Catching the Bug: Pupils and Punched Cards in South Africa in the Late 1970s

Martin S. Olivier

Department of Computer Science, University of Pretoria, Pretoria, South Africa
ms.olivier@olivier.ms

Abstract. This paper describes my memories as one of the pupils taking computer studies in South Africa in the late 1970s, when such programs were still uncommon. The differences between how the subject was approached back then and current approaches are not only due to technological developments; the 1970s curriculum reflects a fundamentally different approach.

Keywords: Computer studies, high school/secondary school, education.

1 Introduction

This paper is an account of my personal experience of studying computing at school level in the late 1970s. Where possible I verified my recollections, but this was not possible in all instances. The ENIAC was arguably the first 'real' computer built and switched on in 1947 [1]. The UNIVAC I followed as the first commercial computer. In 1951 one was installed, in 1952 two, in 1953 three and then, in 1954, twelve [2]. These installations were, however, mostly for government (and, in particular, military) use. The late 1950s saw the introduction of Fortran — a high-level programming language for engineering (and scientific) use [3]. Cobol — ostensibly for business use — was the next landmark event. Its design was completed in 1959 and the first compiler was available in 1960 [4]. But the academic differences — about software rather than hardware — reached boiling point in the 1960s. Algol-68 lead to significant unhappiness and the design of Pascal, not only as a programming language, but as an alternative philosophy to programming language design. IBM's OS/360 operating system was late and over budget [5]. The complexity of Honeywell's Multics operating system inspired Ken Thompson and Dennis Ritchie to design the much simpler Unix. As the issues emerged it became obvious that the problems were hard and required thorough academic inquiry. Often studied in other departments (such as mathematics or engineering) it became clear that computing had its specific problem characteristics that could not be addressed by these other disciplines. The first computer science departments in the world were established in the late 1960s. In addition, the use of the computer spread, providing those who wished to learn more about computing with opportunities — if they were lucky enough. And so it came that the study of computing started finding its tentative roots in schools in many countries around the world. This is my account of being one of the early pupils who enrolled for such a course in 1978 in South Africa.

A. Tatnall and B. Davey (Eds.): History of Computers in Education, IFIP AICT 424, pp. 291–301, 2014.

2 The Beginnings

My father bought his first calculator in about 1972. It could add, subtract, multiply and divide. But, perhaps more importantly, it had a bright eight digit 7-segment LED display. And there were riddles that, when one performed the calculation, yielded answers like 71077345. Turning it upside-down revealed the real answer. This was clearly an amazing machine — and we were into the computer age.

I am not sure when the various education departments in the country introduced Computer Studies as a school subject. It arrived at the neighbouring school in 1977. Back then our schooling started with a grade 1 and grade 2 (or sub A and sub B in some provinces). This was followed by another ten years referred to as standard 1 to standard 10 (with standard 10 also known as matric). In standard 8 one had to select the six subjects that one would take until matric. Computer studies was introduced as a seventh subject for standards 8, 9 and 10 offered at only a few schools. The following year I was in standard 8 and enrolled for computer studies, along with three or four classmates. The subject was to be offered at the neighbouring school, which made it particularly easy for us to attend. A large proportion of the class came from schools as far as 35km away. Classes were scheduled for one afternoon per week from 15:00 to 17:00. (The normal school day ran from 7:45 until 13:45.) Unfortunately for many these classes clashed with sport and cultural activities scheduled for the afternoon. Hence all my classmates except one dropped the subject pretty soon after it started. The two of us who persevered through the first year did continue until the end of matric, but in different schools. However, that is a story for somewhat later in this tale.

When I think back to computer studies 30 odd years ago, the main memories are about programming and programming languages. However, after digging up my old course material I was reminded that the course was about much more than just programming. Let us rather continue with these primary memories.

3 Programming and Programming Languages

Our introduction to programming (and, in fact, most other topics) followed a clear bottom-up approach — something that would probably be frowned upon today, but an approach that did have its benefits. Our language for standard 8 was Samos — an interpreted language based on assembly language. To design a program the logical starting point was a flowchart, which was, of course, hand-drawn using a stencil with all the necessary shapes (including some shapes that I to this day do not know what their purpose was). The primary shapes were the processing rectangles, the decision diamonds, the input/output parallelograms and the beginning and end ovals. The major concession in Samos was the ability to input and output variables with one simple instruction — compared to the usual effort required in a real assembly language.

In fact, Samos was more than a language: Samos was a hypothetical computer, with architecture and programming intermingled. The discussion of Samos begins

with a discussion of ferrite cores and how they could be magnetised in one or the other direction to represent a 0 or a 1. Words on this computer consisted of 61 bits — that is, ten bytes and one bit (at a time when many real computers used 6-bit bytes). The memory consisted out of 500 such words (numbered from 000 to 499).

The memory formed the basis of the comprehensive architecture that followed: The CPU, an arithmetic unit and input and output units were added to complete the picture. A handful of registers in the CPU (and arithmetic unit) were described, and it was said that the input unit is a card reader and the output unit a printer. Encoding data in the 6-bit bytes formed part of this exposition.

At this point we were almost ready to begin to discuss programming, but there was one other obstacle that remained before programming was possible. How would instructions be encoded on this computer?

The lone leftmost bit of the 61-bit word would normally be used to indicate the sign of the number stored in that word; for instructions this was not used. The next three (6-bit) bytes would be used for an opcode. This was followed by a 3-byte indexing value (which is not important for our immediate purposes). The remaining four bytes were the operand address.

The opcode was not the binary value we would (now) expect on a real computer; the three bytes were rather used to encode three characters, which was the mnemonic representation of some instruction. The very first program instruction we encountered was:

```
RWD 50
```

which meant 'read a word and store it in address (or memory word) 50. More specifically, it meant *read the first 11 characters from the next punched card and store it in word 50*.' The first character on that punched card would normally be the sign of a number, but is could also be blank for non-numeric input. The spacing of this RWD instruction was also important. To illustrate consider

```
s  012  34567  89 (Not part of the program)
   RWD         50
```

where s indicates the sign, bytes 0, 1 and 2 the opcode and bytes 6, 7, 8 and 9 the address to be used. Hence, no true assembly was required; this computer was literally programmed in machine code.

The next instruction introduced was `WWD` — write a word. And suddenly we were ready for our first program:

```
RWD 50
WWD 50
```

which would of course read a value from the punched card and print it on the printer.

The remaining instructions included some arithmetic instructions.

- `LDA` — Load accumulator
- `STO` — Store accumulator
- `ADD` — Add to accumulator
- `SUB` — Subtract from accumulator

In each case the operand indicated which value (which memory word) should be loaded, used for storage, added or subtracted. The list continues with MPY (multiply), DIV (divide) and even POW to raise the value in the accumulator to the indicated

power. Later in the course SHL and SHR were added, that would shift the accumulator the indicated number of *bytes* left or right. This was often useful to isolate, say, the month, day and year of a date read in as a single line.

Note again that addressing was direct (rather than immediate). To square a number one would put the value 2 into one of the words and use that word. The program was always read into memory starting at word 0. Hence a program could have looked as follows (with lines consisting out of dots unimportant at this stage).

```
000...........
001+         2
002...........
003...........
004...........
005 POW      1
006...........
```

On the coding sheet the first three columns were used to number statements. The actual content of each word therefore was represented from column 4 onwards. Since word 001 starts with a + that word contains the constant 2. When, at `005` we say `POW 1`, we therefore square the number in the accumulator.

The next set of instructions dealt with flow control, and the names are self-explanatory.

- `HLT` — Halt
- `BRU` — Branch unconditionally
- `BMI` — Branch if minus (that is, if the value in the accumulator is negative)
- `BPL` — Branch if plus
- `BRZ` — Branch if zero

In each case the operand was the destination address to jump to if the condition was true. These instructions were complemented by `CMP` (compare), `BRL` (branch if low), `BRE` (branch if equal) and `BRH` (branch if high) instructions.

At this point it was possible to write basic programs. However one still desperately needed some way of dealing with array-like structures. This was provided by three index registers, which could be loaded, incremented, tested, stored and manipulated in a number of ways. The purpose of the index registers was to modify the operand address to support loops that could indeed process an array of numbers. Recall that bytes 4, 5 and 6 in the instruction word were reserved for indexing. They indicate whether the contents of index register 1, 2 and/or 3 should be added to the operand address. Therefore

```
S 012  3456  789 (Not part of the program)
  STO  010   100
```

would add the contents of index register 2 to 100 and use this sum as the effective address where the contents of the accumulator would be stored. If indexing was not used these three columns were left blank. As an aside, in the case of instructions that do not use an operand (such as `HLT`) the operand field was also left blank.

Note that the convention was to start every program with a `BRU`, followed by words occupied by constants or reserved for variables. The actual program started after this ‘data part’ and was the target of the `BRU` instruction in word 0. This is still

standard practice in those rare situations where programmers work at such a low level. However, in courses I currently teach about malware analysis I note that 'modern' students find this concept (or even the concept of programs just being data that happens to be executed) rather foreign.

As noted, programming consisted of designing a flowchart that represented the solution; the flowchart had to be at a level low enough to easily translate into the (mnemonic) machine code described above. The flowcharts were then duly translated to Samos, where, just like in any other assembly language, every decision diamond was translated into code that contained a conditional jump (to a numeric destination!). This was a decade after Dijkstra's famous letter about the inherent harms of the goto statement [6]. And while Dijkstra was obviously correct in his assessment, I often did wonder if we did not gain some insight into the operation of computers that was missing for those who started by learning structured programming (or any of the subsequent or alternative paradigms). Yes, goto is harmful when developing code — especially if that code needs to be maintained. However, I doubt that the expected outcome of computer studies was that we would be trained programmers — just like the expected outcome of mathematics was not that we would be mathematicians. And those who went on to study computer science after school seemed to make the switch quite easily.

Once the program was flowcharted it was converted to code and this code was copied to coding sheets — in pencil so that writing errors could easily be corrected. And then those coding sheets were sent to the education department's regional headquarters, which had a computer (or remote access to one). There our coding sheets were punched onto punched cards and the programs were run. A few weeks later one would receive the stack of cards back, as well as the printout of this initial run. If the program (based on the test data) was correct, one was awarded full marks — ten out of ten — for that assignment. Else one was allowed to correct the program by (if I recall correctly) inserting notes into the stack of cards. These stacks were returned to the regional centre, where the "punch ladies" made the necessary corrections by punching new cards to reflect the changes we requested and inserting them at the appropriates places in the card stack. The program was run once more using this amended stack, and the cards and printouts arrived back at the school two or three weeks later. If the program worked this time one earned eight out of ten marks for the assignment. If it failed yet again, the teacher had to mark it and assign a mark between 0 and 7 to one's attempt. The process was sometimes slowed down a bit when punching errors occurred and one, in principle, deserved another chance.

One of the elements that would later become part and parcel of programming — test runs and debugging — did not exist for us. Careful coding and checking algorithms using trace tables were implicit in our curriculum. Even in later years — at university — when we could fit in a couple of test runs to debug our programs before submission — each run was time consuming, initially requiring hours per attempt, but later still with a single digit number of attempts that could be tried in a day, careful coding and clear thinking was still the norm. My observation was that this really changed when Borland's Turbo Pascal was introduced at universities, where press of

the F9 button yielded results within seconds, and students could afford to move from a thinking strategy to a trial and error strategy.

In the following year — 1979, standard 9 — we were ready to face a high-level language, and the (obvious?) choice was Fortran. What made it 'obvious' was the fact that not that many compilers were available for the mainframe computer(s) the department of education was operating. Perhaps some perspective is required in this regard. The first microprocessor, the 4-bit Intel 4004 was released at the end of 1971. The Intel 8080, an 8-bit processor that could form the basis of a microcomputer was released in 1974. The first real microcomputer, the famous Altair 8800, was designed in 1975, based on this chip. Various kit-based computers followed. The first complete personal computers followed in 1977 and later (Commodore PET, Tandy Corporation's TRS-80 and Apple Computer's Apple II — all in 1977; the Atari 400/800 in 1978/1979; the Sinclair ZX80 in 1980; the Texas Instruments TI-99 in 1979; the Commodore VIC-20 and Commodore 64 in 1982 and 1980; the Acorn Atom in 1980; the BBC Micro in 1981; and, of course the IBM PC in 1981). The only high-level language available on these microcomputers was Basic, with Microsoft's implementations very common (and, the language that was the reason Microsoft was formed at all). However, for us those microcomputers remained a dream — often costing upwards of 1980 USD 500 it was not even a thinkable option to bring computers into schools. Many universities had access to minicomputers and the esoteric languages, such as Pascal and Algol were running on them. But neither the department of education, nor the schools had access to them. It is also worth noting that the University of Pretoria — my current employer — established its computer science department in 1976. So, in 1979, Fortran was indeed the obvious choice to introduce us to high-level languages.

However, the modern reader will probably not even know about the various flavours of Fortran that existed. In 1979 the FORTRAN-77 standard was already published, but the language in common use was Fortran IV. FORTRAN 77 would introduce some major improvements, of which the block IF-statements would arguably have been the most important for pupils learning to program. However, FORTRAN IV it was, which by implication meant an IF-statement was to be followed by a single statement that would be executed if the tested condition was true. There wasn't really any ELSE part associated with an IF-statement, unless one interprets the *arithmetic* IF-statement in this manner. The arithmetic IF-statement looked something like this:

```
IF (N*K+5) 100,200,300
```

The `N*K+5` could be any arithmetic statement. If this statement evaluated to a negative value, the program would jump to the statement next to the first label following the IF; in this example, a negative results would cause execution to be continued at the statement labelled 100. Similarly, a zero result would transfer control to the statement associated with the second label in the list (which would be 200 in the current example). Finally, a positive result would cause it to jump to the final label.

A 'normal' IF-statement would look something like the following

```
IF (N.GT.2) K=5
```

The `.GT.` in this example is the logical condition and is an abbreviation for *greater than*. One typical way to code a modern `if n>2 then k=5 else k=6;` back then was

```
K=6
IF (N.GT.2) K=5
```

However, in most cases one simply resorted to a `GOTO`-statement after an `IF`-statement, so our experience from the assembly-like Samos did come in very handy. As an aside, another 'strange' feature of Fortran was the fact that spaces were immaterial almost anywhere. Therefore writing `GO TO` was equivalent to writing `GOTO` or, for that matter, `G OT O`.

Our programs were developed in the same manner as Samos programs were — by flowcharting them. The remainder of the process was also similar. Programs were written (in pencil) on coding sheets and would be sent away for punching and *the* run. If errors were found, a second attempt was again allowed, after which a failed attempt meant that the teacher had to assign an appropriate mark.

In matric the language changed again, but in more ways than one. Our standard 8 and 9 teacher was not yet qualified to teach computer studies on matric level. We had two options: either go to the regional centre in Boksburg for matric or attend the English classes which were offered at Springs Boys High — about 10km from my own school. Boksburg was way too far for me, so once a week I cycled the 7km to my school in the mornings, the 10km to Springs Boys' High after school and the 8km back home in the evening — dressed in the fashion we inherited from the British: Trousers, long-sleeved shirt, tie and blazer.

The previous paragraph adds a number of elements to the story that has little to do with computer studies, but does add some historical context. Back then South Africa was a bilingual country with Afrikaans and English as the official languages. My home language was (and is) Afrikaans, which evolved from Dutch which my ancestors spoke when they came to South Africa twelve generations earlier (at various points in the 1600s). When I started my school career in 1969 it quickly became obvious that there was a huge divide between the Afrikaans and English communities with insults (and sometimes other objects) being flung between the two groups of pupils at bus stops and the few other points of contact. It was something I never quite understood. When I started cycling to school in the mid-1970s the problem disappeared from my view since shared bus stops were no longer an issue. But it was with some trepidation that I awaited my final year of education.

My classmates who came from afar could go to Boksburg at little additional cost. The only classmate from my own school could also make a plan to attend the Boksburg classes. About five of us joined the English group in Springs. However, the strife was something of the past and no language-related incidents occurred.

It should perhaps be pointed out that not too far from us incidents that were ostensibly language-related did occur. Just a short while before this — in 1976 — the Soweto uprisings occurred; the stated reason was the fact that some of the courses for black pupils were presented in Afrikaans. Whether that was the true reason for the riots may possibly be answered in due course. While the school children from the townships would probably never have phrased it like this, a better justification for

the riots was arguably the fact that the future for which they were being prepared differed markedly from the future for which I was being prepared. I had the opportunity — with some effort — to study computing. I never asked, but I knew there was no such option in the townships.

In 1980 the *other* must-do computer language was Cobol, which was our programming language to study in matric. The process we followed when programming remained unchanged, apart from the new opportunities introduced by the new language.

However, 1980 was special because I got three opportunities to actually interact with computers. My matric teacher, Martin de Klerk, was a wonderful man who went out of his way to create opportunities for those of us who were clearly interested in the subject. There were a couple of factories in town and I have no idea how many of them had computers. However, one — Ultra High Pressure Units which manufactured diamonds — had a computer and Mr De Klerk arranged a visit for me. At long last I could punch some cards, put them on the hopper, read them in and see my program produce results. This introduced a few challenges because I never before had to deal with JCL (Job Control Language) or whatever the control language was for that particular computer. So, I only managed a few runs during the day, but I was inside a computer room and, for the first time in my life, saw a real computer.

The second event was not related to computer studies as a subject. During the latter part of the year some of us (pupils in general, rather than those taking computer studies) were invited to a weeklong visit to the nearest university — the then Rand Afrikaans University in Johannesburg. During the week we had lectures from the computer science, mathematics and mathematical statistics departments. The computer science department used an internally developed language that they called *Staal* (*Studente taal*, Which translates to *students' language)* to teach us the basics of programming. Staal was a rather simple language based on PL/I (and translated by a preprocessor into PL/I before it was compiled by the PL/I compiler). Given our earlier languages, this language was wonderful: It supported while loops, if-then-else structures that could include arbitrary content in the then and else clauses and this was supported without the 'bloat' of Cobol. We again coded our programs on coding sheets, but stepwise refinement replaced the initial flowcharting. The coding sheets were punched, but we had access to punches and could correct our own programs. We still had to submit them to an operator to get run (but this happened within an hour or so). While the course was aimed at those without programming experience, the few of us who had some experience were given the freedom to explore — an opportunity which I grabbed. The next year I enrolled for my first degree (in computer science and mathematics) at the Rand Afrikaans University. An interesting fact is that the department of computer science there was established in 1970 — quite early in global terms.

The third event was also not related to computer studies per se. I managed to save enough money to buy an HP 33E programmable calculator. The calculator was programmed in RPN (Reverse Polish Notation) and had 49 memory 'steps' to store one's program. This taught me another skill: how to optimise one's programs to fit into that tiny space. I never succeeded to properly program it to play tic tac toe despite several attempts…

4 The Rest of the Curriculum

As noted earlier, the memories that remain of computer studies mostly relate to programming. I have, however, kept a file of the material we received as handouts. (Obviously suitable school-level textbooks were not available at the time.) Unfortunately I have not kept a record of what non-programming material was covered in which year. I therefore list them in the order in which they are in my file. Note that my handouts were in Afrikaans and the translated titles I provide below may not be the title that was in fact used on the English version of the handout. The ideas would be identical.

Not surprisingly, the first handout is entitled *Algorithms and flowcharts*. This consists of 27 A4 pages explaining what an algorithm is and how one represents it using a flowchart. It ends with a document reference IDR/DVN/76.01.13, which suggests that it had been created for a course starting in January 1976; it is very likely that 1976 would have been the first year the subject was offered in the province, meaning I was part of the third cohort.

The second document is the Samos 'manual', explaining how one programs in Samos. It consists of 41 A4 sheets and ends with the reference SJPK/ REKENAARWETENSKAP/DVN/75.09.08.

The third document (undated) is about the *social implications and future development of computers*. Such historical documents about the future often make for interesting reading. Unfortunately this brief (three page) document says more about the history of computing than its future. It does note that the use of electronic data banks (as they were called then) was gaining popularity. It notes that one of the concerns is the storage of confidential data, but expresses the optimistic assumption that we may assume that (freely translated) "suitable protection measures will be introduced." It also notes that the replacement of cash by cards is promising, and notes the "immense value of a permanent record" of an individual's transaction history. Finally, it notes how beneficial an appropriate combination of computing and television may be for education.

A recurring theme is the representation of data in the computer. Almost all handouts start with the reminder that data is stored in binary and then say something about the operation of ferrite core memory (which can be magnetised in one of two directions). However, from that point of departure a rather diverse set of notions is covered. *Numbers in the computer* is a 44 page handout dealing with binary representation, binary operations and conversions to and from binary. This is followed by a brief discussion of the use of any base. Then the bases 2, 4, 8 and 16 get special attention — in particular conversions between numbers represented in these bases. Next (binary) code decimal systems are introduced. The attention then returns to binary for a discussion of fixed and floating point numbers, and a discussion of complementary arithmetic in binary.

Another 19 page handout deals with logical variables. Logical operations, gates and truth tables form the essence of this document (which is also undated).

A thorough 38 page treatise on arithmetic in any base for integers, fixed point and floating point numbers carries the reference IDR:HS:JF:1979-03-01.

A handout about data structures and data storage continues the theme of data representation. Viewed from today's perspective it addresses an almost incoherent range of topics. The first item on the agenda is (logical) files mentioning that they consist of records, fields and characters. Then "expression trees" (in essence, parse trees for arithmetic expressions) are discussed. This is followed by a discussion of physical media such as disc packs (as typically used on mainframes) and floppy discs (which, it is stated, contain 77 tracks divided into 26 sectors. The discussion of tapes that follows continues the theme of physical media, but the emphasis is on how logical data blocks are recorded on tape. Some remarks about tape cartridges follow. Then the logical concepts of sequential and random access are introduced. Finally a few brief remarks about data banks follow. The final paragraph seems to predict the Web. Freely translated it reads as follows:

It is envisaged that telephone will in future serve as terminals and everyday television sets as video screens. Home occupants, students and pupils will then be able to telephonically connect to these databases and extract data from these data banks such that the results will be displayed on the television sets. They will, for example, be able to obtain information about aspects of the weather, information about plants, chemistry, astronomy, etc. by connecting to a specific data bank.

The final document in my file covers Cobol in (only) 24 pages, with three additional pages of handwritten notes (probably written by our teacher) completing our introduction to Cobol.

The fact that my file contains nothing about Fortran implies that my set of handouts is incomplete; however, I think it is complete enough to provide a fairly complete overview of the computer studies curriculum.

5 Conclusion

These were my experiences in the late 1970s in computer studies at school. My future was determined. I enrolled for a degree in computer science, eventually obtained a PhD in computer science and am currently a professor of computer science. I am still fascinated by these wonderful machines and the line between work and fun is often blurred.

In the 1990s, as a lecturer, I experienced classes full of students who grew up with (personal) computers and shared the love of the technology. For me work was fun and (hopefully) for many of them studying was fun. We were a community of people with a shared love who were given the opportunity to discuss it (and call it 'work' or 'studying'). I was just in the fortunate position to get to know something about these machines long before most had the opportunity.

There is also a counterpoint. I am appalled by the extent to which computers are often misused. Perhaps the new apparent simplicity to use them is incommensurate with the impact they yield over people's lives. However, that is a point for another essay.

As far as I know very few of those of us who took computer studies became computer scientists. What value it added to others' lives I don't know. On me the impact was profound.

References

1. McCartney, S.: ENIAC: The Triumphs and Tragedies of the World's First Computer. Walker & Co (1999)
2. Ceruzzi, P.E.: A history of modern computing. MIT (1998)
3. Backus, J.W., Beeber, R.J., Best, S., Goldberg, R., Haibt, L.M., Herrick, H.L., Nelson, R.A., Sayre, D., Sheridan, P.B., Stern, H., Ziller, I., Hughes, R.A., Nutt, R.: The Fortran automatic coding system. Papers presented at the Western Joint Computer Conference: Techniques for Reliability, IRE-AIEE-ACM 1957 (Western), February 26-28, pp. 188–198. ACM, New York (1957)
4. Sammet, J.: The early history of Cobol. ACM SIGPLAN Notices 13(8), 121–161 (1978)
5. Brooks, F.P.: The Mythical Man-Month: Essays on Software Engineering. Addison Wesley (1975)
6. Dijkstra, E.: Go to statement considered harmful. Communications of the ACM **11**(3) (1968) 147–148