SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 17(7), 455–467 (JULY 1987)

Dynamic Compilation in the Unix Environment

M. K. CROWE
Department of Computing Science, Paisley College of Technology, High Street, Paisley, PA1 2BE, Scotland

SUMMARY

A system for dynamic compilation under the Unix operating system is described. The basis of the system is an incremental assembler that can be used statically or during program execution to insert or replace a module in an executable image. All cross-module references are via offsets into a run-time symbol table. All generated code is independent of its location or the location of the symbol table. The symbol table and all modules reside in memory segments compatible with the memory allocator malloc(). **The symbol table origin is maintained in a processor register. Library procedures allow the assembler (or C compiler) to be called to alter the currently executing program, or to place a stub function which acts as a trap, so that when the stub is invoked it causes a file to be dynamically compiled into the executing program to replace the stub with a bona fide procedure. This facilitates the construction of advanced interactive environments using native code. Some example applications, to Prolog and to incremental compilation, are considered.**

KEY WORDS Dynamic compilation Incremental compilation Unix Prolog

INTRODUCTION

This paper presents an approach to dynamic compilation and assembly which has been implemented under the Unix operating system and is proving useful in a variety of contexts. With this system, an executing program exists in memory as a set of independently loaded modules, communicating via a run-time symbol table. Code modules can be added or modified by the program itself.

Intermodule references have an additional layer of indirection, and so are slightly slower than with the normal Unix system.[1*] The system described in this paper is intended for applications where the replacement of a program module is a common occurrence. Such apparently unconventional applications are becoming more common, especially in rapid prototyping and development environments, and generally in the artificial intelligence field. Some examples are included in this paper.

The system has been implemented for the VAX under Unix, by modifying the C compiler and assembler. Some additional utilities have been supplied. The assembler performs all the functions normally associated with the loader (inserting binary files,

* UNIX is a trademark of AT&T Bell Laboratories.

0038–0644/87/070455–13$06.50

Received 10 November 1986
Revised 11 February 1987

extracting needed routines from archives), in addition to supporting mixtures of assembler and binary files. No changes to the Unix kernel have been made, and the system uses its own version of the C library.

DYNAMIC COMPILATION SYSTEMS

The usual system view of a user process is that it consists of a read-only program, constructed once by a compiler, a read/write data and stack area and some information private to the operating system. Read-only programs can be shared among many users.

Such a view is really only tenable for very simple application software. Many users require to be able to tailor their software for their own purposes, and for example many editors have a macro facility to enable such customization to be done on a per user basis. The command processors on many systems, including Unix, support per-user or per-process customization (shell variables, environment, search paths, aliases). The alteration of a macro or search path can make dramatic changes to the behaviour of the software, and is not far removed from the concept of dynamic compilation. The customization is achieved by tables which are interpreted by a common program.

In interpretive systems of the more usual sort, such as BASIC or Prolog, dynamic behaviour is commonplace, but is not considered strange because it is possible to distinguish between the interpreter itself, which is unchanging, and its data (the user's contribution), which is dynamic. But for the user, it is much the same to him whether his BASIC is compiled into machine code, or some compact version suited to interpretation: the former would probably be preferable for speed.

This is one area where dynamic compilation represents a natural progression from existing practice, and is of particular value in Prolog,[2] where the continued addition of rules to a system can make the system very slow, and compilation the more important. Some advanced language systems, such as ML,[3] already use dynamic compilation, but the new machine code is directly constructed by the program (in 'data' space). ML goes to a lot of trouble to do this. By contrast, the software described in this paper makes it a simple matter for any program to be made dynamic.

Other incremental compilation systems have been described,[4–6] which require substitution of procedures or even smaller units within a program being debugged. In general the insertion of breakpoints, or the monitoring of variables, in a compiled system will require code modification. In Reference 7 an incremental system is described where substitution occurs during development of a large project.

In all of these systems, the modifications being interactively made to programs are rigidly controlled by the environment, which ensures that the change is consistent with what has gone before. The system described in Reference 4 is closest in approach to the work described here, where it is envisaged that the dynamic compilation facilities will in practice also be subject to the control of a high-level software environment, such as a Prolog front-end or a structure editor.

The dynamic compiler described in this paper also acts well as an incremental compiler, as it allows the replacement or insertion of a module in an existing executable image file, without changing the existing code. This is not the same as dynamic compilation, as changes are made statically to an existing executable file, rather than dynamically to an executing program.

Overlay programs are another type of self-modifying program. The ETH Modula-2 compiler for the PDP11[8] provided a loader module as standard, which allowed groups of modules to share the same memory. In such systems, changes are made dynamically to an executing program, but the changes are specified in an overlay structure that is specified when the executable file is being created, rather than during execution of the program. Thus this type of modification is much more restrictive than that considered in this paper.

The Berkeley Unix loader ld[1] supports dynamic loading in the sense that a flag can be specified which causes the loader to use the symbol table of another program for the load. The new code and data are relocated and written out to a file which can then be read into the appropriate place in memory when required. This loader can of course be called from an executing program, and so can be seen as a general tool for dynamic compilation.

However, there are two main difficulties. First, there must be a base executable file for the dynamic load operation. This must contain code for calling the dynamic module, and must define any symbols it requires. Typically, the program will have been loaded in stages, with one or more base files being constructed during the load.

Secondly, position in memory of the new module must be specified to ld. If the new position immediately follows the old program, this is simple enough. However, nearly all programs under Unix use the memory allocator malloc() (it is used behind the scenes by the standard i/o library), so that the length of the original program is no guide. Instead, malloc must be asked for a piece of memory the right size for the new code, and then the loader must relocate it to that position. Thus the sequence of operations required for this system is as follows.

Berkeley loader used for dynamic compilation

1. Invoke the assembler behind the scenes to make a loadable version of the new module (in Unix this will have a file name of the form new.o).
2. Examine the header of new.o to determine its size, and malloc enough space for it. Suppose the space begins at location pos.
3. If the base file was called base, invoke the loader using the command ld -A base -T pos new.o to create a loadable version of new.o in the default output file a.out.
4. Copy a.out into memory starting at position pos.
5. Use the symbol table information in a.out to set up pointers to enable the new code to be called.

No assistance with the last step is given by the Berkeley system. The software described in this paper makes this process much easier.

In relation to the use of the memory allocator, it is interesting to note that the DICE incremental compiler[5] constrains all variables to live in the stack, so that global dynamic data are disallowed.

DESIGNING FOR DYNAMIC COMPILATION

From the above discussion, it would seem that a desirable system would allow the addition of a module (in some sense) to an executable file, or into an executing program, with much less fuss than under Berkeley Unix.

From the application point of view, the system described in this paper is very straightforward, as all that is required is to call a standard function assem(), with a file name as parameter. This incorporates the file contents (C or assembly code) into the running program and updates the symbol table automatically. The steps being taken on the application's behalf are as follows.

Steps for dynamic compilation

1. For a given a.out, and an assembler source file thing.s, perform as -A thing.s. This creates a file containing an updated symbol table and a module to be added to memory.
2. Examine the file header to determine the symbol table and code size, allocate space (using malloc()) for the symbol table and new code, and read them into memory.
3. Execute C initializations in the new module. Merge the symbol table with the one in the executing program.
4. Use the new symbol table. Discard (free()) the old symbol table and old code if any.

It can be seen that a considerable saving has taken place. Only one external program (as) need be used, although it is usually convenient to allow insertion of C code, which involves calling the C compiler instead.

Of course, in either system it may be that the new code contains calls to library functions not used before. In both systems it is possible to include archives in dynamic compilation.

Further possibilities should arise from the availability of the symbol table at run time. A program can consult its symbol table to advise on available functions. Some functions can be provided as stubs, so that the first time they are called, the code is dynamically loaded. This provides a simple way of attaining the sort of incremental environment that is standard with many interpretive debuggers.

MODULES

In this system, a module consists of the new code produced by an invocation of the assembler as, as a result of assembling source or binary code into a new or existing executable file. The module is set up in such a way that it looks as if it is in a space allocated by the Unix memory allocator malloc(). This means that a module can be replaced at run-time.

The process of adding a module to an executable file does not involve rewriting the file. At most the symbol table is rewritten to the file, in addition to the code for the new module. Thus the normal assembly process allows a given executable file to be augmented by a new module. This executable file is the file specified for the output of the assembler: if it does not exist, it will be created by the assembler, and will then contain a new symbol table and just one module. (The C compiler first deletes the output file unless explicitly told to use it as a base file.)

If an executable file is used as the input to as, it combines the modules contained in it, and adds the resulting single module to the given base file if any.

For dynamic compilation, the output of as can be restricted to the new symbol table and the code for the new module, by providing a flag to the assembler. In this case the symbol table shows which module is defined by the assembly. The new module can then be incorporated into an executing program, possibly replacing an existing module with the same name. This is done by copying the new code into a new piece of memory, using the standard Unix memory allocator routines malloc() and free().

A module may contain text and data (whether initialized or not). It is possible to have the uninitialized data stored separately, as in the usual Unix loaders; but it would be unacceptable for the uninitialized data area to be extended (and therefore, probably moved) every time a dynamic compilation took place, because all pointers to it would require to be updated. Ordinary incremental loading follows the same philosophy as dynamic loading in the system.

References within modules can be by ordinary displacement. If modules generate dynamic data (using malloc()), pointers remain valid during dynamic compilation. References between modules should be by references through the symbol table and, in this system, compilers obey this convention. If a module is recompiled, all its variables are reinitialized. Initialized pointer variables cannot be allowed in assembly code, so that compilers must ensure that initializations are performed explicitly. These compiler modifications have been made for C and for Ada, as described below.

THE SYMBOL TABLE

This occupies a single malloc'd region. Once a symbol is allocated a position in this region, it retains the same relative position. This ensures that all modules can be sure that references via the symbol table continue to work after a dynamic compilation action. The symbol table itself may move, and a register is reserved to point to it. This means that ordinary base register addressing can be used indirectly for intermodule references.

These simple-looking requirements lead to a very complex structure for the symbol table. It becomes convenient that symbol names are stored along with other symbol table information, as used to be the case in old versions of Unix. A variable length format makes the use of links to the next symbol attractive, particularly if symbols for the same module are chained together. Note that a symbol can start life as an undefined symbol, and then be defined by a particular module, so that such chains of symbols can wander arbitrarily through the symbol table.

The beginning of this chain of symbols (which starts with undefined symbols for convenience) is provided in a header for the symbol table, along with the symbol table size. Provision could be made for deleting symbols from the symbol table: this is problematic, however, since there is usually no good way of ensuring that a symbol is no longer referred to. The exception is on the rare occasions when a local symbol acquires a permanent symbol table position because of the explicit construction of a symbol table reference which uses it: such a symbol can be safely deleted when the module is deleted or recompiled. If symbol deletion were to be supported, a list of free space in the symbol table could also be provided by giving the start of the first such space in the symbol table header.

Additional information is stored in the symbol table for a symbol that is the name of a module: the actual size in bytes of the module (the malloc information at the start

of the module itself may well be rounded, say to the next power of 2), and the position in the symbol table of the next symbol.

Many of the incremental systems in the literature, such as those described in References 4–6, envisage extracting procedures from a database when required. Since a procedure should be extracted only once, a symbol table or equivalent records information about extracted units, and so these systems are seen to be similar to the one described here.

SUPPORT FOR LIBRARIES

As mentioned above, the use of standard libraries makes it desirable to allow load-as-needed archives and binary files. In this system, a binary file contains a symbol table that is used at run time: and once a symbol is given a position in the symbol table it keeps it. This ensures that code referencing the symbol need not be altered if the symbol itself is moved.

In this system, the natural sort of archive is one containing assembler source files together with a table of contents. The table of contents can be examined to see if any library elements should be added to the current program: if so, they are included. Since a module is defined as the code inserted by an invocation of the assembler, the result of processing the archive is to include any needed code in the current module.

However, to improve the speed of including library code, binary files are supported, both for individual files and library elements. These are typically quite short files, containing one or two functions. Since they are constructed by the assembler, they have the form of a module or group of modules. When one or more binary files are inserted into an executable image by a later pass of the assembler (usually when processing an archive), all such modules are combined to form part of the module being assembled. That is, they are compacted into a contiguous piece of memory allocated by the memory allocator.

This code compaction process involves updating the symbol table entries. The way that the symbols are chained together in the symbol table, and the information provided there about module size, is precisely what is needed so that a single pass of the inserted symbol table deals with the symbols one module at a time, enabling the adjusted values to be added to the current symbol table. At the same time, the value in the inserted symbol table can be replaced with a reference to the new symbol, for reasons which are now explained.

The real problem about inserting a binary file is that it was constructed without any knowledge of its future environment, i.e. that the inserted file has its symbols in symbol table positions that are different from the file into which it is being inserted. While rewriting the code, all references via the symbol table must be converted for the new symbol table positions, and this requires a simple relocation table. It also requires that all such intermodule references use a four-byte address within the symbol table. The new -r flag in the assembler ensures both these things.

The relocation table for incremental files is rather simpler than in the usual Unix system. All that is required is a list of addresses in the code where a 4-byte pointer into the symbol table occurs. During pass 2 of dealing with the inserted binary file, the code for each inserted module is copied to its new position, and each symbol table reference noted in the relocation table is replaced with the value found in the old symbol table. This is a very fast operation, as no symbol table look-up is required.

INITIALIZATION

The C language allows external and static data to be initialized to any constant expression including addresses of other objects. In this system, however, addresses of objects cannot be determined by the loader, so that initializations that require addresses must be performed explicitly when the program starts executing.

The C compiler has been modified for this system, and generates such initialization blocks, which are strung together by the dynamic assembler. If a module is deleted or replaced, the corresponding initialization section is bypassed.

If a module is dynamically replaced during execution of a program, its initialization section is performed when it is linked into place.

NEW SOFTWARE

As a result of all the above changes, the following standard Unix utilities required modification, and are effectively replaced in this system:[9]

(a) There is no equivalent of the Unix loader ld.

(b) The assembler as now handles symbol table references: @xyz denotes the position of symbol table entry for xyz relative to the start of the symbol table. as also accepts libraries and binary files as input. Relocation tables are different from the standard Unix format, and the result of assembly is the updating of a named executable file if it exists, or creating it if it does not exist.

(c) The C compiler cc generates relocation-free code for internal symbols, and references via the symbol table for external symbols. Register 6 always points to the symbol table origin in the present implementation.

(d) Some utilities, such as the symbol table analyser and the random archive utility, have been rewritten for the new system.

(e) The C library required a small number of modifications where pointers had been initialized or reserved registers used in assembly language routines. Also, the memory allocator is now initialized from the file header.

(f) Additional utilities were added to the C library to support dynamic compilation.

The last category includes a function which calls the assembler on its file parameter, so that the resulting code is loaded into the current program, as described above. It also includes a function which is given a symbol name and a file name. This defines a new module containing a function with the given name. The function is a stub which, the next time it is invoked, calls the assembler on the given file to replace itself with the result of assembling the file.

An example of a very simple self-modifying program (where the user can type in code in C for incorporation into the program) is shown in an appendix.

It is perhaps worth noting that a linked list involving static data in several modules will become invalid if one of the modules is recompiled. This problem affects C programs almost exclusively, as most high level languages, including Pascal and Ada, only allow addresses of objects on the heap. (In Modula-2 the SYSTEM function ADR() allows this restriction to be overcome.) It is considered, however, that similar problems can arise in C for linked lists of local variables, and that programmers who set up such structures in a dynamic program should take care when using such data structures. Implementations that tried to detect and fix such structures would be very slow.

RELATION TO PROLOG IMPLEMENTATION

In Reference 2 a way of translating a static Prolog program into a high level language such as Pascal is described. It uses the concept of parametric procedure calls, in which an actual parameter is a procedure call which is delayed until the corresponding formal parameter is called:

```
procedure unify(a,b:list; procedure call cont);
begin
  ...
  cont;
  ...
end;
```

Thus the translation of a static Prolog program can be translated into C in a fairly well-understood way. All that is needed are some low level functions to implement parametric procedure calls and the cut. Prolog rules for the same predicate can be gathered together to form a function in C, and backtracking consists of trying each alternative in turn. This approach is interesting in that it reverses the usual trend of treating procedures as data: here instead all facts and rules are coded into procedures.

However, the usual Prolog environment allows new rules and facts to be supplied interactively by the user, and here dynamic compilation facilities are of vital importance. With the system described in this paper, implementing such a Prolog environment is quite straightforward. The resulting Prolog compiler is currently under construction as a student project at Paisley College (David Gibson).

Some Prolog programs make extensive use of the retract and assert primitives to modify the rule base. Such programs are made more efficient if the predicates for which these facilities are used are handled in a slightly different way, with a version of the Prolog source stored together with the code in a linked list structure.

INCREMENTAL COMPILATION

Incremental compilers have been described and implemented now for a number of years. However, it is rare for incremental compilers for real programming languages to generate native code incrementally, for precisely the reasons highlighted in this paper.

In Reference 7, an approach to incremental compilation is described, which is now being combined with the techniques described in this paper to produce true incremental Ada to VAX machine code compilation. The Ada compiler used is the recently validated York University compiler[10] modified to support incremental compilation.

This work, like most incremental compilers, concentrated on the front end, with its interaction with the user, featuring editing operations interacting with the passes of the compiler. The user is given syntax and semantics assistance during editing, and the compiler ensures that its output is patched each time a partial edit occurs on the source code. This is done by synchronizing the compiler so that compilation can be restarted on the smallest possible code fragment surrounding the change made, and semantic checking and code generation take place on the smallest possible fragment surrounding that. The new code generated replaces the code previously generated for that fragment.

The output from the York compiler, as with most high-level language compilers on Unix, is assembler source code, which makes code replacement a simple matter of text editing. However, it means that following the carefully incremental alteration to the Ada source and compiler output, the Unix assembler needs to be called for the whole of the current module, and then the loader must reload all modules in the program. This is rather disappointing, and true incremental compilation, where a small alteration to the executable image results from a small change to the source program, would be much better.

To generate true native code in the usual Unix format, a great deal of additional relocation and symbol information would need to be stored with the executable file. This would then be processed after the binary patch had been inserted in the object code, so that all enclosing branch instructions had their displacements adjusted, all data references were modified, and so on.

With the facilities described in this paper, however, only the current module needs to be reassembled, and the code can be inserted into the executable image by a simple editing process (carried out by the assembler itself). This is possible since all data declared by the module are internal to the module, and all cross-module references are handled via the symbol table. A special version of the code generation phase of the York Ada compiler has been constructed with a view to using the techniques of this paper in the incremental Ada compiler.

This flexibility is of course paid for by the cost of these symbol table references, and by a slightly larger image file. Thus while a program is under incremental development, it will run about ten per cent slower. This is considered acceptable during testing: most programs under test have additional instrumenting code inserted for monitoring purposes in any case.

CONCLUSIONS

An approach to dynamic compilation has been described and implemented. It is useful in environments where recompilation of parts of the system is a frequent occurrence, or where an interactive environment could use native code rather than interpretive techniques. The system is available free of charge for educational or research purposes to any educational Unix licensee, though it will only be useful to sites running Berkeley Unix on a VAX.

ACKNOWLEDGEMENTS

It is a pleasure to acknowledge the helpful comments of staff and researchers at the College, notably those of Brian Lees, in preparing this paper. It was also a pleasure to receive and act on the helpful comments of the referees.

APPENDIX I: THE MODIFIED a.out.h HEADER FILE

```
/* INCREMENTAL ASSEMBLER/LOADER by Malcolm Crowe, August 1986

This file defines the data structures used by the system.

All code should be movable, and live in malloc'd regions.
Movability means that modules can be realloc'd or copied,
provided the symbol table entries are updated. The symbol table
itself is also movable: register 6 points to it during
execution. Displacement addressing can be used only within
modules: otherwise everything is via the symbol table.

Once an object is assigned a position in the symbol table, it
retains that position, which can be referred to in assembler as
@object. Thus the current address of the object is in
@object(r6).

The system includes a new C compiler and C library, and
assembler/loader and namelist utility. Additional library
routines for dynamic compilation can be found in libc/cap. The C
library lib.a is only included when explicitly mentioned by -lc.

The assembler/loader uses the output file as a base file if it
exists.

Additional flags:
cap -b file things...   use file as a base file.
as/as -A things..       code from base file is not copied.
      -m name things..  defines name for the module
      -M things..       requests a primitive load map
      -r things..       put relocation info in a.out file
as/nm -m [file ..]      module information only
*/

/*
 * Header prepended to incremental a.out file.
 */
struct i_exec {
        long    i_magic;        /* magic number 0407 */
unsigned long   i_size;         /* size of program */
        long    i_zero0;        /* must be zero */
unsigned long   i_bss;          /* size of uninitialised data */
        long    i_zero1;
unsigned long   i_entry;        /* entry point */
        long    i_relsiz;       /* incremental relocation table */
        long    i_zero2;
       /* stuff from here on is visible as IHDR->i_sympos etc */
        long    i_zero3;        /* abort */
        struct i_symtab *i_sympos; /* position of symbol table */
unsigned long   i_symsiz;       /* size of symbol table */
        int     i_minbrk;       /* copy of i_size */
        union overhead *i_nextf[28];   /* see malloc() */
};

extern  struct  i_exec  *i_hdr();
#define IHDR    i_hdr()

#define IMAGIC  0407    /* impure format */
#define IMODUL  0477    /* single module: nonexecutable */

/*
 * Macro to check the file has the right magic number and format
 */
#define IBADFMT(x)    (((x).i_magic!=IMAGIC  && \
                        (x).i_magic!=IMODUL) || \
        (x).i_zero0!=0  || (x).i_zero1!=0    || \
        (x).i_zero2!=0  || (x).i_zero3!=0)
```

```
/* Structure of symbol table header */

struct i_symtab {
        int i_syml;     /* offset to first symbol */
        int i_modl;     /* offset to first module */
        int i_symfree;  /* chain of deleted symbols */
};

/*
 * Format of a symbol table entry
 */
struct  i_nlist {
unsigned long   n_value;                /* value of this symbol  */
        short   n_desc;                 /* defining module */
unsigned char   n_type;                 /* type flag, see below */
        char    n_other;                /* unused */
        int     n_next;                 /* see below */
        union {
                char    *n_name;        /* in assembler */
                unsigned char name[1];  /* on disk: [1]=length */
        } n_un;
};

#ifndef N_UNDF
/*
 * Simple values for n_type.
 */
#define N_UNDF  0x0             /* undefined */
#define N_ABS   0x2             /* absolute */
#define N_TEXT  0x4             /* text */
#define N_DATA  0x6             /* data */
#define N_BSS   0x8             /* bss */
#define N_COMM  0x12            /* common */
#define N_FN    0x1f            /* file name symbol */
#define N_EXT   01              /* external bit, or'ed in */
#define N_TYPE  0x1e            /* mask for all the type bits */

#endif

/* n_next fields are used as follows. All symbols in the table
are chained together using this field. i_syml points to the
first symbol in the chain.  The order of symbols in the chain is
as follows:
        Undefined symbols ("module 0")
        Symbols defined in module 1
        ...
        Symbols defined in module n
The first symbol defined in any module has n_type N_FN.
Immediately following it in the symbol table is a 4 byte pointer
(relative to sympos) to the next symbol of type N_FN, i.e the
first symbol defined in the next module. */

/* Information about modules precedes type N_FN symbols: */

struct i_modinfo {
        unsigned int    m_size;   /* size of module */
        int     m_next;           /* next module info */
        struct i_nlist  m_name;   /* symbol of type N_FN */
};

/* Relocation info is located at position 32+i_size in the file,
and consists of i_relsiz/4 integers. Each is the position (less
32) of a symbol table reference (e.g. @thing(r6) ). If the -r
flag is specified all such addresses are guaranteed to be 4
bytes. */
```

APPENDIX II: A SIMPLE SELF-MODIFYING PROGRAM

```
/* Initially func(), defined in the separate file "f.c", is just
   the empty function. The user can supply new code for this
   file, or another file, to be recompiled. In this simple
   program the C library is not reexamined. */

#include <stdio.h>
#include "a.out.h"

char fname[256];

main()
{
        int r;

        for (;;) {      /* until user types ^C to stop */

                /* find out what user wants to do */
                getname();
                getfile();

                /* now fname has the name of a file to load in */

                r=assem(fname);
                if (r!=0) {
                        printf("Return code %d\n",r);
                        exit(-1);
                }

                /* that worked, so go call func */

                func();
        }
}

getname()
{
        int c; char *p=fname;

        printf("What file do you want to (re)compile?\n");
        while ((c=getchar())!='\n')
                *p++ = c;
        *p=0;
}
getfile()
{
        FILE  *f = fopen(fname,"w");
        int c;

        printf("Give contents of %s:\n",fname);
        while ( (c=getchar())>=0)
                putc(c,f);
        clearerr(stdin);
        fclose(f);
}
```

REFERENCES

1. W. Joy, *UNIX Programmer's Manual (4.2bsd)*, University of California, Berkeley, 1980.
2. J. Cohen, 'Describing Prolog by its interpretation and compilation', *Comm. ACM*, **28**, 1311–1325 (1985).
3. R. Milner, 'A proposal for standard ML', *ACM Symposium on Lisp and Functional Programming*, 184–197 (1984).
4. R. Medina-Mora and P. Feiler, 'An incremental programming environment', *IEEE Transactions on Software Engineering*, **7**, 472–482 (1981).

5. P. Fritzson, 'A systematic approach to advanced debugging through incremental compilation', *Proceedings of the Software Engineering Symposium on High-Level Debugging, ACM Software Engineering Notes,* **8**, 130–139 (1983).
6. R. Cook and I. Lee, 'DYMOS: a dynamic modification system', *Proceedings of the Software Engineering Symposium on High-Level Debugging, ACM Software Engineering Notes,* **8**, 201–202 (1983).
7. M. K. Crowe, C. Nicol, M. Hughes and D. Mackay, 'On converting a compiler into an incremental compiler', *ACM SIGPLAN Notices,* **20**, 14–22 (1985).
8. H. H. Naegeli, and A. Gorrengourt, *The Linker for Modula-2 on the PDP-11,* Institut fur Informatik, ETH Zurich, 1981.
9. J. Firth, C. H. Forsyth, L. Tsao, K. S. Walker and I. C. Wand, 'Facts and Figures about the York Ada Compiler', *Ada Letters* (to appear).
10. M. K. Crowe, *cap: A capability-based compiler: User Guide,* Paisley College, 1986.