META II: Digital Vellum in the Digital Scriptorium

Revisiting Schorre's 1962 compiler-compiler

Dave Long

Some people do living history—reviving older skills and material culture by reenacting Waterloo or knapping flint knives. One pleasant rainy weekend in 2012, I set my sights a little more recently and settled in for a little meditative retro-computing, ca. 1962, following the ancient mode of transmission of knowledge: lecture and recitation—or rather, grace of living in historical times, lecture (here, in the French sense, reading) and transcription (or even more specifically, grace of living post-Post, lecture and reimplementations).

Fortunately, for my purposes, Dewey Val Schorre's paper¹⁰ on META II was, unlike many more recent digital artifacts, readily available as a digital scan.

META II was a “compiler-compiler,” which is to say that when one suspects that a production compiler might be a rather large project to write in assembly—and especially if one were in an era in which COTS (commercial off-the-shelf), let alone libre and open source, compilers were still science fiction—then it makes sense to aim for an intermediate target: something small enough to be hand-coded in assembly, yet powerful enough for writing what one had been aiming for in the first place.

Just as mountain climbers during the golden age of alpinism would set up and stock a base camp before attempting the main ascent, and later expeditions could derive benefit from infrastructure laboriously installed by a prior group, the path to the language ecosystem we now use (cursing only on occasion) was accomplished in a series of smaller, more easily achievable, steps. Tony Brooker (who already in 1958 was faced with the “modern” problem of generating decent code when memory access will incur widely varying latencies) wrote the compiler-compiler² (of which Johnson's later, more famous one was “yet another”⁶) to attack this problem in the early 1960's. According to Doug McIlroy, Christopher Strachey's GPM (general-purpose macrogenerator—a macroexpander of the same era) was only 250 machine instructions, yet it was sufficient to enable Martin Richards's BCPL (Basic Combined Programming Language) implementation, later inspiring Ken Thompson to bootstrap C via B, eventually leading to the self-hosting native-code-generating tool chains we now take for granted.

A horse can pull more than a man, but by exploiting leverage, Archimedes can, with patience, move what Secretariat could not. META II is a fine example of a field-improvised lever: one can see how the beam has been roughly attached to the fulcrum and feel how the entire structure may be springier than one would like, but in the end, no matter how unpolished, it serves to get the job done with a minimum of fuss.

WHY STUDY META II?

1. There is not much to examine.
2. There is not much to examine because its parts are simply defined.
3. It enables significant consequences.

I will not go into detail, as nearly all of the interest in this exercise comes from doing it yourself. Programming (when not constrained, as it often is in our vocation, by economic concerns) is not a spectator sport. Donald Knuth, who says a simple one-pass assembler should be an afternoon's finger exercise, might wish to make some additional plans to fill his weekend; it might take closer to four or five evenings if you must first refresh dim memories of a university compiler course. Instead, I will describe the general route of my ascent and why I am confident that I arrived at the same summit that Schorre described well before my birth. By following Schorre's text, possibly aided by mine, you should also find climbing this peak to be an easy and enjoyable ascent. (An alternative for the hardcore: following the Feynman method, ask yourself one question: "what is the square root of a compiler?" and head up the mountain without a guide)

On first reading, Schorre's text may seem horribly naive. We have the benefit of a half-century of experience and a different vocabulary. However, just as it is often amazing how much our fathers seem to have learned in the time between when we turned 14 and when we turned 21, it becomes easy to admire what Schorre accomplished as we follow in his footsteps.

Digression: in examining medieval texts on horses, it is very clear that while equitation has changed very little in the intervening centuries, veterinary science has made giant strides. With this distinction between art and technique in mind—and being thankful that Schorre's text is, albeit in a typewriter font, neither in medieval French nor, worse, handwritten Fraktur—we can take advantage of hindsight to separate the informatics from the technical artifacts of having run on an IBM 1401 (end of digression).

Here is a smattering of the more striking passages to be found:

- "Although sequences can be defined recursively, it is more convenient and efficient to have a special operator for this purpose." With hindsight, we smile and nod as we recognize the Kleene star (cf. the "Thompson construction" *infra*).
- "These assemblers all have the same format, which is shown below:

LABEL	CODE	ADDRESS
1- -6	8- -10	12- -70."

Having grown up after the popularity of fixed column formats, I was introduced to the concept that other people might compute in other ways during high school at a summer job: upon attempting to write a PL/I "hello world" under CMS, I had to bring in older and wiser help who shook their heads, stroked their beards, and gravely informed me that all that needed to be done was to shift my code right one or two spaces, so it would no longer start in what was obviously the "comment" column.

- "Repeated executions, whether recursive or externally initiated, result in a continued sequence of generated labels. Thus all syntax equations contribute to the one sequence." In the modern style, or even in the late 1960s if you were Rod Burstall (cf. his Cartesian product⁴), you might call this monadic composition. In the days of small memories and essentially linear card decks, the flattened sequence was the norm rather than the exception, and in our times Rick Hehner's bunches⁵ are a good example of a case where flattening can make the formulae of "formal methods" more easily manipulable than normally nestable sets.

Note that it has taken only two pages for Schorre to describe what we need for META II. The

remainder of the paper is taken up with a description of VALGOL, which might make a suitable destination for another day. Let us take a brief pause, however, to examine a couple of points:

- “The omission of statement labels from the VALGOL I and VALGOL II seems strange to most programmers. This was not done because of any difficulty in their implementation, but because of a dislike for statement labels on the part of the author. I have programmed for several years without using a single label, so I know that they are superfluous from a practical, as well as from a theoretical, standpoint. Nevertheless it would be too much of a digression to try to justify this point here.” History agrees the digression would have been superfluous; indeed, now it seems strange that it then

FIGURE 1

.SYNTAX PROGRAM

```

OUT1 = '*1' .OUT('GN1') / '*2' .OUT('GN2') /
'*' .OUT ('CI') / .STRING .OUT('CL ' *).,

OUTPUT = (.OUT' '('
$ OUT1 ')' / '.LABEL' .OUT ('LB') OUT1) .OUT('OUT').,

EX3 = .ID .OUT ('CLL' *) / .STRING
.OUT('TST' *) / '.ID' .OUT('ID') /
'.NUMBER' .OUT('NUM') /
'.STRING' .OUT('SR') / '(' EX1 ')' /
'.EMPTY' .OUT('SET') /
'$' .LABEL *1 EX3
.OUT ('BT ' *1) .OUT('SET').,

EX2 = (EX3 .OUT('BF ' *1) / OUTPUT)
$(EX3 .OUT('BE') / OUTPUT)
.LABEL *1 .,

EX1 = EX2 $('/' .OUT('BT' *1) EX2 )
.LABEL *1 .,

ST = .ID .LABEL * '=' EX1
',.' .OUT('R').,

PROGRAM = '.SYNTAX' .ID .OUT ('ADR' *)
$ ST '.END' .OUT ('END').,

.END

```

seemed strange. *Tempora mutantur, nos et mutamur in illis* (times change, and we change with them).

- Finally, Schorre discusses the problem of backup vs. no backup, which is still a current topic, as the recent popularity of the PEG (parsing expression grammar) and other parsers will attest. In our times, however, we are not so interested in avoiding backup, but in avoiding the need to start at the beginning and process linearly until we reach the end. Luckily for compiler writers, whether or not a production can be matched by an empty string is a property that can be determined by divide and conquer... but it is one of the few¹ that are tackled so simply.

The heart of the matter comes in figures 5 and 6 in the original article, “The META II Compiler

FIGURE 2

TST STRING	TEST	AFTER DELETING INITIAL BLANKS IN THE INPUT STRING, COMPARE IT TO THE STRING GIVEN AS ARGUMENT. IF THE COMPARISON IS MET, DELETE THE MATCHED PORTION FROM THE INPUT AND SET SWITCH. IF NOT MET, RESET SWITCH.
ID	IDENTIFIER	AFTER DELETING INITIAL BLANKS IN THE INPUT STRING, TEST IF IT BEGINS WITH AN IDENTIFIER, IE., A LETTER FOLLOWED BY A SEQUENCE OF LETTERS AND/OR DIGITS. IF SO, DELETE THE IDENTIFIER AND SET SWITCH. IF NOT, RESET SWITCH.
NUM	NUMBER	AFTER DELETING INITIAL BLANKS IN THE INPUT STRING, TEST IF IT BEGINS WITH A NUMBER. A NUMBER IS A STRING OF DIGITS WHICH MAY CONTAIN IMBEDED PERIODS, BUT MAY NOT BEGIN OR END WITH A PERIOD. MOREOVER, NO TWO PERIODS MAY BE NEXT TO ONE ANOTHER. IF A NUMBER IS FOUND, DELETE IT AND SET SWITCH. IF NOT, RESET SWITCH.
SR	STRING	AFTER DELETING INITIAL BLANKS IN THE INPUT STRING, TEST IF IT BEGINS WITH A STRING, IE., A SINGLE QUOTE FOLLOWED BY A SEQUENCE OF ANY CHARACTERS OTHER THAN A SINGLE QUOTE FOLLOWED BY ANOTHER SINGLE QUOTE. IF A STRING IS FOUND, DELETE IT AND SET SWITCH. IF NOT, RESET SWITCH.
CLL AAA	CALL	ENTER THE SUBROUTINE BEGINNING IN LOCATION AAA. IF THE TOP TWO TERMS OF THE STACK ARE BLANK, PUSH THE STACK DOWN BY ONE CELL. OTHERWISE, PUSH IT DOWN BY THREE CELLS. SET A FLAG IN THE STACK TO INDICATE WHETHER IT HAS BEEN PUSHED BY ONE OR THREE CELLS. THIS FLAG AND THE EXIT ADDRESS GO INTO THE THIRD CELL. CLEAR THE TOP TWO CELLS TO BLANKS TO INDICATE THAT THEY CAN ACCEPT ADDRESSES WHICH MAY BE GENERATED WITHIN THE SUBROUTINE.

written in its own language” (figure 1 in this article) and “Order List of the META II machine” (figures 2 through 4 here). Now, it would certainly be possible to follow in Schorre’s footsteps directly, using the traditional bootstrap:

0. Hand-code the META II machine—this is basically an assembler-like virtual machine: in other words, a glorified left-fold (mine was about 66 lines of Python).

1. Hand-translate the META II productions to the machine language (211 lines of m2vm opcodes).

FIGURE 3

R	RETURN	RETURN TO THE EXIT ADDRESS, POPPING UP THE STACK BY ONE OR THREE CELLS ACCORDING TO THE FLAG. IF THE STACK IS POPPED BY ONLY ONE CELL, THEN CLEAR THE TOP TWO CELLS TO BLANKS, BECAUSE THEY WERE BLANK WHEN THE SUBROUTINE WAS ENTERED.
SET	SET	SET BRANCH SWITCH ON.
B AAA	BRANCH	BRANCH UNCONDITIONALLY TO LOCATION AAA.
BT AAA	BRANCH IF TRUE	BRANCH TO LOCATION AAA IF SWITCH IS ON. OTHERWISE, CONTINUE IN SEQUENCE.
BF AAA	BRANCH IF FALSE	BRANCH TO LOCATION AAA IF SWITCH IS OFF. OTHERWISE, CONTINUE IN SEQUENCE.
BE	BRANCH TO ERROR IF FALSE	HALT IF SWITCH IS OFF, OTHERWISE, CONTINUE IN SEQUENCE.
CL STRING	COPY LITERAL	OUTPUT THE VARIABLE LENGTH STRING GIVEN AS THE ARGUMENT. A BLANK CHARACTER WILL BE INSERTED IN THE OUTPUT FOLLOWING THE STRING.
CI	COPY INPUT	OUTPUT THE LAST SEQUENCE OF CHARACTERS DELETED FROM THE INPUT STRING. THIS COMMAND MAY NOT FUNCTION PROPERLY IF THE LAST COMMAND WHICH COULD CAUSE DELETION FAILED TO DO SO.
GN1	GENERATE 1	THIS CONCERNS THE CURRENT LABEL 1 CELL, IE., THE NEXT TO TOP CELL IN THE STACK, WHICH IS EITHER CLEAR OR CONTAINS A GENERATED LABEL. IF CLEAR, GENERATE A LABEL AND PUT IT INTO THAT CELL. WHETHER THE LABEL HAS JUST BEEN PUT INTO THE CELL OR WAS ALREADY THERE, OUTPUT IT. FINALLY, INSERT A BLANK CHARACTER IN THE OUTPUT FOLLOWING THE LABEL.
GN2	GENERATE 2	SAME AS GN1, EXCEPT THAT IT CONCERNS THE CURRENT LABEL 2 CELL, IE., THE TOP CELL IN THE STACK.
LB	LABEL	SET THE OUTPUT COUNTER TO CARD COLUMN 1.
OUT	OUTPUT	PUNCH CARD AND RESET OUTPUT COUNTER TO CARD COLUMN 8.

FIGURE 4

CONSTANT AND CONTROL CODES

ADR IDENT	ADDRESS	PRODUCES THE ADDRESS WHICH IS ASSIGNED TO THE GIVEN IDENTIFIER AS A CONSTANT.
END	END	DENOTES THE END OF THE PROGRAM.

2. Machine-translate the META II productions to the machine language (using the output from step 1).

Note that Schorre's character set does not include ";" hence his quasi-BNF (Backus-Naur Form) is written with the sequence ".,". Those in search of verisimilitude may wish to use a keypunch simulator to create a "deck" from figure 1. Type-ahead is anachronistic, however, so if you are going to wear the hairshirt, it may be better to try talking someone else into being your keypunch operator.

Before condemning APL for excessive terseness, you may want to remember both that it was formed before standard character sets, and that at 110 baud, you have much more time to think about each character typed than you do with an autocompleting IDE (integrated development environment). Before condemning Pascal for excessive verbosity, you may wish to recall that the Swiss keyboard has keycaps for not only the five English vowels, but also the French accented vowels, as well as the German umlauted vowels, and hence does not offer so much punctuation. Before condemning Python and Haskell for whitespace sensitivity, recall that Peter Landin came up with the "offside rule" in 1966,⁷ which "is based on vertical alignment, not character width, and hence is equally appropriate in handwritten, typeset, or typed texts." This was not only prescient with regard to the presentation of code in variable-width fonts, but presumably also catered to the then-common case of one person keypunching code that had been handwritten on a coding sheet by a different person.

As Schorre himself notes, because of the fixpoint nature of this process, it can, if one is fortunate, be forgiving of human error: "Someone always asks if the compiler really produced exactly the program I had written by hand and I have to say that it was 'almost' the same program. I followed the syntax equations and tried to write just what the compiler was going to produce. Unfortunately I forgot one of the redundant instructions, so the results were not quite the same. Of course, when the first machine-produced compiler compiled itself the second time, it reproduced itself exactly."

Being lazy, however, I chose to take a switchback on the ascent, bootstrapping via Python. Much as the Jungfrauoch or the Klein Matterhorn can now be approached via funicular and gondola instead of on foot, we can take advantage of string and named tuple library facilities to approach the same viewpoint with little danger of arriving out of breath. The pipeline I first set up was structured as follows:

0. Lexical analysis (unfolding the character-by-character input string into a sequence of tokens and literal strings).
1. Syntax analysis (unfolding the linear lexical list into a syntax tree).

2. Code generation (in a traditional syntax-directed style).

Depending on your programming subculture, you may prefer to call this *syntax-directed translation*, a *visitor pattern*, or even an *algebraic homomorphism*. No matter what it is called, the essence of the matter is that the mapping of a composition can be expressed as the composition of mappings, and we use this distributive property to divide and conquer (advice which was probably passed on to Alexander by Aristotle—showing that in certain things the ancients anticipated Hoare and Brelloch by a few millennia), pushing the problem of translation out to the leaves of our syntax tree and concatenating the results, thereby folding the tree back down to a sequence of output characters.

Each stage is motivated by a structural transformation: the first two steps take structure that was implicit in the input and make it explicit, while the final step uses this explicit structure to guide the translation but then forgets it, leaving the structure implicit in the generated code string. Had we included a link phase (in which we would be concerned with flattening out the generated code into a word-by-word sequence), the building up and breaking down of structure would be almost perfectly symmetrical.

Note that you can easily cut corners on the lexical analysis. Schorre notes, “In ALGOL, strings are surrounded by opening and closing quotation marks, making it possible to have quotes within a string. The single quotation mark on the keypunch is unique, imposing the restriction that a string in quotes can contain no other quotation marks.” Therefore, a single bit’s worth of parity suffices to determine if any given nonquote character is inside or outside of a string.

Schorre was even more frugal when it came to numeric literals: “The definition of number has been radically changed. The reason for this is to cut down on the space required by the machine subroutine which recognizes numbers.” Compare Schorre’s decisions with those taken in Chuck Moore’s “Programming a Problem-Oriented-Language”⁸ for an example of how much thought our forebears were prepared to put into their literal formats when they had to be implemented on these, by current standards, minuscule machines. (Such frugality reminds one of the Romans, who supposedly, during the negotiations to end the first Punic war, multiplexed a single set of silverware among everyone scheduled to host the Carthaginian delegation.)

The syntax analysis can also profitably cut corners. In trying to arrive at a system that can process grammatical input, you don’t actually need the full machinery to analyze the grammar from which you start. In fact, if you are willing to ignore a little junk, the grammar in figure 5 can be parsed as an expression entirely via precedence climbing, with “.”, “=”, and “/” being the binary operators and “\$” and “.OUT” being unary.

All of these cases are good examples of a general principle when bootstrapping: because you are initially not creating the cathedral, but merely putting up ephemeral scaffolding, you can save a good deal of effort by doing the unavoidable work (while still at the lower level, where everything is relatively difficult) in a quick and dirty manner, allowing you to do the desired work later in the proper manner (presumably much more easily, once you have a system operating at the higher level). Schorre’s paper takes two more steps in this manner, moving from META II to VALGOL I to VALGOL II all in the span of a few pages.

Another reason I took this route, rather than Schorre’s direct ascent, is because I had the luxury (much like discovering a fixed line left in place by a previous expedition) of having the skeleton of a precedence-climbing parser left over from a previous project; hence, parsing Schorre’s expressions was simply a matter of changing the operator tables. In this case, my luck was due to having been

inspired by Martin Richards's simple parsers⁹. Richards was a pioneer in the technique of porting and distribution via virtual machine, and his expression parsers are often under a dozen lines each; mine was left over from a reimplement in `sed(1)`, and so (having eschewed integer arithmetic) is comparatively bloated: a score of lines.

At this point, I've climbed a bit and can look down with some satisfaction at the valley below, but the switchback means I've moved a good deal sideways from the original line of ascent. I am parsing Schorre's original file and generating code, but the code is for his VM (virtual machine), which I have not yet rewritten. Again, rather than aiming directly for the summit, I took another switchback. In this case, it was to rewrite Schorre's grammar to generate Python code rather than META II. This is another invaluable property of good intermediate positions: I have not yet properly reconstituted Schorre's system, but there is enough of the machinery in place to use it as intended, as a seed that can be unfolded in different ways to solve different sorts of compilation problems.

Sure enough, Schorre's system was flexible enough to generate code in a language that would not even have been started until a quarter century later. Because of additional `.LABELS` for the import boilerplate, and an expansion of `EX2` to `EX2` and `EX25` so I could trivially express META II's sequential composition in Python as short-circuit conjunction (`and`) with identity (`True`), the Python-generating META II grammar grew to 33 lines instead of 30. Now I needed to implement the functionality of the META II VM in Python. The advantage was that by generating Python code, I could implement each piece using a full high-level language, essentially a form of "big step" semantics. This consisted of about 85 lines of code, developed largely by the mindless method of iteratively rerunning the program and implementing each operation as execution reached the point where it became necessary. Debugging the null program is not to everyone's taste, but as A. N. Whitehead remarked: "Civilization advances by extending the number of important operations which we can perform without thinking about them. Operations of thought are like cavalry charges in a battle—they are strictly limited in number, they require fresh horses, and must only be made at decisive moments."¹⁴

At this point, I was able to use the Python-generating META II to regenerate itself. This was still a good deal laterally removed from the direct route to the summit, but it gave me confidence that I was heading in the correct direction, and perhaps more importantly, I have far more frequent occasion to use generated Python code than code generated for Schorre's META II VM.

Most importantly, I now had a good idea which data structures were necessary and how they fit together. (The vocabulary of programming changes as frequently as hemlines rise and fall, but the importance of structured data remains constant. Frederick P. Brooks said, in the language of his times, "Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious,"³ and before him, John von Neumann not only delineated control flow, but also meticulously tracked representation changes, in his 1947 flow diagrams¹³.) With this structure, it was obvious how to take Schorre's list of opcodes for his VM and create a Python version. Having gained some experience, this version was not only cleaner, but also shorter. Each of Schorre's opcodes turned out to be simply implementable in one to three lines of Python, so it was a relatively painless process. I had effectively implemented small-step semantics instead of a big step. To the extent that one could have arrived here directly, by following Schorre's description immediately from the paper, the switchbacks have been a waste of time. I found the diversion useful, however, because instead of needing to work out small-step

semantics from scratch, or to read and understand what Schorre had written, the direction to take at each step (as if I were following a well-blazed trail) was almost forced by the data given.

By this time, I appear to have reached a peak. In the distance, I can see the other peaks that Schorre discussed, VALGOL I and VALGOL II, as well as an entire chain of different peaks that might be more attractive to modern sensibilities. But how can I be sure (especially if the clouds have come in, and in the absence of a summit book) that I am standing where Schorre did half a century ago? This is the first time I might actually need to use some intellect, and luckily for me it is known that self-reproducing systems are fixed points, and bootstrap processes should therefore converge. Little need for intellect then: you merely need to confirm that running Schorre's program in figure 1 through a program for the machine given in figures 2-4 reproduces¹² itself. In fact, if you follow a similar set of switchbacks to mine, you will find that all of the possibilities converge: not only does META II via META II reproduce itself, but Python via Python (as noted supra) reproduces itself, and the two cross terms check as well: META II via Python produces the same output as META II via META II, and Python via META II is identical to Python via Python.

Note well the importance of self-reproduction here. It is not difficult to find self-referential systems: we may take the 1839 Jacquard-woven portrait depicting inventor Joseph Marie Jacquard seated at his workbench with a bunch of punched cards, or the fictional Baron Münchhausen pulling himself up by his pigtail (rather than by his bootstraps; having needed to lift his horse as well as himself, bootstraps were never an option — he sought a greatest rather than a least fixed point) as entertaining examples, but META II is a useful example of self-reference: it derives almost all of its power, both in ease of propagation and in ease of extension, from being self-applicable: from being the square root of a compiler.

What has this exercise accomplished? It has resulted in a self-reproducing system, executing both on the original META II VM (working from the original listing) and on Python or another modern language. Obviously, I could use the same process that I followed to bootstrap from the Python to the META II machine not only to port to yet another underlying technology, but also to become self-hosting. Less obviously, the basic problem I have solved is to translate (in a “nice” manner) one Kleene algebra (consisting of sequences, alternations, and repetitions) to another, which is a pattern that, if not ubiquitous in computing, is certainly common anytime we deal with something that has more structure than a linear “shopping list” of data. Compare Thompson's NFA (nondeterministic finite automaton) construction,¹¹ in which a search problem is solved by parsing a specification that is then executed on a virtual (nondeterministic) machine, with the twist that the nondeterministic virtual code has been further compiled into actual deterministic machine code.

Finally, remember that META II lends itself well to this kind of exercise precisely because it was designed to be bootstrapped. As Schorre says in his introduction: “META II is not intended as a standard language which everyone will use to write compilers. Rather, it is an example of a simple working language which can give one a good start in designing a compiler-writing compiler suited to his own needs. Indeed, the META II compiler is written in its own language, thus lending itself to modification.”

I hope the exercise of implementing your own META II will have not only the short-term benefit of providing an easily modifiable “workbench” with which to solve your own problems better, but also a longer-term benefit, in that to the extent you can arrange for functionality to be easily bootstrappable, you can help mitigate the “perpetual palimpsest” of information technology, in

which the paradox of bitrot means many artifacts effectively have a shorter half-life than even oral history.

After all, barbarians may be perfectly adapted to their environment, but to be part of a civilization is to be aware of how other people, in other places and times, have done things, and hence to know how much of what one does oneself is essential and how much accidental. More specifically, barbarians must learn from their own mistakes; civilized people have the luxury of learning from other people's. Very specifically, for engineers faced with ephemeral requirements, it is often helpful to avoid thinking of the code base at hand as a thing in itself, and instead consider it only a particular instantiation of the classes of related possible programs.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

REFERENCES

1. Backhouse, R. 2006. Regular algebra applied to language problems. *Journal of Logic and Algebraic Programming* 66; <http://www.cs.nott.ac.uk/~rcb/MPC/RegAlgLangProblems.ps.gz>.
2. Brooker, R. A., MacCallam, I. R., Morris, D., Rohl, J. S. 1963. The compiler compiler. *Annual Review in Automatic Programming* 3: 229-275.
3. Brooks, F. P. 1975. *The Mythical Man-Month*. Addison Wesley.
4. Burstall, R. M. 1969. Proving properties of programs by structural induction. *Computer Journal* 12 (1): 41-48.
5. Hehner, E. C. R. 1993. *A practical theory of programming*. Texts and Monographs in Computer Science. Springer.
6. Johnson, S. C. Yacc: yet another compiler-compiler; <https://www.cs.utexas.edu/users/novak/yaccpaper.htm>.
7. Landin, P. J. 1966. The next 700 programming languages. *Communications of the ACM* 9(3): 157-166; <http://doi.acm.org/10.1145/365230.365257>.
8. Moore, C. H. 1970. Programming a problem-oriented-language; <http://www.colorforth.com/POL.htm>.
9. Richards, M. 2007. *The MCPL Programming Manual and User Guide*. 58, 63; <http://www.cl.cam.ac.uk/~mr10/mcplman.pdf>
10. Schorre, D. V. 1964. META II: a syntax-oriented compiler writing language. In *Proceedings of the 19th ACM National Conference*: 41.301-41.3011; <http://doi.acm.org/10.1145/800257.808896>.
11. Thompson, K. 1968. Programming techniques: regular expression search algorithm. *Communications of the ACM* 11(6): 419-422; <http://doi.acm.org/10.1145/363347.363387>.
12. Thompson, K. 1984. Reflections on trusting trust. *Communications of the ACM* 27(8): 761-763; <http://doi.acm.org/10.1145/358198.358210>.
13. von Neumann, J., Goldstine, H. H. 1947. *Planning and Coding of Problems for an Electronic Computing Instrument*. Princeton, NJ: Institute for Advanced Study.
14. Whitehead, A. N. 1911. *An Introduction to Mathematics*. New York, NY: Henry Holt and Company.

DAVE LONG began his career with language tool chains, from supercomputers to prototype microprocessors, but rapidly turned to the dark side of consumer online services. He now divides his

time between developing equine area network sensor systems as an application of current technology to the problem of training horses for a millennia-old mounted game, and simply enjoying playing it. (The technology is not so advanced, the force feedback is at times more than one would wish, but the frame rate is always excellent.)

© 2014 ACM 1542-7730/14/1200 \$10.00

“J’ai seulement fait ici un amas de fleurs étrangères, n’y ayant fourni du mien que le filet à les lier.” (“I have gathered a posy of other men’s flowers, and nothing but the thread that binds them is mine own.”)
—Montaigne, *Essais*, 1595