Strachey's General Purpose Macrogenerator

Andrew Herbert

8th May 2014

Introduction

As part of my on-going project to resurrect Elliott 903 software I regularly receive files from Terry Froggatt containing copies of paper tapes from his large collection. In one of the more recent batches there was a SIR assembly code tape labelled "GPM10 9/5/77, 900-Series Telecode" [1] and I guessed, correctly as it turned out, that this might be an implementation of Christopher Strachey's General Purpose Macrogenerator (GPM). This article gives an account of GPM; of errors found in the Computer Journal paper describing its use [2]; a fatal flaw in the implementation given by the same paper; and an improved version of GPM developed fixing both problems.

GPM

GPM was a well-known and popular macro generator dating from the early 1960's. GPM was initially conceived by Christopher Strachey when working as part of the Cambridge team designing the CPL programming language for the then new Cambridge Titan computer. Strachey proposed GPM as a tool to be used as an intermediate language for the compiler. The idea was to translate CPL into GPM macro calls that would in turn be expanded into machine code. According to David Hartley this didn't work too well in practice – GPM was a more powerful tool than required for the task and was too easily abused, resulting in impenetrable code.

The GPM notation is deceptively simple. A string such as:

```
§DEF,Plus,<ADD ~1 ~2>;
```

introduces a new macro "`Plus`" which when called with two arguments, `A` and `B` say, would generate:

```
ADD A B
```

Such a call would be written:

```
§Plus,A,B;
```

The symbols § and **;** delimit a macro call. `DEF` is a built-in macro that defines its second argument as having the value of its third argument. Commas separate the arguments of the call.

The symbol ~ followed by a digit `n` substitutes the *n*-th argument of the current macro, if any. `~0` substitutes the macro name.

The symbols < and > are used to “quote” text – i.e., to stop it being expanded by the macroprocessor. The symbols are stripped off as the text it read, this in the definition of `Plus`, the macro body is quoted to prevent argument substitution, but when `Plus` is called, the quotes are missing and so argument substitution occurs.

The notation is completely general: macro definitions can be nested, arguments can include macro calls (including calls to `DEF`), quotes can be nested, etc, etc. There are a handful of built-in macros: `DEF` to define a new macro, `UPDATE` to replace a macro definition with a new body, `BIN` to convert a textual representation of an integer to binary form, `DEC` to convert a binary integer to an textual one and `BAR` which allows simple arithmetic on binary quantities. There is also a macro `§VAL,name;` that returns the unevaluated string bound to `name` as its result.

These simple facilities are sufficient to write quite complex programs. Here for example is a conditional:

```
    §A,§DEF,A,<X>;§DEF,B,<Y>;;
```

This defines A to be <X>, then B to be <Y> and then immediately evaluates §A. If A=B the result is Y, otherwise the result is X. Thus we have a template for

```
    if A=B then X else Y
```

From this we can progress to more exotic forms such as:

```
    §DEF,Succ<0,1,2,3,4,5,6,7,8,9,10,§DEF,1,<~>~1;;>;
```

The effect of a call such as `§Succ,3;` is to make a temporary definition of a macro `1` and defining string `~3` and then call it with the arguments `2,3,4,...10,` producing the result `4`. So, the effect of `§Succ,`*r*`;` where 0<*r*<10 is to calculate the numerical successor of *r*.

There are further subtleties to do with how GPM handles superfluous arguments and the interplay between permanent and temporary macro definition for which the reader should refer to Strachey’s paper.

The reader can perhaps start to see the veracity of David Hartley’s observation the GPM is perhaps too clever for its own good. That said, GPM was of considerable interest during its time as an object of academic study and is acknowledged as an important ancestor of the m4 macro generator used by Unix systems today.

Bugs

In his paper on GPM, Strachey develops a more complicated successor function

```
    §Successor,a,b,c;
```

that computes the *c*-th successor to the two digit number *ab*.

The definition is given as:

```
    §DEF,Successor,<§~2,§DEF,~2,~1<,§Suc,>~2<;>;
              §DEF,9,<§Suc,>~1<;,0>;;>;
```

but, when given to GPM10, the macro failed to work as expected and pulling on this loose thread unravelled spectacularly. To work correctly the definition should have been:

```
    §DEF,Successor,<§~2,§DEF,~2,~1<,>§Suc,>~2<;>;
              §DEF,9,<§Suc,>~1<;<,>0>;>;
```

i.e., with some of the commas quoted, otherwise `DEF` is called with too many arguments and the wrong result produced.

This error was probably a simple transcription error from Strachey's original to the final typeset paper. Such errors are far less likely these days when we can cut and paste from program files to camera-ready copy.

At first, when I found the GPM10 program wouldn't produce the expected result from the `Successor` macro, I suspected a bug in the code. Careful checking revealed the SIR code was a correct transcription of the CPL code given in the original paper, so then I began to doubt the CPL, especially as Strachey notes in the paper that the GPM implementation used at Cambridge was different to that specified in the paper. This led me to search the World Wide Web to see if there were other implementations of GPM still extant. I discovered just one, in BCPL, maintained by Martin Richards, a now retired lecturer from the Cambridge University Computer Laboratory, a former colleague from my own time at Cambridge.

I learned that Richards had been Strachey's PhD student, hence his interest in GPM. Most readers will know Richards wrote BCPL as a simplified version of CPL while visiting MIT in the early '60s; that BCPL was a popular systems programming language in the '60s, '70s and '80s; and is an ancestor of C. Richards continues to maintain BCPL [3] and in his collection of BCPL examples there was a program `bgpm` that claimed to be an implementation of GPM. Frustratingly `bgpm` also wouldn't run the Successor macro and worryingly `GPM10` wouldn't run many of the test cases for `bgpm`. This led me to change tactics and code up my own `Successor` macro and in so doing I found the error in the original.

Looking at the `bgpm` it was apparent Richard's program was a different algorithm to Strachey's. In particular Strachey's CPL uses a single stack to evaluate macros and puts temporary macros on that stack using a linked list to maintain an environment chain that can be scanned to find the most recent definition of a given name. Richards by contrast uses two stacks – one for macro evaluation and a second for temporary definitions. I contacted Richards and he explained there was a fatal flaw in Strachey's algorithm: the special markers used to delineate environment chain information could be left on the evaluation stack after a call of `DEF` and give rise to errors. (Strachey indirectly acknowledges this in section 2.6 of his paper). By using a separate stack for the environment, Richards sidestepped the problem. However, in changing the data structures, Richards had inadvertently changed the binding rules for temporary macros and this is what led to his examples not running on `GPM10` and my proposed correction to Strachey's `Successor` macro not running on `bgpm`.

Following our discussion Richards wrote a BCPL equivalent of the CPL version in Strachey's paper. With this program, which included a useful backtrace facility and better error reporting that `GPM10` I was able to confirm the error in `Successor` and its correction.

`bgpm` Evolution

Following on from writing the BCPL version of Strachey's algorithm, Richards went back and updated `bgpm` to handle temporary macros identically to Strachey, but importantly still with two stacks, so that problems with environment chain markers couldn't arise. This version thus correctly expands all of Strachey's examples and, with some minor recoding, all of Richards' examples (including those which would have crashed Strachey's original due the environment chain marker problem).

Richards `bgpm` contains several other improvements. In particular arithmetic is handled by a macro `eval` that understands simple numerical expressions. Thus

```
§DEF,x,1; ... §UPDATE,X,§BAR,+,§VAL,x;,§BIN,1;;;;
```

becomes:

```
[def\x\1] ... [set\x\[eval\[x]+1]]
```

(Richards uses different delimiter symbols better suited to the ASCII character set.)

Richards' form is easier to read and importantly avoid the risk of leaving arbitrary binary values on the evaluation stack which might confuse the scanner (as does indeed occur with Strachey's algorithm if certain negative numbers are computed).

Two new macros `lquote` and `rquote` are provided that allow left and right quotes to be generated in the output stream, a completeness feature curiously missing from Strachey's original.

Argument naming is also tightened up. A ~ can be followed by any integer allowing arbitrary numbers of arguments, whereas in Strachey's GPM argument numbers greater than 9 had to be encoded using the symbols that followed digit 9 in the Titan character code, which is obscure.

Perhaps most importantly of all, `bgpm` allows comments to be inserted in GPM input. A ` (grave) symbol causes the scanner to skip all remaining text on the current line and continue to skip white space until some other symbol is found. This is valuable in allowing complex macros to be documented inline and layout be used to improve readability without impacting adversely on the layout of the output.

Elliott 903 GPM

Having resolved the problems running Strachey's examples and working with Richards on the convergence of `bgpm` and Strachey's semantics, I wrote a translation of `bgpm` in SIR as a replacement for `GPM10` and this is available from my Elliott 900 series software and documents archive to any who might be interested. (Writing a 903 code generator for BCPL is a project for another day).

The origins of `GPM10` itself are unclear: it was not issued by Elliotts: there were GPM like facilities in the Elliott MASIR (macro SIR) assembler, but not the complete set. And, indeed why `GPM10`? I recall there being a GPM program for the 903 at Leeds University when I was a student there from 1972-5. The `GPM10` tape however is dated 1997. I could easily imagine there were several versions in circulation at various times given the popularity of GPM in those days and the suitability of implementing GPM as a final year student project. If any CCS member can shed further light on these matters I'd be pleased to hear from them.

Summary

An early and influential macro generator has been preserved[1] with two implementations, one in Elliott SIR, the other in BCPL, which although now a vintage programing language, is still supported on most current operating systems including, most recently, the Raspberry Pi. During the preservation a number of bugs in the implementation and illustrative examples have been corrected, offering a much-improved version for the modern user. The Computer Journal has published a record of the errors on its web site.

References

[1] `GPM10` can be found in the archive of Elliott software and documentation at: http://homepage.ntlworld.com/andrew.herbert1/andrew_herbert/elliott.html.

[1] Taking David Holdsworth's definition of "software preservation," namely being able to run the software on current platforms.

[2] “General Purpose Macrogenerator”, C.J. Strachey, **The Computer Journal** (1965) 8 (3): 225-241.

[3] `bgpm` and `bgpmcj` can both be downloaded from: http://www.cl.cam.ac.uk/~mr10/BCPL.html.