20·10·62

published by ~~Infotech~~ ACIP summer school.

SINGLE-PASS COMPILATION

Mr. C. A. R. Hoare

Mr. HOARE: Single-pass compilation. There are many different ways of going about writing translators nowadays that it is advisable at the outset to give a reasonable definition of what is meant by a single-pass compilation. The first principle of single-pass compilation is that as much work of translation as possible is done in a single scan through the source code, which produces as output an intermediate code which is as close as possible to that which the machine obeys; and the second principle which is observed in designing single-pass compilers is to get as much information out of store as quickly as possible during the first scan, and not to fill the store with too much information, certainly not more than is absolutely necessary.

Now the circumstances in which single-pass compilation is the best way of going about things can be clearly defined. It is a fairly simple way of designing a compiler, which is to be implemented by a few people, and to a fairly short time scale. So where early implementation is a criterion, single-pass compilation should be considered. Secondly, it is particularly useful in dealing with a computer which has a reasonable amount of immediate access core storage, but no magnetic backing storage; and the only input/output medium is paper tape, say, or possibly punched cards. To give a rough estimate, if there is sufficient core storage for between 8,000 and 16,000 single-address instructions, and a paper tape only configuration, then single-pass compilation is highly advisable, because it minimises the amount of paper tape output which is required. Thirdly, on a configuration with sufficient main core store or backing store to hold the intermediate output, single-pass compilation can give extremely high speeds, probably higher than any other possible method. High speed of translation, together with compactness of the compiler, is particularly valuable in a multiaccess environment.

In order to achieve single-pass compilation, the language must possess certain properties. In fact, it should be a language in which the compiler knows most of what it needs to know at the time when it needs to know it. It would not be recommended for a language which allowed the programmer to put all his declarations at the end of the programme, or to omit them altogether. For such a language, it would definitely be recommended to use a "one-and-a-half-pass system", which involves a preliminary scan of the source programme in order to collect all the information on identifiers that is required, and in which the main part of the compilation is performed in a second scan of the source programme. The one-and-a-half-pass system has been used by Dijkstra in his pioneering implementation of ALGOL 60 for the X.1. The solution adopted by Elliott on the 803 and 503 was to make minor restrictions to the Algol language which made it possible to be sure that practically everything is declared before it is used. For example, all procedures declared in the head of the block must follow all other declarations in the head of that block, so that when an identifier occurs in a procedure statement, the compiler knows whether that identifier is in fact a local identifier of the block or global to the block. So single-pass compilation is feasible only for a language which behaves itself according to certain rules.

The organisation of the translator is defined by the way in which the whole task being performed can be split up into separate tasks which can be tackled independently; for this is a very important feature of any major computer program. For a multi-pass translator, the job naturally splits itself up into the implementation of the separate passes; but for a single-pass translator, the split-up of tasks is quite orthogonal to that of the multi-pass translator. In fact, the translator is split into independent "compiling routines", each of which deals with one of the syntactic categories of the language. In an ALGOL 60 translator there would be about sixty such routines, each of which is capable of compiling a particular class of texts, and each named after the class of texts which it compiles: for instance, "compile statement", "compile arithmetic expression", "compile Boolean expression", "compile for statement". All these procedures are written separately, but of course they are permitted to enter each other as subprocedures. In practice, they use each other to compile the sub-components of the syntactic construction which they are responsible for compiling as a whole.

In greater detail, there are two fundamental utilities, "advance" and "issue" which look after the input of source text and the output of object codes; in between these, there exist procedures for compiling various syntactic units. "Advance" is responsible for inputting and assembling the next basic unit of the programme. It has two buffers: one is a single-character buffer which is internal to it; and the second one is a single-word buffer, which is used to transmit a unit of source program to the rest of the translator. The basic task of "advance" is to read through the next few characters of the source programme and assemble them together into a single "word" (for example, a basic word, a combination of symbols like "colon equals", an identifier, or a number); and having assembled this information together, it usually looks it up in a dictionary of basic symbols and current identifiers. Thus, for each source program word which is input, certain basic information is made available - for instance, whether it is an identifier; and if it is an identifier, what type it is, and what address has been allocated to it. The various compiling procedures will frequently enter the advance routine to step the source programme through the tape reader, to deliver the next unit of compilation, and to place it in the single-word buffer.

At the other end of the operation, the actual issuing of the intermediate code is done by a set of issuing routines, which deal with object code instructions and addresses. The issue routines take the address part of the instruction which they are issuing direct from the information supplied by "advance". Consequently, the compiling routines never have to worry what the actual addresses associated with the identifiers are. As well as issuing the actual instructions of object code, the issue routines are also available to issue instructions to the programme loader. For example, the instruction "issue nested jump" tells the loader to do something, as well as causing an actual instruction to be placed in the object programme.

Each of the compiling procedures has to conform to certain conventions. For example, each of them expects to be entered with the first word of the text it has to deal with in the buffer; and each of them is designed to exit with the buffer containing the first word which is <u>not</u> part of the syntactic category for which it is responsible. We take as an example, the procedure "compile statement", which is given in figure 1, which has been taken direct from the program documentation of the Elliott 803 and 503 ALGOL compilers. It begins with the name of the procedure (line 1) which indicates that the procedure is designed to compile statements, and, just for the benefit of the person who is putting this procedure into machine code, we mention that it is recursive and that he had better push down the link. The first test that is made is, "Is the word in the buffer a label?", and that is tested by the first if clause, (line 6). The word label here may be regarded as a Boolean procedure which looks at certain markers made available by "advance", and delivers the value true or false in accordance with whether those markers indicate that the word is a label or not. If it is a label, then we check in the Elliott implementation whether the block level of the label is the same as the current block level. This test in Elliott ALGOL prevents the programmer from jumping into the middle of a block, which is not permitted in the language.

We now come to our "issue" instruction, which causes a jump instruction to be placed in a jump table which is being accumulated by the loader. For every label that is declared, a location is allocated to hold an instruction which is going to be a jump to the place where that label has been prefixed to a statement, and "issue a label" issues an instruction code to the loader to place a jump to this part of the code in the position of the jump table which has been allocated to the given label. Next, we need to check that the next symbol on the source programme is a colon. This is done by a routine of which we call "check", which checks that the content of the buffer is a colon. Finally, we do another advance to read the next word again into the buffer, and go back to the beginning, because we are still expecting a statement from the source text. Having gone back to the beginning, of course, we can deal with repeated labels on the same statement. If, however, the first word of the statement being compiled is not a label, then there are a great many other things that it could be. For instance, it could be a procedure identifier. If it is a procedure identifier, there are still two alternatives: it is either a type procedure identifier, occurring on the left-hand side of an assignment statement; or it is an ordinary procedure identifier, introducing a procedure statement. The discrimination between those two cases is made by the conditional statement beginning on line 16. Notice that when we enter, for instance, compile assignment statement, the first word of the whole statement is still in the buffer, and therefore it is still in the buffer after entry. This satisfies the rule for communication between the compiling routines.

The next line (line 19) determines whether to compile an assignment by testing if the buffer contains an identifier for a variable. Lines 20 to 25 use the utility procedure "cs", which stands for "current symbol" and takes a string parameter. It yields the value <u>true</u> if the content of the buffer is equal to the parameter, <u>false</u> otherwise. The interpretation of these lines is obvious; however, it is worth noting that their basic simplicity and efficiency is due to the fortunate characteristic of ALGOL, that the syntactic class of a portion of text is nearly always obvious from its first symbol. This characteristic is not shared by FORTRAN.

Lines 25 to 32 illustrate one of the few cases where this is not so, and one of the ad hoc techniques which can be used to get round the difficulty. If the current symbol is begin, we do not know whether this introduces a block or a compound statement. So after ignoring one or more possible comments, we examine the next symbol to see if it is a declarator. If it is, we compile a block; otherwise, we compile the compound tail. Finally, the case of the empty statement is dealt with simply by exiting from the procedure, without having done anything.

To illustrate the mutual dependence of procedures, we now consider the procedure "compile compound tail". This procedure is recursive, since it both enters and is entered from "compile statement". Note that the procedure has been entered with the first symbol of the compound tail in the advance buffer. A "compound tail" is defined as a sequence of statements separated by semi-colons and terminating with the symbol "end". So when we are trying to compile a compound tail, we first of all compile its first statement. The first word of the compound tail is in fact in the buffer, and this is also the first word of the first statement, and so the entry satisfies the rules. Having compiled the statement, the buffer will contain the first symbol which is not in fact part of the statement, i.e., the semi-colon, which terminates it, or possibly the <u>end</u> which terminates the entire compound tail. If in fact it is a semi-colon, (line 6) then we advance again to read the symbol after the semi-colon. If it is a comment, then again we have to ignore till the following semi-colon and then go back again to read the symbol after the semi-colon which terminates the comment. The test is repeated, in case several comments have been written in a row. As soon as we encounter a symbol which is not <u>comment</u>, we go right back to the beginning again to compile the next statement of the compound tail. If the symbol following a statement of a compound tail is not a semi-colon, then it must be an <u>end</u>. There is no other alternative for it. So all we have to do if the current symbol is not a semi-colon (line 12) is to check that it is, in fact, an <u>end</u>. Finally, we have to ignore the comment which Algol permits after the <u>end</u>, and that is done by line 13.

The first point of interest is that "compile compound tail" contains a call of the procedure "compile statement", which we have already analysed, so that this is a simple case of a procedure to compile a syntactic unit (i.e. a statement) calling a procedure to compile a sub-unit (i.e. a compound tail), which in its turn calls the original procedure again. This is a very characteristic feature of the organisation of these compiling routines. In fact, the calling structure of these recursive routines mirrors almost exactly that of the syntactic definitions of the Algol Report.

The syntactic definition of a statement is made in terms of an assignment statement, procedure statement, for statement, conditional statement, block, and compound statement; and these are exactly the procedures which the procedure "compile statement" calls. Similarly, the syntactic definition for a compound tail shows that a compound tail is defined in terms of statement; correspondingly, we have "compile statement" called from within "compile compound tail". For this reason, the compiling technique described here is sometimes known as "syntax-directed". It differs from other syntax-directed compilers in two respects: firstly, it is completely single pass, and secondly the syntactic structure of the language is actually embodied in the machine code instructions which analyse and translate the language, rather than in a "syntax table", which is used interpretively to analyse the language. Because everything is done explicitly in machine code, rather than by looking up in a table, very high speeds of analysis and translation can be achieved. However, it is not recommended to use recursion unnecessarily, when simple iteration would suffice. Thus "compile compound tail" uses a conventional loop to deal with its component statements, although the ALGOL definition uses direct recursion.

Another point to notice is that in compiling a syntactic unit the procedure to compile it looks after every aspect of the compilation, the syntactic analysis, the diagnostics, checking, and the issuing of compiled code; and these aspects of the job are not split up into separate passes as they would be in a multi-pass compiler.

The account given above has concentrated on the broad outlines of single-pass compilation, without dealing with any of the minor difficulties arising in practice as a result of incomplete knowledge at certain stages in compilation. There are a number of places, even in a well-structured language like Algol, where it is just impossible to know exactly what the object code is going to look like, or exactly where it is going to go in the store, until the entire process of compilation is completed. There are two well-known areas in which this is obvious: the first is forward jumps to labels; and the second is the final position in store of the work space of the programme, the constants, and the code itself. The point here is that we want to be able to separate in the store of the computer at one time the areas allocated to the work space of each of the blocks which we compile. We wish to separate this, in its turn, from the constants which are being used, and this, also from the code which is being produced. Since we do not know until we have finished compiling the programme how many constants there are going to be, how many words of work space there are going to be, or how long the code is going to be, we cannot make a final allocation of machine addresses to these things during compilation.

The solution to both these problems and other problems of a similar nature is usually very obvious. In each individual case, there will be found some ad hoc technique which is suitable; this will usually involve some elaboration of the structure of the intermediate code, and the inclusion in it of certain types of information, which lead to some appropriate action when the intermediate code is being loaded. For example, the problem of forward jumps can be dealt with by a technique of allocating a separate table with an entry for each label. A go to statement referencing the label is compiled as a jump to the table entry itself; and when the label is encountered by the translator, it issues an instruction to the loader to place a jump to the current program position in the corresponding table entry. Thus at run time, each jump to a label is executed as two jumps, one to the table entry, and one to the destination. The overhead of object time and space is quite small, but this particular technique is not recommended for indiscriminate use.

In designing the expedients to be adopted to solve the minor difficulties, it is a great help if the intermediate pass produced by the translator can be read in backwards by the loader. This will ensure that any information which is not known early enough at time of compilation, will be known to the loader ahead of the requirement; so that the loader can always produce the final absolute machine code without any backtracking or subsequent filling in. As an example of the usefulness of backward loading, we take the problem of jumping in conditional and for statements. These can be dealt with by a technique known as "nested jumping", which does not involve any of the run time overheads of a jump table. A conditional construction in Algol takes the form: <u>if</u> ..... <u>then</u> ..... <u>else</u> ..... The condition is placed between an <u>if</u> and a <u>then</u>, and obviously the compiler first constructs code to evaluate the condition. Next, corresponding to the <u>then</u>, it wishes to issue a conditional jump instruction which will jump to the <u>else</u> if the condition is <u>false</u>, and will allow control to pass sequentially if the condition is <u>true</u>. But of course the compiler does not know at this stage how much code is going to be generated by the limb of the conditional which follows the <u>then</u>. It therefore issues a specific instruction called a "nested jump", which tells the loader to fill in the destination of the jump from information that will be given later. If there is no <u>else</u>, the end of the conditional is indicated by the end of the statement, and we issue an instruction called "answer nested jump" which tells the loader that "This is the place where you were supposed to jump to when I last told you that you were going to jump somewhere". The loader then will take note of those instructions and behave in the manner instructed. Now the fact that the intermediate code which has been produced by the translator is read backwards into the store of the computer, means that the loader will have got the answer to the nested jump before the question has even been asked, so that at all times it knows the destination of a jump at the time when the jump is loaded into store.

The reason why it is called a "nested jump" is as follows: the text in between the <u>then</u> and the <u>else</u> may itself contain conditional constructions, and therefore it is quite possible, if there is another <u>then</u> in between, and that there will be issued yet another nested jump instruction and another answer. Consequently, there are two nested jumps, and two answers. However, they are properly nested, because the innermost nested jump has got to be linked up with the innermost answer, and the outermost one has got to be linked up with the outermost answer, in the same way as nests of brackets are linked up in arithmetic expressions. This means that the loader has got to maintain a stack of these addresses given by the answers; whenever it reads a nested answer it puts the address of the most recently input instruction into the stack, and increments the stack pointer. Whenever it reaches the nested jump to which this was the answer, it takes the address concerned out of the stack, puts it into the required conditional jump instruction, and decrements the stack pointer.

The question now arises: "How does the loader know how long to make this stack?". In fact, there are several other stacks which the loader has got to maintain, for example, a constants stack, and a stack for the work space of each independent block or procedure. Since the intermediate code is read in backwards, this presents no problem. The compiler can keep account of the maximum that is ever reached by any of these stacks, for example, the maximum depth of nesting that is ever reached by the stack of nested jumps; and at the end of compilation it can output a number indicating the maximum depth of nesting which has ever been encountered in the given programme. Since the intermediate code is read in backwards, the loader will read this information first of all, and can therefore allocate exactly the right amount of space to hold each of the stacks.

In a case like this, it would have been perfectly acceptable to avoid the complexity of counting the maximum depth of nesting, by setting an arbitrary limit of say 64 on how many _thens_ are permitted before reaching an _else_. In fact it is possible to construct an Algol compiler without making such arbitrary restrictions. In the Elliott 803 ALGOL compiler there is virtually no table that can be exhausted before the store is full anyway. At compile time a double-stack system is maintained, and practically everything is put in one of the stacks; when the two stack pointers meet, then there is no space being wasted anywhere in the store of the computer. On the 803 it is very uncommon to run out of space at compile time. There is an "error" number which indicates inadequacy of space, and is interpreted as "Programme is too long or complicated to be translated". This is the error message that appears when the store is completely full of nested information, dictioneries, and so on. It was quite difficult to construct a test case to see that this error was properly detected. In the end, it was done pressing the run-out button of a Creed teleprinter on the open-bracket symbol, thereby making a tape containing a long sequence of open brackets. This was presented to the translator to see how far it would read. In the first attempt, there were not enough brackets. At the second attempt, the "error" was correctly detected after about four hundred consecutive open brackets. As far as I know, this error number has never been given in normal use of the compiler.

Questions

Q. It does not seem necessary to maintain a jump-table in order to solve the forward jump problem. It is sufficient to place in each compiled jump instruction the address of the previous jump to the same destination. The loader can then chain through all jumps to a given label, and fill in the addresses accordingly.

A. I think that is a technique which is equally suitable for solving the forward label problem.

Q. Yes, and also the conditional jumps as well.

A. It will solve that one as well, but of course the nesting mechanism is slightly simpler in this case, since for each nested jump, there is only one answer. You only ever have to fill in one address, and therefore nesting technique takes advantage of the circumstances of the case;

this is characteristic of all good ad hoc programming techniques.

Q. Would you say that the techniques required to get over the individual difficulties would be very machine-dependent, because it is probably going to involve directions to the loader?

A. If the loader is machine oriented, then you are right, but in our case the loader was designed specifically to load Algol programmes.

Q. Isn't one taking a grave risk in any system which is designed to input the object programme backwards? It is probably all right on quite a lot of paper tape, or even punches, but very likely to go wrong if you have a system of only punched cards, and on many handlers it is not possible to read the magnetic tapes backwards. And, in fact, it is quite possible to avoid backward reading by the loader.

A. Of course, but backward reading is quite a useful technique for the particular circumstances in which it was employed. As I mentioned at the beginning, the techniques described are oriented towards certain kinds of configurations, certain kinds of compiling problem, and certain kinds of language. I would not advocate it on every computer or for every language. For example, the Gier Algol is a nine-pass system, and that is dictated by the fact that they have got a good fast drum and only a thousand words of core store; and for that machine this obviously is the best way to organise the compiler.

Q. You have got the routines here expressed in Algol notation - it may be pure Algol, I am not quite sure. Did this come before or after the machine code routine was produced?

A. Before.

Supplementary Question

Q. So that it was written out like this beforehand as a specification from which to code.

A. I would not necessarily say in all cases, but certainly for the recursive parts of it. The bits like "advance" and "issue" I think were written in machine code first. They are a lot more scrappy, and much less elegant from the point of view of documentation than the compiling routines.

Q. And was this found to be a successful way of expressing the compiler?

A. Yes. It is relatively easy to find your way through what is going on. For maintenance purposes, of course, you need annotated programme sheets as well. But a syntax-directed compiler organisation means that you only have to know the syntax of Algol to be able to find out where any particular fault is likely to have occurred.

Q. How do you define the difference between a half pass and a whole pass?

A. A half pass consists only of input, without output of intermediate code. It is a pre-scan of the source code, collecting up information, and the source code is input a second time to do the actual compilation.

Q. But this is a whole pass. You actually have to read the whole text.

A. Yes. You have to read the whole text in, but you do not have to output any of it. A real two-pass system would involve two sets of intermediate code instead of only one.

```
procedure  compile statement;
comment    this is a recursive procedure so link is pushed
           down at begin and popped up at end;
begin
X: if label then
begin      check (block level of label = current block level);
           comment check label declared in inner-most block;
           issue label;
           advance;
           check (':');
           advance;
           go to X
end
else if procedure identifier then
     begin if type procedure then compile assignment statement
           else compile procedure entry
     end
else if variable identifier then compile assignment statement
else if cs ('go to') then compile go to statement
else if cs ('read') then compile read statement
else if cs ('print') then compile print statement
else if cs ('for') then compile for statement
else if cs ('if') then compile conditional statement
else if cs ('begin') then
```

```
A: begin advance;
        if cs ('comment') then
        begin ignore till (';')
        go to A
        end
     else if declarator then compile block;
     else compile compound tail
  end block or compound statement
end compile statement;

procedure  compile compound tail;
comment    this is a recursive procedure so link is pushed
           down at begin and popped up at end;
begin start:  compile statement
     if cs (':') then begin L: advance;
                          if cs ('comment') then
                             begin ignore till (';');
                                go to L
                             end;
                          go to start
                       end;
     check ('end');
     ignore till 3 (';', 'else', 'end')
end
```