Management of block-structured programs

T.A. Marsland

Computing Science Department,
University of Alberta,
Edmonton T6G 2H1
Canada

DRAFT for Software - Practice and Experience
submitted: March 1981.
returned: March 1984.
revised: June 1984.

ABSTRACT

Experience with a technique for compiling subsets of the procedural elements of AlgolW programs is described. The method does not require modifications to the compiler, but does presume the existence of a random access file system. Complete management of the user's source and object files is provided by supplementary directory and environment files. Because of its nature, the basic method should be equally applicable to other block-structured languages, such as Pascal, which also require recompilation of the whole program whenever a small change is made.

Acknowledgements

The separate compilation technique described here has been developed over the past decade. Several people have revised the support programs including: Wayne Chelak, Grant Crawford, Ron Kostuik, and Earl Culham. More recently Terry Crocker and Murray Campbell developed a better way of generating the environment.

1. Introduction

In many block-structured languages, like Algol and Pascal, the support facilities are such that individual procedural components of a program cannot be compiled. Non block-structured languages do not suffer this problem. For example the scope of a variable in the 'C' language is the file in which it is declared. [1] A large program may thus be distributed over many files; alterations to part of the program then only require recompilation of the file with the altered code. In 'C', program-wide global variables are made available by a method analogous to the Common statement in Fortran. In Algol, on the other hand, a minor change to the source code usually requires the subsequent recompilation of the whole program. With small programs this is of no great consequence, but with increasing size the extra processing time may be substantial. The related language Pascal suffers from similar difficulties. [2]

2. Procedural Element Recompilation

Both AlgolW [3] and Pascal/VS [4] contain, as part of the language specification, a facility for independent compilation of those procedures which communicate solely via their parameter list. Although it is entirely possible to write complete programs which enforce this communication convention, it is rarely done. The problem is especially acute with large programs, containing many procedures, that manipulate complex data structures. Prohibitively long parameter lists for procedures are burdensome, especially in the case where a subprocedure is spawned simply because the amount of code generated exceeds some arbitrary size restriction (e.g. 8K bytes of code for AlgolW and Pascal/VS). Of course, if all procedures were required to communicate via parameters, there would be little need for global variables, which in turn means that the whole concept of block-structure and variable scope is lost. Alternative solutions to this maintenance problem have been proposed, [2, 5] but typically these entail modifications to the compiler itself. The scheme described here, however, requires no such support, but a programming convention or discipline is imposed, to ensure a logical consistency between the source program and object code modules.

In languages which allow the programmer's view of the procedure name to be an arbitrary sequence alpha-numeric characters, one other small difficulty has to be faced. An association must be made between the user's "external name" for the procedure and the compiler's "internal name". For practical reasons the compiler's view of the procedure name is as a short sequence of characters, e.g. eight. These internal names must be unique, and for AlgolW take the form AWXSCnnn, where nnn represents a three digit integer which in some sense measures the relative position of the corresponding procedure in the source program. There is nothing special about the prefix AWXSC; it could equally well be the first five characters of the external or programmer's name for the procedure. It was chosen perhaps for simplicity and for easy differentiation from AlgolW run time library routines, and from other system wide functions.

Those procedures which are not dependent on global variables, and so can be used by any application program, also need a unique internal name. In AlgolW this is achieved by taking the first five characters of the procedure name (padded with '#' as necessary) and appending '001' to form an eight character internal name. To avoid ambiguity some care must be taken in the choice of procedure names. Even though independent procedures cannot access global variables/functions directly, they can access other independent procedures.

In the general case, for procedures that manipulate global variables, a separate compilation capability becomes desirable whenever the cost of compiling a single typical procedure is small (less than 5%) compared to the cost of recompiling the whole program. The facility is essential for all large programs, those which cannot be compiled as a whole because of compiler design limitations. For one version of AlgolW, [6] executing under the Michigan Terminal System (MTS), [7] this was the case for any program with more than about 900 occurrences of 'BEGIN', 'PROCEDURE' and 'FOR', almost independent of application. The probability of this happening rises dramatically when the number of statements in the program exceeds 3000, but can occur at any time for a variety of reasons. This then is one definition of a large program, even though it would be modest by commercial standards.

In order to overcome these restrictions on program size, and also to reduce computational requirements for re-compilations, I have developed a source program management scheme which draws on the notion of a "working program environment". The basic facility provides methods for updating the source and object modules and for producing the working environment. All these components have been drawn together into a single program, hereafter referred to as the Manager, making the whole maintenance process automatic, eliminating the possibility for error and significantly reducing the need for the user to be familiar with the finer details of the process. A general outline and more specific information about the Manager's requirements can be found in a report. [8]

It should be noted that the Manager relies heavily on the random access properties of the MTS file system. [9] Of particular importance is the use of line numbers to ensure that parts of one file may be overlaid by another. This is done by making

use of the indexed read/write capabilities of MTS line files, so updates may be made in an arbitrary non-sequential manner. Extension of our technique to systems which support only sequential files may therefore be difficult or expensive.

To simplify the management process a naming convention for the files accessed was established. Thus the "root name" for the source file of the user's AlgolW program is shared by the maintenance files produced by the Manager. If the selected root name is 'prog', then the user's source file must be named 'prog.ALG', and the Manager will initially produce (and subsequently presume exists) three permanent files 'prog.OBJ', containing executable object modules, 'prog.ENV', to hold the skeletal structure of the AlgolW program, and 'prog.DIR', to show the update status and the names and locations of the procedures. In addition, four temporary files are created to hold error messages, source listing, compilation environment and object codes.

3. Environment Creation

An important feature of the Manager is the generation of an AlgolW compilation environment. To do this, certain AlgolW reserved words are recognized using the state transition table presented in our report. [8] The major features of the environment are:

(a) Nested (bottom level) procedures are converted to dummy procedures; i.e. the procedure heading and parameters (if any) remain, but the procedure body is replaced by a suitable null argument. For example, a proper procedure would have the body replaced by a semicolon, whereas a function procedure would need an appropriate expression type, say TRUE for a logical function.

(b) A block (procedure) containing nested procedures retains all its declarations and its spanning BEGIN-END, though the (procedure) body itself is either omitted or, for a function, is replaced by a suitable expression.

(c) The outer block declarations (globals) are included, as well as the spanning BEGIN-END.

(d) Global labels which are accessed from internal procedures must be included by hand, giving another reason such labels are undesirable.

(e) Line numbers for the working environment correspond exactly to the those of the original source.

The environment thus created is a syntactically correct AlgolW program which, when compiled, produces object modules with a one to one correspondence to the original program's object modules (though of course the program bodies are effectively null programs). Consider the sample program of Figure 1 and its corresponding compilation environment of Figure 2. Figure 1 illustrates a program with nested procedures and functions, and has been primarily designed to show the form of the working environment, Figure 2. The important points to note are the correspondence between the line-numbers of the sample program and its environment, and the form of the null program body that is provided for each function/procedure.

```
 1     BEGIN   COMMENT contrived purely for illustration;
 2
 3     PROCEDURE FIRST;
 4     BEGIN
 5        WRITE(" This is a Factorial program!");
 6        FACTORIAL;
 7     END;
 9     PROCEDURE FACTORIAL;
10     BEGIN                              COMMENT an outer procedure;
11        INTEGER NUMBER, SPARE;
11.5
12        LOGICAL PROCEDURE VALID;
13        BEGIN                           COMMENT a nested procedure;
14           IF NUMBER > 12 OR NUMBER < 0
15           THEN FALSE ELSE TRUE
16        END;
17
18        WRITE(" Input a number!"); WRITE(" ");
20        READ(NUMBER);
21
22        IF VALID
23        THEN WRITE(" Number was", NUMBER, "Factorial is",
24                   PHASE1(NUMBER))
26        ELSE WRITE(" This number is inappropriate!");
27     END;
29     INTEGER PROCEDURE PHASE1(INTEGER VALUE NUMBER);
30     BEGIN
31        INTEGER PROCEDURE PHASE2(INTEGER VALUE NUMBER);
32        BEGIN
33           IF NUMBER = 0 THEN 1
34           ELSE NUMBER * PHASE1(NUMBER-1)
35        END;
36
37        IF NUMBER = 0 THEN 1
38        ELSE NUMBER * PHASE2(NUMBER-1)
39     END;
40
41        COMMENT   Main program ;
42        FIRST;
43     END.
```

Figure 1: prog.ALG, a sample program with nested procedures.

```
 1    BEGIN   COMMENT contrived purely for illustration;
 2
 3    PROCEDURE FIRST; ;
 9    PROCEDURE FACTORIAL;
10    BEGIN                              COMMENT an outer procedure;
11       INTEGER NUMBER, SPARE;
11.5
12       LOGICAL PROCEDURE VALID; TRUE;
27    END;
29    INTEGER PROCEDURE PHASE1(INTEGER VALUE NUMBER);
30    BEGIN
31       INTEGER PROCEDURE PHASE2(INTEGER VALUE NUMBER); 0;
39     0 END;
43    END.
```

Figure 2: prog.ENV, a working environment for the sample program.

The environment is created directly from a legal version of the user's AlgolW source program. The creation process consists of a subroutine which is called with a parameter giving the address of a table into which certain data is to be placed. This table will be 16(N+1) bytes long, where N is a count of the explicit procedures in the AlgolW program. Information returned has the following format:

Procedure Name (8 bytes)	First Line (4 bytes)	Last Line (4 bytes)

The "Procedure Name" is an eight character, left justified version of the corresponding procedure's external name. It is either truncated, or padded on the right with blanks to get the 8-byte length. The definition of the "First-Line" field differs depending upon whether the procedure is at the bottom level or not. For nested (bottom level) procedures, this field is the line-number of the procedure heading, whereas for "outer" procedures it is the line-number of the first available source line after the declarations. The "Last Line" field is always the line-number of the closing END for that procedure.

The file 'prog.DIR' is a directory and each entry has five components:

(a). The external and internal names of the procedure.

(b). The starting and ending line numbers of the object code for the procedure.

(c). A count of the times the procedure has been modified.

(d). The date and time of the last update.

(e). The starting and ending line numbers of the source code for the procedure.

A directory of the type shown in Figure 3 is produced by the Manager from information extracted from the table entries formed during the environment creation. Note that there is one entry for every procedure in the program, including one for the outermost block, '(MAIN)^^'.

external	internal	lines object		mod	last update date	time	lines source	
(MAIN)	AWXSC001	5.0	11.0	1	JUL 27,	...:26	39.1	43.0
FIRST	AWXSC002	40.0	46.0	1	JUL 27,	...:26	3.0	7.0
FACTORIA	AWXSC003	27.0	34.0	2	JUL 28,	...:45	16.1	27.0
VALID	AWXSC004	34.0	40.0	1	JUL 27,	...:26	12.0	16.0
PHASE1	AWXSC005	11.0	19.0	1	JUL 27,	...:26	35.1	39.0
PHASE2	AWXSC006	19.0	27.0	1	JUL 27,	...:26	31.0	35.0

Figure 3: prog.DIR, a directory file for the sample program.

The information in the directory is used to maintain the source and object files in the following way. The environment is copied to a temporary file, which in turn is overlaid by the contents of the selected procedures. This subset of the program is then compiled. The object file produced is scanned for the selected modules and these are extracted, and the directory again used to delete the old object modules prior to their replacement. In our system it is possible to do this replacement with an in situ update, so there is no need to change the line number range associated with the object modules. The modification count and the date and time to the update are finally altered in the directory.

4. Programming Conventions

To avoid some of the hidden features of our compiler it was necessary to program in a disciplined way. The primary purpose of the following conventions was to ensure a one to one correspondence between the procedures generated by the compiler and those declared by the user.

(a) All external (independent) procedures must be declared as globals

This restriction is not severe, since there is no disadvantage in making the external procedures global.

(b) Any construct which causes the compiler to produce additional (implicit) procedures, transparent to the user, is not allowed.

The elimination of these implicit object modules is really a blessing in disguise. In our version of AlgolW, [6] these hidden procedures can be created two ways. Whenever a redundant BEGIN-END block is built, a hidden procedure is generated and is given the universal internal name of <BLOCK>. Construction of redundant BEGIN-END blocks is sometimes done deliberately within a large block, e.g., in order to overcome a restriction that no procedure generate more than 8K bytes of code. Rather than forcing the compiler to produce those procedures implicitly, it is better for the user to do so explicitly, so that one always knows where procedures are being invoked. More bizarrely, these <BLOCK> procedures are generated wherever a block expression is used. For example, the sequence:

```
WHILE (BEGIN <statements>; <expression> END)
DO <statement> ;
```

causes a procedure to be created and invoked whenever the while condition is tested. Our management system requires that such constructs be modified so that a function is used explicitly. In the past, an error in the AlgolW compiler itself lead to the unnecessary construction of hidden procedures for innocuous IF-THEN-ELSE statements. By a simple revision to the 1975 version of the compiler [6] that effect was eliminated.

(c) After initial environment creation some "hand-fixing" may be necessary.

Global labels which are accessed from inner procedures must be inserted by hand. If one forgets to do this, a later attempt to recompile a procedure that needs an external label will fail. At that time the label may be inserted into the environment, and the compilation repeated. Such labels should be uncommon, since they are normally used only for error exits. Of course the adherents to go-to-less programming will never have problems here.

(d) Variables and procedures must be added in a controlled way.

During product development, aside from revising source program logic, one is constantly creating and deleting variables. Under the management scheme described here, these alterations require some thought. Deletions of procedures or global variables can be done simply by converting them into dummy variables (place holders). Creation of new globals must be done carefully, using the following technique. Either new declarations are added to the end of the appropriate declaration list, or previously allocated dummy entries (place holders of the right type) are renamed to form operational variables. The introduction of procedures can similarly be done by reactivating previously deactivated procedures, or by appending new procedure declarations to the declaration list of the outer block. These techniques are adequate to preserve the established correspondence between internal and external procedure names, and to ensure that the allocated locations of existing globals is not altered. Clearly creation of new variables must be done to both the source program and the environment. However, alteration to the directory file will be necessary only if new procedures are generated. Although these entries can be created simply, it may be better to treat this case as a major revision, and recompile the whole program under the control of the Manager, so that all the support files will be re-initialized.

5. Large Program Problems and Costs

Usually, it is more expensive both to maintain and to execute large programs. Almost every operation is affected. For example, it costs more to delete or move lines in a large source file, because the line directory portion of the file undergoes more substantial revision. One solution to that problem is to distribute the source program over several small files. Although some cost savings are possible by this means, there is an increase in complexity of the management process.

5.1. Subfile hierarchy: The following proposal generalizes the Manager's functions, and reduces the cost of handling large files. The method involves the use of a Manager to manipulate a hierarchy of files, in a way which is transparent to the user. The benefits of this hierarchic structure are carried across to manipulations on object files, which will be structured in a way analogous to the source. In the simplest case, the original file 'prog.ALG' could be broken down into ten subfiles: 'prog.0.ALG' to 'prog.9.ALG'. For sanity one would want to follow a convention which requires that line numbers in a specific range appear only in a single subfile. If, for example, line numbers in the range 2000 to 2999 were restricted to the subfile 'prog.2.ALG', the editor could be invoked through a small filter (preprocessor) which, in turn, would have a simple criterion for switching from file to file automatically. Although this approach is clean and conceptually satisfying it is not a technical necessity, since all the information regarding the ranges of every procedure already exists in the file 'prog.DIR', so there are no practical difficulties here. As a matter of discipline, however, one would probably want to keep these subfiles disjoint and non-overlapping.

6. Manager Efficiency

The Manager was originally developed to support a program made up of 109 AlgolW procedures (average size 40 statements), and a few small independent Fortran and PL360 procedures for system dependent functions. In the following discussion, the cost of compiling these independent components was ignored. Compilation of this program as a whole under our AlgolW compiler was not possible, because of a block table overflow. By replacing six major procedures with dummy statements, so that the overflow condition was not violated, the program was handled and from this a lower bound on the compilation time was estimated at 16 seconds Amdahl V/6 time. Thus without the Manager the application would have been abandoned. On the other hand, the initialization phase to generate a compilation environment and a directory, to compile the original source in two parts, and to merge the resultant object modules into 'prog.OBJ' required 33.5 seconds under our facility. While this appeared to be expensive it provided a working program and was done only once; thereafter procedures were recompiled a few at a time. Recompiling and updating a single sample procedure, consisting of about 90 statements, required 2.85 seconds. Of this about 2.25 seconds was the overhead associated with the Manager, including opening files, compiling the environment and replacing a null program. In fact, for this application, direct compilation of the environment alone required 1.04 seconds. One might therefore deduce that at least 16/(2.85^-2.25)^=^26 procedures could be recompiled at one time, before a compilation of the whole program would be cheaper. The more detailed statistics of Table 1 support that view, although the estimate itself is dependent on procedure sizes.

An alternative viewpoint can be obtained from a smaller constructed application with 59 identical procedures, each of about 50 statements, which required 8.4 seconds to compile and generate object code. The initialization phase of building the

supplementary files, compiling the whole program and updating the object modules took 18.27 seconds. For this case the overhead was 2.09 seconds and since updating 10 procedures took 4.15 seconds, up to 84/(4.15ˆ-2.09)ˆ=ˆ40 procedures could be replaced at a time before the process became uneconomic. This same experiment was repeated with a second sample, a program of 110 procedures, each of 25 statements, and the results summarized with the other two examples in Table 1.

	Chess prog.	Sample 1	Sample 2
No. procedures	109	59	110
Ave. proc. size	90*	50	25
Initialization time (secs)	33.5	18.27	18.16
Stand alone compilation (secs)	16.0(est)	8.40	7.60
Overhead time (secs)	2.25	2.09	2.57
Time to update 5 procs. (secs)	4.90	3.05	2.91
Time to update 10 procs. (secs)	7.05	4.15	3.63
Est. max. no. updates	33	40	71

* Average for (1-10) procedures updated in this test.

Table 1: Separate compilation/replacement of procedures.

7. Conclusions

The Manager is a cost-effective and convenient way of maintaining large AlgolW programs. An equivalent system to handle Pascal could be built. The major change would be a new state transition table to accommodate the different nature of Pascal's block structure. Our approach is efficient, though dependent on the average procedure size. It is especially valuable if the program is still undergoing substantial revision to a subset of its procedural components. Since the overhead does not vary significantly with application, we assert that any AlgolW program that needs more than about 2.5 seconds of Amdahl V6 CPU time to compile and generate object modules is sufficiently large to warrant the use of a Manager. Our data shows that any program of more than 15 procedures or 800 statements can be handled more efficiently, reliably, and quickly than by any other locally existing support program. [10] The use of a directory to maintain not only the association between the user's name for procedures and the compiled name, but also relative locations of the various object modules, is responsible for this efficiency improvement.

A logical extension to the present scheme has also been described. It involves dividing a large source file into a set of subfiles, and thus benefit from the reduced cost of manipulating smaller files. A special filter to interface the user to the editor would have to be built.

REFERENCES

References

1. D.M. Ritchie and K. Thompson, "The UNIX time-sharing system," *CACM*, 17, pp. 365-375 (1974).
2. R.J. LeBlanc, "Extensions to PASCAL for separate compilation," *SIGPLAN Notices*, 13, 9, pp. 30-33 (1978).
3. Computing Centre Staff, *MTS Vol16: Algolw in MTS*, document R25.0881, University of Alberta, Edmonton (August 1981). 505pages.
4. IBM Staff, *Pascal/VS Language Reference Manual*, IBM Corporation, document SH20-6168-1 (1980). 172pages.
5. D.R. Hanson, "A simple technique for controlled communication among separately compiled modules," *Software - Practice and Experience*, 9, pp. 921-924 (1979).
6. Computer Centre Staff, *An AlgolW compiler*, Computer Centre, The University, Newcastle upon Tyne, U.K. (May 1975). 200pages.
7. D.W. Boettner and M.T. Alexander, "The Michigan Terminal System," *Proc. IEEE*, 63, 6, pp. 912-918 (1975).

8. T.A. Marsland, T. Crocker, and M. Campbell, “Partial compilation and support functions for block structure language programs,” TR78-6, Computing Science Dept., Univ. of Alberta (1978). 35pages.

9. G.C. Pirkola, “A file system for a general purpose time sharing environment,” *Proc. IEEE,* 63, 6, pp. 918-924 (1975).

10. Computing Centre Staff, *Object File Manipulation,* document R15.0682, University of Alberta, Edmonton (1982). 126pages.