SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 12, 351–366 (1982)

A Machine-Independent Linker*

CHRISTOPHER W. FRASER AND DAVID R. HANSON
Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, U.S.A.

SUMMARY

Linkers, although a well-established component of language translation, are typically machine-dependent, idiosyncratic, and hard for many users to understand. This paper describes a machine-independent linker and object language. The linker embodies those linking functions that are machine-independent and centralizes them in a single tool, simplifying compilers, assemblers, and loaders. Included are descriptions of its operation, implementation, and application.

KEY WORDS Linkers Loaders Portability Code Generation Object Code

INTRODUCTION

For most high-level languages, translating source language programs into executable code involves three steps: compiling, linking, and loading. Compilers translate source code to object code or to assembly language that is assembled into object code. Linkers search libraries and combine separately compiled object modules. Loaders translate object code into executable form or simply load object code into memory in a form ready for execution.

Typical compilers, linkers, and loaders are machine-dependent. Many techniques for compiler retargetting have been developed (c.f. References 1 and 2), but linkers, loaders, and object languages remain highly machine-dependent and idiosyncratic. Even tutorial treatments rely heavily on machine-specific examples.[3-5] The UNIX[6,7] and Thoth linking loaders[8,9] have been ported to several architectures, but they achieve some of their portability because they are ported with complete operating systems. This approach is not always available or even desirable. For example, compiler researchers using a large shared computer with existing software may be unwilling or unable to port an entire operating system but might benefit by replacing just an awkward linker.

This paper shows one way to make linkers machine-independent by isolating the few inherent machine-dependencies of linking and loading in a small loader. The attendant generalizations simplify the implementation of typical linking operations. The design has resulted in a precise separation of the machine-independent functions that can be performed by a linker and the machine-dependent functions that must be performed by a loader. The techniques are demonstrated by the machine-independent linker link, a 700-line Y[10] program that is a part of an emerging portable programming environment. Although link is used primarily with Y, it can handle more widely known languages such as Fortran and Cobol and newer languages such as Ada.[11] link attempts to distill the fundamentals of linking and consequently lacks some of the more exotic capabilities often found in commercial linkers, such as overlay management and report generation.

*This work was supported by the National Science Foundation under Grant MCS-7802545.

0038–0644/82/040351–16$01.60

Received 9 September 1981
Revised 2 November 1981

LINKING AND LOADING

The terminology of linking and loading varies between systems. Definitions given in this section appear to be common and show how link relates to traditional linkers.

Object code represents machine code in which addresses may have not been completely bound and is typically the output of compilers and assemblers. Examples include '.o' files on the PDP-11 under UNIX, 'TEXT' files on the IBM 370 under CMS, 'REL' files on the DEC-10, 'relocatable' files on the CDC Cyber, and 'object' files on the Eclipse under AOS. Many object codes specify addresses assuming a starting location of 0 and include information indicating which locations must be augmented to reflect the actual location of the program when loaded.

Executable code represents machine code ready for loading and execution by the operating system. Typically, most or all addresses are bound. Examples include 'EXE' files on the DEC-10, 'absolute' files on the CDC Cyber, and 'core images' on the Eclipse under AOS. On some systems, executable code has the same format as object code. In other words, an executable file is an object file in which most or all addresses have been bound. UNIX uses this convention, for example.

Loading reads executable code into memory in preparation for execution. Because most addresses have been bound, loaders need do little more than simple i/o, though some details of this process may be extremely machine-specific. For example, some loaders must adjust hardware paging and segmentation tables, and others must execute their last few instructions from fixed memory locations to avoid being overwritten by the incoming program. Since loaders are typically short but faced with accommodating hardware idiosyncracies, there is little to be gained from trying to make them portable.

Linking resolves inter- and intra-module references and binds addresses. Almost all linkers accept object code as input. Some linkers (e.g., Link-10 on the DEC-10 and the Cyber loader) produce executable code, either in memory or in a file that may be loaded directly. Such programs may be called 'linking loaders', 'relocating loaders', or 'relative loaders'. Other linkers (e.g., Linkage Editor on the IBM 370) produce an object file that may be re-linked. Such programs may be called 'link editors' or 'linkage editors'. Some linkers can operate in either mode. For example, ld on the PDP-11 under UNIX and binder on the Eclipse under AOS can produce an unlinkable executable file or a relinkable object file.

Most linkers provide some form of library searching in order to centralize commonly used runtime routines. A *library* is a collection of object files in a form that permits identification of individual files. An example is an 'archive' file[12] whose members are object files. Another, less common, example is a directory of object files.[9] In this case, the members of the library are the files in the directory, and ubiquitous file system tools may be used to manage libraries. Some linkers also provide more comprehensive capabilities, such as overlay construction and patching of object code.

link is a linkage editor. It accepts and produces a machine-independent object code. Although the compilers, assemblers, and loaders that produce and accept link's object code may be machine-dependent, they can be simplified by having link perform many of the typical machine-independent operations that they would otherwise have to perform themselves. Examples of such operations, which are detailed in the following sections, are segment coalescing and resolution of forward references.

THE OBJECT LANGUAGE

link is best introduced through samples of its input and output. link uses a machine-independent object language consisting of lines of text. As described below, a conventional

binary format would do almost as well. Object code *commands* are interleaved with *object text*—symbolic expressions whose values will, eventually, occupy some memory cell.† The following grammar defines the object file format.

file	→	*line file* | *line*
line	→	*command* | *expression*
command	→	.def *identifier expression* .def -s *identifier expression* .seg *identifier* .org *constant* .len *identifier constant*
expression	→	*expression expression operator* *identifier* *constant*
operator	→	+ | - | < | >

Identifiers consist of an arbitrary number of letters, digits, and several special characters, and constants may be decimal, octal, hexadecimal, or textual. Expressions are given in postfix, which avoids issues of precedence and simplifies processors such as loaders that read and evaluate expressions. Since object code is produced only by translators, and since they invariably analyze expressions anyway, the production of postfix is no hardship.

link makes no architectural assumptions about the data it manipulates. It has no concept of 'word' or addressable unit, and it is independent of word size, address size, and addressability considerations. link's shifting operations (< for left shift, > for right) permit relocatable addresses to be placed at arbitrary positions within an expression, but translators typically find it easiest to generate—and loaders typically find it easiest to load—expressions that denote as many bits as are in an address. On the PDP-11, words and addresses are 16 bits, so link expressions denote 16 bits. On the DEC-10, each half of a 36-bit DEC-10 word may be relocated separately, so link expressions on the DEC-10 denote 18 bits, and loaders take two adjacent lines of object text to fill one word. They shift the value of the first left 18 bits and add the value of the second. The use of addition allows DEC-10 translators to avoid splitting large non-relocatable constants. They generate a zero for the top half and the full 36 bits for the bottom, and the loader obtains the proper value when it combines them. This technique is also used to pack 15- and 30-bit instructions into the 60-bit words of the large CDC computers. link expressions denote 15 bits, and 18-bit addresses correctly overflow from one half of the instruction into the other. link expressions also handle encoded addresses. On the IBM 370, instructions encode addresses using a 4-bit register index and a 12-bit offset, so expressions denote 16 bits. If the address lab is reached through base register 9, which is loaded with the address base9, then it might be encoded as

```
lab base9 - #9000 +          ('#' flags a hex constant)
```

The 'lab base9 -' forms the offset, and the '#9000 +' adds in the register index. Authors of translators and loaders are free to assign different meanings to link expressions; those described

†The input format resembles input to typical document formatters and other text processors. Indeed, it was first implemented by extending a conventional macro processor, and it may be viewed as an object code formatter.

above are just convenient for the machines indicated.

link could use a more conventional binary format, which would increase machine-dependence and decrease processing time and object file size, though not so much as might be expected. For example, link's object files tend to be only slightly larger than equivalent files in DEC-10 binary format. This is partly because the DEC-10 object code is, like most object codes, sub-optimal, and partly because text is sometimes shorter than a fixed binary code. For example, link's object code takes 14 bits to represent a half-word 1: seven bits for the character '1' and seven bits for the line terminator. The DEC-10 binary format takes 19 bits for the same datum: 18 bits for the half-word, and 1 bit for relocation flag. The binary code is superior for longer constants, but the typical preponderance of small constants helps reduce the size of link code.

A textual object code is also convenient for systems programmers. Many systems provide special tools to display object files, to determine the size of the program in an object file, to display or delete symbol tables, etc. With link code, compiler writers can view object code directly, and many of the functions above can be performed by existing editors and text processors. In addition, link's object code has little impact on loader complexity, which is dictated largely by the form of machine-dependent executable code. For example, the loader for the DEC-10 is about 150 lines of Y and 30 lines of assembly language.

Segments

seg commands divide the object text into *segments*. A segment is a block of object text to be loaded into a contiguous block of memory. The command seg *identifier* causes link to start, or resume, placing text in segment *identifier*. Intra- and inter-segment references are made by referring to the base location for that segment, which is simply the name of the segment. link accepts text for any number of segments, interleaved in any way. It outputs each segment as a contiguous block of text. The ultimate location of each segment and the meanings of their names may be machine-dependent and must be coordinated among the translators that produce input for link and the loaders that read its output. For example, compilers might use link to separate code from data on machines that support code sharing.

len commands specify segment lengths. link uses the len commands to adjust segment base locations so that references are relocated to reflect the final location of the segment. For example, consider two object files that contribute to segment code. If the first contributes 100 words, in processing the second link replaces code with code 100 +, to reflect the fact that the contribution of the second comes after that of the first. link cannot determine lengths by counting expressions because the width of each expression is not known. The len commands are needed to convey this machine-dependent data.

org commands eventually cause the loader to place subsequent text at different locations in the current segment, relative to the beginning of the object file in which the command appears. They are typically used to skip over large, uninitialized blocks of data.

Symbols

def commands define symbols. link replaces defined symbols in expressions with their corresponding values, and then evaluates the expressions. For example,

```
.def stack data 75 +
```

defines 'stack' to be cell 75 in the data segment. A subsequent line of object text

```
stack 1 +
```

will become

```
data 76 +
```

More precisely, link is driven by three sets of symbols: **R** is the set of symbols referenced in the object code; **D** is the set of symbols whose values are defined in the object code; and **S** is the set of symbols that identify segments. The set **D** is constructed by executing **def** commands, references to symbols in expressions form **R**, and **seg** commands form **S**.

An object file for which $\mathbf{R} \subseteq \mathbf{D} \cup \mathbf{S}$ is said to be *resolved*. Given the locations of the segments—the values of the symbols in **S**—a resolved file can be loaded and executed. An *unresolved* object file references undefined symbols, sometimes called 'external references'. link combines unresolved object files into a single resolved object file, if possible. Note that linking is an iterative process—an unresolved output file is a valid output from link. An object file for which $\mathbf{R} \cup \mathbf{S} \subseteq \mathbf{D}$ is said to be *absolute*. It contains no undefined symbols, including segment base locations. As a result, it must be loaded at a location compatible with the sets **D** and **S** that resulted in its absolute status.

link also resolves references by searching libraries. It typically receives several file names from the command that invoked it. Most object files are included in their entirety, modifying the sets **R**, **D** and **S** accordingly. Specification of a library causes selective linking. A library member is linked if and only if it contains a definition for any of the symbols in **R - D - S**, that is, the current set of symbols that are referenced but not defined (and not a segment name). For example, the UNIX-style command

```
link a.o b.o -l fortran.lib
```

is a typical invocation of link specifying two object files and the Fortran runtime library. Another example is

```
link c.o d.o -l y.lib e.o -l fortran.lib
```

which links c.o and d.o, searches the Y library, links e.o, and finally searches the Fortran library.

Including a library member may modify the sets **R**, **D** and **S**. For archive-style libraries read in one pass, the object files must appear in topological order according to inter-file references. This restriction can be avoided by constructing an index or directory for the library as it is read.

Output

link commands may appear in any order within input object files, although there must be at least one len command for each segment. link arranges the input commands so that the output object file has the following form.

```
.len commands
.def commands
.seg segment_1
(text for segment_1)
 ...
.seg segment_k
(text for segment_k)
```

This form permits the output to be loaded in one pass. len commands are placed at the beginning of the output file because typically the length of all segments must be known before any but the first can be loaded. def commands appear next so that all symbols can be defined

before they are used. def commands that included the -s option are suppressed; the next section describes applications of this feature. The segments themselves appear last, sorted so that there is no switching between segments, though there may be org commands skipping about inside segments. Note that the output consists solely of object code, which may be used as an input to a subsequent invocation of link.

Linking is not always necessary. Occasionally one encounters a program so simple that a compiler directly produces a resolved file in which the segments appear as above. Such object files can be loaded without processing by link.

APPLICATIONS

Some applications of link may not be immediately apparent, particularly those that simplify the machine-independent and multiple-pass aspects of translation and loading. Compilers, assemblers, and loaders that use link may focus on the machine-dependent aspects of translation and can often avoid a 'last' pass. For instance, the emerging programming environment of which link is a part has only one-pass compilers and no assembler.

Assembly

A popular approach to compiler implementation is to generate assembly language and let an assembler produce the desired object code,[13] since generating assembly language is typically much easier than generating object code. This organization is used in the C compiler,[1,14] and in several versions of the Y compiler. link's handling of symbols obviates the need for an assembler without complicating the code generation process. Opcode mnemonics may be defined as link symbols so that the output looks very much like assembly language, or is at least as easy to generate. For example, the DEC-10 assembly language for the Y expression

```
f(a + b, c)
```

where a, b, and c are integers and f is a procedure is

```
push    17,c      ; push c onto argument stack
move    2,a       ; load a
add     2,b       ; compute a + b
push    17,2      ; push a + b
pushj   17,f      ; call f
adjsp   17,-2     ; remove arguments
```

The Fortran compiler generates similar code. The link code for the first instruction, which is generated as 18-bit expressions, is

```
push 0740 +       ; push c onto argument stack (0 flags an octal constant)
c
```

The position of the register specification on the DEC-10, while easy to generate, is difficult to read. Symbolic names for the registers can be used to improve readability. In addition, opcode definitions need not correspond to actual opcodes. For instance, the stack register (17 in the above example) can be defined as a part of the various stack instructions. Assuming such

definitions, the link code becomes

```
push            ; push c onto argument stack
c
move r2 +       ; load a
a
add r2 +        ; compute a + b
b
push            ; push a + b
02
pushj           ; call f
f
adjsp           ; remove arguments
0777776
```

The use of definitions and link expressions to effect assembly is even more attractive on computers with complex addressing modes for which instruction assembly is tedious. For example, the PDP-11 assembly code for example above includes

```
mov @#c,-(sp)   ; push c onto argument stack
mov @#a,r2      ; load a
 ...
jsr pc,@#f      ; call f
 ...
```

The corresponding link code is

```
mov .A+ M+ sp+  ; push c onto argument stack
c
mov .A+ r2+     ; load a
a
 ...
jsr .pc+ A+     ; call f
f
 ...
```

The symbols A and M denote absolute and auto-decrement addressing modes, respectively, and r2, sp, and pc denote registers. These symbols are defined for use in destination addresses, and the corresponding symbols preceded by a period are for use in source addresses. Note that these symbols may appear in any order. As for the DEC-10, opcodes that include addressing modes can be defined. For example, on the PDP-11 if push is defined by the command

```
.def -s push mov M+ sp+
```

then

```
push .A+
c
```

can be generated for the first instruction in the example above.

Experience with the various Y compilers indicates that link code is as easy to generate as assembly language and much easier than typical binary formats. On the DEC-10, for example, the module for generating link code is 15 percent shorter than that for the assembly language generator and 54 percent shorter than that for the DEC-10 binary object code generator.

Local Symbol Resolution

Typical linkers do not resolve local symbols because local symbols from separately compiled modules may collide. Assemblers and compilers that use such linkers typically make an extra pass to resolve local symbols.† Since link permits long symbol names, translators can get it to resolve local symbols by adding codes to make their names unique.‡ For example, the following Y declarations

```
module stack
   import print from "lib"
   export push, pop to "lib"

   integer sp
   integer stack[MAXSTACK]

   push(x)
      ...
   end
   pop()
      ...
   end
end
```

identify symbols print, push, and pop as global and symbols sp and stack as local to the stack module. The compiler leaves print, push, and pop alone, but prefixes the module name and an underscore to every use of sp and stack in the object code, ensuring that they will not collide with local symbols from other modules. For example, it translates the code above into

```
.seg data
.def -s stack_sp data
.def -s stack_stack data 01 +
.seg code
.def push code
...
.def pop code 021 +
...
.len data 025
.len code 043
```

Another use of local symbols is for forward references. For example, typical code for

```
while ( expression ) statement
```

has the form

```
L1      jump to L2 if expression is false
        statement
        jump to L1
L2
```

†On a few machines, a second assembly pass may be needed, but not to resolve locals. For example, on the IBM one is needed to assemble base-displacement addresses.

‡A similar technique can be used to perform some limited type-checking.[15]

As an example, the loop

```
while (a < 0)
   a = f(a, b)
```

in a module named stack results in the following DEC-10 link code.

```
.def -s stack_L1 code 011 +
move r2 +                    ; load a
a
cail r2 +                    ; skip if a < 0
0
jrst                         ; jump to L2
stack_L2
 ...code for a = f(a, b)...
jrst                         ; jump to L1
stack_L1
.def -s stack_L2 code 022 +
```

In both of these examples, all references to local symbols are resolved by link, and since the local symbols are defined using the -s option, definitions for them do not appear in the output from link.

Another way to handle local symbols, typically used in machine-dependent linkers, is 'backpatching'. The compiler inserts linker directives in the object code that instruct the linker how to backpatch forward references. Directives typically specify address modifications to various locations. While backpatching is not needed because of link's handling of symbols, it can be accomplished via the org command, permitting translators to generate link code that is ready for loading.

Normally, expressions completely specify the contents of a location, e.g.

```
data 3 +
```

specifies that the value of the expression be placed in the current location. More precisely, after evaluation the value at the top of the evaluation stack is placed in the current location. However, link implicitly assumes that the *current* value of the current location is pushed onto the evaluation stack prior to evaluation. Thus, an expression such as

```
code +
```

specifies that the value of code is to be added to the value in the current location. Backpatching is accomplished by using an org command to get back to the desired location and an expression that makes the desired modification. For example, consider the while statement given above. Using backpatching, the link code is as follows. The numbers on the left represent locations in the code segment.

```
location  link code

011       move r2 +          ; load a
          a
012       cail r2 +          ; skip if a < 0
          0
```

```
013     jrst                ; jump to L2
        0
        ...code for a = f(a, b)...
021     jrst                ; jump to L1
        code 011 +
        ...
        .org 013
013     code 022 + +        ; patch forward reference to L2
        ...
```

Note that backpatching for backward references (e.g. to L1) is unnecessary. While link permits the specification of backpatching, it is the loader that ultimately implements it because only the loader has the actual value and location.

Finally, another way to handle local symbols is to process the compiler output with link as soon as it is generated. As a result, all local symbols are replaced by their values before further linking with other modules, avoiding local name collision. The disadvantage of this approach is, of course, the overhead of always using link for the last pass of the compiler.

Segment Switching

link's ability to switch between segments also simplifies translators. A compiler can use different segments for code, uninitialized static data, initialized data, and literals. Text can be emitted into any segment at any time. It is easier to generate such code than it is to collect literals and dump them at the end of the translation. link allows translators to interleave code, data, and literals; it sorts the interleaved information. Indeed, link code is easier to generate than assembly language for assemblers that lack such features, such as the UNIX assembler. For instance, the typical DEC-10 translation of

```
print("? stack full")
```

is

```
.seg code
movei r2 +                  ; load address of literal
stack_L4
.seg lit                    ; switch to literal segment
.def -s stack_L4 lit 01 +
 ...code for the string "? stack full"...
.seg code                   ; resume code segment
push                        ; push address of literal
02
pushj                       ; call print
print
adjsp                       ; remove argument
0777777
```

and link produces

```
.seg code
movei r2 +                  ; load address of literal
lit 01 +
```

```
push                        ; push address of literal
02
pushj                       ; call print
print
adjsp                       ; remove argument
0777777
 ...
.seg lit
 ...code for the string "? stack full"...
 ...
```

By placing literals in a separate segment the actual area into which they are loaded can be deferred until loading and need not be known by the compiler. For example, the Y compilers generate code for three segments, code, data, and literals, even though on the DEC-10 the literals are placed at the end of the code area, and on the PDP-11 they must be placed in the data area.

Link-Time Binding

Symbol suppression and iterative linking can be used to obtain some unusual, but useful, effects. An example is the reduction of so-called 'name space congestion'—the proliferation of global names—that often plagues large software systems, especially those fabricated from numerous independent modules. This is accomplished by linking several modules together and suppressing the definition of those global symbols that are no longer needed. The result is a 'package' that can be linked without fear of name collisions with those global symbols that were used in its construction.

One example is the construction of a debugger. In this case, it is important that the global symbols used within the debugger do not collide with those of the program with which it is being used. When the debugger is linked, such symbols are simply suppressed, permitting it to be linked with any program.

Another example is the construction of abstract data type packages. For instance, consider the following four object files. istack.o contains object code that implements an integer stack abstraction, rstack.o contains object code for a real stack abstraction, parse.o contains an expression parser that uses the integer stack, and eval.o contains an expression evaluator that uses the real stack. Both stacks are accessed via the three global functions init, push, and pop. The apparent problem is that the use of two different kinds of stacks results in multiple definitions for each operation. The solution is to view the program as only two abstractions—a parser and an evaluator. This can be accomplished without modification of either stack module by using symbol suppression and two iterations through link. An object file for the parser abstraction is constructed by the command

```
link parser.o istack.o -s push -s init -s pop -o parser.a
```

The -o option specifies the output file. The -s options operate exactly like the -s option on def commands and prevents definitions for the named symbols from appearing in the output file. Since all references to the integer stack functions are within parser.o, the -s options result in the omission of the operation names from parser.a. Similarly,

```
link eval.o rstack.o -s push -s init -s pop -o eval.a
```

constructs the evaluator. Linking parser.a and eval.a yields the final program:

```
link eval.a parser.a -o program
```

Since neither input file contains definitions or references to init, push, and pop, no name collision can occur.

This kind of 'information hiding' capability is provided by several recent languages, e.g. CLU[16] and Ada. Linking parser.o and istack.o to form parser.a is very similar to the production of 'packages' in Ada, for example. link provides a mechanism for implementing such capabilities in a machine-independent and, perhaps more importantly, a language-independent fashion. Centralizing these kinds of features in tools like link makes these features more widely available and simplifies the implementation of compilers for languages that must provide them since they can simply run link to achieve the desired results.

Machine-Independent Cross Assembly and Linking

Since link is insensitive to the meaning of expressions within its object code, it can be used as a cross assembler and linker. It can serve, for instance, as a linker for several targets other than its host system, especially microprocessors that have poor linkers or lack them altogether.

As an example, link has been used to assemble and link code for the Motorola 6502 microprocessor. A Y code generator was written that ran on the PDP-11/70 and produced assembly code for the 6502 in a form similar to that described at the beginning of this section. link, also running on the PDP-11/70, was used to assemble and combine independently compiled Y modules. Its output was then down-loaded to the 6502 and executed or saved. Because link code for the 6502 is as easy to read as the regular assembly language, the runtime support routines were hand-written in link code. This approach permitted link to output resolved files, which required only a very simple loader running on the 6502. The end result was the ability to produce relatively complex Y programs for a system that formerly offered only assembly language and BASIC.

The ultimate target of link's output need not be a real machine. link can, for example, link object code for an abstract machine. Operations in the abstract machine language must be represented as text expressions, which differ only syntactically from conventional abstract machine languages.[17] This capability is a step towards reconciling the conflicts between modularity and portability. Because of the wide variations in linkers among systems, modularity tends to complicate the installation of portable software.[18,19] Indeed, installation of a heavily modular system can be thwarted completely if the system was developed in an environment with linking concepts that poorly match those in the target environment. Most successful portability projects address this problem. BCPL, for example, has a mechanism for runtime linking of separately compiled modules,[20] and the entire macro implementation of SNOBOL4 is distributed as a single monolithic module.[19,21]

Using link, a portable system can be developed in a modular fashion and distributed in a monolithic fashion. For example, the portable version of the Y compiler, which generates code for an abstract machine, consists of about 20 modules. Each of these modules can be compiled separately producing object files of abstract machine code. link can combine these to produce a single object file for the compiler. Instead of loading this file, it is translated to, say, assembly language for the target system using typical abstract machine modelling techniques. Once running, the compiler can be tuned to the target environment. Such tuning can be done with the compiler in its modular form rather than in its distributed monolithic form. Similar comments apply to other portable systems based on abstract machine modelling techniques, such as the 'P-code' implementations of Pascal.[22]

IMPLEMENTATION

link makes two passes over its input. The first pass reads the input object files and produces several temporary files. The second pass reads the temporary files and produces the output object file. The following two object files are used to illustrate the operation of each pass.

a.o

```
.seg code
.def start code
0263 f +
.seg data
.def -s a_l data
data 01 +
.org 06
code
.len code 01
.len data 07
```

b.o

```
.seg code
.def f code
0201 b_l +
.seg data
.def -s b_l data
start
.len code 01
.len data 01
```

File a.o defines global symbol start and local symbol a_l, and refers to external symbol f; b.o defines global symbol f and local symbol b_l, and refers to external symbol start. Both files contain text for two segments, code and data. In a.o, note the references within expressions in the data segment to the base locations of the two segments. Subsequent discussion assumes that these files are linked by the command

```
link b.o a.o -o ba.o
```

The first pass distributes text into segment temporary files and builds the sets **R**, **D**, and **S**. Figure 1 outlines the general operation of pass one. The temporary files are†

code temp file

```
.seg code
0201 b_l +
.len code 01
.len data 01
0263 f +
.len code 01
.len data 07
```

data temp file

```
.seg data
start
.len code 01
.len data 01
data 01 +
.org 06
code
.len code 01
.len data 07
```

Note that len commands are not 'executed' until *after* an object file is processed. This permits them to appear in any order, even before the segments as in link's output. In addition, len commands for *all* segments appear in each temporary file in order to correctly relocate inter-segment references during the second pass. Finally, org commands specify offsets *relative* to the base locations for the file in which they appear.

Libraries are searched when they are encountered in the command invoking link. Linking a library member is exactly like linking an input object file, except that the selection of a member is based on the current value of the set **R**.

†The characters comprising a symbol do not actually appear in the temporary files; a pointer into the symbol table is used. The symbol is used here for clarity.

```
for each input object file do {
    for each segment id do set length_id to 0
    for each line in the object file do
        if line is .seg id then {
            if id refers to a new segment then {
                create temporary file for id
                output '.seg id' to the temporary file
                set base_id and length_id to 0
                }
            establish id as the current segment
            }
        else if line is .def id expr then
            establish value of expr as definition of id
        else if line is .def -s id expr then
            establish value of expr as 'suppressed' definition of id
        else if line is .len id n then {
            if id refers to a new segment then {
                create temporary file for id
                output '.seg id' to the temporary file
                set base_id and length_id to 0
                }
            increment length_id by n
            }
        else if line is .org n then
            output '.org n' to the current temp file
        else {
            add symbols appearing in line to R
            output line to the current temp file
            }
    for each segment id do
        increment base_id by length_id
    for each segment id1 do
        for each segment id2 do
            output '.len id2 length_id2' to id1's temp file
}
```

Figure 1. Operation of Pass One.

The second pass combines the text in the temporary files into the output file. One product of the first pass is the final value of the set **D**. As the second pass reads the temporary files, it evaluates expressions, and replaces symbols with their values. The operation of pass two is sketched in Figure 2. The final result of linking b.o and a.o is ba.o:

```
.len code 02
.len data 010
.def start code 01 +
.def f code
.seg code
0201 data +
0263 code +
.seg data
code 01 +
data 02 +
.org 7
code 01 +
```

```
for each segment id do
   output '.len id base_id'
for each symbol do
   if symbol is not 'suppressed' then
      output def command for the symbol
for each segment id do {
   for each segment id do
      reset base_id to 0
   for each line in id's temp file do
      if line is .seg id then
         output '.seg id'
      else if line is .len id n then
         increment base_id by n
      else if line is .org n then
         output '.org base_id + n'
      else
         output value of expression
   }
```

Figure 2. Operation of Pass Two.

In the input to the second pass, org commands are processed so that their effect is relative to the file in which they originally appeared. In the final output file, however, they are adjusted to be relative to that file—the one in which they now appear.

link processes object text at approximately 200 lines per second on the DEC-10 and at 185 lines per second on a PDP-11/70 running UNIX. The implemention of link is straightforward, although some care was taken in the symbol table and expression routines. Symbols are kept in a hash table, and expressions are compiled into an intermediate form that avoids subsequent table lookups and reduces space requirements. The first pass writes expressions to the temporary files in this form. As a result of these techniques, the speeds mentioned above are nearly independent of the number of symbol definitions and references.

CONCLUSIONS

link demonstrates the feasibility of a machine-independent object language and linker. While it may be unlikely that the detailed aspects of object languages and linkers will ever be standardized, *functional* standardization would be of significant benefit. Experience with link suggests that functional standardization of linkers would simplify translators and loaders by centralizing many of their machine-independent capabilities. Such standardization would make these capabilities more useful, more widely available, and better understood.

ACKNOWLEDGEMENTS

The Y code generators for the PDP-11/70 and the Motorola 6502 were written by David Weatherford and Bruce Conrad, respectively. The comments of Ralph Griswold and of the referees were very helpful in presenting this work.

REFERENCES

1. S. C. Johnson, 'A portable compiler: theory and practice', *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, 97-104 (1978).
2. P. C. Poole, 'Portable and adaptable compilers', in *Compiler Construction: An Advanced Course*, F. L. Bauer and J. Eickel (eds.), Springer-Verlag, New York, 1976, 427-497.

3. D. W. Barron, *Assemblers and Loaders*, 3rd edn., Elsevier North-Holland, New York, 1978.
4. R. M. Graham, *Principles of Systems Programming*, John Wiley and Sons, New York, 1975.
5. L. Presser and J. R. White, 'Linkers and loaders', *Computing Surveys*, **4**, 149-167 (1972).
6. D. M. Ritchie and K. Thompson, 'The UNIX timesharing system', *Communications of the ACM*, **17**, 365-375 (1974).
7. S. C. Johnson and D. M. Ritchie, 'Portability of C programs and the UNIX system', *Bell System Tech. J.*, **57**, 2021-2048 (1978).
8. D. R. Cheriton, M. A. Malcolm, L. S. Melen and G. R. Sager, 'Thoth, a portable real-time operating system', *Communications of the ACM*, **22**, 105-115 (1979).
9. G. R. Sager, *The Thoth Linking Loader*, Tech. Rep. CS-77-15, Dept. of Computer Science, Univ. of Waterloo, Waterloo, 1977.
10. D. R. Hanson, 'The Y programming language', *SIGPLAN Notices*, **16**, 59-68 (1981).
11. J. G. P. Barnes, 'An overview of Ada', *Software—Practice and Experience*, **10**, 851-887 (1980).
12. B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, Reading, Mass., 1976.
13. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977.
14. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, New Jersey, 1978.
15. R. G. Hamlet, 'High-level binding with low-level linkers', *Communications of the ACM*, **19**, 642-644 (1976).
16. B. H. Liskov, A. Snyder, R. Atkinson and C. Schaffert, 'Abstraction mechanisms in CLU', *Communications of the ACM*, **20**, 564-576 (1977).
17. M. C. Newey, P. C. Poole and W. M. Waite, 'Abstract machine modelling to produce portable software—a review and evaluation', *Software—Practice and Experience*, **2**, 107-136 (1972).
18. F. C. Druseikis, 'Influence of modularity on program portability', *Proceedings of the European Conference on Software Systems Engineering*, London, 1976.
19. R. E. Griswold, 'The macro implementation of SNOBOL4', in *Software Portability, An Advanced Course*, P. J. Brown (ed.), Cambridge University Press, London, 1977.
20. M. Richards, 'The implementation of BCPL', in *Software Portability, An Advanced Course*, P. J. Brown (ed.), Cambridge University Press, London, 1977.
21. R. E. Griswold, *The Macro Implementation of SNOBOL4; A Case Study in Machine-Independent Software Development*, W. H. Freeman, San Francisco, 1972.
22. P. A. Nelson, 'A comparison of PASCAL intermediate languages', *Proceedings of the SIGPLAN Symposium on Compiler Construction*, 208-213 (1979).