Tony Brooker and the Atlas Compiler Compiler.

Simon Lavington and many others.
lavis@essex.ac.uk
February 2014, and as revised April 2016.

Introduction and acknowledgements.

Of all the software associated with the Ferranti Atlas computer, two systems were destined to be remembered as landmarks. One was the Supervisor, which was the first multi-tasking, multi-user operating system. The other was the Compiler Compiler which, when provided with a syntactic and semantic description of a programming language, would automatically generate a compiler for that language. Both the Supervisor and the Compiler Compiler (CC) were conceived and developed at the University of Manchester between about 1960 to 1964, in the spirit of making a high-performance computer more efficient and user-friendly. The Compiler Compiler was the idea of R A (Tony) Brooker.

This article begins by describing the pre-history of the CC and then goes on to discuss its implementation and use on Atlas. We try to capture the spirit of the time by including an account of the practical difficulties with which the researchers had to deal and the comings and goings of compiler team members. The wider CC picture is then briefly described. Finally there is a substantial technical Appendix, specially written by Tony Brooker, which describes the CC methodology and gives Tony's comments on contemporary research. Unless otherwise stated, all the quotes in the main text come from Tony Brooker – see [1] in the list of References given at the end (page 26 and following).

One of the aims of this article is to record personal views and the background information that never gets into the formal published papers. Many ex-Atlas people have contributed their memories. Besides Tony Brooker himself, particular thanks are due to John Clegg, Robin Kerr and Iain MacCallum. Alas Derrick Morris and Jeff Rohl, two other members of the original joint Ferranti/University compiler team at Manchester, are no longer with us. Several (non-Manchester) Atlas software implementers, particularly George Coulouris, Bob Hopgood, Ian Pyle and Dik Leatherdale, have also been most helpful in illuminating the wider CC picture.

1. Pre-history.

The Compiler Compiler was conceived by Tony Brooker, who had arrived at Manchester in October 1951. The Ferranti Mark I had been delivered to the University in February of that year and Tony's job, he recalls, "was to make it useable". When he arrived, the Mark I programming system was "extremely neat and very clever but pretty meaningless and very unfriendly". Programs had to be written in 'backwards binary' as 5-track teleprinter characters, with guidance from a manual written by Alan Turing that was full of minor errors.

An obvious improvement would have been to develop some sort of symbolic assembler for the Mark I, but Tony looked deeper. Whilst helping the scientific and engineering users with their problem-solving he became aware that two fundamental issues were causing difficulties even for experienced programmers: firstly, the need to scale numerical calculations involving real quantities; secondly, the need for memory-management between the small but fast primary store and the larger but slower drum backing store. The scaling issue could be solved by implementing a library of interpretive routines for floating-point arithmetic – so long as the invoking of these routines was made easy. The second issue could be eased by providing systems software which gave the end-user the impression of a *one-level store* – again, so long as the user was treated gently.

Tony's answer was Mark I Autocode, which allowed arithmetic expressions to be written down directly using named integer and real variables without the user being aware of where those variables were stored. The *one-level store* could be implemented in a balanced way on the Mark I because the time to transfer a block of information from the drum was approximately the same as the time taken by a software-implemented floating-point operation running in the fast store.

For the Mark I users of the day, floating-point calculations dominated their problem-solving. Tony's autocode pre-allocated some alphabetic symbols {r, s, t, … z} to real variables and other symbols {i, j, k, ..} to integers. A library of standard functions and organisational routines for printing, etc., was permanently stored on the Mark I's drum. When running Autocode programs, 128 tracks on the drum were reserved for instructions and 128 tracks for variables – which was adequate for all but the most ambitious problem-solving at the time [ref. 2]. The autocode system was released in March 1954. Whilst Mark I Autocode is now recognised as one of the earliest high-level languages, its benefits to end-users at the time were perhaps more organisational than linguistic.

The next Manchester computer, the Ferranti Mercury, had floating-point hardware but the memory-management problems were still present. By 1958 Tony had developed Mercury Autocode, which extended the linguistic richness of Mark I Autocode whilst failing to provide a general solution to the problem of Mercury's two levels of storage (core and drum). Tony later recalled that "In a way I missed out... I introduced the idea of chapters where the programmer would break his huge programme up into chapters, and usually the chapter was sufficient …[Only one chapter could be in the main memory at a time and it was the programmer's responsibility to transfer control between the chapters]. As regards numbers... I've forgotten offhand how I solved the problem of numbers. Well I didn't really solve it".

By this time (1958) the MUSE project, subsequently called Atlas, was under way at Manchester University and Tony Brooker was placed in charge of software provision. The engineers were in the process of solving the problem of the Atlas one-level store via hardware page-address registers and the drum learning program – though Tony had certainly contributed to the discussions – see [ref. 3]. Tony later recalled that "the only problem left for me to solve" was what high-level language(s) should be implemented for Atlas. "I suppose I'll have to come up with some, some kind of autocode for the Atlas", he said. Fortran was of course a contender, though Tony added somewhat dismissively that he "wouldn't have called Fortran a high level language".

At this time there were proposals in America and Europe for a universal programming language, initially known as *International Algorithmic Language, IAL.* Detailed specifications were put forward by the ACM and also by GAMM (Gesellschaft für Angewandte Mathematik und Mechanik). It was decided to hold a joint meeting to combine the two proposals. The meeting took place from 27th May to 2nd June 1958 in Zurich, as a result of which ALGOL 58 was defined. Historians now view the Zurich meeting as seminal. Tony recalls: "I never received any invitation to the conference at Zurich and I had no contact whatever with the authors of the 1958 report…. By that time I was resigned to ploughing my own furrow. Although I had attended BCS conferences I was really quite naive about overseas activities".

When he read the Zurich report, Tony thought that ALGOL 58 was "an incredibly inefficient language in some ways" but he nevertheless considered devising a strict subset of ALGOL 58 for Atlas. Robin Kerr, who implemented an Atlas compiler for Algol 60 several years later, thinks [ref. 4] that Tony was initially put off by the difficulty of writing a one-pass compiler for ALGOL 58. Tony himself says his trouble lay in "finding a strict subset [of Algol 58] which was at the same time a useful programming tool". In the end, of course, Tony designed and implemented Atlas Autocode which was "perfectly usable and it had recursion …the only thing wrong with the Atlas Autocode was that it wasn't ALGOL 60".

Back in 1958, Tony put off the decision about choice of languages for Atlas. "I thought, well, before jumping in, why don't I think about a system for writing *any* kind of compiler, so that whatever turns out to be popular, I can do it". Tony started by considering the question of how one could describe autocodes in general. Thus was born the idea of the Compiler Compiler.

Tony and his former research student Derrick Morris began to publish relevant CC research from 1960 onwards – see [refs. 5 to 9] – long before an operational CC system had been implemented. Their first paper [ref. 5] used the phrase "an autocode to write autocodes". Tony's first UK publication with the words *Compiler Compiler* in the title did not appear until 1963 [ref. 11]. Shortly before this Saul Rosen, an enthusiast at Purdue University, "decided to rewrite my account in terms that Americans could understand". Rosen's paper, which drew attention to what Rosen called the "elegance and generality" of the CC, appeared in a journal that was much more widely-read [ref. 11]. Henceforth the by-line *Brooker and Morris* became better-known.

The fact that anything which describes a high-level language is, of course, itself a high-level language was what first attracted interest from the theoreticians. Interest from software implementers was slower to materialise. The Compiler-Compiler as a tool for writing compilers was intrinsically inefficient because it worked top-down and was very recursive – (see also the description given in the Appendix). "You've got to keep invoking all this heavy apparatus, which is very time-consuming….to make the CC practically useful we had to put in an artificial addition which says, first check that it isn't a simple expression, when it can be done in a blinding flash". (See also Tony's retrospective comments in the Appendix).

2. Other early research.

The need to provide software tools for the production of compilers was not, of course, confined to Manchester. Rosen's paper [ref 11] briefly mentions the PSYCO project at Princeton and the syntax-directed compiler of E T Irons, also at Princeton, both projects

dating from January 1961. Another early innovator in this area (not mentioned by Rosen) was Alick Glennie, who worked for the Atomic Weapons Research Establishment at Aldermaston. In a July 1960 report [ref. 12], written under an Office of Naval Research contract whilst Alick was on sabbatical at Carnegie Institute of Technology, Alick proposed a set of descriptive primitives which he applied to the syntax of a source language. These primitives, which formed what he called the *Syntax Machine,* reflected the actual computational primitives of a target computer. The *Syntax Machine* was simulated on an IBM 650, with code for an IBM 709 as the target.

Glennie's report was available to US Government-funded Research projects and is referenced as a major input to the META Series of Translator Writing Systems (referred to below) which were developed in the period 1962 to 1968. Glennie's paper was cleared for public release in September 1977. It remained unknown to Tony Brooker.

It is not surprising that Tony Brooker and Alick Glennie were thinking along somewhat similar lines, for each had implemented an autocode for the Ferranti Mark I computer a few years earlier. Tony has recently remarked that "everything Alick did at that time was very hush-hush". Bob Hopgood, who was familiar with Alick's Fortran compiler work at Aldermaston in the early 1960s, has said [ref. 13]: "I remember many conversations where Alick emphasised the need to have a fast compilation system even while attempting to generate good code… The system used for generating the S1 to S3 Fortran compilers [for IBM computers at Aldermaston] had quite a few resemblances to the *Syntax Machine* system…. I think the CC was more elegant than Alick's work while Alick's was more efficient and could in principle handle a greater range of languages". The Aldermaston work is referred to again in section 5.3.

Other early compiler-generation systems included META II, described in 1964 [ref. 14]. In Bob Hopgood's view [ref. 13], this was much simpler than the CC, but rather closer in concept to the CC than Glennie's *Syntax Machine.*

Returning to the progress of Tony Brooker at Manchester, implementations of compilers for the Ferranti Atlas computer will shortly be described. Firstly, however, it is helpful to set the scene by reviewed the initial engineering difficulties in getting Atlas fully-operational and by introducing some of the people who joined Tony Brooker on the software side.

3. Compiler-writing activity at Manchester: facilities and people.

Back in 1958 Tom Kilburn's team of engineers was still developing the high-speed arithmetic circuits and storage devices and so no usable Atlas hardware yet existed. A Ferranti Mercury had been installed by the end of 1957 in the Electrical Engineering Department's building at Manchester University and this machine was used for preliminary Atlas software development whilst prototype Atlas hardware was assembled in an adjacent room. By pushing forward the frontiers of digital technology, Atlas was designed to be about two orders of magnitude faster than Mercury. Inevitably, there were implementation delays for such an ambitious – some said foolhardy – project.

By the end of 1960 a Pilot Model of Atlas had been assembled that successfully demonstrated the ALU, B-store, the 1K word Working Store and the Fixed Store – but little else. This Pilot Model was moved to Ferranti's West Gorton factory in January 1961, so that its space at the University could be refurbished and fitted with air conditioning and a

false floor. Production Atlas hardware started to fill this refurbished room in the summer of that year. During the first half of 1962 both the Supervisor team and the Compiler team had to manage with only the 1K-word Working Store available. It was not until the autumn of 1962 that sufficient hardware units had arrived from West Gorton to constitute a reasonable platform for realistic software development. Even then, there were times when both the Supervisor team and the Compiler team had to develop their code within relatively constrained amounts of operational storage.

Incredible as it now seems (though typical of the age), the Atlas Compiler group never numbered more than six full-time programmers and the Supervisor group never more than seven. As 1962 drew to a close, pressure within these Atlas development teams mounted. By prior arrangement with the University Grants Committee (UGC), Manchester University's Mercury computer was due to be moved to Sheffield University in January 1963 – at which point Atlas was intended to support both Manchester University's Computing Service and Ferranti's Computing Service (the latter catering for industrial users). In early 1963, only a limited user service was provided. Not only was the Supervisor still being refined but the original Ferranti drums were proving unreliable; these had to be replaced by Bryant drums imported from America. In the event, it was not until January 1964 that the full Atlas Supervisor was declared operational and a full user service was running.

Whilst Tony Brooker led the CC team and determined the overall strategy for Atlas compiler implementations, he was assisted at various times by various people – both University-based and Ferranti-based. John Clegg [ref. 15] remembered Tony as "the senior father of compiler software development, but he also lectured on numerical analysis, even extending invitations to the Ferranti people to attend. This was typical of Tony – I've never forgotten one of his sayings 'I don't mind whose ideas I use as long as they are good ones'. This was not plagiarism, just natural team-building".

Table 1 *(next page)* gives a summary of the comings and goings of the original Atlas joint compiler team members (in alphabetical order).

Others who helped in one way or another with Atlas compiler work included (in alphabetical order):

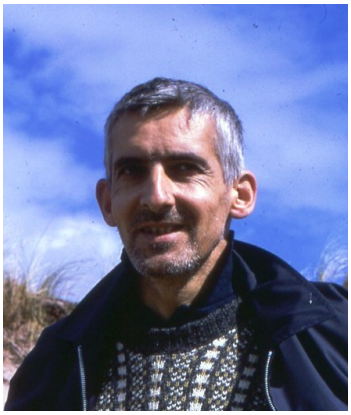
- Stan Clark (a research student who worked with Jeff Rohl on Atlas Autocode from about 1964 to 1966);
- Vic Forrington (who worked for Tony Brooker as a numerical analyst within the Computing Machine Laboratory from 1961 to 1964);
- Ted Gaitskell (who worked for Manchester University Computing Service from 1963 - 65 before joining the Ferranti compiler group which he left in 1967);
- Jan Olejnikzak (who joined Ferranti in about 1964);
- Charles Outred (who joined Ferranti in about 1964, leaving to join the Department of Computer Science in 1966 and then moving to Burroughs in 1971);
- Graham Sencicle (who joined Ferranti in 1965).

(Table 1 on next page)

Name of team member	***When and why he became involved***	***When and why he left the team; subsequent career.***
Tony Brooker	Right from the start (ie from well before 1959, with pioneering work on autocodes that led to the Compiler Compiler).	**Autumn 1967:** departs to become the founding Professor of Computer Science at the University of Essex. Retired from the University of Essex in 1988.
John Clegg	**Oct. 1961:** joins Ferranti Ltd. as a fresh Mathematics graduate and was assigned to Robin Kerr's compiler team.	**July 1966**: departs to join ICI; later returned to the University of Manchester Regional Computing Centre. Left in 1990 to become head of Information Systems at Granada Television. Left Granada to start his own consultancy in 1993. Retired in 2003.
Robin Kerr	**1956 – 59**: Ph.D. student under Tony Brooker, working on (a) the automatic evaluation of integrals, (b) development of the library system for Mercury Autocode. **1959 – 64:** joined Ferranti Ltd. and was initially assigned to Tony's compiler team.	**July 1964:** departs to work for Control Data in Australia, and from thence to the USA to work successively for GE's Corporate Research Labs and Schlumberger. Retired in 1995 and now living in the USA.
Iain MacCallum	**Oct. 1961:** enrolled on the Diploma in Numerical Analysis course under Tony Brooker. By December, had begun working on the Compiler Compiler, transferring to an M.Sc by thesis supervised by Derrick Morris. **Oct. 1962:** transferred to Ferranti Ltd. to continue the CC work and apply it to Algol, under Robin Kerr.	**Jan. 1964:** departs to join the CEGB in Bramhall. Joined Tony Brooker at the University of Essex in 1967, retiring as a Reader in Computer Science in 2002.
Derrick Morris	**1955 – 1959: P**h.D. student jointly supervised by Tony Brooker, working on Civil Engineering calculations. **1959:** employed as a Research Assistant in the Computing Machine Lab., working with Tony Brooker on the Compiler Compiler.	Remained at Manchester University in the Atlas team. In 1964 he joined the staff of the newly-formed Department of Computer Science; worked on the MU5 project and its operating system MUSS. Derrick was promoted to Professor of Computer Programming in 1972. He transferred to a chair at UMIST in 1985 and died in post on 3rd January 1997.
Jeff Rohl	**1960 – 1963:** Ph.D. student under Tony Brooker. Title of thesis: *Atlas Autocode and the compiler*. **From 1963:** employed as a Research Assistant in the Computing Machine Lab., working with Tony Brooker on the Compiler Compiler and on Atlas Autocode and its extensions.	Remained in the Atlas team, in due course becoming a lecturer in the newly-formed Department of Computer Science in 1964. Worked on the MU5 project. Transferred to UMIST in 1972 as Professor of Computation. Moved back to Australia in 1976 to become the Foundation Professor of Computer Science at the University of Western Australia, Perth. Retired in 1999. Died on 30th December 2003.

Table 1. Names of the principal people who worked on the Compiler Compiler or on the implementation of compilers for the Manchester Atlas.

Here *(on the next page)* is how the team members in Table 1 looked at about the time of working on Atlas (or as near to this as surviving photos allow). In Tony's case, the photo shows him in about 1968, after leaving Manchester. A recent photo of Tony, taken in 2013, is shown on page 15.



Tony Brooker

Derrick Morris

Robin Kerr

Jeff Rohl

Iain MacCallum

John Clegg

It can be deduced from Table 1 that the number of active team members fluctuated as other duties intervened. For example Robin Kerr [ref. 16] recalls that, during 1960 – 61, he spent time "doing odd jobs such as technical sales support [for Atlas], providing programming support for the engineers on the Pilot Machine, writing the Intermediate Input Assembler and getting the automated back wiring project started. In 1962 as well as working on the Algol compiler I was once again involved as needed with the hardware engineers and technical sales support. Being the senior Ferranti [compiler] person on site I seemed to get stuck with a lot of administration chores and meetings both in Manchester and London".

Tony himself had other academic commitments from time to time. To the surprise of some of his colleagues, Tony accepted an invitation from Michael de V. Roberts to spend a year at IBM's prestigious Yorktown Heights Laboratory in America. He took leave-of-absence from October 1962 to September 1963 – a critical period in the Atlas project when (as he later admitted) it was "all hands on deck".

Returning to a chronological account of compiler development, it is convenient to describe the (parallel) activities of the various Atlas language sub-groups at Manchester. Contemporary to this, there were several semi-independent language development projects at the other Atlas sites, notably at London, Chilton (adjacent to the UKAEA's Harwell Laboratory), Cambridge (for Titan) and AWRE Aldermaston. There was naturally some interaction between the sites, as mentioned later. Firstly, though, we concentrate on the joint University/Ferranti endeavours at Manchester.

4. Timeline of implementations at Manchester.

4.1. The Compiler Compiler.

From 1959 onwards, Tony Brooker was joined firstly by Derrick Morris, then by Robin Kerr and then Jeff Rohl. Tony recalls that "Derrick was involved because I was his Ph.D. supervisor, jointly I supposed with someone from the Civil Engineering Department (but if so I never met him!). I recall his Head of Department asking me if his thesis was going to be OK and if the external examiner was appropriate. In fact it was David Wheeler [of Cambridge University]. After that Derrick seemed the obvious person to recruit on the project". Derrick obtained his Ph.D. in 1959; the title of his thesis was *Automatic solution of boundary value problems.* Derrick was the co-author on all of Tony's early Compiler Compiler publications.

The theory and detailed structure of the Compiler Compiler were laid down soon after Derrick joined in 1959. Implementation of the code received a boost when, at the end of 1961, Iain MacCallum became involved. Initial testing of the CC began in March 1962 [ref. 17] and continued for about a month while only the 1K word Working Store was available for Atlas programs. The first version of the CC became operational in June 1962, even though it was not until October that the first 8K words of Atlas main core store were working. The first languages to which the CC was applied in earnest were Mercury Autocode and Extended Mercury Autocode (see below).

From 1963 various minor improvements were made to the Compiler Compiler, as operational experience was accumulated. Naturally the main objectives for each source language were to reduce compile-times and to increase the efficiency of the run-time code. The penultimate CC modification, a new feature required for the second release of an Algol compiler, was completed by Derrick Morris early in 1964. A later modification (strictly, a later version) was made to the CC in about 1965 so that it could generate code for the smaller Atlas 2 machines – see section 5.3 below.

Tony recalls that "As for myself I can't recall using [the CC] for actually implementing a compiler after I returned from IBM in October 1963. Instead I wrote a conventional compiler for Atlas Autocode in the Autumn term of '63 as a back up for Jeff Rohl's CC version in case for whatever reason it proved too slow. In fact both versions were used but I was able to tweak mine to provide statistics about the relative usage of different statements".

4.2. Mercury Autocode.

Users of the Ferranti Mercury, both in academia and industry, were looking forward to transferring their Mercury Autocode programs more or less 'unaltered' to Atlas. Some of these users, for example Ann Moffatt from Kodak Ltd. and Owen Mills and Rowland Benson from Manchester University, were skilled programmers with high-priority jobs. A few such privileged customers had been given limited access to Atlas at Manchester from late 1962 onwards. There was thus an immediate demand for a usable Mercury Autocode (MA) system.

Robin Kerr recalls that "My first project [after obtaining my Ph.D. in 1959] was the Mercury Autocode extensions (EMA). There was no question from either the University or Ferranti that a Mercury Autocode compiler was needed for Atlas. The questions were how should it be implemented and should features be added to exploit the Atlas one level store and

index registers. There were some concerns by Ferranti sales that relying on the unproven CC was a risk and adding features just added to the risk. Tony prevailed and we went ahead with both the CC and the extensions." In the autumn of 1961 John Clegg joined Ferranti Ltd. and started work for Robin Kerr on the implementation of Extended Mercury Autocode for Atlas.

50 years later, at the time of writing this paper, the precise timescales of MA/EMA development are not quite clear. It is thought that work on both systems was proceeding in parallel, with Tony Brooker and Derrick Morris pressing ahead with the standard Mercury Autocode for the academic community whilst Robin Kerr and John Clegg worked on a full-strength production version of Extended Mercury Autocode.

We do have one documented insight, from Iain MacCallum's M.Sc. thesis: "On 31st December 1962 when only 16K words of store could be used [presumably no drums available on that particular day?], the first simple program was compiled and obeyed. Less than a week later, most of Mercury Autocode with the exception of matrix, double length and complex facilities were tested and working, having used less than 20 hours useful machine time" [ref. 17]. Robin Kerr has recently commented: "I am guessing that that the CC MA programs used by Iain at the end of 1962 were obtained from our EMA program. My guess is based on three things: first, I had complete confidence that the CC worked because it had been used to implement MA; second the short time John and I took to get EMA working and third Iain mentions double length, complex and matrix operations which I cannot remember being in Mercury Autocode".

It is believed that a production version of Extended Mercury Autocode was released by Ferranti Ltd. in about March 1963. EMA did not impinge much on the University Computing Service but Ferranti actively promoted it as a product. By 1965 [ref. 18] EMA was described as: "possibly in wider use in European scientific computing centres and industrial research establishments than many other programming languages…. Compilers are already available for Mercury, Orion, Atlas I and I.C.T. 1101/1301 computers and will be made available for the I.C.T. 1900 series". Meanwhile, although Mercury Autocode had been in use within the academic community from early 1963, by March of that year Atlas users at Manchester were being encouraged to switch to Atlas Autocode.

4.3. Atlas Autocode.

This language was Tony Brooker's response to Algol. Atlas Autocode, or AA, was a block-structured language of considerable power and efficiency. At first sight, AA and Algol appeared similar in scope and form. Amongst the subtler differences, AA used *call-by-reference* whereas Algol used *call-by-name.* Atlas Autocode is fully described in [refs. 19 and 20].

From mid- 1962 to mid- 1963 Jeff Rohl implemented AA using the Compiler Compiler. During this time Jeff remained in touch with Tony (his Ph.D. supervisor) at IBM. Tony remembers that "In fact I discovered a flaw with the Compiler-Compiler when I was at IBM, and I remember writing letters …". John Clegg has said that Jeff Rohl "coupled great technical skills with a good sense of humour. I recall that Robin could occasionally be somewhat avuncular, which led Jeff to refer to him as 'Uncle Robin' in their banter exchanges!"

The first internal course on programming in Atlas Autocode was held for staff and research students at Manchester in September 1962, shortly before Tony departed for America – though at that time no AA computing service existed. The first bound copies of the AA programming manual were issued in February 1963, the authors being R A Brooker and J S Rohl, after which regular courses were offered. AA remained the most popular high-level language for the Manchester Atlas and was also well-used at Edinburgh University where, between 1966-1969, AA was extended to form the IMP system programming language. In the wider word of the 1970s, AA faded in popularity as the popularity of Algol increased (amongst academics, at least). Edinburgh IMP remained in use until the 1990s.

The main AA compiler was developed using the Compiler Compiler. Another version called AB, was developed by Tony Brooker – (see the Appendix). The data-preparation device of choice at Manchester was the Friden Flexowriter, which produced punched paper tape. Delimiters in AA were underlined thus: begin, integer, end, etc., which required Flexowriter users to punch lots of 'backspace' and 'underscore' characters. The AB compiler allowed upper-case letters to be used in delimiters.

In the mid-1960s an extension of AB called ABC was developed by Jeff Rohl and colleagues – but without using the Compiler Compiler. (This ABC should not be confused with the language of the same name produced in 1987 at CWI, Netherlands). The Manchester ABC supported (amongst other things) "complex, double, multi-length precision, CC-style syntax analysis, an early form of *struct* and much besides" [Ref. 21].

4.4. Algol 60.

Robin Kerr took the initiative with Atlas Algol, which was probably the most challenging application for the Compiler Compiler. Robin remembers that: "Tony, Derrick and I had been working on and off on Atlas Algol since 1960 and we really did not get it right until the end of 1963. I spent most of 1960 and 61 working the Algol issues". By about September 1963 visitors from Chilton [ref. 22] reported that there were apparently two Algol compilers on the stocks: Robin's and Derrick's, though the latter was described as "mainly an exercise" and it did not turn into a product.

In October 1962 Iain MacCallum had joined Robin and the implementation of Ferranti's Atlas Algol gathered pace. By the end of December 1963 the first version of the compiler was in use and the design of the second version was complete. On 19th March 1964 ICT, who had taken over Ferranti's mainframe computer interests in the previous September, announced to its customers that "the first version of the Atlas Algol compiler is working and available for use by the computing service" [ref. 23].

John Clegg recalls that "The second version addressed the problem in Version 1 of the huge analysis record (which restricted the size of the Algol programs which we were able to handle) by performing an initial 'shallow analysis' of the entire program and thereafter processing each statement one by one". This second version required a new CC feature which was ready early in 1964. For this, Derrick Morris changed the CC to two passes. The first pass extracted all the definitions by name, type and scope and the second pass used these definitions to generate the code for the arithmetical, logical and control statements that were not analysed on the first pass. For a technical description of the CC change, see the description of [BASIC STATEMENT] in [ref.23, page 346]. The second Atlas Algol compiler was released in May of that year.

4.5. Fortran 2 and ASA Fortran.

It is believed that FORTRAN 2 was essentially taken as the first exercise in using the CC. Although no production-strength compiler was released into service, the exercise contained enough realism for comparative statistics (size of compiler, efficiency of object code, etc.) to be presented in [ref. 24]. The implementation of a compiler for Fortran IV, the most up-to-date version of the language in the early 1960s, was at first given low priority at Manchester. This reflected not only the language preferences of local Manchester users but also the intense Fortran activity already planned for the London and Chilton Atlas sites – see sections 5.1 and 5.2 below.

It was the Ferranti side of the Atlas software team that eventually wrote the compiler for ASA Fortran (the version of Fortran IV that had been approved by the American Standards Association in 1966). They began in mid-1964, by which time Ferranti's mainframe computer interests had been taken over by ICT. John Clegg led the ASA Fortran development after Robin Kerr had left to work in the USA.

4.6. Summary of the language development at Manchester for Atlas.

John Clegg has recently summarised the Atlas compiler activity at Manchester thus: "My lasting memory of this period is the remarkable integration which was achieved between the University and Ferranti personnel who worked on the Atlas project. This was not unique to the compiler team, but applied equally to the Supervisor team and to the hardware people. It probably stemmed from the top level agreements between Tom Kilburn and Peter Hall [the Manager of Ferranti's Computer Department]. In our case Ferranti provided resources, but did not interfere at higher management level with the way those resources were used on a day-to-day basis. That said, the two compiler-writing teams were distinct, at least from the date when I joined [October 1961], although Robin had been Tony's research student before joining Ferranti. The distinction lay in the different compiler products which the two teams created. In summary, I believe the respective end-user products were:

University Team:	CC, Mercury Autocode, Atlas Autocode
Ferranti Team:	Extended Mercury Autocode, Algol 60, ASA Fortran.

As far as other Atlas sites were concerned, the Chilton (Harwell) Atlas team was the front-runner in providing a good-quality Fortran programming environment. This is part of the wider picture, which is considered in Section 5 below.

The CC was used to generate compilers at Manchester for other languages. Examples are Elliott Autocode Mark III [ref. 24] and SNAP, a natural language system [ref. 24]. It was also used at Manchester and elsewhere to produce special applications-specific systems. For example, using the *Compiler Special* facilities in Atlas Autocode, a speech-processing package SPP1 was given the same status under the Supervisor as any normal compiler [ref. 26]. Another example was the special compiler *Log,* written by Sam Moore at Manchester and used daily to process a paper tape containing the statistics of every Computing Service job run on Atlas in the last 24 hours [ref. 24].

By 1967, when the Manchester Atlas had matured and was half way through its working life, there were 25 standard 'compilers' occupying a total of approximately 250K words of code [ref. 27]. Most of these 'compilers' had been written using the Compiler Compiler [ref. 28]. No distinction was made between compilers, assemblers and loaders, since the system was normally run in 'compile and go' mode. The standard compilers were held on

a single Supervisor magnetic tape, along with the code for the Supervisor itself and other logging, testing and diagnostic systems. Whenever the Supervisor was restarted, the main part of it was transferred from the Supervisor tape to the primary (one-level) store. (Of course, key Supervisor routines such as those handling the higher-priority peripheral interrupts, were held permanently in the Fixed Store). Compilers were transferred as required by user programs. Once a particular job's compile-phase had finished the compiler was normally 'lost' from primary store. However, a batch processing facility existed whereby successive compilations could be initiated. Great flexibility was offered by the Atlas *Call Compiler* and *Define Compiler* extracodes [ref. 28].

5. Compiler Compiler usage elsewhere.

Activities related to the Compiler Compiler were also pursued at other Atlas 1 and Atlas 2 sites, as is now briefly summarised.

5.1. London Atlas.

After Manchester, the University of London Atlas Computer Service (ULACS) and its neighbour establishment the University of London Institute for Computer Science (ICS) were the most enthusiastic users of the Compiler Compiler. Indeed, David Hendry (ULACS) eventually devised BCL, an unrelated language which had some concepts in common with the CC. Unfortunately BCL was not judged a success. Much more successful were the following three London developments using the CC.

CPL. This was a combined Cambridge-London implementation of Christopher Strachey's original proposal for a new programming language and compiler for the Titan (Atlas 2) computer at Cambridge University. At first CPL stood for *Cambridge Programming Language*, then later *Combined Programming Language* when the London-Cambridge collaboration was set up. CPL was intended as a powerful Algol-like language that catered for an extremely wide spectrum of applications – ranging from low-level programming for industrial process-control to commercial business data processing. George Coulouris of the Institute for Computer Science began work on a compiler for CPL in mid-1963 and a working version was released in the autumn of 1964 [29].

Atlas Commercial Language (ACL).. It was decided not to implement COBOL itself on Atlas, because of predicted inefficiencies with character-manipulation. Instead, a COBOL-like language called ACL was designed in which all numerical data was held in binary fields. Conversion (to/from decimal, sterling, etc.) only took place when reading cards or printing lines. David Hendry of ULACS implemented ACL.

Fortran V. David Hendry was also responsible for implementing Fortran V, a superset of the then-available Fortran IV. It embodied all the various enhancements to Fortran IV which had been added by various (independent) developers over the years. Interestingly, Fortran V included the facility for dynamically-allocated arrays.

5.2. Chilton Atlas.

Bearing in mind the UK Atomic Energy Authority's imperative for using Fortran on IBM machines for nuclear physics problems, the Chilton team (closely related to UKAEA Harwell) prepared for the arrival of Atlas by writing a cross-compiler for the IBM 7090. This produced cards which could be loaded onto Atlas. In due course a special Fortran 2 compiler was then written for Atlas. The Chilton group knew of the Compiler Compiler but

decided not to use it for two main reasons: (a) FORTRAN 2 was not at that time defined formally, so there was no strict language definition to feed into the CC; (b) the CC (and the Atlas Supervisor) was essentially 'compile-and-go', making no provision for the separate compilation of sub-programs (written either in Fortran or in machine code). The Chilton Fortran compiler [ref. 30] ran within the Hartran system, which allowed for sub-programs. Hartran eased the transition of computing work from IBM computers to the new Atlas. Overall statistics reveal that about 70% of jobs run on the Chilton Atlas used Fortran.

The Chilton group decided not to write its own Algol compiler but Bob Hopgood wrote a pre-processor for the Manchester Algol compiler [31]. The pre-processor handled the various Algol dialects and data-preparation media being used in UK academia at that time, and supported differing hardware representations of the Algol reference language. This contrasted with the Manchester Algol compiler, which normally expected 7-track paper tape produced on a Flexowriter. Handling different dialects and representations was important for the Chilton site, whose users came from a wide background.

5.3. Atlas 2 at Aldermaston.

The Atlas 1 Compiler Compiler was modified and ported to Atlas 2, which had no automatic paging mechanism and no Fixed Store. John Clegg, who left ICT in July 1966, says: "The background was that the instruction to port the compiler came from my (then) immediate boss, Peter Hunt, based at ICT Bracknell. Peter gave Derrick a contract to make the necessary changes to the CC. I think these were mainly the replacement of extracodes, not available on Atlas 2, by sequences of standard instructions, and we had to make similar changes to the object code planted by the compiler. It was planned that three of us, Derrick, myself and I think Jan Olejniczak, would make an initial visit [to the Atlas 2 at AWRE Aldermaston) of one week to install the amended compiler. In the event on Day 2 of the visit, I was able to telephone an incredulous Peter to tell him 'Job done'…"

It is not clear what the modified CC was used for at Aldermaston. It seems unlikely that Alick Glennie and the AWRE programmers would have used it, since their main concern was the production of the S1 to S3 series of Fortran 2 compilers using Glennie's own insights – (see section 2 above).

It is possible that ICT was contracted to provide Fortran IV and/or Algol compilers via CC for the Aldermaston Atlas 2. However, a December 1966 brochure [ref. 32] puts the emphasis elsewhere: "Two compilers have been written by AWRE staff, with assistance from the manufacturers. In fact AWRE has concentrated its small team of software specialists on this field. The very large internal storage of both STRETCH and Atlas 2 has made possible a new sort of compiler, one which is both fast and produces efficient programs in machine language. The S2 compiler for STRETCH and the S3 compiler for Atlas 2 are the result: they compile from the same Fortran source language and hence either STRETCH or Atlas can be used for a program written in this source language….. In addition to the S2 and S3 compilers for Fortran 2, a Fortran IV compiler written by IBM is available on STRETCH. Compilers for other languages may be added as they become available from outside sources". Bob Hopgood has added that the IBM Fortran IV compiler for STRETCH "was notoriously slow and was abandoned by Aldermaston". It seems likely that Fortran 2 was the standard at AWRE for several more years.

5.4. Titan and Atlas 2 at Cambridge.

Titan, the prototype for the production Atlas 2, was a collaboration between Ferranti Ltd. and the Cambridge University Computer Laboratory. Peter Woodsford [ref. 33] remembers that: "The Brooker/Morris Compiler Compiler was used in the CAD Group of the Cambridge University Computer Laboratory to create and support the Systems Assembly Language (SAL)…. I think SAL [ref. 34] may have been the only use of CC on the Titan". Charles Lang [ref. 33] adds: "Three of us in the CAD Group contributed to the SAL/CC effort. I designed the language, Heather Brown made the first implementation and Phil Cross made the CC work on Titan and hugely improved its efficiency. I recall being dazzled at the time when - if my memory serves me right - he speeded up the CC 4x working from an octal version of the code that came from Manchester!"

A special Computer-Aided Design Centre was set up by the Ministry of Technology at Madingley Road, Cambridge, in 1967 [ref. 35]. The CAD Centre was equipped with the last production Atlas 2 computer, together with the novel multi-access operating system that had been developed for Titan at the Cambridge University Computer Laboratory. There was obvious synergy between the aims of the new CAD Centre and the expertise of the University.

Peter Woodsford continues: "One of the main projects for which SAL was successfully used was the implementation of GINO (specifically GINO-2 and GINO-3)…. In the Computer Laboratory the implementation was on the Titan (prototype Atlas 2). The Compiler/Compiler, SAL and GINO all went to the CAD Centre, along with the Titan multi-access system…. As you know GINO-F was then implemented by CAD Centre, but I don't know which Fortran compiler they used or how it was generated. I do recall that there was a very strong emphasis on the use of 'standard' Fortran as the whole rationale was to make GINO as ubiquitous as possible". By 1974 GINO-F was using ANSI standard Fortran – which was what ASA Fortran had by then become [ref.36]. The GINO-F 3D Graphics Software Package is still very much in use today.

Alan Clark has said [ref. 33] "I joined CADC in September 1972, and it was definitely standard Fortran IV from day one". It is believed that both Titan and the CAD Centre standardised on a Cambridge-produced Fortran compiler known as T3, which was compatible with Fortran IV. Its production had no connection with the CC. Mike Williamson [ref. 32] is sure that the CC itself was never used at the CAD Centre.

6. Recreating the Compiler Compiler.

In 2013-14 Dik Leatherdale and Iain MacCallum collaborated in an exciting project to recreate a working demonstration of the CC. Their starting-point was a folder of original program listings and lineprinter output that had survived from December 1963 [ref. 37]. This is supported by Iain's M.Sc. thesis [ref. 17] and some archival material from the University of Manchester which includes copies of a CC index and flow diagrams of the major CC routines painstakingly produced by Jeff Rohl.

Iain and Dik carefully scanned the original bootstrap listings, consisting of 30 pages of octal input, 107 pages of what looks like Atlas Assembler Code (but is more than that), and 75 pages of program written in the Compiler Compiler language itself. On 21st November 2013 the entire script was input to Dik's Atlas Emulator [ref. 38] for the first time, starting with the 30 pages of octal and finishing with the directive "DEFINE

COMPILER". The output from the emulator was character for character (except for line numbers) the same as that of a run that Iain had recorded on 23 December 1963.

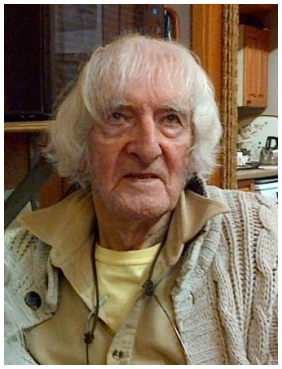
Dik and Iain intend to define a simple source language and a sample object program to leave on the web for historically minded computer scientists to play with. They also intend to write up their project for publication.

7. Aftermath and summary.

In a paper published in 1999 [ref. 39], Jeff Rohl put his Atlas compiler-writing experience into stark perspective. Quoting from his Abstract: "Atlas was a wonderful machine, with over 1,000 instructions, 128 registers in an architecture that allowed double modification, and an address space of 24 bits—at a time when stores were measured in kilobytes. Unfortunately, compiler writers were unable to use these facilities to the full—or anywhere near it. Almost all of the bits of compiled code were zeroes".

By 1966 Tom Kilburn's team at Manchester University had started to work on the design of a successor to Atlas. A proposal was published in 1968 for a supercomputer to be called MU5 [ref. 41]. Programmers played a prominent part in ensuring that MU5's instruction set could indeed be fully utilized. In particular Derrick Morris and Jeff Rohl, two important members of the Atlas compiler team, exerted a strong influence on MU5's architecture and software. This in turn had a major impact on the design of the machine that became the ICL 2900 Series [ref. 41]. John Buckle of ICL, who worked closely with Derrick and Jeff during this period, remembers their "insistence on maximum support for block structured languages and the wealth of experience that enabled them to say both why and how this should be done" [ref. 42].

All this does not in any way detract from the originality of the Atlas Compiler Compiler. George Coulouris has recently [ref. 43] described the compiler compiler as *"A quite amazing achievement in terms of the innovations that it contained and the effectiveness of its design and implementation".* At the time it was unique. Later, there was a series of less-ambitious parser-generator systems, perhaps the best-known of which (YACC (*yet another compiler compiler*) was written in 1975 for UNIX) – (see also the Appendix). Such systems basically parse an input language into trees and provide routines to interpret what they mean. These routines can be replaced by compiled versions for efficiency. This is essentially how XSLT, the XML transformation language, works. XSLT currently supports the world wide web. Tony Brooker's insights have lived on. His retrospective views, presented in a special technical Appendix, follow on the next page.



Tony Brooker in October 2013, aged 88.

Appendix.

The Compiler Compiler (CC) Revisited.

R A Brooker, January 2014.

The CC was a program which, provided with the syntactic and semantic descriptions of the types of statements constituting a programming language, would generate a regular compiler for that language, for the computer in question. For example if the language was Fortran it would yield a Fortran compiler, one which translates user's Fortran programs into machine code for that computer (or an assembly language which is (effectively) in 1-1 correspondence with it).

The resulting compiler (Fortran or whatever) works as follows. When supplied with a user program it reads the constituent statements, one at a time, matching each of them syntactically with one of the possible statement types (i.e., parsing them) ; and when it finds a match it looks up the corresponding semantic description, which states how to process it.

If it is a declarative type of statement e.g., **real** Ident1, Ident2, etc) it stores this information; which will subsequently be used when an executive type of statement occurs which feature one or more of these identifiers. With this information the CC will formulate and compile the appropriate instructions to execute that statement in the resulting computer program.

An Example
We shall describe only one type of statement here, the (executive) A*ssignStatement* and its constituent phrases, the *Variable* and *Expression* (henceforth abbreviated to *Var* and *Expr.)*, each of which have appropriate subphrases. We can describe it thus

AssignStatement => Var := Expr

where *AssignStatement* is a CC **pointer variable**. It gives the address of its **descriptor** *Var := Expr* which, like all descriptors henceforth, is underlined. *Var* and *Expr* will in turn point to their own descriptors with subphrases such as *Term, Factor*, *Ident* . In effect *Var := Expr* is the **node** sitting at the top of a **Tree** describing the *AssignStatement* **.**

Processing such a statement is the purpose of the function

ProcessAssignment(St1) *=> Begin ……. End*

where *St1* is some statement pointer. This function will access any relevant declarations and plant the machine code (or equivalent) instructions to execute it.
(Note: => is a CC symbol; and := is a program symbol.)

Symbols of the 'CC' Language
In addition to the *italic identifiers* (characterised by initial capital letters for component words) and the pointer symbol => the language uses just five further italic symbols pertaining to descriptors, thus

{ } to enclose compound descriptors,
/ to separate alternate descriptors within *{ }* 's,
and **?* to indicate 0,1,2 … possible instances of a descriptor (* indicates one or more, *?* indicates at most one, and **?* indicates zero or more.)

Any other symbols normally used in programming languages (operators, brackets, punctuation) which are also needed in a CC context, such as a procedure, will appear in *italics* (although this may not be typographically apparent!).
Thus we may assume for the purpose of this article that there are NO italic symbols in the program being compiled, whereas ALL the symbols in a CC program are italicised.

Describing the Syntax of a general Expression
The following is the syntax definition of a somewhat simplified arithmetical expression (*Expr*), a construct which appears in most programming languages. (It is simplified in the sense that it makes no provision for exponents and functional values.)

(In what follows the LHS of each definition is an identifier (pointer) and the RHS is the descriptor.)

Expr => {+|-}? Term { {+|-} Term }?*
where the RHS means a *Term* (possibly preceded by + or –) and possibly followed by(plus or minus) further *Term*s.

Term => Factor { {×|÷ } Factor }?*	e.g., 2 × A[i]
Note a x b ÷ c x d means ((a x b) ÷ c) x d	
Factor => {Number|Var|(Expr)}	e.g., 0.1 , B , (A + B)
Number => Integer { .Integer }?	e.g., 3 , 3.14159
*Integer => Digit **	e.g., 1 , 12 , 3003
Var => Ident {Subscript}?	e.g., A , A[i] , B[2]
Subscript => [*{Ident|Integer}*]	e.g., [k]
Ident => Letter {Letter|Digit}?*	e.g., a , a1 , ab12c

The foregoing serve as the multi branching nodes of the *Expr* tree referred to earlier. Thus *Term* in the (RHS) descriptor for *Expr* is a pointer to its own (RHS) descriptor. The recursive appearance of *Expr* in the definition of *Factor* is a pointer back to its own (node) descriptor. The components of a node are referred to as units and are stored in a (possibly) linked list. When we say that a pointer refers to a descriptor node we mean that it points to the first syntactic unit of that node.

AssignStatement ----------> Var := Expr
Var ----------> Ident {Subscript}?
Expr ----------> {+|-}?Term{{+|-}Term}?}*
and similarly for all the other phrase definitions, *number, Subscript* etc*. Letter and Digit* are as yet undefined.

Before leaving this section we introduce, for the sake of brevity, two more phrase names
MoreTerms? => {{+|-} Term}? MoreFactors? => {{x|/}Factor}*?*
(We have taken the liberty here of (also) using '?' as an alphabetical character, which we trust won't cause any ambiguity).

The named pointers on the LHS of all syntactic definitions for statement types and phrase types (for a specific programming language) are assembled into two open ended lists, which are themselves identified (by the pointers) *StatementTypes* and *PhraseTypes.* Along with all pointers to descriptors we also store pointers to their names (as character strings). This last will enable monitoring the progress of compilation. Both these and the descriptors could (in principle) be stored in consecutive storage locations because they remain invariant when the CC is actually translating a program (in the language for which it is designed).

However during the compiling process we need open ended Trees and Lists because the size and shape of structures arising from actual program statements cannot be known in advance. They nevertheless have to be mapped onto the consecutively numbered cells of a computer. This means storing the nodes as open ended lists of units. It is not the purpose of this article to go into the details of such a system, such as finding and recovering space, and whether lists should be bidirectional. Suffice to say that it will provide both for creating such structures (when parsing statements) and provide for appropriate functions to explore the parsing records (i.e., accessing their nodes and units.)

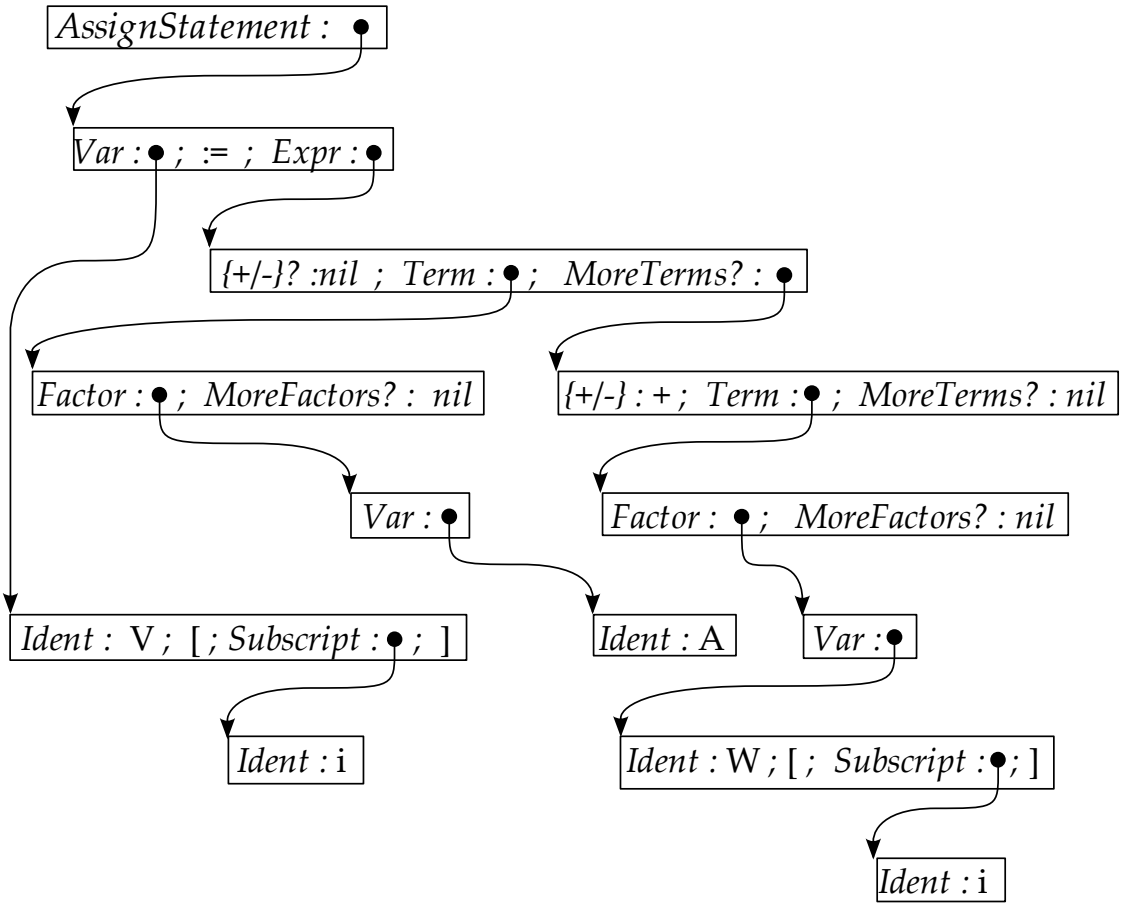
In addition to *StatementTypes* and *PhraseTypes,* we need (pointer) variables *Type, Statement* and *ParsingRecord.* The latter will point to the record of a statement after it has been successfully matched to one of the *StatementTypes*. (Those most likely to occur would be listed first).

BEGIN read Statement;
FOR ALL Type in StatementTypes DO
IF Statement Matches Type THEN Statement => ParseTree END
END; PROCESS(Statement)
The loop finishes with *Statement* pointing to the result of the parsing process, which is then processed, i.e, compiled. Note the use of the (italicised) key words *BEGIN, END* which mimic their Algol equivalents.

An Example of the Tree resulting from the Parsing Process
Now consider how an *actual* assignment statement, for example V[i] := A + W[i] would match the general descriptor
Var := *Expr* where *Expr => {+|-}? Term MoreTerms?*

The following is the record which would result from the simple example just given. (It is of course quite distinct from the statement descriptor tree.) The parsing process itself is described later on.

(See diagram on next page …)



Here a *nil* indicate the absence of anything in the source program to match a descriptor qualified by a *?* such as a *Subscript*, *MoreFactors?* or *MoreTerms?* . Note too that the symbols of the statement being matched, namely V, i, A, W are not italicised because they are source language characters.

The elongated 'boxes' represent the nodes of the tree, the heads of the subtrees (in a multibranching node) being separated by (*italicised*) semicolons. Each head is represented by a twin celled unit, the first cell specifies the phrase type and the second cell (which is initially nil when the head is first created) points to the subtree for that phrase or its terminal; the terminals in this example being V,i,A,W. More precisely the second cell (if not nil) points to the first cell of the subtree node, because where multiple branching is possible (a *Term* could have several *Factors*) there is a twin celled unit for each branch, and together they constitute the node (of the multi-branch tree 'below' them).

A final word before describing the matching process. The content of the first cell of the unit [{+|-}?/*nil*] is, strictly speaking, a pointer to a supplementary descriptor *plusorminus?* That of the second cell (nil) remains unaltered to confirm its absence. If present it could be '+' or '-' .

The Parsing Process

The CC works by scanning the syntax descriptor in a topdown manner, left to right fashion, at the same time trying to match it with the object text. Thus it looks first for a *Var*, which

then means looking for an *Ident*, possible followed by a *Subscript*. Looking for an *Ident* it has first to look for a letter which it finds in the form of a V. Finding no further letters (or digits) it backs up a level to look for a possible *Subscript*, and finds the '[' which heralds the subscript proper. After this it can confidently look for an *Ident* or an *Integer*, and indeed finds the *Ident* in the form of the single letter 'i'. Single because the next character is the closing ']'. The scanner then backs up further to confirm that it has recognised a *Var*. The CC then looks for the assignment symbol ' := ' and finding it proceeds to examine the right hand side *Expr* .

The recognition process for the latter proceeds in much the same way, top down, left to right fashion as that which we have already described; except for the structured descriptors enclosed in {…}. For example after matching the A to *Factor* the next symbol is + but we have to check that this doesn't herald a second factor. However this is not the case because that would have implied an *'x' or '/'* (although the descriptor {{x|/}*Factor*}*?) has to be inspected to verify this). Thus the head representing this unit is *nil*.

Likewise the structure *{{+|-} Term}*?* has to be unpacked to yield a *{+|-} Term}* to provide a match for + W[i].

Finally the presence of an 'end of statement' symbol (say ';') brings an end to the matching process for this statement, so that the *nil* in the second cell of the head *MoreTerms?* (on the 4th line of the tree diagram) remains unaltered.

Processing the Parse Tree

The result of the matching process (shown above) is similar to the descriptor record of the AssignStatement but contains only those phrases which are present in the statement presented for recognition.

At the end of the matching process we are presented with the (pointer) variable *Statement* pointing to the *ParseTree* presented earlier. Knowing that this takes the form
Var := *Expr* where *Expr => {+|-}? Term MoreTerms?}*
we can identify the component subtrees with pointer variables of our own choosing by using the *LET* function, thus

LET(Statement) = {VAR := EXPR}
LET(EXPR) = {SIGN TERM MORETERMS?}

which assign the pointer variables *VAR, EXPR, SIGN, TERM* and *MORETERMS?* to the subtrees in question, so that, e.g., *TERM* points to the subtree for *Term.*

With the aid of these pointers a processing function can find its way into the depths of the *ParseTree*.

As explained earlier each unit of this recognition record (being a node or part of a node) consists of just two elements, a descriptor name and a pointer to (the node of) a subtree which terminates in the relevant characters of the statement being processed. Thus a processing function will know the nature of each phrase encountered and the means to access it. This includes the case of alternative structures where the unit concerned records in its second cell either a pointer to the alternative subtree actually present (or to

its value). There is an example of this in the *ParseTree*, specifically the choice of sign {+|-} before the second *Term*.

When a pointer refers to the first unit of a node we can access the following units in the (linked) list using the *NEXTUNIT* function. (Depending on how the list is implemented it may be possible to move in either direction.) Thus

NEXTUNIT(SIGN) points to the subtree for *TERM* and
NEXTUNIT(TERM) points to the subtree for *MORETERMS?*
NEXTUNIT(MORETERMS?) returns *nil*

There is also the function *NUMBOF* which applies to any structure of the form *{….}*?* . It returns the actual number of such structures present, 0,1,2, …. as the case maybe. Thus in our example *NUMBOF(SIGN)* returns 0, and *NUMBOF(MORETERMS?)* returns 1.

There is also the function *NTH(I, {…..}*?)* which returns a pointer to the *I* th instance of the qualified item if *I* is in range, else *nil*. With the above functions we can find our way about any tree structure, and if necessary compose further functions.

For example the function *VAL* applied to any unit (say a *Factor*) follows the pointer to its subtree(s) where the terminal characters constitute either a *Number* or a *Variable* or a bracketed (sub)*Expr(ession*). The result will be to plant instructions to either evaluate the *Var* (at runtime) or return the Number itself. The third alternative is to plant instructions to evaluate the subexpression.

With the foregoing function we can evaluate a *Term* by evaluating its component *Factor*s and multiplying/dividing them together. The final step, to evaluate an *Expr,* is to evaluate its *Term's* and add/subtract them together, thus :-

```
BEGIN
LET EXPR = {SIGN  TERM  MORETERMS?};
ACC:=  VAL(TERM);
IF  VAL(SIGN?)  = '-' THEN ACC:= -ACC END;
        BEGIN
        IF MORETERMS? ~=  NIL THEN
        LET MORETERMS?  =  {SIGN  TERM  MORETERMS?} ;
        INCREMENT := VAL(TERM);
        IF SIGN =  '+'  THEN ACC := ACC + INCREMENT
                        ELSE ACC:= ACC – INCREMENT   END;
        END
END
```

The function *VAL(TERM)* is evaluated in much the same way employing the function *VAL(FACTOR)* which, as outlined earlier, is the value of a *VAR* or *NUMBER* (or possibly a subexpression). Thus

LET MORETERMS? = {SIGN TERM MORETERMS?}
becomes
LET MOREFACTORS? = {SIGNX FACTOR MOREFACTORS?}
Where now SIGNX means a ***multiply or divide*** *sign, as distinct from a plus or minus.*

(Note that in both the preceding *LET Statements* the *MORE…* pointers are *themselves* replaced.)

Now the purpose of an assignment statement is to evaluate the r.h.s *Expr* and store the result in the location associated with the l.h.s *Var*. The evaluation instructions are planted in the compiled program by the CC commands such as

ACC:= VAL(TERM); ACC := ACC + INCREMENT
the ACC being some working store location set aside for this purpose. These evaluation instructions will need to know the storage locations allocated to the variables involved and for this they refer to the declaration statements for these variables, which could, for example, take the form **real** V, W, A; **integer** ;

The same applies to the storage of the final result which is achieved by the CC assignment *STORE*(*Var*) *:= VAL*(*Expr*) where *STORE(Var)* plants instructions to replace the current value of the l.h.s *Var(iable)* by the value of the r.h.s *Expr.*

The Efficiency of the Parsing and Compiling Process
This point was considered in our 1967 paper [ref. 24] "Experience with the Compiler Compiler", thus:
"The parsing algorithm has sometimes been criticized because it operates top-down instead of bottom-up. However, the grammatical definitions are usually such that it would make little difference which method is used if the *c*omplete parsing record is to be determined. More to the point is the fact that complete parsing of a fairly simple statement such as *X :*= 1 with respect to [VARIABLE] = [EXPR] involves the construction of trees with many empty branches. It is the time spent in this activity, and on the subsequent inspection of the empty branches in the processing routines, that makes our approach inherently less efficient than (say) the technique of operator precedence for arithmetical expressions. If necessary, however, we can always fall back on such methods through the use of phrase routines."

The syntax tree for a statement is the same whatever method of parsing is used. As with top-down parsing the bottom-up process scans the symbols of the statement from left to right; but instead of reading the entire statement it only proceeds until a recognisable (sub)phrase is encountered, for example a term or factor. (This may entail 'looking ahead' in the input sequence.) In this way the parsing process 'builds' the tree starting from the bottom left end, and works its way upwards and rightwards. Such parsers that need only look one symbol ahead are classified as LR(1).

It may at the same time process the nodes as they are discovered *without ever creating an actual data tree*. In this sense the bottom-up method is more efficient because it only deals with the structure that is present, not the structure that might be present. Some illustration of what this kind of 'compiler generator' entails is given later on. (**Parser generators.**)

Wikipedia ("bottom-up parsing") illustrates how the complete parse tree for an assignment statement is assembled, both top-down and bottom-up, to illustrate the difference in the sequence in which the nodes were visited.

As regards the example quoted in our 1967 paper [ref. 24], X:= 1;

top-down parsing would yield (for the *Expr* tree)
Expr -> Term -> Factor -> Number -> 1, where *Term* and *Factor* are essentially redundant. Fortunately the terminating semicolon curtails further redundancy.

The same paper presents a table of performance figures for the Compiler Compiler when used to write compilers for (Extended) Mercury Autocode, Fortran, Algol and Atlas Autocode (AA) – (see below). In this last case it also compares the same figures with a conventional ('hand coded') version of the same language (AB). Somewhat surprisingly the average compiling times for AA and AB programs differed only by a factor of 2.

Table 2 from [ref.24]: Breakdown of compiling times, as percentages of total compiling times.

	EMA	**ALGOL**	**FORTRAN 2**	**AA**	**AB**
(1). Input				13%	19.8%
(2). Line reconstruction and pre-editing					
{(1) + (2)}	17.3%	27.1%	11%	23%	44.8%
(3). Analysis	57.9%	39.2%	29.3%	46%	
(4). Processing	24.8%	33.9%	59.7%	18%	
{(3) + (4)}					35.4%
Speed A	**1,452**	**1,100**	**2,275**	**537**	**298**
Speed B	**303**	**169**	**355**	**197**	**112**

Notes.
1. *Speed A* is the number of machine instructions obeyed per instruction compiled. *Speed B* is the number of machine instructions obeyed per character of program tape. These are average figures for a number of programs. In each case *Speed A* showed the most individual variation (the speeds for some programs being as much as ± 50% about the mean). The other measures, *Speed B* and the distribution ratios, showed much less variation.
2. One or two miscellaneous figures are also available: *Speed A* for an early version of AA was approximately 1,500. The overall compiling times for the same program run with EMA and CHLF3 was in the ratio of 3.9. (CHLF3 is a hand-coded compiler for a language amounting to a subset of EMA).
3. In none of these compilers were any special measures taken to optimise object code. (FORTRAN 2 was essentially a first exercise in using the CC).

We shall now deal with the matter of 'phrase routines' without which things could become DRAMATICALLY worse.

Phrase Routines
Recall that the syntax definitions for *Expr (Term, Factor,* etc) stopped at *Letter* and *Digit*. It was somehow implied that these units were basic.

We could however have taken them further and defined
Letter => {A|B|C| ……|Z} *Digit* => {1|2| …… |9}
One's mind boggles at the time involved in parsing a statement top-down in such detail.

Instead of course, syntax units such as *Letter* and *Digit* are 'built in', meaning that instead of searching a list of alternative letters (or digits) the recognition program simply 'looks up' the 8-bit representation of the characters concerned in a table of (say) 256 entries which in effect say 'this is a letter' or 'this is a digit' or 'this is not a letter or digit'. Hence the term a

'table driven' program. It was a small step then to provide special recognition routines for related (higher) level units such as *Ident*, *Subscript* and *Var*, and similarly *Integer* and *Number.*

A possible next step of course would be to write built in routines for *Factor*, *Term* and *Expr*, and other syntax units such as *Boolean* and even whole statements. (Indeed one can provide the entire compiler in this way!)

Thus far in this article we have only been concerned with the recognition and compilation of expressions which are where the Compiler Compiler (and indeed the resulting compiler itself) spends most of its time. Expressions appear not only in assignment statements but also in Boolean statements and as parameters in function calls.

(Note: as regards the efficiency of the *compiled program* this is quite a different matter: as with any programming language the CC can be used to write good programs or bad programs.)

Comparison with Backus-Naur form.
The phrase structure notation we have used in the foregoing is somewhat different from the standard Backus-Naur form. In that notation our definitions of Number and Ident would appear as

<number> ::= <integer>|<integer> . <integer>
<integer> ::= <digit>|<integer><digit>
<ident> ::= <letter> | <ident><letter> |<ident><digit>

This makes explicit the recursive nature of the definitions but where recursion implies repetition, we felt that it was preferable to have a short hand notation for that, and * and ? provided it.

Parser-Generators
According to Wikipedia compiler generators based on bottom-up parsing & processing are known as parser-generators precisely because they combine parsing with code generation.

It is appropriate to demonstrate the method by processing the LHS of an assignment statement (using the phrase definitions already introduced). Assume that the ':=' has already been processed and instructions have been compiled to allocate some storage location E (initially zero) in which to build the value of the expected expression. The phrases we might expect to follow are listed below together with the processor actions implied. The underlined characters in what follows are not input but signify the end of the preceding component phrase (of the expected expression).

{+|-}? *Ident* [	implies a subscripted Ident (*Var*). The subscript and closing ']' must be read before processing can proceed.
{+|-}? *Ident* {+|-}	implies E := E {+|-} VAL(*Ident*)
{+|-}? *Number* {+|-}	(*Val* meaning *Ident* or *Number*) implies E := E {+|-} VAL(*Val*)

We can encapsulate the last two cases by writing

{+|-}? *Val* {+|-}	(*Val* meaning *Ident* or *Number*) implies E := E {+|-} VAL(*Val*).

The next two cases embody the precedence of {x|÷} over {+|-}

{+|-} *Val* {x|÷}	implies the start of an implicit subexpr (pushdown current E {+|-} and set new E :=VAL(*Val*))
{x|÷} *Val* {+|-}	implies end of implicit subexpr. update (old E := {+|-}) new E {x|÷}VAL(*Val*);

The next two cases deal with explicit subexpressions

{+|-|x|÷}? (	implies an explicit subexpr. (advance the pushdown list, E:= 0)
{+|-|x|÷} *Val*)	implies the end of explicit subexpr (backup the pushdown list). (old E := {+|-}) new E {+|-|x|÷} VAL(*Val*)
{x|÷} *Val* {x|÷}	implies E := E {x|÷} VAL(Val)
;	implies end of statement

The matching process entails looking ahead in the input sequence for an underlined symbol and acting accordingly. The action to be taken in each case could clearly be expressed by a CC procedure such as those used earlier.

Case recognition would be performed by a cyclic CC procedure which matches the source text with each of the foregoing cases (held in a list).

All this amounts to saying the Compiler Compiler could itself be used to write a Parser-Generator, which should be no great surprise because the CC is a compiler oriented language and a parser-generator is just another kind of compiler. Indeed the CC could no doubt be used to write itself.

Bottom–up parsers became the basis of all subsequent ‘compiler generators’ in the USA. Thus the concept of a top-down ‘compiler compiler’ rather faded from the screen. Nevertheless the name survived. (It was in fact coined quite early on by my colleague Derrick Morris.) It appears in the widely read report of a parser-generator developed at Bell Labs, Murray Hill, N.J and described in “Yacc (Yet Another Compiler Compiler)” by Stephen C. Johnson. (It is available online.) Yacc was originally written in a version of the system language ‘C’.

It is difficult to compare the two approaches to compiler writing because the foregoing attempt to explain the parser-generator approach is the author’s only experience of the latter. It is tempting therefore to fall back on poetry. One gets a better view of the local street plan from the top of a building. The use of low level phrases such as {+|-} and *Val* in the foregoing explanation of the bottom-up approach suggests that even a view from a ‘first floor’ window can be helpful.

The reader is recommended to read at least the concluding paragraphs of the 1967 paper [ref. 24] “Experience with the Compiler Compiler”, which point that, efficiency aside, it lends itself to quickly roughing out working versions of compilers for one-off projects.

Parser-generators on the other hand with their focus on the 'ground floor' may well be the answer to generating really efficient compilers. In this context the work of A.Glennie [ref.12] is probably the best contribution from the UK, work which, for security reasons (the author worked for AWRE), did not surface for long after it was proposed in 1960.

Experience of using the Compiler Compiler

This is contained in the 1967 paper already referred to. By this time several compilers had been written using it, both in the Computer Science Department at Manchester and in London University. They are all reported in this final paper together with acknowledgements to the writers concerned. Regrettably the present author was working for IBM Research in the academic year 1962-3 when the Compiler Compiler itself was actually commissioned by Derrick Morris and Iain MacCallum. They were so successful in this task that the CC was soon being to used to commission (an extended) Mercury Autocode (which had been in use since 1958, and was in great demand because the Mercury itself was about to be decommissioned) and Atlas Autocode (an Algol-like compiler). I myself wrote the machine code version of an AA compiler (called AB) to see how it compared with the CC version (commissioned by J. Rohl). The results are contained in the 1967 paper.

References.

1. The main source of quotations in this paper is the British Library extended interview of Tony Brooker, carried out by Thomas Lean between 16th February and 10th June 2010. This is number C1379/09 in the BL's *National Life Stories* series, accessible via http://sounds.bl.uk A few additional quotations have been taken from a series of e-mail exchanges between Tony Brooker and Simon Lavington between 6th September 2012 and 28th October 2013.

2. Brooker, R A, *The autocode programs developed for the Manchester University computers.* Computer Journal, Vol. 1, number 1, 1958, pages 15 to 21.

3. The Atlas One-level Store. R A Brooker & F H Sumner. May 1959. 19-page typed foolscap document. Available from the National Archive for the History of Computing, box-file S1. This paper gives program fragments and estimates likely swapping-times for the following algorithms: matrix multiplication; matrix transposition; roots and vectors of a symmetrical matrix by the Jacobi method; cross correlation & partial differential equations; solution of simultaneous equations. It is thought that this document, which represents a step in the evolution of the final Atlas system, was for internal discussion only.

4. Robin Kerr, responding to a question from the audience at the Atlas 50th Anniversary Symposium, Manchester, 5th December 2012.

5. Brooker, RA and Morris, D. *An assembly program for a phrase structure language.* Computer Journal, vol. 3 (1960), page 168.

6. Brooker, RA and Morris, D, *Some proposals for the realization of a certain assembly program.* Computer Journal vol. 3 (1961), pages 220 - 231.

7. Brooker, RA and Morris, D . *A description of mercury autocode in terms of a phrase structure language*. In *Second Annual Review of Automatic Programming,* Pergamon, New York, 1961.

8. Brooker, RA and Morris, D, *A general translation program for phrase structure languages*. J. ACM vol. 9 (Jan. 1962), page 1.

9. Brooker, RA, Morris, D and Rohl, J S. *Trees and routines*. Computer Journal, vol. 5 (1962), page 33.

10. Brooker R A , MacCallum I R, Morris D and Rohl J S, *The Compiler Compiler.* Annual Review of Automatic Programming Vol 3, 1963, page 229. Published by Pergamon, London.

11. Rosen, S. (1964). *A Compiler-Building System Developed by Brooker and Morris*, Comm. A.C.M., Vol. 7, No. 7, July 1964, pages 403 - 414.

12. Glennie, A E, *On the Syntax Machine and the Construction of a Universal Compiler.* 10th July 1960. Computation Center, Carnegie Institute of Technology, Technical Report No. 2. Prepared under Contract Nonr-760(18) (NR 049-141) for Office of Naval Research. Available at: http://www.chilton-computing.org.uk/acl/literature/reports/p024.htm

13. F R A Hopgood, e-mails to Simon Lavington, December 2013.

14. Schorre, D V, *META II: A Syntax-Oriented Compiler Writing Language***.** In Proceedings of the 19th ACM National Conference, 1964, ACM Press, New York, NY, 41.301-41.3011.

15. John Clegg, e-mail correspondence with Simon Lavington, July to November 2013.

16. Robin Kerr, e-mail correspondence with Simon Lavington November 2011 to November 2012.

17. MacCallum, I.R. (MSc Thesis, Manchester University, January 1963, catalogue number Th7476 at the Joule Library): *Some Aspects of the Implementation of the Compiler Compiler on Atlas.*

18. H W Gearing, *Review of Extended Mercury Autocode*. Computer Journal vol. 23 no. 8 1965, page 23.

19. Brooker, R A, Rohl, J S and Clark, S R, *The main features of the Atlas Autocode.* Computer Journal, Vol. 8, 1965, pages 303 – 310.

20. Brooker, R A, Morris, D and Rohl, J S, *Compiler Compiler facilities in Atlas Autocode..* Computer Journal, vol. 9, number 4, February 1967, pages 350 – 352, 1967.

21. Dik Leatherdale, e-mail communication with Simon Lavington, 6th November 2013.

22. Fossey, B and Churchhouse, R F, *The case for writing an Algol compiler.* Internal report, Atlas Computer Laboratory, Chilton, 20th December 1963. Available at: http://www.chilton-computing.org.uk/acl/applications/algol/p001.htm

23. *Algol Bulletin number 1.* ICT Atlas Computing Service, 19th March 1964. Anon, but Robin Kerr's initials are included at the end of this five-page typewritten document.

24. Brooker, R A, Morris, D and Rohl, J S, *Experience with the Compiler Compiler.* Computer Journal, vol. 9, number 4, February 1967, pages 345 – 349, 1967.

25. Napper, R B E, *Some proposals for SNAP, a language with formal macro facilities*. Computer Journal, Vol. 10 number 3, 1967, pages 231 – 243.

26. Lavington, S H and Rosenthal, L E, *Some facilities for speech processing by computer.* Computer Journal, vol. 9 no. 4 1967, pages 330 – 339.

27. Morris, D, Sumner F H, and Wyld, M T, *An appraisal of the Atlas Supervisor.* Proceedings of the ACM National Meetings, 1967, pages 68 – 75.

28. Morris, D and Rohl, J S, *The Atlas compiler system.* Computer Journal, Vol. 10, number 3, 1967, pages 227 – 230.

29. Coulouris G.F, Goodey T.J., Hill R.W., Keeling R.W. and Levin D, *The London CPL1 Compiler,* Computer Journal Volume 11, Issue 1, 1968, pages 26-30.

30. Pyle, I C: *Implementation of Fortran on Atlas*, *in* "Introduction to System Programming", *ed*. P Wegner, Chapter 6, pages 86-100, Academic Press (1964).

31. Hopgood, F R A and Bell, A G, *The Atlas Algol pre-processor for non-standard dialects.* Computer Journal, vol. 9, number 4, February 1967, pages 360 – 364, 1967.

32. *Digital computing facilities at AWRE Aldermaston*. 12-page illustrated brochure produced by AWRE in December 1966.

33. E-mail exchanges between Simon Lavington and the following former associates of the Mintech CAD Centre: Alan Clark, Charles Lang, Mike Williamson and Peter Woodsford, on 3rd and 4th December 2013.

34. Lang Charles. A., *SAL - Systems Assembly Language*. Proc. Spring Joint Computer Conference, American Federation of Information Processing Societies, May 1969, pages 543 - 555.

35. See: *Ferranti Atlas 2 computers at AWRE Aldermaston and at the CAD Centre.* Simon Lavington. August 2014. See: http://elearn.cs.man.ac.uk/~atlas/docs/Atlas%202%20at%20AWRE%20and%20CAD%20Centre%20Final.pdf

36. GINO-F – an appreciation. 19-page technical brochure dated 8th May 1974, issued by the Computer Aided Design Centre, Madingley Road, Cambridge.

37. A scanned copy of the Compiler Compiler source code is available at:

http://elearn.cs.man.ac.uk/~atlas/docs/Compiler%20Compiler%20source%20code An explanation is given here:
http://elearn.cs.man.ac.uk/~atlas/docs/An%20explanation%20of%20the%20Compiler%20Compiler%20listings.pdf

38. The Atlas emulator is available at: http://www.leatherdale.net/atlas.html

39. J.S. Rohl, *The Influence of Programming Languages on the Design of MU5*. IEEE Annals of the History of Computing, vol. 21, no. 1, pp. 34-37, Jan.-March 1999.

40. (a). T Kilburn, D Morris, J S Rohl and F H Sumner, *A system design proposal.* In Information Processing 68 (the proceedings of the 1968 IFIP Conference at Edinburgh), published by North Holland, Amsterdam, 1969.
(b). D Morris and R N Ibbett, *The MU5 computer system.* Macmillan Press, 1979.

41. J K Buckle, *The ICL 2900 Series.* Macmillan Press, 1968.

42. J K Buckle, e-mail to Simon Lavington, January 2014.

43. George Coulouris, *The Compiler Compiler - Reflections of a User 50 Years On.* November 2013. See:
http://elearn.cs.man.ac.uk/~atlas/docs/CC_Reflections-v4%20final.pdf