension. For instance, complex add might be defined, using McIlroy's notation, as

```
COMPLEXADD, A, B, C, D, E, F = FETCH, A
                               ADD, C
                               STORE, E
                               FETCH B
                               ADD D
                               STORE F
```

with the obvious substitution when called with

```
COMPLEXADD, U, V, W, X, Y, Z
```

From macros as a way of extending languages it was a short step to the idea of extensible high-level languages. From 1966–1968 I took computer science courses at MIT as a part-time graduate student and eventually did Master's thesis work (never completed) on a capability for extending a high level language. Unfortunately, I have lost the complete draft of the thesis report (and I never finished the accompanying program). However, I am sure that my thesis literature research and thinking influenced the next project I had at BBN.

In 1967 to 1968, with the assignment to think about a programming language for what became BBN's PROPHET system (a tool to help medicinal chemists and research pharmacologists), I looked deeply into extensible languages. I studied and reported on McIlroy's ideas[6] and the ideas and implementations of several other researchers.[30] Eventually, as a proof of concept, I translated James Bell's Proteus from the Fortran implementation in his thesis into PDP-10 assembly language. I turned this effort over to Fred Webb, who eventually replaced what I had done with a fresh implementation of an extensible language named PARSEC.[31] PARSEC was used in various versions (and later as the basis for RPL for the RS/1 system) by a multitude of people for many years.

This note on macros is not the place to go more deeply into extensible languages. The references in note 30 are a decent introduction to the state of the art circa 1968. For the state of the art a decade later, John Metzner's graded bibliography on macro systems and extensible languages is relevant.[5]

## 6 Midas, macros, and the ARPANET IMP program

At BBN I was part of the small team that in 1969 developed the ARPANET packet switch.[1,32] We developed the packet switch software for a modified Honeywell 516 computer using the PDP-1d Midas assembler, using lots of macros, etc., to adapt Midas to know about the 516's instruction set and paged memory environment. Bernie Cosell primarily constructed this hairy set of conversation macros. Our three person software team (Cosell, Will Crowther,

and me) also used Midas macros to facilitate development of the packet switching algorithms to run on the 516. All this may be seen in a listing of the IMP program available on the web.[33] We also used Midas macros to generate a concordance for the IMP program[34] as well as to reduce the probability of writing time-sharing bugs.[35] A contemporary version of this Midas macro assembler written in Perl by James Markevitch is also available for study.[36]

## 7 Ratfor and Infomail

The next project on which I saw something like a macro processor was an effort in the early 1980s to develop a commercial email system that would run on a variety of vendor platforms, e.g., DEC VAX, IBM CICS, IBM 360, and Unix on the BBN C/70. For portability we decided to implement our email system (called InfoMail) in Fortran, for which compilers already existed for the target platforms. However, to avoid having to actually write Fortran code, we developed the system using Ratfor (Rational Fortran).[37] Ratfor was not actually a macro processor but rather a programming language that acted as a preprocessor to emit a Fortran program that could be compiled by a standard Fortran compiler. Nonetheless, that seems to me to be quite a lot like what a macro processor does. Also, Chapter 8 of the *Software Tools* book (see previous note) describes the implementation in Ratfor of a macro processor (based on the macro processor for the programming language C).

Also during my BBN years between 1967 and 1995, I briefly used the C language, which includes a macro processor,[38] which optionally can be used independently of the rest of C. But from 1982 on I did no real computer programming and thus didn't track what was happening in the world of macro processors.

## 8 TEX, macros for typesetting

People who use (LA)TEX macros may already know much of what is in this section. However, some of it may be new to some readers.

After retirement from BBN in 1995, I had started using (LA)TEX in place of a word processor such as Microsoft Word, particularly for documents that were more than a one-page, one-time letter. This brought me in contact with the very sophisticated and complex macro processor embedded in the TEX typesetting system. I have now been using this system and its macro processor for typesetting for nearly 20 years, the longest in my life I have used any single macro processor.

Here is an example of a TEX macro.

```
\def\Greeting#1{Hi #1! I hope you're well.}
```

If this is called with `\Greeting{Dave}`, it results in

```
Hi Dave! I hope you're well.
```

The text "`#1`" tells the macro definition processor (a) that there is one argument, and (b) where the call-time argument is to be substituted in the body of the macro. If the macro definition allowed three arguments, for instance, the sequence "`#1#2#3`" would appear after the macro name and before the open curly bracket of the definition.

The TEX macro processor is enormously powerful and flexible, in its unique way, and a comprehensively documented piece of software.[39,40,41,42] Massive programs have been (and continue to be) written in its macro language. For example, LATEX is implemented entirely with TEX macros, as are other variants or supersets of TEX (called "formats" in TEX jargon) as well as thousands of LATEX "packages" which extend or modify the capabilities of LATEX. One modest-size example of the use of TEX macros to extend LATEX can be found on pages 6–10 of "The `bibtext` Style Option".[43]

Personally, I tend to do my LATEX extensions using packages that other people have already written, although sometimes I make minor modifications to existing packages. More commonly I use TEX macros (or a LATEX variation) to replicate small snippets of LATEX code which are used repeatedly — for consistency, and also to easily allow later changes of such snippets as I figure out what I actually want them to do. In 2004 I published an example of one such use of a TEX macro;[44] I encourage readers to take a look at it as it describes (far from completely) various ways TEX macros are defined and can be called.

Historically, there has been an interesting set of pressures around TEX's macro capability. Originally Donald Knuth included only enough macro capability to implement his typesetting interface. However, he was persuaded ("kicking and screaming"[45]) by early users to expand that macro capability. That expanded capability allowed users to construct pretty much any logic they wanted on top of TEX (although often such add-on logic was awkward to code using macro-type string manipulations). On the one hand, TEX and its macro-implemented derivatives have always been very popular and there have been nonstop macro-based additions for over 30 years. On the other hand, users despair at how annoying coding using macros is, moan about "why Knuth couldn't have included a real programming language within TEX", and otherwise cast aspersions on TEX's macro capability.

Over the years various attempts have been made to link TEX to a "real" programming language, typi-

cally invoking the programming language from TEX or the reverse; however, none of these efforts have come into widespread use. Over the past few years, however, a small group of TEX hackers have accomplished an apparently successful merger of the Lua programming language and TEX, called LuaTEX.[46] LuaTEX maintains TEX's macro processor (there is really no way, and no reason, to get rid of it while keeping TEX). Thus, the full power of the TEX macro processor is available for the many situations in which it is the best tool, and the Lua language is available for things which can be done much more easily in a procedural language.

The history of the TEX macro processor partially explains the above. Knuth has made the point that he was designing a typesetting system that he didn't want to make too fancy, i.e., by including a high level language. He has also noted that when he was designing TEX he created some primitive typesetting operations and then created a set of macros for the more complete typesetting environment he wanted. He expanded the original macro capability when fellow Stanford professor Terry Winograd wanted to do fancier things with macros. Knuth's idea was that TEX and its macro capability provided a facility with which different people could develop their own typesetting user interfaces, and this has happened to some extent, e.g., LATEX, ConTEXt, etc.

It is perhaps worth discussing a few of the things that make the TEX macro capability different from, for example, the capability of GPM.

GPM has a simple and unchanging definition and calling syntax, as was described in Section 3. Macro definitions can include other macro definitions, and macros can have recursive calls (and without going back to study the GPM paper carefully, I assume that the scope of macro definitions happens in the natural and obvious way). The definition associated with a macro name can be "looked at" without evaluating the definition; the definition associated with a macro name can be assigned to a different macro name; and there is a capability for converting numbers between binary and decimal formats and for doing binary arithmetic. No limit is specified for the number of arguments a macro definition and call can have. Finally, GPM is a stand-alone program; it processes its input, and it is up to the user what happens next with its output.

The TEX macro capability can do all those things; but it cannot be used independent of TEX, and in its straightforward form its macro definitions and calls are limited to nine arguments. TEX also has a way of defining a macro call to have a much more free-form format, and with some programming (or use of an appropriate TEX macro package) macro call arguments can be specified with attribute-name/value pairs. There are explicit commands in TEX creating local or global definitions, as well as various other definition variations, such as delayed definition and delayed execution of macro calls. TEX has a rich rather than minimal set of conditional and arithmetic capabilities (some related only to position in typesetting a page). There are also ways to pass information between macros and, more generally, to hold things to be used later during long, complicated sequences of evaluation and computation. These capabilities allow big programs to be written in the macro language, and thus TEX also has a capability to trace the flow of macro definition and execution.

TEX has another unusual capability that is sometimes used with macros, although it is a capability more closely related to lexical analysis than to macro definitions and calls; this is the TEX "category code" feature. TEX turns its input sequence of characters into a list of tokens. A token is either a single character with a category code (catcode) or a control sequence. For instance (using an example from Knuth's *The TEXbook*), the input "`{\hskip␣36␣pt}`" is tokenized into

```
{ hskip 3 6 ␣ p t }
```

where the opening brace is given the catcode of 1 (begin group), `hskip` gets no catcode because it is a control sequence, `3` and `6` get catcodes of 12 (other), the space after `36` gets catcode 10 (space), `p` and `t` get catcodes of 11 (letter), and the closing brace gets catcode 2 (end group). (There are 16 such category codes in all.)

The next step in the TEX engine decides what to do with these different tokens, e.g., put the numbers together into a numeric value and the letters together into a unit of measurement, execute the primitive `hskip` command, and so on. The backslash has a catcode of zero indicating an escape character, thus telling TEX that the following letters (five in this case) form a control sequence; the space after the command name delimits the end of the name and doesn't otherwise count as a space.

TEX also has the capability of changing which character(s) have which catcode(s). For instance, the dollar sign (by default having catcode 3, indicating a shift to math mode) could be given the escape character catcode, and the backslash could be given the math shift catcode. This capability is routinely used in TEX program libraries to define macro names internal to a program in the library which users of the library cannot use when (normally) defining their own macro names. Basically, the entire input

language of TEX can be changed.

The typeface design system, METAFONT, that Knuth developed in parallel with TEX has a more powerful macro capability (my memory is that he says somewhere that this was because he assumed more sophisticated people would be using METAFONT than TEX).

When Knuth rewrote TEX and METAFONT using literate programming techniques and targeting the Pascal programming language,[47] he included a macro capability in his literate programming WEB system for two reasons: simple numeric and string macro definitions were used to simplify coding given the limitations of Pascal; another kind of macro supported literate programming by allowing source material to be presented in an order suitable for human readers, which was then reordered by the system into the order needed for compiler processing.[48] (When literate programming systems targeting C were later developed, WEB's numeric and string macros were no longer strictly needed since C has its own macro processor, as noted above. The macros supporting literate programming continued to be an essential part of the system.)

I got to wondering when and where Donald Knuth got his own introduction to macro processors. He said:[49]

> I worked with punched cards until the 70s. My first "macro processor" was a plugboard for the keypunch, setting up things like tab stops! The assembler that I wrote as an undergrad, SuperSoap for the IBM 650, had a primitive way for users to define their own "pseudo-operations"; but it was extremely limited. For example, the parameters basically had to be in fixed-format positions.
>
> I learned about more extensible user-definability with the first so-called "compiler-compilers", notably D. Val Schorre's "META II" (1964)[50] and E. T. Irons's syntax-directed compiler written at Yale about the same time.[51] Later I knew about the sort-of macros in other compilers, e.g., PL/I.
>
> But the first really decent work on what we now call macro expansion was done I think by Peter Brown ... it was his book *Macro Processors* (Wiley, 1974)[2] that was my main source for macros in TEX, as far as I can remember now.

## 9 M4

In about 2001, I was looking for a macro processor to help me format an HTML list of the articles in the *Journal of the Center for Quality of Management* that were relevant to the chapters of a book I co-authored.[52] I found the m4 macro processor.[53]

The following first entry in an HTML table:[54]

| | | | | |
|---|---|---|---|---|
| 28.2¤ | click·here¤ | Vol.·1·No.·1,·1992¤ | Tom·Lee·and·David·Walden·¤ | What·is·the·Center·for·Quality·Management?¤ |

was created with the following m4 macro definition that outputs HTML markup:

```
define(‘_te’,‘
<TR VALIGN="top">
<TD ALIGN="center"><FONT FACE="Arial">$1</FONT></TD>
<TD ALIGN="left"><FONT FACE="Arial">$2</FONT></TD>
<TD ALIGN="left"><FONT FACE="Arial">$3</FONT></TD>
<TD ALIGN="left"><FONT FACE="Arial">$4</FONT></TD>
<TD ALIGN="left"><FONT FACE="Arial">$5</FONT></TD>
</TR>
’)
```

which was called as follows (except all on one line):

```
_te(28.2,_url(00100),_i(1,1,1992),
    Tom Lee and David Walden,What...)
```

where "`_url`" and "`_i`" are other m4 macro calls. I gave the m4 macros names beginning with underscores so the names would be recognizable in the m4 source file which didn't otherwise use underscores.

Use of m4 brought me full circle, in a sense, as the m4 macro processor is somewhat derived from GPM.[55] It was also used for and with Ratfor.

One incarnation of m4 is a freely available program under the GNU GPL, and thus that implementation is also available for study and other use.[56]

My writing and publishing work using a LATEX engine involved me in publishing work flows, particularly the need to have one source file be processable into multiple publication formats. For example, in the case of the TEX Users Group Interview Corner,[57] we use m4 macros to define a new markup language which can, with different sets of definitions, be targeted to output markup language code for either LATEX for paper printing or HTML for web posting. In our interview sources files, we might write the macro call "`_i(Book Name)`". With our HTML set of m4 definitions, that would convert to "`<i>Book Name</i>`". With our LATEX set of m4 definitions, that would convert to "`\textit{BookName}`". We have hundreds of m4 definitions in each of the LATEX and HTML m4 definition files that implement this ad hoc markup language that targets the two methods of output.

A language independent macro process such as GPM or m4 can be used at any point in a workflow:

- as a preprocessor for a programming language (as in many of the examples in this note)
- as a post processor (Martin Richards has noted that a GPM-like macro processor was used to

convert the O-code output of his BCPL compiler into machine language)[58]

- as an intermediate step in the work flow (for instance, a Perl program processes an input driver file to generate `m4` macro calls, the macro calls generate LaTeX markup, and the LaTeX engine generates PDF files from the LaTeX markup).

## 10 Reflections

**General reflections.** A computing historian who is a potential reader of this note might ask, "Why should I care about macro processors?" There are a couple of answers to this. First, macro processors played an important role in the history of programming languages. They were used with early assemblers, before higher level languages became widespread. In fact, they allowed assembly language users to create higher level language constructs by grouping together useful sequences of assembly language instructions under a single name, for example, `complex-add`. As higher level languages came into use (e.g., Fortran, ALGOL), the usefulness of macro processors was carried over into higher level languages (e.g., to create yet higher level linguistic constructs). Soon stand-alone macro processors were created that could act as a preprocessor for any programming language rather than being embedded within a particular programming language. Second, both embedded and stand-alone macro processors continue in widespread use today. We also find macro processors embedded in the user interface of many computer tools other than programming languages, for instance, as part of text editors, operating system shells, makefiles, and other applications and software packages (one example is Actions in Photoshop — repeatable sequences of Photoshop commands).

The reader might next ask, "If macro processors have been important in the history of programming, why isn't more written about them?" I think the answer to that question may have to do with the relative simplicity of many macro processors, which substitute one character string for another. That is not an area that requires much research, and generally useful implementation methods have been well known since the 1960s or earlier; also, it is not too hard to do an ad hoc implementation of a macro processor as part of some piece of software one is writing. Being mostly implemented as string manipulators, macro processors don't have to have all of the debatable research topics that a full programming language has (side effects, call by name, scope, etc. — although the sophisticated macro processors such as `m4` and TeX have parallels for these sorts of programming-language-philosophy topics). Also, as macro processors tend to operate at compile time, sometimes in ways that increase runtime efficiency, there probably hasn't been as much interest in optimizing macro processor performance as there has been with programming languages.

From the point of view of the professional programmer (versus historian), macro processors are just another development tool, and they are available as preprocessors or embedded in whatever programming language or other development tools the programmer is using. Details about how a macro processor works may annoy but probably won't deter such working programmers. They just use what is available as best they can, and don't write the theoretical articles. Practical use of macros also tends to be awkward, and describing their use can be messy. Thus, it is difficult in a short paper to describe serious uses of macros.

On the other hand, people who might be reluctant to use a "real" programming language may use macros (e.g., editor macros) in quite sophisticated ways. In the multi-chapter interview of Knuth in his *Companion to the Selected Papers of Donald Knuth* (page 161),[59] Knuth says of his wife, "I don't think she will ever enjoy programming. She is good at creating macros for a text editor, sometimes impressing me with subtle tricks that I didn't think of, but macro writing is quite different from creation of what we call 'recursive procedures' or even 'while loops.'"

Macro processors are also not as tractable a topic for historical or theoretical writing as high level languages perhaps are. While many macro processors may be for relatively straightforwardly extending a programming language or aiding in a development task, some macro processors (such as `m4` and the macro capability in TeX) have a big set of capabilities for dealing with sophisticated programming situations — they are set up to write big complicated programs, just as regular higher level languages are, but often in more clumsy-to-use ways. There is a 160-page manual to describe all the capabilities of the stand-alone `m4` macro processor. In addition to Knuth's own writings,[39,41] Eijkhout's book[40] has more than 50 pages on the extensive macro capabilities embedded in TeX, and Stephan von Bechtolsheim wrote (perhaps excessively) a 650-page book[60] on using TeX macros. (There is probably an interesting paper to be written comparing the issues inherent in these powerful macro processors with similar issues in regular procedural programming languages.)

The immediately preceding discussion brings to mind the question, "Where on the spectrum of programming languages do we put macro processors?"

Early on they were used as ways to extend assemblers and compilers. Quickly they were used to implement programming languages (e.g., WISP and SNOBOL). They were used to simulate different "computers" (e.g., BCPL's compiler of O-code, the assembly language of the Honeywell 516 IMP computer). I personally have used them to define and process little special purpose markup languages (as described in the `m4` and TEX sections above).

Many macro processors are Turing-complete and in this sense can compute anything a typical higher level language can compute. However, they are distinct from many higher level language compilers (discounting macro capabilities they might have, or other compile-time evaluation capabilities, as languages which execute interpretively often have, e.g., Lisp, Perl) in that much computation in macro processors inherently goes on at "compile" time. This can make their syntax and semantics harder to specify.

In Knuth's paper "The Genesis of Attribute Grammars", reprinted in his *Selected Papers on Computer Languages*,[61] Knuth explains (pages 434–435) that he didn't use an attribute grammar to specify the semantics of TEX because he hadn't been able to think of *any* good way to define TEX precisely except through the implementation using Pascal. He also couldn't think of how "to define any other language for general-purpose typesetting that would have an easily defined semantics, without making it a lot less useful than TEX. The same remarks also apply to METAFONT. 'Macros' are the main difficulty. As far as I know, macros are indispensable but inherently difficult to define. And when we also add a capability to change the interpretation of input characters, dynamically, we get into a realm for which clean formalisms seem impossible."

**Personal reflections.** Macro overview books, such as those by Brown and Campbell-Kelly, try to categorize the uses of macro processors, for example, to extend a programming language, to allow late binding of variables to values, to communicate with the operating system, to implement desirable-but-nonexistent machine instructions, to insert debugging code into a program, and to simply abbreviate long strings of text. Below is a short list of the ways I feel I have made use of the macro processors described in this note.

- The macro processor described in McIlroy's paper, Strachey's GPM, and Mooers' TRAC were for the study of programming languages, understanding of how macro processors work, and for coding practice.
- Midas was about implementing a macro processor and using one for production work including retargeting Midas to a new computer and putting hints in a real-time program to reduce the probability of time-sharing bugs.
- TECO and other editor macros were to reduce editing keystrokes, especially when regular expression searches and replacements are needed.
- Ratfor and C were for programming efficiency.
- TEX macros are for production typesetting, creating extensions to LATEX (e.g., the style for a particular book), and as another sophisticated macro processor to study.
- `m4` use has been to create new markup languages with which to target document source files to different output media.

In addition to assembly languages for different computers, Fortran for various mathematical projects, and the languages mentioned above, I briefly programmed in Lisp and BCPL years ago. Lisp has a macro capability and I may have used it, but, if so, I don't remember anything about it.[62] While writing these notes, I have learned that a version of Strachey's GPM existed for BCPL, known as BGPM.

In recent years I have used Perl for several large and many small projects. Apparently Perl has a capability for redefining parts of the Perl language (the equivalent of macros and more), but it confused me when I tried to read about it and am not likely to try it. I suppose I can just use `m4`, which I already know, as a preprocessor for Perl; but that doesn't stop me from wishing that a conventional macro capability was built into Perl.

**Acknowledgments**

Chuck Niessen helped me remember the capabilities of the Univac 1219 macro assembler we wrote. Ralph Alter and Will Crowther responded to queries relating to macro capabilities on Lincoln Laboratory computers. Stan Mazor confirmed my memory about whether or not we used macros in the early 1960s on the IBM 1620 at San Francisco State College. Tom Van Vleck reminded me of what languages with macro processors were available at MIT in the late CTSS and early Multics era. Jim Wood reminded me of some things about PARSEC. Bill Aspray and Martin Campbell-Kelly provided guidance on how this paper might be developed into a more traditional academic computing history paper. Bernie Cosell pointed out half a dozen subtle issues or possibilities in the paper. Ralph Muha gave me his copy of Gimpel's SNOBOL4 algorithms book. Most helpfully, Karl Berry read drafts of this note, noted typos, made suggestions for improvement, and provided additional insight about and references for a number

of macro processors, particularly TEX. Karl, along with Barbara Beeton, also edited the final draft of the paper to ready it for publication.

## Notes

[1] http://walden-family.com/impcode/imp-code.pdf

[2] [*Macro overview*] P. J. Brown, *Macro Processors and Techniques For Portable Software*, John Wiley & Sons, 1974.

[3] [*Macro overview*] Martin Campbell-Kelly, *An Introduction to Macros* (Computer monographs, 21), MacDonald & Co., London, 1974.

[4] [*Macro overview*] Peter Wegner, *Programming Languages, Information Structures, and Machine Organization*, McGraw-Hill Book Company, 1968, section 2.6 and section 3.1–3.4, pp. 130–180.

[5] [*Macro overview*] John R. Metzner, A graded bibliography on macro systems and extensible languages, *ACM SIGPLAN Notices*, Volume 14 Issue 1, January 1979, pp. 57–64.

[6] [*McIlroy's model*] M.D. McIlroy, Macroinstruction Extensions for Compiler Languages, *Communications of the ACM*, vol. 3 no. 4, 1960, pp. 214–220.

[7] In 1967 McIlroy's paper was reprinted in the now classic book *Programming Systems and Languages*, edited by Saul Rosen, McGraw-Hill, New York, 1967. For this paper I was looking at the reprint of Rosen's book. In the reprint McIlroy doesn't say anything about the macro name in this example also being used within the definition but not apparently as a recursive call of the macro, i.e., the 3-argument call can be distinguished from the 1-argument instruction.

[8] [*Macros at Bell Labs*] May 4, 2009, http://deememorial.blogspot.com/2009/05/doug-mcilroy-recalls-bell-labs.html

[9] [*Other macro systems*] M. Barnett, Macro-directive Approach to High Speed Computing, Solid State Physics Research Group, MIT, Cambridge, MA, 1959; D.E. Eastwood and M.D. McIlroy, Macro Compiler Modifications of SAP, Bell Telephone Laboratories Computation Center, 1959; I.D. Greenwald, Handling of Macro Instructions, *Communications of the ACM*, vol. 2 no. 11, 1959, pp. 21–22; M. Haigh, Users Specification of MICA, SHARE User's Organization for IBM 709 Electronic Data Processing Machine, SHARE Secretary Distribution SSD-61, C-1462, 1959, pp. 16-63; A.J. Perlis, Official Notice on ALGOL Language, *Communications of the ACM*, vol. 1 no. 12, 1958, pp. 8–33; A.J. Perlis, Quarterly Report of the Computation Center, Carnegie Institute of Technology, October. 1969; Remington-Rand Univac Division, Univac Generalized Programming, Philadelphia, 1957.

[10] [*Other macro systems*] This article was originally published in Volume 7 of the *Encyclopedia and Computer Science and Technology*, published by Michel Dekker in 1977. The article is reprinted as chapter 1 of Donald E. Knuth, *Selected Papers on Computer Languages*, Center for the Study of Language and Information, Stanford University, 2003.

[11] Page 6 in the volume of Knuth's selected papers.

[12] Page 42 of the selected papers volume.

[13] Martin Campbell-Kelly told me [email of 2013-11-23], "Maurice Wilkes developed a macro-based system called WISP for list processing (http://ai.eecs.umich.edu/people/conway/CSE/M.V.Wilkes/M.V.Wilkes-Tech.Memo.63.5.pdf) ... Peter Brown[2] was Wilkes' PhD student."

[14] [*GPM*] C. Strachey, A General Purpose Macrogenerator, *The Computer Journal*, vol. 8 no. 3, 1965, pp. 225–241, http://comjnl.oxfordjournals.org/content/8/3/225.full.pdf+html.

[15] See Section 1.2, Historical References, at http://www.gnu.org/software/m4/manual/

[16] [*GPM*] Another implementation of GPM can be found in Section 8.8 of James F. Gimpel's book *Algorithms in SNOBOL4*, John Wiley & Sons, 1976. (SNOBOL4 itself was implemented using a collection of macros which create a virtual machine for the purpose of language portability: Ralph E. Griswold, *The Macro Implementation of SNOBOL4*, W.H. Freeman and Company, San Francisco, 1972. SNOBOL4 is sometimes known as "Macro SNOBOL".)

[17] Email of November 16, 2013.

[18] [*Midas*] MACRO Assembly Program for Programmed Data Processor-1 (PDP-1), Digital Equipment Corporation, Maynard, MA, 1963, http://bitsavers.informatik.uni-stuttgart.de/pdf/dec/pdp1/PDP-1_Macro.pdf

[19] Email of October 28, 2013.

[20] [*Midas*] http://bitsavers.informatik.uni-stuttgart.de/pdf/mit/rle_pdp1/memos/PDP-1_MIDAS.pdf

[21] [*Midas*] http://bitsavers.trailing-edge.com/pdf/mit/tx-0/memos/M-5001-39_MIDAS_Nov62.pdf

[22] Ralph Alter wrote a version of TECO, Dan Murphy's paper Tape Editing and Correcting Program: Dan Murphy, The Beginnings of TECO, *IEEE Annals of the History of Computing*, 31(4), October–December 2009, pp. 225–241. Will Crowther wrote a version of Alan Kotok's DDT debugging program. Chuck Niessen and I wrote a version of Robert Saunder's macro assembler: Chuck wrote the basic assembler and I wrote the macro-processor.

[23] I have the line printer assembly listing and my handwritten flow chart for this macro processor. It is tempting to try to make this macro processor run again on a 1219 emulator.

[24] This was the first program in which I used stacks and for which I thought about recursion in a profound way. I think this was also the first time I wrote code for managing a symbol table.

[25] I don't remember using this capability in TECO at Lincoln Lab.

[26] I won't bother with further mention of macros in other editors (or macros in various job control languages, shells, or makefiles) in the rest of this note.

[27] [*TRAC*] Calvin Mooers and Peter Deutsch, TRAC: A Text Handling Language, *Proceedings of the 20th ACM National Conference*, 1965, pp. 229-246.

[28] As part of my investigation, I visited TRAC creators Calvin Mooers at his Cambridge office (of the Rockford Research Institute) and Peter Deutsch at the Cambridge home of his parents.

[29] Brown's book[2] in fact classifies TRAC as a macro processor.

[30] [*Extensible languages*] Niklaus Wirth, On Certain Basic Concepts of Programming Languages, Technical Report No. CS 65, Computer Science Department, Stanford University, May 1, 1967; B.A. Galler and A.J. Perlis, A Proposal for Definitions in ALGOL, *Communications of the ACM*, vol. 10 no. 4, April 1967, pp. 204–219; T.E. Cheatham, Jr., A. Fischer, and P. Jorrand, On the Basis for ELF — An extensible language facility, AFIPS Fall Joint Computer Conference, 1968, pp. 937–948; J.V. Garwick, J.R. Bell, and L.D. Krider, The GPL Language, Programming Technology Report TER-05, Control Data Corporation, Palo Alto, CA; Thomas A. Standish, A Data Definition Facility for Programming Languages, PhD thesis, Carnegie Institute of Technology, 1967; James Richard Bell, The Design of a Minimal Expandable Programming Language, PhD thesis, Stanford University, 1968.

[31] [*Extensible languages*] F. Webb, The PROPHET System: An Overview of the PARSEC Implementation, BBN Report 2319, September 1, 1972; PARSEC User's Manual, Bolt Beranek & Newman, Cambridge, MA, December 1972.

[32] F.E. Heart et al., The Interface Message Processor for the ARPA Computer Network, AFIPS Conference Proceedings 36, June 1970, pp. 551–567.

[33] `http://walden-family.com/impcode/c-listing-ps.txt`

[34] `http://walden-family.com/impcode/d-concordance.pdf`

[35] `http://walden-family.com/impcode/detect-interrupt-bugs.pdf`

[36] [*Midas*] `http://walden-family.com/impcode/midas516.txt`

[37] [*Ratfor*] Kernighan, B. and Plauger, P., *Software Tools*, Addison-Wesley, 1976.

[38] [*C preprocessor*] `http://gcc.gnu.org/onlinedocs/cpp/`

[39] [*TEX*] Donald Knuth, *The TEXbook*, Addison-Wesley, 1986, particularly chapter 20.

[40] [*TEX*] Victor Eijkhout, *TEX by Topic*, Addison-Wesley, 1991, particularly chapters 11–14, `http://mirror.ctan.org/info/texbytopic`

[41] [*TEX*] Donald Knuth, *Computers & Typesetting, Volume B, TEX: The Program*, Addison-Wesley, 1986.

[42] [*TEX*] Donald E. Knuth, *Digital Typography*, Center for the Study of Language and Information, Stanford University, 1999.

[43] [*TEX*] `http://mirror.ctan.org/macros/latex209/contrib/biblist/biblist.pdf`

[44] [*TEX*] `http://tug.org/TUGboat/tb25-2/tb81walden.pdf`

[45] Peter Seibel, *Coders at Work*, Apress, 2009, p. 597.

[46] [*TEX*] `http://www.luatex.org/`

[47] [*TEX*] Donald E. Knuth, Literate Programming, `http://literateprogramming.com/knuthweb.pdf`

[48] Email of December 5, 2013, from Karl Berry.

[49] Private communication, January 11, 2014; the footnotes in the quotation are from the author of the present paper, not from Knuth.

[50] D. V. Schorre, META-II: A Syntax-oriented Compiler Writing Language, D. V. Schorre, *Proceedings of the ACM, 19th ACM National Conference*, ACM, New York, 1964, pp. D1.3-1–D1.3-11, `http://ibm-1401.info/Meta-II-schorre.pdf`

[51] Edgar T. Irons, A syntax-directed compiler for ALGOL 60, *Communications of the ACM*, vol. 4, 1961, pp. 51–55; Edgar T. Irons, The structure and use of a syntax-directed compiler, *Annual Review of Automatic Programming 3*, 1962, pp. 207–227.

[52] `http://walden-family.com/4prim/`

[53] [*M4*] `http://www.gnu.org/software/m4/manual/`

[54] The full table is at `http://walden-family.com/4prim/archive/issues-list.htm`

[55] [*M4*] `http://en.wikipedia.org/wiki/M4_(computer_language)`

[56] While `m4` was originally developed by Brian Kernighan and Dennis Ritchie in 1977 and released as part of AT&T Unix, the GNU version mentioned here was a complete rewrite by René Seindal with continuing updates by many others. The GNU `m4` manual's history section[15] has a good bit of additional history.

[57] `http://tug.org/interviews/`

[58] Martin Richards, Christopher Strachey and the Cambridge CPL Compiler, *Higher-Order and Symbolic Computation* (a special Christopher Strachey memorial issue), 13, 2000, pp. 85–88.

[59] Donald E. Knuth, *Companion to the Selected Papers of Donald Knuth*, Center for the Study of Language and Information, Stanford University, 2012.

[60] [*TEX*] Stephan von Bechtolsheim, *TEX in Practice, Volume III: Tokens, Macros*, Springer-Verlag, 1993.

[61] Donald E. Knuth, *Selected Papers on Computer Languages*, Center for the Study of Language and Information, Stanford University, 2003.

[62] Tim Hart at MIT added a macro capability to Lisp in 1963. Macros in Warren Teitelman's BBN Lisp, which I used, were perhaps more well developed. In some sense the original Lisp interpreter functioned a bit like a macro processor.

◇ David Walden
`http://walden-family.com`