WORKING NOTE AC128

Alan Creak
2000 June 26

THE STUBOL STORY

The body of this Working Note was written somewhere around – probably before – 1980. It was a draft of a paper for publication; the paper was never completed, because the university got rid of the Burroughs B6700 on which the compiler depended, and replaced it with an IBM4341. This Note has been prepared to preserve the material, as there has been some recent interest in it.

The material has been scanned from the original, and is unchanged except for a few spelling corrections, some corrections marked on the original by hand, one or two minor rephrasings to clarify points which had become obscure in the intervening years, and the typeface. (It used to be Letter Gothic and was printed by a typewriter with a golf-ball head.)

In reading the Note, it is useful to bear in mind that the Stubol system was devised when essentially all computing was dependent on batch processing, with a minimum turn-around time of an hour at best, and students could not sit at a computer trying several different solutions to a problem until one worked. A significant aim for the system was that the student should get at least some benefit from every attempt, even if the programme was seriously wrong.

A COMPILER FOR TEACHING COBOL

A specialised Cobol compiler has been developed, in which the redundancy inherent in a Cobol programme is exploited and certain default properties are introduced to facilitate, in particular, the early stages of teaching the language. A beginner may compile and execute "programmes" consisting of only a Procedure Division, while, for more advanced students, a realistically large subset of Cobol is implemented.

It is widely accepted that Cobol is one of the more difficult high-level languages to teach – and, correspondingly, to learn. One reason for this appears to be that, before a student can run his first Cobol programme, he must somehow attach appropriate Data and Environment Divisions to the (comparatively straightforward) Procedure Division, so that a great deal of the formal structure of the programme must perforce be present from the outset. It is interesting to compare this aspect of Cobol with the corresponding requirements of other languages. Fortran requires no declarations as such, although, unless the compiler used permits free-format input and output, the complexities of the FORMAT statement can be something of a stumbling block; PL/I is surprisingly easy for beginners, as the default declarations of variables and the list-directed input and output statements are adequate for simple programmes; while Basic, where problems of declarations and formatting need not arise at all, is probably the easiest language of its type to learn.

There are at least two reasonably straightforward solutions to the Cobol problem, each of which leaves something to be desired. On the one hand, the instructor may elect to teach at least the elements of the Environment and Data Divisions, as well as the Procedure Division, before letting the students try their hands at writing a programme; this takes rather a long time, and there is a fair risk that at least some students will lose heart at the prospect of mastering a moderately complex system of interlinked information before getting to grips with the computer. On the other hand, the instructor may provide standard Environment and Data Divisions, together with rules for their use; this method is perhaps to be preferred to the alternative, at least insofar as it allows the student to begin practical programming sooner, but suffers from the more subtle defect that the student is encouraged to write his Procedure Division to fit in with the declarations rather than – more satisfactorily – designing the declarations to suit the needs of the Procedure Division (and, possibly, constraints imposed by requirements for compatibility with existing files).

Aims of the new compiler

Contemplation along the lines sketched out above, coupled with past experience of teaching Cobol and the prospect of more to come, led to the development of a third approach. The desirability of students being able to write their own programmes as soon as possible without the constraints imposed by a standardised set of declarations suggested that a compiler which could accept and run a Procedure Division alone would be a valuable teaching instrument : the possibility in principle of such a compiler is attested to by the existence of Fortran compilers, in which essentially the same problems (those of determining the nature of an identifier from its context, rather than from formal declarations) are successfully solved. While it is clear that only a limited portion of Cobol can satisfactorily be implemented in this way, this is not a serious drawback : the object of this aspect of the design is to get the student off the ground rather than into the stratosphere, and a rather small repertoire of facilities is sufficient.

At the same time, it is clearly desirable that the beginner should be led on, if not up to the stratosphere, at least in its direction. The compiler should therefore be capable of handling a subset of Cobol containing at least the more important aspects of the language, so that moderately ambitious assignments can be tackled in a reasonably realistic way. Another feature under this heading is that errors should not merely be reported, but should, wherever possible, be corrected. The implementation of this proposal is fraught with pitfalls. It is easy enough to determine that a statement is syntactically incorrect, and it is often reasonably straightforward to see how it can be made correct; it is markedly more difficult to ascertain whether the proposed correction coincides with the programmer's original intention. It seemed, nevertheless, that the attempt should be made, but that the corrective action taken should be described in an appropriate error report.

A special case of error correction, with one eye on the beginning student, is that of the undeclared identifier. In order to achieve the aim already set out, some set of properties must be associated with the identifier : there is therefore no obstacle to the construction of the corresponding Cobol declaration, and it seemed reasonable that this declaration should be inserted at the appropriate point in the programme. Taken to its conclusion, this proposal requires a compiler which, when presented with a Procedure Division alone, will construct and print with the programme listings suitable Environment and Data Divisions : this is of considerable value in demonstrating, first, the sort of thing that the programmer should have written, and, second, the point that, subject to certain constraints, the form of the early divisions is largely determined by what is in the Procedure Division.

Implementation

I have successfully implemented a compiler which, in general terms, achieves the aims laid out above; it is now in daily use by students of the Commerce Faculty of this University. The compiler runs on the University's Burroughs B6700 machine, and is written in B6700 Extended Algol. The subset of Cobol implemented by the compiler is, in summary :

IDENTIFICATION DIVISION
PROGRAM-ID, AUTHOR, and DATE-WRITTEN paragraphs.
(The run date is also printed on the listing)

ENVIRONMENT DIVISION
CONFIGURATION SECTION : SPECIAL-NAMES paragraph only, for printer carriage control functions.

INPUT-OUTPUT SECTION : FILE-CONTROL paragraph only; one card-reader, one line-printer, and up to 6 disc files (RANDOM or SEQUENTIAL) may be declared.

DATA DIVISION
FILE SECTION and WORKING-STORAGE SECTION : Conventional. PICTURE, USAGE (restricted to DISPLAY for characters or COMPUTATIONAL for binary numbers), VALUE, OCCURS, and REDEFINES clauses may be used. Condition-names may be associated with elementary items. Level 77 items may be defined in the WORKING-STORAGE SECTION.

PROCEDURE DIVISION
Sections and paragraphs. ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE. READ, WRITE, OPEN, and CLOSE. GO, IF, PERFORM, EXIT, and STOP RUN. MOVE and EXAMINE. SORT, RELEASE, and RETURN.

In addition, any user may maintain a file library, which may contain data files generated by his programmes, or source programmes in card image form which may be used in conjunction with a variant of the COPY statement.

In the interests of simplicity, the compiler makes a single pass over the source programme. Separate arrays are used for executable code and for storage; this device is particularly convenient in this situation, as some – or, indeed, all – of the storage requirements may not be defined at the beginning of the Procedure Division. In the event of successful compilation, the code is executed interpretively. Although this technique unavoidably results in rather slow execution, it is retained to permit much more careful error checking then would be possible using the hardware of the computer.

Coping with errors found during compilation

The more interesting features of the compiler are those which compensate for errors in, or omissions from, the source programme. With the exception of some very straightforward cases (such as statements beginning in the wrong area of the card, which are reported but otherwise cause no trouble), the situations which arise can be classified as consequences of missing or incomplete declarations of files or data items, or as syntax errors. In both cases, the philosophy of the compiler is, wherever possible, to act in such a way as to produce an executable programme – with luck, the programme which the user thought he was writing. There is clearly a limit to the extent to which a garbled statement can be ungarbled, but in practice the compiler seems to cope reasonably well with most of the more common errors.

In the case of a missing declaration, the aim is to provide a suitable declaration of the item. To achieve this aim it must be possible, first, to recognise the nature of an undeclared identifier in a Cobol statement, and, second, to assign to it a suitable set of default attributes.

The first of these requisites can, in most cases, be accomplished rather easily; the rigidity of Cobol syntax ensures that the type of an identifier (file, procedure name, data name etc.) can be determined unambiguously from its context. Differentiation of detail within this type is not always so simple : thus, while it is reasonable to assume that an undeclared file name appearing in a READ statement denotes a card reader file, it is less obvious whether an undeclared data name appearing as the source for a MOVE statement refers to a numeric or non-numeric item. More or less arbitrary criteria must be established to handle such cases; in the example, all data names other than names of file records are assumed to represent numeric items, unless they appear as destination items in a MOVE for which the source is known to be alphanumeric or alphabetic.

Assignment of default attributes turns out to be a matter for common sense and experiment. All undeclared data names, in the absence of indications to the contrary, are taken to represent numeric items of 10 digits with two decimal places. Undeclared filenames present a rather more tricky problem, as there may be insufficient information available when the name is first encountered to determine the hardware device with which the file should be associated : for example, if the filename is first encountered in a READ statement, then it can plausibly be assumed to be a card-reader file; but a filename first met in an FD statement has no similar contextual indication of its type, and it is therefore necessary to provide for partial definition of the file at this point in the compilation.

It is less easy to summarise the ways in which the compiler handles syntax errors, as most errors require individual treatment. To maintain the principle that an executable programme should be produced if possible, the compiler must try to guess what the programmer really meant. In some cases, the correction is obvious; for example, if a statement begins

```
WRITE filename
```

then it is almost certainly adequate to interpret the filename as the name of the file record defined in the Data Division. Similarly, though perhaps somewhat less reliably, if the statement

```
READ filename
```

is not followed by AT END or INVALID KEY, then AT END (or INVALID KEY) STOP RUN may be assumed without serious risk of calamity. In other instances, the requisite correction is less apparent; consider the statement

```
MULTIPLY A B  – – – – –
```

where A and B represent words not known to the compiler. In accordance with the rules already set out, A is assumed to be an undeclared identifier, and interpreted as a numeric data item, but B presents a dilemma. In the first implementation of the compiler, it was assumed that an unrecognised word in this position was in fact a mispunched BY, so that the next word should be a dataname; in the event, it turned out that a much more frequent error was the omission of the BY altogether, no doubt by analogy with the ADD statement, so the compiler was modified accordingly.

This example illustrates a point of some importance. It is feasible in principle to try to distinguish between these two possibilities by inspecting the next word : if this should be a reserved word (GIVING, ROUNDED, or the keyword of the next statement) or a fullstop, then it could be inferred that BY had been omitted; whereas if it should be a dataname or an unrecognised item, then the mispunched BY interpretation could be adopted. It is perhaps hardly surprising that a number of ambiguous cases of this type can be elucidated by considering a wider context; at the same time, the value of incorporating such extensions in the compiler is questionable. The existence of the error will in any case be reported to the programmer (subject to the reporting mechanism described below), and he would then normally amend and resubmit the programme whether the assumed interpretation were correct or not. To avoid undue complexity in the compiler, it therefore seemed reasonable to restrict its error-correcting capabilities by requiring that it make its decisions on the basis of what it already knows at the point where the decision is required.

After disposing of the correctable errors, there remains a residue which, for one reason or another, the compiler cannot patch up. Again, this assertion must be taken as an approximation : for example, the statement

GO TO SPECIAL-NAMES

falls into this category, although it is quite possible to envisage an extension to the compiler whereby it would be able to distinguish between distinct usages of a word (in this case, as a reserved word and as a procedure name). As a rule, such extensions have not been incorporated, both in the interests of restraining the compiler's growth, and because a good proportion of the errors in this group represent fairly serious violations of Cobol propriety, suggesting lacunae in the programmer's comprehension of the language which are best drawn to his attention in a fairly forcible way. These errors are therefore regarded as fatal.

Errors arising during execution

The principle that a programme should be made to run if possible is carried forward from the compiler phase into the interpreter. Two sorts of error are recognised : those which the interpreter can get round, and the rest. As for compilation errors, the distinction is drawn partly on grounds of possibility and partly on expediency : for example, an attempt to read from or write to a file which has not been opened causes the file to be opened in the appropriate mode, while an attempt to read a record from a file which has reached its end but has not been closed is taken to suggest a potentially serious flaw in the programme logic, and is fatal. The interpreter also keeps track of the execution time and the number of pages printed, and terminates any job in which either of these factors exceeds its permissible maximum.

Error reporting

The mechanism used in this compiler for reporting errors is slightly more complicated than the conventional automatic production of a more or less informative message; although a suitable message is defined for each error detected, it is only printed if a certain condition is satisfied. The need for selective error reporting arises from the dual function of the compiler : it is intended both as an "easy way in" to Cobol for beginning students, and as a practical language for those who have passed the introductory stage. The more advanced students require a full range of error messages, but a beginner who submits a simplified programme of the sort for which the compiler is designed gains nothing from being told that all the simplifications are wrong – he may indeed have some difficulty in finding his source programme among the welter of error messages which are produced.

The errors are therefore divided into three groups, on the lines already mentioned in the foregoing discussion. The groups are : group 0, containing the fatal errors; group 1, containing errors which the compiler can correct, but with a possible change in the performance of the programme; and group 2 which the compiler can correct without affecting the performance. For each batch of students' jobs, the compiler is supplied with a parameter called the level, which takes one of the values 0, 1, or 2; an error is reported only if its group number does not exceed the level.

This approach has worked reasonably well in practice, although some anomalies remain. These are particularly evident, although fortunately of infrequent occurrence, in programmes executed at level 0. As only fatal errors are reported at this level, any programme which is executed gives rise to no error messages; it is therefore somewhat disconcerting to a student to find that his results are not those which he expected, which can happen if his programme contains errors from group 1.

To take a case in point, suppose that the first READ statement in a programme with no Environment or Data Division is

READ CARDFILE,

Provided that the identifier CARDFILE has not been previously recognised as a dataname, the compiler will decide that it represents a card reader file, and will invent a record name of the form NAMEnnnn! (where nnnn represents a 4-digit integer). As the value of nnnn depends on the position of the job in the batch, and is therefore not predictable, the programmer has no way of referring to the record just read – unless he compounds his error by subsequently using CARDFILE as a dataname.

This sort of problem seems to be, if not unavoidable, at least difficult to eliminate completely, and the most satisfactory solution appears to be to explain the problem in the lecture room, and to emphasise that it is advisable to use the form

READ CARDFILE INTO AN-IDENTIFIER.

The same line is taken in the instruction manual which the students use for reference. A consequence is that, until the student can tackle the Data Division, only one data item (generally a number, although a particularly pertinacious student can force a character string) can be read from each card. This is accepted, as alternative solutions – such as the provision of some species of free-format input statement – would involve the introduction of non-Cobol statements, and some effort has been made to ensure that programmes which run, while they may not be very good Cobol, at least preserve the Cobol syntax.

CONCLUSIONS

The new compiler has proved to be very successful as a teaching instrument. In particular, it has permitted students to start writing programmes right from the start of the lecture course : one student, with no previous programming experience, had his first programme ready for submission within half an hour of the end of the first lecture.

The possibility of writing the compiler emphasises the high degree of redundancy present in the Cobol language. An obvious example concerns the majority of the standard division, section, and paragraph headers ; as most of the statements in the language are unambiguously identified by their first syntactic items, their proper positions in the programme can be determined without reference to the standard headings. This is not to say that the redundancies have no value; as is well known from Information Theory, redundant codes have valuable properties in the context of error detection and correction. Most Cobol compilers exploit the redundancies of the language to detect errors; the new compiler extends this exploitation into the field of error correction. Regarded as an error-correcting code, Cobol is not markedly efficient, in the sense that the redundancy is very unevenly spread. For example, it is easy to correct a statement of the form

```
BLOTCH DIVISION   ;
```

it is possible, although much harder, to correct the statement

```
BLOTCH A, B TO C   ;
```

while it is impossible to correct

```
BLOTCH A BY B GIVING C   ,
```

as the reserved word could be either MULTIPLY or DIVIDE. It is interesting to observe that much of this potentially useful redundancy arises from the attempt to make Cobol read like a natural language; it is tempting to speculate on the possibility of designing a viable programming language which incorporates sufficient redundancy to be essentially error-correcting throughout.

It is also possible to regard the compiler as defining three languages, one corresponding to each level of error reporting, and consisting of the set of programmes which are compiled and executed without error messages at each of the three levels; using the terminology of Algol 68, there is a strict language (say, L2) defined by the level 2 error reports, and two extended languages (L1 and L0), such that L1 is a proper subset of L0, and L2 is a proper subset of Ll. An Algol 68 compiler is required to recognise an extended language; the new Cobol compiler demonstrates that it is not difficult to perform a similar service for Cobol.

REFERENCES.

No references are present in the original; one or two are implied, but hardly worth following up more than twenty years after the event. Stubol is not nearly as well documented as I'd like it to be (and never will be); it was always unfinished business. After the disappearance of Burroughs Algol I spent some time trying to rewrite the compiler in Fortran, but the demand had evaporated and there wasn't enough spare time. (There wasn't any spare time.) The two publications identified below are all that remain, so far as I know.

G.A. Creak : *Cobol using the Stubol compiler* (Auckland University Computer Centre, 1975).

G.A. Creak : *The Stubol system : An error-correcting Cobol compiler for student use* (Auckland University Computer Centre, Technical Report #2, 1978).