The Design and Implementation of a Dynamic Binding Feature for a High-Level Language

Rex E. Gantenbein
Department of Computer Science, University of Wyoming

Douglas W. Jones
Department of Computer Science, University of Iowa

Most high-level programming languages are unable to control the bindings between names and separately compiled implementations of those names at run time, while checking the correctness of such bindings at compile time. This facility is necessary for the use of the language in a monolingual programming environment. This paper describes the semantics of a dynamic binding feature for a block-structured, strongly typed language and the incorporation of the feature into a Pascal implementation.

1. INTRODUCTION

An area of software engineering receiving increased attention lately is the *programming environment*, a variation on the operating system that is primarily concerned with the support of high-level program development. A common thread through much programming environment research is the trend toward "monolingual" environments, in which no system-specific job control language or shell is needed to provide the connections between the programs making up a large application. Heering and Klint [1] show that the convergence of programming and control languages depends on the ability of the unified language to treat (1) procedures and programs and (2) files and types as equivalent. Many high-level languages, some of which are mentioned in Ref. 1, have this ability; however, any language that is to be used to write a command interpreter must provide some sort of dynamic binding facility to link programs together, a trait that is much less common.

Address correspondence to Rex E. Gantenbein, Department of Computer Science, University of Wyoming, Box 3682, University Station, Laramie, WY 82071.

Although most operating systems provide dynamic linkage services that allow one running program to cause other programs to be loaded and run, such services are not usually provided in high-level languages. Instead, control over binding at execution time usually involves accessing the binding facilities of the underlying operating system. The Multics dynamic linker, for example, is automatically called whenever a program references an unbound external name; after the linker has bound the name, the program may call a system service to terminate the binding so that the next reference to that name will relink it, possibly to a different object [2]. It should be noted that system services like this are outside the semantics of the programming languages involved; the linkers do not perform type checking, and programs using these linkers are not portable.

There appears to be no high-level language that provides dynamic control over bindings between names declared within a program and objects external to the program while checking the consistency and correctness of the bindings. Algol 68 [3] allows strongly typed dynamic binding within the confines of a single program. In this language, it is legal to define procedure variables, or pointers to procedures; these pointers may be bound to any compatible procedure (where compatibility is determined by the structural equivalence of the argument list and the result), but no provision is made for binding a procedure name to an external object. Mesa [4] allows somewhat more flexibility; here, the language definition includes separate compilation, with strong type checking enforced across compilation-unit boundaries by the Mesa binder. Furthermore, as in Multics, the binder can be dynamically invoked to add new code to a running program. Unfortunately, once a

The Journal of Systems and Software 8, 259–273 (1988)

0164-1212/88/$3.50

binding has been established, it cannot be safely broken. As a result, running Mesa systems tend to grow and must be periodically restarted to prune off unneeded linkages.

A complete implementation of dynamic binding in a strongly typed language would involve all the aspects described above. The language would have to allow programs to be constructed from separately compiled pieces, and it would have to include constructs for dynamically controlling the bindings between internal names and these external pieces. If the problems illustrated by Mesa are to be avoided, the language should provide some method of unbinding or rebinding a previously bound name. In addition, the data-typing and data-sharing rules of the language should not depend on whether program pieces are bound together at compile time or at run time.

The following program fragment illustrates how a dynamic binding mechanism can provide a monolingual programming environment:

```
program Shell (...);
type a = ...;
  ...
var b: a;
    FileName: string;
  ...
procedure c (var b: a) ; dynamic;
begin
  repeat
    ...
    link 'c' to FileName;
    c(b);
  until ...;
end (* Shell *).
```

This example is written in a version of Pascal extended to support dynamic binding. The program repeatedly binds the dynamically bindable procedure c to implementations stored in object files named by successive values of the string variable FileName. As such, Shell serves as a control program to mediate the communication between the procedure (program) implementations linked to the name "c"; since these can be written in the same language as the control program, this is a monolingual environment. Furthermore, the global variable b has a lifetime determined by the lifetime of Shell and not by the duration of a call to c or the lifetimes of the object files containing implementations of c. In this way, dynamic binding can achieve a form of control over persistent data (data that outlive the program that creates them), which has been shown to be one way of implementing typed files [5]. There is nothing to prevent the recompilation or replacement of implementations of c between calls, and given an appropriate checkpointing mechanism, Shell can run forever.

2. SEMANTICS OF DYNAMIC BINDING

The process of designing a programming language is not well understood; furthermore, the evaluation of various design approaches is far beyond the scope of this paper. However, it has been succinctly noted by Hoare [6] that the design of a language feature is innovation, while the design of a language is consolidation of the best features in existence. Following this paradigm led us to view dynamic binding as a feature with which to extend existing languages rather than as the basis for an entirely new language. We did, however, try to avoid letting the choice of a particular base language influence the design of our feature by first considering several alternatives for the various aspects of the feature independently of implementation issues. This approach follows a language design methodology developed by Marlin [7] that has been successfully used in other design projects.

It became clear early in the design process that certain general decisions had to be made before attention could be paid to more detailed decisions. In particular, the design was guided from the outset by a desire to work within a strongly typed, block-structured, high-level language, so that advantage could be taken of the structuring and static correctness-checking facilities in such languages. Separate compilation was also seen necessary to support the redefinition and rebinding of names during the course of a program's execution. Once these decisions had been made, the design could progress to the consideration of alternatives for the distinct aspects of dynamic binding.

The operations involved in dynamic binding can be summarized with respect to compile-time and run-time issues as follows:

Compile time.

1. Establish the environment for compilation.
2. Compile the program that declares the name, and compile the object.
3. Check the correctness of the compilation with respect to the compilation environment.

Run time.

1. Determine the safety and consistency constraints on the binding specification.
2. Locate the object in compiled form.
3. Add the binding between the name and the object to the run-time environment.

These issues are independent in the sense that there are

many different ways of approaching each issue, and the particular approach taken in dealing with one issue has little effect on the approaches that may be used to deal with the others. Furthermore, they are independent of the other semantics of the language and can be considered without choosing a specific base language. The advantage of this approach is that it avoids the distractions found in designing a complete language and concentrates on the problems associated with the new feature.

Each of the above issues is related to one (and only one) of three identifiable aspects of dynamic binding. Establishing an environment and binding an object to a name are part of the *binding mechanism*; compiling programs in separate parts and locating the compiled objects at run time require the use of *interfaces*; while type checking by the compiler and safety checks by the run-time system are both part of *safety and correctness*. There are several alternatives for the semantics of each of these aspects; the following discussion addresses some possible choices for each area.

The compile-time semantics of a dynamic binding mechanism primarily describe what things can be dynamically bound or rebound. A natural way to answer the question of what constructs should be bindable with this mechanism is to list all the kinds of things that have names and determine which of them should be bindable. Procedures, functions, and abstract data type implementations are all reasonable candidates, since each is a natural unit of separate compilation. Simple variables may be viewed as dynamically bindable constants, and pointer variables may be viewed as dynamically bindable variables. Languages with label or type variables may be considered to support dynamic binding of labels and types. Type names, however, probably should not be dynamically bindable, as the strong typing we have assumed would be either easily violated or expensively maintained.

At the heart of any dynamic binding mechanism, there must be a loader to include separately compiled material in the address space of the running program, as well as a linkage mechanism to connect dynamically bindable names with newly loaded objects. If this linkage is made through an indirect link, as in the Multics "combined linkage region," dynamic bindings are easily changed. If the linkage is made by modifying the address fields of each instruction that references the dynamically bound object, bindings are harder to change. If user programs may obtain and store actual code addresses in user-managed data structures, bindings are almost impossible to change. If the indirect links are stored in activation records, many bindings of a name may coexist (perhaps as many as there are activations of the declaring block). Of course, multiple bindings can coexist only if the loader is able to load different bindings of a name into different memory locations.

The combination of block structure with separate compilation in our as-yet-unspecified base language provides a simple abstraction mechanism. However, to make such abstraction useful, there must be an interface between the use and the implementation of an abstraction. This interface serves both as a uniform reference to an object, regardless of the different representations the object may take, and as a specification of the object by which the correctness of the binding can be checked. The interface is thus the means by which the using program's independence from differences in representations for an object can be preserved [8].

Interfaces between program units can be maintained as entities separate from programs, as is done in Mesa [9] and Modula-2 [10]. This approach provides an explicit interface that permits both uses and implementations to be written and compiled in any order and allows the revision of one unit without requiring revision or recompilation of the other. However, the use of explicit interfaces is frequently complex, often involving libraries as in CLU [11]. Interfaces can also be defined within the text of the using program as definition *stubs* for code segments whose compilation can be deferred. This approach is used, in slightly different ways, in both the Modcap language [12] and the UW-Pascal system [13]; generally, the approach is simpler to implement than the explicit approach, although it is less flexible. (Note: Ada includes both explicit and implicit interfaces in its *package* and *is separate* constructs, respectively [14].)

In a dynamic binding system, the run-time interface is taken to mean a mechanism for locating a separately compiled implementation from within an executing program. Locating an object to be bound on most computers undoubtedly means locating an object-code file produced by the compiler; thus, the locating scheme must involve searching the file system directory structure for a file name specified by the linkage instruction, most probably as a string. This string can represent, in its simplest form, the complete (relative to the directory structure) name of the implementation file. In another approach, the name of the identifier to be dynamically bound could be used as a file name, as is done in Multics. In either case, the file must be located in some directory that can be searched via the file system.

Compile-time correctness checking of dynamic bindings consists primarily of checking that all implementations written for a particular name satisfy the specification given by the interface. This sort of checking is usually done as a matter of course in languages with explicit separate compilation like Modula-2; however, we must also be concerned with the issue of recompila-

tion of separate units when the interfaces in which they are involved are modified, whether generated explicitly or implicitly.

Compile-time checking can guarantee the syntactic correctness and consistency of declaring and implementing units for a dynamically bindable name, but certain errors can be detected only at run time, especially in languages where names can be dynamically rebound after one binding has already been established. Brosgol [15] defines two such problems: that of converting instantiations from one form to another when rebinding occurs, and that of type identity when multiple bindings are allowed to coexist and interact. The DMERT operating system developed by Bell Laboratories [16] solves the problem of dynamically replacing individual procedures very simply: No binding may be changed except when all instances have been deleted. This is described as the "clear stack" condition, and it is up to the programmer to indicate when this condition is true in a program. More complex solutions may involve canonical forms for instantiations so that conversion is possible. An intermediately complex solution involves disallowing the rebinding of names when any instances of that name exist; although some run-time checking will undoubtedly be required, compile-time checking may be able to partially support such a solution.

3. AN EXPERIMENTAL IMPLEMENTATION

In order to more fully explore the semantic issues involved in dynamic binding, a programming language incorporating the feature was implemented. Pascal was selected as the base language for this experiment because it is well known and because easily modified partial implementations such as Pascal-P [17] are available. This particular implementation compiles Pascal code into intermediate or "P-" code, which is then assembled and executed by a separate interpreter. Both the compiler and interpreter are themselves written in Pascal, a factor that greatly influenced their use in this experiment. The resultant language, db-Pascal, was implemented by modifying the Pascal-P compiler to support separate compilation, a linking instruction, and compile-time safety checks, and by modifying the P-code interpreter to include loading, linking, and run-time safety checking. Separately compiled object modules are stored as P-code files.

The discussion that follows illustrates the choices made in implementing db-Pascal from among the semantic alternatives previously mentioned and presents the rationale behind these choices. A brief description of the language system and an example of a db-Pascal program can be found in the appendix. Further details can be found in Ref. 18.

Of the named objects supported by Pascal, procedures and functions are the obvious candidates for dynamic binding. An important question is: Do results obtained for procedures in Pascal apply to abstract data type implementations such as those provided by Ada packages or Modula-2 modules? Linden [19] argues that procedures and modules are equivalent, while Jones [20] claims that modular and object-oriented encapsulation mechanisms provide equivalent support for abstract data type implementations. Intuitively, this equivalence can be seen by noting that the operations on an abstract data type may all be formulated as procedures and that rebinding the implementation of an abstract data type is equivalent to rebinding all of the procedures (including those that allocate space for instances of the type).

The mechanisms already in Pascal for *forward* procedure declarations provided useful direction for the introduction of separate compilation into the language. The Pascal *forward* directive allows the compiler to check the correctness of code that calls a procedure or function before the compiler has encountered the procedure or function body. Dynamically linked procedures can use the same facility, although an additional mechanism is needed to allow the compiler to check that the separately compiled body conforms to the definition used by the calling program.

The "compiler suspension" model of separate compilation used in Ref. 13 is also used here. In this model, the (abstract) compiler saves the known environment (i.e., all known names and their associated attributes) whenever it encounters a stub defining a separately compiled object. Compilation of the declaring segment continues until complete, possibly saving many different environments in the meantime; in effect, the compiler is "suspended" at the point of definition of the stub and must be "resumed" (that is, the environment restored and compilation restarted) to compile the implementation of the stub. The result is that the semantics of the separately compiled object are the same as they would have been had the text of the object been inserted immediately after the stub. Thus, stubs may appear at any nesting level.

In order to implement separate compilation and dynamic binding together, a new compiler directive, *dynamic*, was added to the Pascal-P language. When the compiler encounters this directive, it dumps its symbol table into a file with the same name as the stub in a directory associated with the one containing the source file (usually a subdirectory). This symbol table contains definitions of all label, type, variable, procedure, and function names that are accessible from within implementations bound to the stub as well as the formal parameter list of the stub. Each compilation unit begins with an "environment heading" indicating the stub for

which a body is being compiled. When the compiler encounters this header, it initializes its symbol table from the associated file. The standard Pascal *program* heading can be viewed as an environment heading for the stub of the system command language interpreter.

A new simple statement, *link*, provides program control over the bindings between internal procedure stubs and external procedure bodies. This statement specifies the name for the procedure and the access path to the file containing the object code for the procedure body. The access path to the object file is assumed to start in the directory in which the source code of the declaring program is located. The compiler stores the access path as part of the compiled P-code for the *link* statement; the use of the path name thus eliminates the need for a run-time search of directories, but it requires a hierarchical file system organization.

The extensions of Pascal-P that characterize db-Pascal are summarized in the syntax specifications below. Symbols not in the original grammar are underscored. The notation is that of the IEEE Pascal Standards Committee [21], whose notion of "extension" is also used here. Note that "program" is no longer the start symbol of the grammar.

```
compilation-unit = program | external-procedure
external-procedure = environment-heading "."
                     procedure-identification ";"
                     procedure-block ".".
environment-heading = ( "environment dynamic" |
                        "environment separate")
                        file-name.
file-name = { directory-identifier "/" } file-identifier.
directory-identifier = identifier.
file-identifier = identifier.
simple-statement = empty-statement |
                   assignment-statement |
                   procedure-statement |
                   goto-statement |
                   link-statement.
link-statement = "link" procedure-name-string "to"
                 procedure-body-string.
procedure-name-string = " ' " procedure-identifier " ' ".
procedure-body-string = " ' " file-name " ' ".
```

In this specification, the directive *dynamic* is not explicitly included, in accordance with the standard, which does not specify directives except to say that *forward* is required by all implementations.

At run time, execution of the *link* statement initiates the dynamic linkage mechanism. The P-code interpreter assembles the indicated object file (located via the path name in the compiled form of the statement) and loads it into an unused code segment in its internal code array. The current implementation permits only one implementation of a given procedure to exist at any time, so relinking a previously linked name can reuse the space allocated to the previous implementation in the code array. A global indirect link suffices to transfer control to dynamically linked routines.

The only threat to consistency in db-Pascal involves attempts to rebind a procedure that is active. The decision to allow only one implementation of a given stub to exist at any time is the root cause of this problem; if a procedure were to rebind itself or call another procedure that rebinds it, chaos would result. The db-Pascal compiler contains static checks to prevent any procedure from rebinding itself or any statically enclosing procedure. The db-Pascal P-code interpreter maintains, with each indirect link to a dynamically bound routine, a count of the number of activation records currently in existence for that routine; any attempt to rebind a routine with a nonzero count will cause a fatal run-time error.

The implementation of the db-Pascal system was carried out on a VAX 11/780 computer running Berkeley 4.2 UNIX during the summer of 1985. The

introduction of separate compilation into the compiler required approximately 2 weeks of work. Another 4 weeks was spent implementing the dynamic binding mechanism in the interpreter and adding the associated changes for static safety checking in the compiler. The new implementation is neither complete nor efficient, but it demonstrates the relative ease with which dynamic binding can be added to an existing language.

4. CONCLUSION

Although the implementation of db-Pascal presented in this paper only allows the dynamic binding of procedures, it does indicate the feasibility of extending strongly typed, block-structured languages with dynamic binding. The system, as currently implemented, allows programs like those discussed at the beginning of the paper to be written; if the language had a checkpointing mechanism as well, it would allow "persistent programming" as described by Atkinson and Morrison [22]. Implementing a complete monolingual programming environment would also require an exception-handling mechanism like that proposed for Pascal [23] so execution errors would not cause exits from the environment. This mechanism can easily be implemented using nonlocal *gotos*, which unfortunately are not implemented in the Pascal-P compiler from which the system was built. Also required would be a global command interpreter (which could be written in db-Pascal) under which all user programs, including the compiler, could run.

Research is under way that will correct some of the current limitations of db-Pascal and investigate more applications in which dynamic binding might prove useful. In particular, the use of dynamic binding to support fault-tolerant applications, in which code found to cause errors can be replaced without requiring the termination of the application program, is being studied. Part of this research may involve a reimplementation of the language to compile programs into machine code rather than P-code in the hopes of increasing their efficiency. Dynamic binding is also being incorporated into languages other than Pascal (a version of C has been implemented), an undertaking that may lead to more insight into the semantic choices that can (or should) be made among the alternatives for the feature.

REFERENCES

1. G. Heering and P. Klint, Towards Monolingual Programming Environments, *ACM Trans. Program. Lang. Syst.* 7(2), 183–213, 1985.
2. *Multics Programmers' Manual and Reference Guide*, Honeywell Information Systems Document No. AG91, 1975.
3. A. van Wijngaarden et al., *Revised Report on the Algorithmic Language Algol 68*, Springer-Verlag, New York, 1976.
4. J. G. Mitchell, M. Maybury, and R. Sweet, *Mesa Reference Manual, Version 5.0*, Xerox Corporation, 1979.
5. M. P. Atkinson et al., An Approach to Persistent Programming, *Comput. J.* 26(4), 360–365, 1983.
6. C. A. R. Hoare, *Hints on Programming Language Design*, Stanford Univ. Tech. Report STAN-CS-73-403, 1973.
7. C. D. Marlin, *A Methodical Approach to the Design of a Programming Language and Its Application to the Design of a Coroutine Language*, Univ. Iowa Tech. Report 83-05, 1983.
8. C. M. Geschke and J. G. Mitchell, On the Problem of Uniform References to Data Structures, *IEEE Trans. Software Eng.* SE-1(2), 207–219, 1975.
9. H. C. Lauer and E. R. Satterthwaite, The Impact of Mesa on System Design, *Proc. 4th Int. Conf. Software Engineering*, Munich, 1979.
10. N. Wirth, *Programming in Modula-2*, Springer-Verlag, New York, 1985.
11. B. Liskov et al., *CLU Reference Manual*, MIT Tech. Report MIT/LCS/TR-225, 1979.
12. M. B. Wells, M. B. Hug, and R. Silver, Libraries as Programs Preserved within Compiler Continuations, *SIGPLAN Notices* 20(7), 83–92, 1985.
13. R. J. LeBlanc and C. N. Fischer, On Implementing Separate Compilation in Block-Structured Languages, *SIGPLAN Notices* 14(6), 139–143, 1979.
14. *Reference Manual for the Ada Programming Language*, U.S. Dept. of Defense Document No. ANSI/MIL-STD-1815A, 1983.
15. B. M. Brosgol, Some Issues in Data Types and Type Checking, in *Design and Implementation of Programming Languages: Lecture Notes in Computer Science*, vol. 54 (J. H. Williams and D. A. Fisher, eds.), Springer-Verlag, New York, 1977.
16. R. H. Yacobellis et al., The 3B20D Processor and DMERT Operating System: Field Administration Subsystems, *Bell System Tech. J.* 62(1), 323–339, 1983.
17. U. Ammann, The Zurich Implementation, in *Pascal—the Language and Its Implementation* (D. W. Barron, ed.), Wiley, New York, 1981.
18. R. E. Gantenbein, *Dynamic Binding of Separately Compiled Objects Under Program Control*, Ph.D. Thesis, University of Iowa, Iowa City, Iowa, August 1986.
19. T. A. Linden, The Use of Abstract Data Types to Simplify Program Modification, *SIGPLAN Notices* 11 (Spec. Issue), 12–23, 1976.
20. D. W. Jones, The Systematic Design of a Protection Mechanism to Support a High-level Language, Ph.D. Thesis, University of Illinois, Urbana, Illinois, 1980.

21. *An American National Standard: IEEE Standard Pascal Computer Programming Language*, IEEE Pascal Standards Committee of the IEEE Computer Society and ANSI/X359 of the American National Standards Committee X3, 1983.
22. M. P. Atkinson and R. Morrison, Procedures as Persistent Data Objects, *ACM Trans. Program. Lang. Syst.* 7(4), 539-559, 1985.
23. T. N. Turba and M. W. Whitelaw, The Pascal Exception Handling Proposal, *SIGPLAN Notices* 20(8), 99-106, 1985.

APPENDIX. The db-Pascal Language System

The primary extensions to Pascal-P found in the db-Pascal system are the separate compilation of procedure bodies and the dynamic binding of these bodies into an executing program. The extensions required modification of both the Pascal-P compiler and interpreter.

Upon encountering a *dynamic* compiler directive associated with a procedure name, the db-Pascal compiler saves the current compilation environment by writing all currently known names (which the compiler represents as Pascal records) and their associated bindings (represented as pointers to records) to a text file. This text file serves as the interface between each procedure stub and its implementation(s). When compilation of a separately defined procedure body begins, the environment saved in the interface is restored by reading in the text file and recreating the previously defined structures and their associated bindings. Once these bindings have been reestablished, variables based on these structures can be reinstantiated and reset to the appropriate values.

The interfaces between compilation units are stored within the UNIX hierarchical directory structure. The environment for each declaration of a dynamic procedure is written to a file named "sytab" in an immediate subdirectory of the directory containing the declaring program or procedure. This subdirectory has the same name as the dynamic procedure and contains the code for the procedure body or bodies to be bound to that procedure name. Multiple implementations of the same procedure can use the same environment and even the same name; separation is maintained by the directory organization. Dynamic procedure declarations nested within dynamic procedures continue this pattern. An *environment* statement precedes every compilation unit (except the top-level program) and lists the expected chain of directories from the one containing the top-level program to the one containing the unit. This statement is used by the compiler to check that the procedure body is in the correct directory and that all higher-level procedures in the definition chain are defined in the procedure's environment.

The Pascal-P interpreter on which the db-Pascal interpreter is based actually operates in two phases: assembly and execution. The P-code statements generated by the compiler are input from a file; each statement is assembled as it is input, and the resultant code is stored in an internal code array. Once assembly is complete, instructions in the code array are executed.

The extensions to the Pascal-P system to support dynamic binding involve the generation and interpreter support of two new P-code instructions, "lnk" (link) and "cdp" (call dynamic procedure). The compilation of a db-Pascal *link* instruction produces a P-code "lnk" as well as instructions that push the names of the declared procedure and the separately defined body onto the interpreter's internal stack. The execution of a "lnk" instruction initiates the loading of the named procedure body; the assembly phase of the interpreter is reentered at this point to assemble and load this code into a segment of predefined size in the code array. The use of segments allows the relinking of a procedure without requiring compaction of the code array and subsequent revision of references for names that are not relinked.

Once the assembly and loading of the dynamic procedure are complete, execution of the program recommences. When the dynamic procedure is called via the "cdp" instruction, a jump to the address associated with the loaded procedure body takes place in a manner similar to that for statically bound procedures. Relinking the procedure thus simply requires replacing the code in the segment allocated to a procedure body with new code from the file specified by the source code *link* statement. Note that once a dynamic procedure is allocated a segment in the code array, it does not relinquish it; there is no *unlink* statement in db-Pascal.

An example program written in db-Pascal follows. For each compilation unit, the source code and P-code compiler output is given. An interpreter-defined execution trace is also given that shows both the assembly and execution phases of the interpreter. Each trace step shows the P-code instruction being acted upon and its index in the code array; assembly-phase instructions are surrounded by asterisks (*), and execution-phase instructions are surrounded by dollar signs ($). Note that assembly of *separate* procedure code (which is much like *dynamic* code except that it cannot be relinked) takes place immediately after the inline code has been assembled and is initiated by an instruction at the end of the compiled code for the declaring segment. Assembly of dynamic code, however, is initiated by the execution of a "lnk" instruction and thus occurs while the program is running.

```
Jun  5 15:02 1986  main/testl.src Page 1


(*T+,C+*)
program testl(input,output);
const i = 'i2';
procedure dl ; dynamic;
procedure sl ; separate;
begin
  link 'dl' to 'il'; dl;
  link 'dl' to i; dl;
  sl
end (* testl *);


Jun 10 13:00 1986  main/code Page 1


1   5
 ent    1    1    6
 ent    2    1    7
 ldci             2
 lca'il                    '
 ldci             2
 lca'dl                    '
 ldci             0
i   10
 lca'                      '
 lnk    0    9    3
 mst              0
 cdp    0    9    3    0
 ldci             2
 lca'i2                    '
 ldci             2
 lca'dl                    '
 ldci             0
 lca'                      '
i   20
 lnk    0    9    3
 mst              0
 cdp    0    9    3    0
 mst              0
 cup    0    1    4
 retp             0
1   6=           11
1   7=           25
q
i   0
 mst                 0
 cup    0    1    5
 stp
q
s   4 * sl
```

Exhibit 1. A top-level unit.

```
Jun  6 11:05 1986    main/d1/i1/src Page 1

(*T+,C+*)
environment dynamic d1.
procedure i1;
  procedure d2; separate;
begin
  writeln(output,'hello world');
  d2
end (* d1 *).
```

```
Jun 10 13:00 1986  main/d1/i1/code Page 1

1   3
 ent    1    1    5
 ent    2    1    6
 lca'hello world
 ldci             11
 ldci             11
 lda    1          6
 csp              wrs
i   10
 lda    1          6
 csp              wln
 mst                0
 cup    0    1     4
 retp               3
1   5=             10
1   6=              9
q
s    4 d1/i1 d2
```

Exhibit 2. A dynamic procedure body "dl/il."

```
Jun  6 10:36 1986  main/d1/i1/d2/src Page 1


(*T+,C+*)
environment separate d1/i1/d2.
procedure d2;
begin
  writeln(output,'hello from me');
end.


Jun 10 13:00 1986  main/d1/i1/d2/code Page 1


1   4
 ent    1    1    5
 ent    1    1    6
 lca'hello from me
 ldci             13
 ldci             13
 lda    2          6
 csp             wrs
i   10
 lda    2          6
 csp             wln
 retp              0
1    5=            9
1    6=            9
q
```

Exhibit 3. A separate procedure "d2" declared in "il/dl."

```
Jun  6 10:38 1986  main/d1/i2/src Page 1

(*T+,C+*)
environment dynamic d1.
procedure i2;
begin
  writeln(output,'hello there');
end.
```

```
Jun 10 13:00 1986  main/d1/i2/code Page 1

1   3
 ent    1    1    4
 ent    2    1    5
 lca'hello there
 ldci             11
 ldci             11
 lda    1          6
 csp             wrs
i   10
 lda    1          6
 csp             wln
 retp              3
1    4=            9
1    5=            9
q
```

Exhibit 4. A dynamic procedure body "dl/i2."

```
Jun  6 10:44 1986  main/sl/src Page 1

(*T+,C+*)
environment separate sl.
procedure sl;
begin
  writeln(output,'hello world');
end.
```

```
Jun 10 13:01 1986  main/sl/code Page 1

1   4
 ent    1    1    5
 ent    2    1    6
 lca'hello world
 ldci           11
 ldci           11
 lda    1        6
 csp           wrs
i   10
 lda    1        6
 csp           wln
 retp            0
1   5=           9
1   6=           9
q
```

Exhibit 5. A separate procedure "sl" declared in the top-level unit.

```
ASSEMBLY (*) AND EXECU

*ent   *          3
*ent   *          4
*ldc   *          5
*lca   *          6
*ldc   *          7
*lca   *          8
*ldc   *          9
*lca   *         10
*lnk   *         11
*mst   *         12
*cdp   *         13
*ldc   *         14
*lca   *         15
*ldc   *         16
*lca   *         17
*ldc   *         18
*lca   *         19
*lnk   *         20
*mst   *         21
*cdp   *         22
*mst   *         23
*cup   *         24
*ret   *         25
*mst   *          0
*cup   *          1
*stp   *          2
*ent   *         26
*ent   *         27
*lca   *         28
*ldc   *         29
*ldc   *         30
*lda   *         31
*csp   *         32
*lda   *         33
*csp   *         34
*ret   *         35

$mst   $          0
$cup   $          1
$ent   $          3
$ent   $          4
$ldc   $          5
$lca   $          6
$ldc   $          7
$lca   $          8
$ldc   $          9
$lca   $         10
$lnk   $         11
```

Exhibit 6. Assembly and execution trace of "testl."

```
*ent  *        36
*ent  *        37
*lca  *        38
*ldc  *        39
*ldc  *        40
*lda  *        41
*csp  *        42
*lda  *        43
*csp  *        44
*mst  *        45
*cup  *        46
*ret  *        47
*ent  *        48
*ent  *        49
*lca  *        50
*ldc  *        51
*ldc  *        52
*lda  *        53
*csp  *        54
*lda  *        55
*csp  *        56
*ret  *        57
$mst  $        12
$cdp  $        13
$ent  $        36
$ent  $        37
$lca  $        38
$ldc  $        39
$ldc  $        40
$lda  $        41
$csp  $        42
hello world$lda     $        43
$csp  $        44

$mst  $        45
$cup  $        46
$ent  $        48
$ent  $        49
$lca  $        50
$ldc  $        51
$ldc  $        52
$lda  $        53
$csp  $        54
hello from me$lda     $        55
$csp  $        56

$ret  $        57
$ret  $        47
$ldc  $        14
$lca  $        15
$ldc  $        16
```

Exhibit 6. (continued).

```
$lca   $        17
$ldc   $        18
$lca   $        19
$lnk   $        20
*ent   *        36
*ent   *        37
*lca   *        38
*ldc   *        39
*ldc   *        40
*lda   *        41
*csp   *        42
*lda   *        43
*csp   *        44
*ret   *        45
$mst   $        21
$cdp   $        22
$ent   $        36
$ent   $        37
$lca   $        38
$ldc   $        39
$ldc   $        40
$lda   $        41
$csp   $        42
hello there$lda     $        43
$csp   $        44

$ret   $        45
$mst   $        23
$cup   $        24
$ent   $        26
$ent   $        27
$lca   $        28
$ldc   $        29
$ldc   $        30
$lda   $        31
$csp   $        32
hello world$lda     $        33
$csp   $        34

$ret   $        35
$ret   $        25
$stp   $         2
```

Exhibit 6. (continued).

The Journal of Systems and Software

Volume 8, Number 4, September 1988

Contents