THE DESIGN AND IMPLEMENTATION OF A RUN-TIME ANALYSIS AND INTERACTIVE DEBUGGING ENVIRONMENT

by

MARK SCOTT JOHNSON

B.A., University of California, Santa Barbara, 1973
M.S., University of California, Santa Barbara, 1974

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard.

THE UNIVERSITY OF BRITISH COLUMBIA

August, 1978

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Mark Scott Johnson

Department of Computer Science

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date 1978 August 14

Abstract

The design and implementation of a language-independent, interactive system to facilitate the analysis and symbolic debugging of computer programs written in high-level languages is presented. The principal features of the system, called RAIDE, are:

(1) Host source language independence is supported by the abstraction of language entities and constructs (for example, variables, constants, procedures, statements, and events) with a language interfacer providing language-dependent details;

(2) Translators can cooperate with RAIDE at varying levels of detail;

(3) The user interacts with RAIDE and an executing object program using an extendable debugging language, called Dispel;

(4) Primitive debugging actions are kept to a minimum and nonprimitive actions (for example, tracing, snapshots, and postmortem dumping) are provided by user-supplied and library procedures written in Dispel; and

(5) The implementation is aided by simulation of a virtual debugging machine, called SPAM.

To demonstrate RAIDE's feasibility, a prototype implementation was undertaken, including a SPAM simulator and the modification of two language translators (namely, Asple and BCPL) to interface with RAIDE. Besides describing the external and internal designs of the debugging system, the abstract machine,

and the debugging language, the thesis also discusses the advantages and shortcomings of each of these components. Numerous examples of debugging commands written in Dispel are given. Two significant side-effects of the research are reported: reflections on the software tools supporting the implementation, and suggestions for translator design to facilitate run-time debugging.

The thesis contains a substantial annotated bibliography and an extensive index.

Key Words and Phrases

Primary terms:

debuggers, debugging, debugging languages, debugging systems, Dispel, error detection, interpreters, program testing, RAIDE, SPAM, virtual machines

Additional terms:

abstract machines, automated programming aids, command languages, debugging-oriented machines, diagnostic systems, dynamic debugging, error analysis, error checking, execution profiles, interactive debugging, interactive systems, program analysis, programming systems, programming tools, pseudo-machines, simulators, software debugging, software tools, special-purpose languages, symbolic debugging, tracing

Computing Reviews categories: 4.13, 4.29, 4.42, 4.6

Table of Contents

List of Tables

List of Figures

Preface

/* The language is perpetually in flux: it is a living stream, shifting, changing, receiving new strength from a thousand tributaries, losing old forms in the backwaters of time. To suggest that a young writer not swim in the main stream of this turbulence would be foolish indeed.
-- William Strunk, Jr., [Stru 59:69] */

Altho readers of this thesis may at first be surprised to encounter certain unusual spellings, they should find after reading thru it that any initial difficulties will disappear even tho their academic training may still inhibit their acceptance of such usage.

Acknowledgements

/* Aliquid per Omnes.

-- motto of Calistralia */

Work on this thesis was made possible thru the financial support of the Department of Computer Science and the Computing Centre of the University of British Columbia. The final year was supplemented by a UBC Graduate Fellowship.

This dissertation was prepared using the Texture document processor [Text 78], which is based on the work of Peter van den Bosch [Vand 74]. Members of the Texture Support Group, particularly Tom Rushworth and Ted Venema, generously gave their time to assist me in understanding the occasionally gordian workings of Texture.

For their help in the formative years of my computer science training at the University of California, Santa Barbara, I am thankful to Chuck Dana, Leon Presser, and John R. White. Their encouragement significantly affected the course of my academic peregrination.

Several fellow graduate students made special contributions to my work: Greg Wilbur was one of my earliest and most ardent rock throwers. Not only did he point out shortcomings in the early proposals, but he also suggested many additions and changes. Ted Venema was likewise lavish with his constructive criticisms, particularly of the abstract machine, SPAM. Bill Appelbe was instrumental in shaping the direction of the early

drafts of this thesis. Its ethereal witticisms and ubiquitous acronyms are largely credited to (blamed on?) him. Graeme Hirst graciously accepted the onerous task of editing and proofreading. Altho I was at first overpowered by the magnitude of his suggestions ("enraged at his vicious attacks" more accurately describes my initial reaction), I am nevertheless confident that his editing has significantly improved the quality of this dissertation.

The extradepartmental members of my examining committee deserve commendation for their perserverance in studying what must have been to them a somewhat translucent document: J.J.F. Fournier, the chairperson of the examining committee; Helene E. Kulsrud, the external examiner; and Peter Lawrence, the public examiner. The members of my guidance committee also deserve special mention for their supervision: Alan Ballard, Sam Chanson, Hugh Dempster, Bob Fraley, Mabo Ito, and John Peck. I especially acknowledge the counsel of my faculty advisor, Harvey Abramson.

Finally, this thesis would not have been completed without the constant encouragement and moral support of Bill Appelbe, fellow Calistralian. When I reached the depths of the academic pit, his obstinate consolation gave me the stimulus to climb out and continue on. Bill, together with a host of friends too numerous to name, have made my sojourn in Canada a most memorable and blessed event.

Chapter I. Introduction

/* The best may slip, and the most cautious fall;
He's more than mortal that ne'er erred at all.
-- Rev. John Pomfret, Love Triumphant over Reason */

A. The Problem

The computer program development cycle consists basically of six phases: specification, functional decomposition, coding, testing, debugging, and evaluation. In recent years software engineers have concentrated on the second and third of these phases. This is exemplified by the attention paid to language design and the advocacy of structured programming. Altho these two points are certainly important in the production of quality software, their emphasis has detracted from progress in the other phases. It is clear that the best manner of dealing with a programming bug is to prevent its presence in the first place. Emphasizing the design of languages which make the inclusion of logic errors difficult and programming methodologies which encourage error-free programming is important. Nevertheless, it is still necessary for all production programs to enter the testing and debugging phases [Halp 65].

/* Debugging poorly designed and coded software systems is
veterinary medicine for dinosaurs.
-- Harlan D. Mills, [Mill 76b:271] */

This thesis presents the design and implementation of a language-independent, interactive system to facilitate the analysis and symbolic debugging of computer programs written in

high-level languages. The primary purpose of the debugging system is to provide an environment in which the computer programmer can detect the presence of errors and trace their cause and, thereby, reduce the total time spent on the debugging phase of the program development cycle. The concern of this thesis is the isolation and correction of errors during program execution. Detecting syntactic errors is not of interest here.

The debugging process contains three major components: recognition, diagnosis, and correction. Recognition of a program error requires knowledge of the anticipated results and program behavior, and realization of when these expectations are not met. Diagnosis involves the identification of the cause of an error, based on its symptoms. Altho the diagnostic process is largely intuitive, it is directed by answering such questions as "What precisely is wrong with the program?", "Where was the error detected?", "Under what conditions does the error manifest itself?", and "How extensive is the error?" [Brow 73:ch. 9]. To be useful, a debugging system must help to answer these questions. Debugging is akin to detectivery; a debugging system is akin to a magnifying glass.

A debugging system should be just one component of a collection of software aids available to programmers to improve the quality and quantity of their software [Balz 74, Chea 72, Clap 74, Davi 75, Ledg 76, Stoc 67, Teit 69, Wilc 76, Wino 75]. A debugging system must maximize the amount of usable information available to the programmer. The need for such a

system is apparent from the proliferation of language processing systems which provide inadequate run-time debugging aids. The user of such language processors must often resort to machine-language dumps or extensive user-supplied source language debugging and monitoring routines.

B. Previous Work

/* No vehement error can exist in this world with impunity.
-- James Anthony Froude, Spinoza */

This section describes briefly previous work in the area of debugging systems. The reader who is interested in a survey of debugging techniques and concepts should consult various of the following general references: [Brow 73, Evan 66, Fong 73b, Gain 69, Gell 75, Koch 69, Ledg 75, Mann 73, Pool 73, Rain 73, Satt 75, Schw 71, Scow 72, Seid 68, VanT 74].

The concept of an interactive, run-time debugging system is not new. Before the invention of batch processing, debugging was carried out directly at the operator's console using the console switches and lights to provide clues as to why a program behaved incorrectly. After the development of batch processing, this activity was automated to produce memory and register dumps [Kirs 74] and program traces. In the early days of interactive computing, the value of interactive debugging systems, which were first applied to the detection of errors in machine-language programs, became apparent. Virtually every interactive computing environment provides some such aid [Ball 77, Bern 68, Blai 71, Cris 65, Evan 65, Evan 66, Gain 69, Gall 74, Jöns 68,

Jose 69, Kuls 69, Stoc 65, UWM 73, Zimm 67]. The basic facilities provided by such systems include the ability to inspect the state of the run-time environment at particular points, to set breakpoints in the program, and to trace the execution of procedures and the values of variables. These systems often allow one to patch object programs during execution and to change the values of variables.

Machine-language debugging systems are of limited use to the person who programs in a high-level language. The trend away from machine-language programming has emphasized the need to develop debugging tools which provide the user with the ability to monitor the execution of a program within the terms of some high-level source language. The user wants to refer at run-time to the program using source-level names and notations without regard to the results of the translation process.

/* How truly sad it is that just at the very moment when the computer has something important to tell us, it starts speaking gibberish.
-- Gerald M. Weinberg, [Wein 71:208] */

One common way in which some run-time debugging aid for high-level languages has been provided is thru language-processor-supplied symbolic postmortem dumps (e.g., ALCOR Algol 60 [Baye 67, Ferl 71], PL/C [Conw 73, Morg 71], Snobol4 [Gris 71b], Algol-W [Satt 72, Site 71], and IMP [Step 74]). A symbolic postmortem dump gives the user a picture of the state of the execution of a program at the point of abnormal termination by printing the source-level names of variables and

their values in a form resembling the terminology of the source language. A symbolic traceback of the dynamic procedure invocation history is usually supplied as well. Using this information, it should be possible for the programmer to discern the cause of abnormal termination after the fact.

A significant advance in high-level language debugging has been afforded by the implementation of special diagnostic translators. These translators, often called student compilers because of their orientation toward novice programmers, sacrifice efficiency to provide extensive run-time checks to aid in the detection of program logic errors. Such errors include out of range subscripts, mismatched formal and actual parameters, and accessing uninitialized variables. Examples of translators oriented toward students include PLUTO [Boul 72], PL/C [Conw 73, Morg 71], and SPLINTER [Glas 68] for PL/I programs; DITRAN [Moul 67] and Watfiv [Cres 70] for Fortran programs; Watbol for Cobol programs; Algol-W [Satt 72, Site 71]; and FLASC [Thom 76] for Algol 68 programs.

Another common way in which the high-level language user has been given access to debugging tools is thru extensions to the host source language itself [Bair 75, Bull 72, Conw 73, Glas 68, Gris 71b, Gris 75, Hans 75, IBM 72, Kemm 76, Leed 66, Pull 69, Wolm 72]. These extensions have taken the form of debugging subroutines which the user can incorporate into a source language program, and syntactic extensions of the language which provide explicit debugging facilities. In either

case, it is the user's responsibility to code into the program information to facilitate its debugging. This approach suffers from the need to carefully preplan the debugging phase of the program development cycle. When the programmer discovers some abnormality in the program, it often becomes necessary to modify, retranslate, and reexecute the program to obtain the desired debugging information.

To overcome the disadvantages of user-programmed debugging aids, systems have been developed to provide high-level language users with essentially the same facilities available to the machine-language user in low-level debugging systems. High-level, interactive debugging systems have almost invariably been developed around one particular source language (e.g., MANTIS and Fortran [Ashb 73], EXDAMS and PL/I [Balz 69], the INTERLISP system [Bobr 72], PL/CT and PL/I [Conw 77, Moor 75], IBM's PL/I Checkout Compiler [Cuff 72], BUGTRAN and Fortran [Ferg 63], AIDS and Fortran [Gris 70], the LISP/MTS system [Hall 75, Wilc 74], HELPER and Fortran [Kuls 71], IF and Fortran [ORei 76], SIMDDT and Simula [Palm 77], DDS and Coral 66 [Pier 74], BAIL and Sail [Reis 75], TALK and CS-1 [VerS 64], and PL/I under Multics [Wolm 72]). Altho some of these systems have involved integrating debugging capabilities directly into an interpretive environment (e.g., INTERLISP, the PL/I Checkout Compiler, and IF), others have been designed explicitly as run-time systems manipulating translated code (e.g., EXDAMS, AIDS, HELPER, and DDS). Nevertheless, in all cases a debugging environment has

been established which is applicable to a single high-level language.

In view of the current state of interactive debugging, the advantages of a single system which is capable of dealing with programs written in various source languages should be obvious. A language-independent debugging system minimizes the duplication of effort needed in providing a debugging environment with the introduction of new programming languages. Such a system also minimizes the user's overhead in learning a new debugging system for each new language. A language-independent environment allows collections of programs written in more than one source language to be debugged in a uniform manner. Multilingual systems are not currently common, due in part to the language interface and debugging problems involved. The creation of a language-independent programming environment should result in software being written in the best available language for each subtask, rather than in the single language deemed most generally suitable to the entire project. Finally, such a debugging system serves as another off-the-shelf translator writing system resource available to language implementors in much the same way general-purpose lexic analyzers and parser generation systems are currently available.

C. Concurrent Work

Concurrent with the research reported in this thesis has been the design and development of the DAD (Do-All Debugger)

debugging system [Vict 76a, Vict 76b, Vict 77]. It is designed for use in a multiprogramming, multiprocessing network environment, in which components of the system being debugged may be written in different source languages and may be executing on different machines. This is accomplished primarily thru a rigid protocol which must be adhered to by all operating systems and language translators involved in the debugging process.

Chapter II. The Debugging System

```
/* As a famous philosopher once almost said,
   "Give me a suitable debugging environment and a
    tool-building facility powerful (and simple) enough,
    and I will debug the world."
                          -- Robert M. Balzer, [Balz 69:567] */
```

The debugging system described in this thesis is called RAIDE (Run-time Analysis and Interactive Debugging Environment), named after another product successfully employed to eliminate bugs from the user's environment.

A. Design Criteria

```
/* A good interactive debugging system must be difficult
   for the beginner to master [sic]. Its emphasis must
   be on completeness, convenience and conciseness,
   not on simplicity.
                          -- Butler W. Lampson, [Lamp 65:478] */
```

Many criteria have been taken into account in the design of RAIDE [Conw 73, Gain 69, Gris 71a, Mann 73, Pool 73]. The most important of these is that the system should be language-independent over a large class of user source languages. This criterion virtually dictates that the system run using translated code since to provide a system which can interpret a broad class of source languages is currently infeasible.

Altho the debugging system should be language-independent, it should appear language-dependent from the user's point of view, that is, the terminology should be that of the source language as much as possible. For example, if an array bound is exceeded during program execution, RAIDE should respond with a

message couched in the terminology of the source language. Thus, for Algol 68 the message "INDEX EXCEEDS THE BOUNDS SPECIFIED IN THE DEFINING OCCURRENCE OF THE MULTIPLE VALUE" might be produced.

Another goal is that RAIDE should be usable on multilingual collections of programs. Thus, the user can debug a set of programs written in more than one source language.

Another major design criterion is that the system should be oriented toward interactive processing, but it ought also to be usable in a batch processing environment without substantial difference from the user's viewpoint [Goul 75, Gran 66, Sack 68]. Obviously, many of the interactive features will be of marginal value from the batch stream. Nevertheless, there still exists a kernel of debugging facilities which are applicable to both environments.

Another major requirement is that all debugging should be done within terms of the source language(s). Knowledge of the underlying machine environment should not be necessary. Since the system will always respond to inquiries in symbolic terms, there is no need to provide core dumps, register dumps, and other similar machine-language debugging facilities.

One consequence of the preceding criterion is that language translators will need to supply the debugging system with a substantial amount of information concerning the source program. Data such as the identifier table, the type table, and even the

source code itself will need to be provided. Nevertheless, it should be possible for translators to provide information in increasing layers of completeness. This will enable RAIDE to be used even when the translators are not completely cooperating. If the user makes a request to which the system is unable to respond because of lack of translator-supplied information, RAIDE should return a message to that effect and allow the user to continue with other requests.

Another design criterion is that the system should be capable of supplying extensive information concerning the state of program execution. Such capabilities will exceed those of any preceding debugging system. It is for this reason that RAIDE is a run-time analysis and debugging environment, as opposed to simply a debugging system. Altho some analysis features (e.g., execution profiles [Conr 70, Inga 72, Knut 71]) border on the domain of program testing, as opposed to program debugging, such facilities are needed in a powerful program debugging environment; indeed, the dividing line between program testing and debugging is unclear. Nevertheless, the debugging information supplied should never be overwhelming. That is, the user should see only what is relevant with detailed information being provided upon a more explicit request.

The kernel of the system should be minimal, yet sufficient. That is, the system must include a set of primitives sufficient to carry out all of the desired debugging actions; but this set should not contain primitives which can easily be simulated

using a combination of the others. Some overlap may be necessary, however, to produce a set of primitives easily usable by the programmer. This design criterion should minimize the effort required to implement and transport the system.

To further minimize the implementation effort, the system must avoid duplicating resources provided by the host operating system. It is assumed the user will be familiar with the operating system so that alternative facilities built into the debugging system will merely be a source of confusion.

One final design criterion is that the user should not be required to make modifications to the source program to carry out debugging using RAIDE. It should be possible to specify at run-time everything the user may need to facilitate the debugging and testing of a program. This should not, however, preclude the programmer from designing some debugging aids into the program since doing so is a desirable implementation strategy [Ledg 75].

In summary, and for ease of future reference, the design criteria discussed above are listed in Table I.

B. Basic RAIDE Concepts

/* Programmers are habituated to sesquipedalian utterances.
-- Richard L. Wexelblat, [Wexe 76:333] */

The user interfaces with RAIDE solely thru the debugging system language. Before it is possible to describe this language, however, it is necessary to define some terms and to

1. The system should be source language independent.
2. The user interface should be language-dependent.
3. The system should be usable on multilingual collections of programs.
4. The system should be interactive-oriented, but usable in batch.
5. Debugging should be done symbolically in source language terms.
6. Translators should be allowed to supply information in successive layers of completeness.
7. The system should provide extensive analytic information at run-time.
8. Information supplied by the system should be concise and pertinent to the user's request.
9. A small, usable, and sufficient set of primitive actions should be supplied.
10. Operating system resources should not be duplicated.
11. No translation-time modifications to the source program should be necessary to carry out debugging.

Table I. Debugging System Design Criteria

explain some basic RAIDE concepts which are reflected in the debugging system language itself.

A program is a collection of procedures which interact to perform one primary task. Thus, the term as applied here is equivalent to system of programs used by many people. In other words, RAIDE is designed to debug a single program during one interactive session. This program may, however, consist of a

main procedure and any number of subprocedures. It should be remembered that, from RAIDE's viewpoint, it is not necessary for these subprocedures to have all been written in the same high-level language.

One concept which is basic to a proper understanding of RAIDE is the distinction between a specific and a generic. A specific is a reference to a particular entity in the user's program. For example, X might be one particular variable, 10 might be one particular constant, and X := 10 might be one particular statement. Thus, X, 10, and X := 10 are specifics. On the other hand, a generic is a set of entities within the user's program all of one homogeneous variety. For example, VARIABLE is a reference to a class of entities of which X is one particular member. Similarly, CONSTANT and STATEMENT are examples of generics. Furthermore, it is convenient to divide generics into two classes. A segment-generic is a generic which refers to some executable segment of a user's program code. For a block-structured language, typical segment-generics are PROCESS, PROCEDURE, BLOCK, and STATEMENT. A data-generic refers to a particular class of data which the user's program can manipulate. Thus, for a block-structured language, VARIABLE, PARAMETER, and CONSTANT are examples of data-generics.

Generics are host-language-dependent. The only presupposition which RAIDE makes concerning them is that there are two classes: segment and data. For each language which is to be interfaced to RAIDE, it is necessary to supply a set of

generics. Table II contains examples of generics which might be defined for several well-known programming languages.

language	segment-generics	data-generics
ALGOL 68	PROCESS	VARIABLE
	PROGRAM	IDENTITY
	ROUTINE	DENOTATION
	CLAUSE	YIELD
	UNIT	
FORTRAN	PROGRAM	VARIABLE
	SUBROUTINE	ARGUMENT
	FUNCTION	ARRAY
	STATEMENT	CONSTANT
LISP 1.5	FORM	ATOM
	FUNCTION	ARGUMENT
		PROPERTY_LIST
PL/I	TASK	VARIABLE
	PROCEDURE	PARAMETER
	BLOCK	RETURNED_VALUE
	ON_UNIT	CONSTANT
	STATEMENT	
SNOBOL4	FUNCTION	VARIABLE
	PREDICATE	ARGUMENT
	OPERATOR	LITERAL
	STATEMENT	KEYWORD

Table II. Examples of Possible Generics for Various Languages

The reader may notice a subtle bias of RAIDE which is evidenced by the distinction between segment- and data-generics. Languages which make little or no distinction between executable code and data (e.g., Lisp) are not natural candidates for inclusion within RAIDE even tho their inclusion will not necessarily be precluded by the system design.

Related to the concept of a generic is that of an incident. An incident is some activity which is associated with a generic. The system defines both segment-incidents and data-incidents to correspond to segment- and data-generics. The activities which can be associated with the segment-generics are ENTRY and EXIT. Thus, it is possible to speak of PROCEDURE ENTRY or STATEMENT EXIT. Similarly, the activities associated with the data-generics are ACCESS and UPDATE for referencing and changing the value of an item of data. Note that it is not necessary that every data-generic possess both data-incidents. In particular, CONSTANTs cannot be UPDATEd. Unlike generics, incidents are fixed within RAIDE and are not specified by a host language interfacer.

Another system concept is that of an event. An event is any occurrence which can cause the user's program to stop execution leaving RAIDE in the interactive request mode. Examples of possible events are pressing the attention interrupt key on the keyboard, an attempt to access a nonexistent element of an array in a program, and changing the value of some program variable. An event which is defined independently of some particular source program (e.g., ATTENTION_INTERRUPT and OVERFLOW) is called an exception. Other events are described by expressions (e.g., "**when** x>y" to represent the event occurring when the value of the variable 'x' exceeds that of the variable 'y'). A number of exceptions are predefined, but language-dependent exceptions can also be defined. For example, a

language interfacer for Snobol4 might define STATEMENT_FAILURE.

When the user is interacting directly with RAIDE, many possible actions can be requested. An *action* is any primitive operation which the system can perform. For example, **display** is an action which causes information to be displayed to the user. **execute** is another primitive action; it initiates execution of the user's program by leaving the interactive request mode. A *deferred action* is any action which does not occur immediately upon its specification. Deferred actions allow the user to set traps (or demons) within the program. Whenever the event associated with a deferred action occurs, the action is initiated by the system. Any action which RAIDE can perform immediately can also be deferred. The system maintains all deferred actions on the *deferred action list*. Normally, the user must explicitly remove an action from this list using the **cancel** action; nonetheless, some deferred actions are automatically removed from the deferred action list when the associated event occurs. Such an action is called a *transient deferred action*.

Altho the basic actions of the system are fixed, system extendability is provided by debugging procedures. A *debugging procedure* is a subroutine written using the primitive actions of RAIDE. A debugging procedure can be called in any context in which an action can occur. Thus, it is possible to provide the user with a library of debugging procedures which perform any standard debugging operations which are not provided as

primitive actions in RAIDE. For example, there is no primitive RAIDE action to cause all procedure invocations to be traced, altho it is possible to write a debugging procedure to accomplish this. The system maintains all debugging procedure (and variable) declarations on the declaration list.

In addition to the debugging procedures provided by a standard library and those written by the user, the system itself provides several built-in functions. These functions are primarily concerned with interrogating the current state of the debugging system, such as obtaining the name of the most recently executing user procedure. The built-in system functions are listed in Appendix A.

To allow the user greater freedom in identifying specifics and generics within the executing program, the system defines a default reference point. The reference point always indicates the program segment currently active when a RAIDE action is initiated. Thus, the reference point is used to disambiguate generic and specific references. For example, if the program contains three procedures, each of which contains a distinct variable named 'foo', the user would always have to qualify a reference to 'foo' to indicate which of the 'foo's is desired if there were no reference point. If execution of the program is suspended, a reference to 'foo' will automatically (thru the reference point) refer to the 'foo' in the procedure in which execution was suspended, or to the closest one accessible at the point of suspension. A user can also establish a reference

point different from the default.

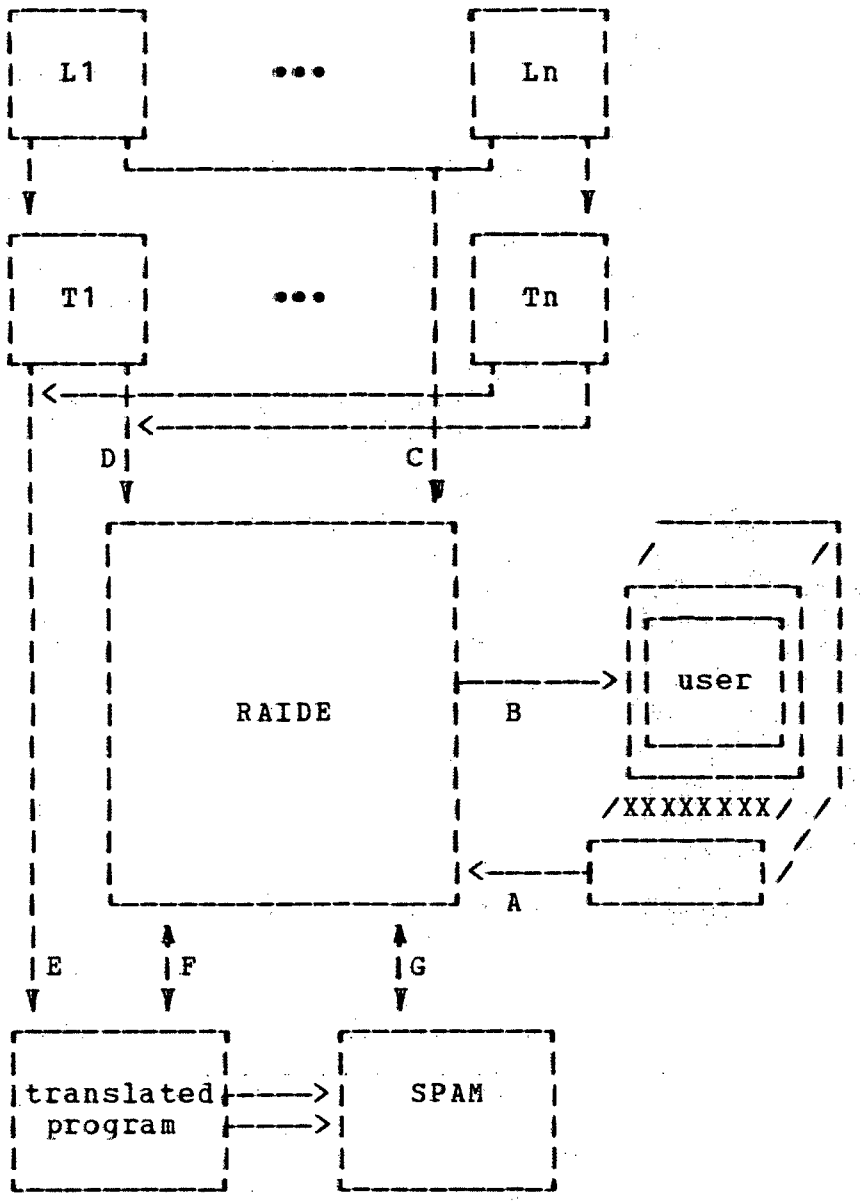
Finally, the environment is the state of the executing system at any particular time. An environment includes the values of all of the data in the user's program, the default reference point, and the deferred action list. It is possible in RAIDE to save the current environment by naming it and placing it on the accessible environment list. Using a primitive system action, the current environment can always be replaced by one on the accessible environment list. This facility allows the user to checkpoint the state of the system and to subsequently restore it.

Once the basic concepts described above are understood, the RAIDE user should be capable of learning the system debugging language.

C. Overview of the Implementation

Before proceeding with a description of the debugging system language, it is desirable to present an overview of the implementation of the debugging system to insure an understanding of its components when they are presented in detail. A complete description of the implementation is contained in Chapter V.

Figure II-1 illustrates the debugging environment provided by the system. In this figure, the lines connecting boxes indicate intermodular communication. Each interface is assigned



A requests from the user (written in Dispel)
B RAIDE's responses to user requests
C information supplied to RAIDE by the language interfacers
D information supplied to RAIDE by the translators
E information supplied to SPAM by the translators
F analysis and modification of the user program by RAIDE
G interactions of RAIDE and SPAM

Figure II-1. Overview of the Implementation

a letter for identification. The boxes L1, ..., Ln represent the languages which are interfaced to RAIDE. For each language using the debugging system, language interfacers must provide RAIDE with information that is dependent on each host language (C). This information, which is summarized in Table III,

1. definitions of the segment- and data-generics
2. run-time error messages
3. identification of any language-dependent exceptions
4. definitions of generic-related and language-dependent system functions

Table III. Information Supplied to RAIDE by the Language Interfacers

includes the definitions of the segment- and data-generics of each language, the set of language-oriented run-time error messages, identification of any language-dependent exceptions, and the definitions of any generic-related and language-dependent system functions.

For each language Li interfaced to the system, there must exist at least one translator Ti which has been constructed or modified to furnish RAIDE with information to facilitate run-time debugging. As stated previously, the system runs using translated source code to expedite implementation and to insure source language independence. To accomplish this, a virtual

debugging machine (which is called SPAM) has been defined and is described in Chapter IV. Each language translator interfaced to the system must supply information both to SPAM and directly to RAIDE. Table IV outlines the information which the translators

1. SPAM machine code for the user program
2. descriptions of all program code segments and data items
3. run-time type table
4. run-time bounds table

Table IV. Information Supplied to SPAM by the Translators

must furnish to SPAM (E). Translators using RAIDE must map the user's source program into SPAM machine code. In addition, they must supply the virtual machine with a description of all program code segments and data items, a type table, and a bounds table for array subscript checking. The translators must furnish RAIDE (D) with symbol table information, a description of the static (i.e., textual) structure of the source program, and the source program code itself. This information is outlined in Table V.

The user directs the debugging system (A) from an interactive terminal using the debugging system language, which is called Dispel. This language is described in detail in Chapter III. RAIDE's responses to user requests (B) are directed back

1. run-time symbol table
2. static structure of the program in terms of the segment-generics
3. source program code

Table V. Information Supplied to RAIDE by the Translators

to the terminal. To process the user's requests, RAIDE uses the information supplied to it by the language interfacers (C), the language translators (D), the translated program (F), the virtual machine (G), and its own record of program and system execution. To set program traps, to obtain run-time analysis information, and to modify the values of program variables, RAIDE must interact directly with the translated program (F). Nevertheless, the translated program is primarily controlled by the virtual machine. When SPAM detects an error during program execution, it supplies RAIDE with information concerning the error and the state of the program (G). Likewise, RAIDE can interact directly with SPAM to inquire concerning the state of program execution and to make certain modifications to it (G).

System design criterion 6 states that translators should be allowed to supply information in successive layers of completeness. The discussion concerning the information which must be supplied to RAIDE, as outlined in Tables III thru V, seems to imply that fully cooperating translators are necessary; however, this is not the case as the discussion so far has centered

around the most complete debugging configuration. Table VI outlines the levels of support at which the debugging system is capable of running.

Seven successively more complete layers of debugging support are defined. In the basic machine level, RAIDE is not available and the virtual machine runs with the minimum of overhead. At this level the translators are only required to generate SPAM code and descriptors, and to supply the type and bounds tables. The basic machine level corresponds to a user program running on a bare machine that is more sophisticated than many currently in existence. The next level of support is the machine debugging level. Altho RAIDE is again not available, the virtual machine will make additional run-time checks (e.g., scope violations and mismatched parameters) which are not performed in the most basic mode of machine execution. These additional checks are not required for the machine to execute properly, but are intended to further the debugging process.

RAIDE itself is not a proper component of the system until the third level of debugging support, the simple symbolic level. At this level RAIDE is only capable of reporting errors and executing simple interactive requests, such as inspecting the value of a variable. The language interfacers must supply RAIDE with enough information to properly interpret and report the error conditions detected by SPAM. The translators must supply some symbol table information so that detailed error messages

level of support	module	additional activity
1. basic machine	translator	generates SPAM code and descriptors, and supplies type and bounds tables
	SPAM	executes code normally checking for basic run-time errors
	RAIDE	not available
2. machine debugging	SPAM	makes additional checks on operands and interrupt flags
	RAIDE	not available
3. simple symbolic	interfacer	supplies generic information and error messages
	translator	supplies some symbol table information
	RAIDE	available for simple interactive requests
4. symbolic debugging	interfacer	identifies language-dependent exceptions and defines generic-related system functions
	translator	supplies complete symbol table
	RAIDE	available for more complex interactive requests
5. full debugging	translator	describes static program structure
	RAIDE	can respond to requests involving static program structure
6. full analysis	RAIDE	keeps run-time analysis statistics
7. deluxe debugging	translator	supplies source program code
	RAIDE	can respond to requests to display source code at run-time

Table VI. Levels of Debugging Support

can be generated in symbolic terms. If the language interfacers identify the language-dependent exceptions and define the generic-related system functions, and the language translators supply the complete symbol table, RAIDE is capable of initiating more complex interactive user requests, such as displaying the current state of the executing program. This corresponds to the symbolic debugging level.

If the translators supply a description of the static structure of the user program (e.g., information concerning the nesting of segment-specifics and the scope of variables), the full debugging level is entered. At this level, RAIDE is capable of initiating requests which involve the static structure of the program (e.g., dynamic flow tracing, postmortem tracebacks, and the setting of intricate traps). In the full analysis level, RAIDE keeps run-time analysis statistics, such as how many times each program statement is executed or each variable is accessed. In the deluxe debugging level, translators supply the source program code. This enables RAIDE to respond to a user request to display portions of the source code at run-time.

For the most part, the user will not need to be concerned with the levels of debugging support, since the system will always inform the user if some requested action cannot be performed due to lack of information.

Chapter III. The Debugging System Language

For the user to communicate with RAIDE, it is necessary to provide a language embodying the primitive system actions and supporting future extensions to the debugging environment. The debugging language of RAIDE is called Dispel (DebuggIng SPEcification Language), a name chosen more for its mnemonic than for its acronymic value. The first section of this chapter describes Dispel's design criteria. This is followed by a description of the syntax and semantics of the language itself, a section containing several examples of debugging requests coded using Dispel, and finally a discussion of how well the language fulfills its goals.

A. Design Criteria

The most obvious design criterion of Dispel is that it should reflect the design criteria of RAIDE itself and should embody the basic RAIDE concepts. In particular, the language must be interactive-oriented and should only contain a small number of primitive actions. It is also necessary, however, that the language satisfy other criteria [Brad 68, Gris 71a, Mann 73, Schw 71].

Dispel should be uniform. Uniformity implies that the syntax is clean and consistent since it connotes the existence of few syntactic exceptions and special cases. A construct is always used in the same way regardless of its context. Impor-

tant attributes of a uniform language are that it can be described succinctly (cf., the Algol 68 syntax chart [Watt 74]) and that it is easy to learn and to implement [VanW 76:sec. 0.1.2].

Altho the language should be uniform, it must also be reasonably simple to use. Uniformity must not introduce unnecessary semantic complexity (or deleterious superfluities, as the Algol 68 Revised Report [VanW 76] says). If the language is complex, the programmer may have to debug the debugging procedures! This is clearly undesirable since the overall intent of the system is to reduce the total debugging effort. A complex system is frequently more of a liability than an asset.

In addition to simplicity, the language must be readable and should appear to be a natural extension of the host source languages. But care must be taken to insure that the debugging language does not become easily confused with the host languages.

To fulfill system design criterion 8 (viz., conciseness and pertinence of the information supplied by the system), the syntax of Dispel should discourage voluminous output. The user should be encouraged to carry out the debugging process thoughtfully and to avoid debugging by deluge. Designing a syntax which discourages excessive output may conflict with the goal that the language be concise. It is not obvious where the compromise between these two criteria should be made.

The final, important design criterion of Dispel is that the syntax should allow for possible extensions of the debugging environment to meet new needs not currently envisioned. Language extendability will be provided thru debugging procedures. If the debugging system and its language supply sufficient primitive actions, debugging procedures will afford the requisite language extendability. Thus, it should not be necessary to change the syntax of the language itself to fulfill future requirements.

In summary, the design criteria discussed above are listed in Table VII. After detailing the syntax and semantics of Dispel and giving a number of examples, a discussion of how well these criteria are met will be made.

1. The language should be interactive-oriented.
2. Only a small set of primitive actions should be supplied.
3. The language should embody the basic debugging system concepts.
4. The syntax should be uniform.
5. The language should be simple to use.
6. It should be easy to read and understand debugging programs.
7. The language should discourage voluminous output.
8. The language should be extendable.

Table VII. Debugging Language Design Criteria

B. Syntax and Semantics

In the ensuing discussion, an outline of the syntax of Dispel is presented using a variation of the syntactic metalanguage Backus-Naur Form (BNF). Two extensions to BNF are used: square brackets ([ ]) to delimit an optional construct and braces ({}) to group together several constructs which are treated as a single unit. Also, the keywords of Dispel (i.e., the terminal symbols) are in boldface and nonterminal symbols are delimited by angle brackets (<>). A more complete syntactic specification of Dispel, using a syntax chart, is in Appendix B. For lack of a less prosaic substitute, the semantics of Dispel is presented using the semantic metalanguage English. In the examples to follow, upper-case identifiers represent language-dependent entities (e.g., generic names), incidents, exception names, and the names of system functions. Lower-case identifiers represent program-dependent entities (e.g., specific names) and the names of debugging variables and procedures.

The fundamental syntactic entity is an <utterance>.

```
<utterance> ::= <explanation> .
              | <inquiry> .
              | <declaration> .
              | <definition> .
              | <command> .
```

The <utterance> is the basic unit of interactive input. Until the full-stop (.) is encountered, the <utterance> is checked only for syntactic correctness; the full-stop initiates semantic interpretation.

The syntax of an <explanation> is:

<explanation> ::= **explain** <keyphrase> .

An <explanation> furnishes the user with an explanation of some component of the system, as signified by the <keyphrase>. The possible <keyphrase>s will not be listed, but they include such terms as "command", "break", "display", "specific", and "deferred action". The <explanation> <utterance> provides the rudiments of an interactive document querying facility.

An <inquiry> has the syntax:

<inquiry> ::= **inquire** <sentence> .

An <inquiry> enables a link into a natural language interrogation system which provides information concerning the debugging system and the state of program execution by means other than thru Dispel <command>s. The following is a sample <inquiry>.

inquire "In the procedure FOO, what is the value of the variable Z the first time X is greater than Y?".

An <inquiry> can be used to extract information which the user prefers to express in natural language terms. The syntax of a <sentence> is unspecified as far as the debugging system is concerned.

A <declaration> specifies debugging variables, as opposed to user program variables.

```
<declaration> ::= <integer-declaration>
                | <specific-declaration>
<integer-declaration> ::= integer [ ( <expression> ) ] <id-list>
<specific-declaration> ::= specific [ ( <expression> ) ] <id-list>
<id-list> ::= <identifier> [ , <id-list>]
```

As indicated above, the two types of debugging variables are integers and specifics. The optional <expression> denotes the declaration of debugging array variables; it must evaluate to an integer value indicating the size of the array. Elements of debugging array variables are selected using subscripts. Examples of debugging variables are given in the last section of this chapter, at which time their utility will become more apparent.

A <definition> identifies a debugging procedure.

```
<definition> ::= define <procedure-id>
                    [ ( <declaration-list> ) ] as <command>
<declaration-list> ::= <declaration> [ ; <declaration-list>]
```

The optional <declaration-list> specifies the formal parameters of the debugging procedure. Whenever initiation of the procedure (as identified by its <procedure-id>) is indicated, the <command> associated with the <definition> is initiated.

The most important <utterance> of Dispel is the <command>.

```
<command> ::= [<when-clause>] <action>
<when-clause> ::= [<label-id> :] <when>
<when> ::= when <condition>
         | on <exception-list>
         | before <specific-incident-list>
         | after <specific-incident-list>
<exception-list> ::= <exception> [, <exception-list>]
<specific-incident-list> ::= <specific-incident>
                              [, <specific-incident-list>]
```

The <when-clause> of a <command> causes the associated <action> to become a deferred action. The <label-id> of the <when-clause> is used to remove the action from the deferred action list. For a deferred action, whenever the specified event occurs, the associated <action> is initiated. <action>s which

are not deferred are initiated immediately upon entry by the user.

If a <when-clause> is contained within a debugging procedure, any actions deferred by execution of the procedure are not canceled when the procedure is exited. Thus, the establishment of deferred actions is independent of procedure scope. Furthermore, all <label-id>s are global and can be referenced regardless of their textual scope. If several <when-clause>s have been executed for the same event, only the most recently deferred action will be initiated when the event occurs. Deferred actions for the same event are stacked; the <cancel-action> is used to remove an action from the stack.

There are four forms of the <when-clause>. The **when** <condition> form specifies that the associated <action> is to be initiated whenever the indicated <condition> becomes true. <condition> can be any expression which yields a boolean value. For example, "**when** x>y **break**" means that the interactive request mode is to be entered (i.e., a break is to occur) whenever the value of the variable 'x' is greater than that of 'y'. The means by which this trap is established is implementation-dependent. The **on** <exception-list> form of the <when-clause> specifies that the associated <action> is to be initiated whenever one of a set of possible <exception>s occurs. The possible <exception>s are not listed here, but they include ATTENTION_INTERRUPT, OVERFLOW, and ZERODIVIDE. For example, "**on** ATTENTION_INTERRUPT **quit**" will cause RAIDE to be exited if the

attention interrupt key is pressed. The **before** and **after** forms of the <when-clause> cause the associated <action> to be initiated before or after some particular incident occurs. For example,

after each STATEMENT **in** foo **break.**

sets a trap after each executable statement in the procedure 'foo'. "**each** STATEMENT **in** foo" is a <specific-incident>; it pinpoints the setting of a trap. The complete syntax of <specific-incident> follows.

```
<specific-incident> ::= <specific> [<generic-incident>]

<specific> ::= <variable>
             | [each] <generic> [<segment-qualifier>]
<generic-incident> ::= <segment-incident> | <data-incident>
<segment-incident> ::= ENTRY | EXIT
<data-incident> ::= ACCESS | UPDATE

<variable> ::= [<generic> :] <unqualified-variable>
                 [<segment-qualifier>]
<unqualified-variable> ::= <subscripted-variable>
                              [. <unqualified-variable>]
<subscripted-variable> ::= <variable-id> [ ( <expression-list> ) ]
<segment-qualifier> ::= in <variable>
<expression-list> ::= <expression> [, <expression-list>]
```

The **each** form of the <specific> is used to transform a <generic> into a sequence of <specific>s. The possible <generic>s are language-dependent. It should be noted that "**each**" generates a sequence of <specific>s based on their textual organization. For example, "**each** VARIABLE **in** foo" identifies all variables declared in the procedure 'foo', not those referenced in 'foo'. Likewise, "**each** PROCEDURE **in** foo" identifies the procedures declared local to 'foo', not those invoked from 'foo'.

It is important that the user understand at what times <specific>s are evaluated since this greatly affects the specification of <utterance>s. There are three forms of <specific> evaluation: static, dynamic, and continual. During static evaluation, <specific>s are evaluated once, upon specification of the <utterance> containing them. The <specific>s in the **before** and **after** forms of the <when-clause> and all those not deferred or contained within debugging procedures are evaluated statically. During dynamic evaluation, <specific>s are evaluated once, upon initiation of the <command> containing them. The <specific>s contained within deferred actions and within the bodies of debugging procedures are evaluated dynamically. During continual evaluation, <specific>s are evaluated repeatedly until the deferred actions containing them are canceled. The <specific>s within the **when** <condition> form of the <when-clause> are evaluated continually.

The basic actions of RAIDE are specified in Dispel as follows.

```
<action> ::= <compound-action>     | <quit-action>
           | <break-action>        | <reference-action>
           | <call-action>         | <restore-action>
           | <cancel-action>       | <save-action>
           | <display-action>      | <set-action>
           | <execute-action>      | <skip-action>
           | <for-action>          | <system-action>
           | <if-action>           | <while-action>
           | <input-action>
```

These <action>s are described below in alphabetic order of the keyword which begins each.

The <compound-action> is a syntactic device to allow the grouping of multiple actions into one.

<compound-action> ::= **begin** [<declaration-list> ;]
<command-list> **end**
<command-list> ::= <command> [; <command-list>]

<declaration>s within a <compound-action> have a scope and lifetime local to the defined block. The <compound-action> allows an entire series of <command>s to be specified wherever a single <action> can be specified.

The <break-action> indicates that processing is to revert to the interactive request mode. The <break-action> is designed for use within deferred actions and debugging procedures only. Its syntax is:

<break-action> ::= **break** [<message>]

When a <break-action> is initiated, the deferred action or debugging procedure containing the **break** is terminated, the <message> is displayed on the output display device, and RAIDE enters the request mode waiting for the specification of some <utterance>.

The <call-action> causes a debugging procedure to be called after evaluation of its arguments.

<call-action> ::= **call** <procedure-id> [(<expression-list>)]

Recursive debugging procedures are supported. The parameter passing mechanism is call by reference, as in PL/I [IBM 70].

The <cancel-action> causes one or more deferred actions to be removed from the deferred action list.

<cancel-action> ::= **cancel** [<cancel-list>]
<cancel-list> ::= {<label-id> | <variable>} [, <cancel-list>]

cancel without an argument is used only within deferred actions; it causes the deferred action which contains it to be canceled. Deferred actions which were given <label-id>s can be canceled by listing the <label-id> (e.g., **cancel** foo). If a deferred action was not labeled, the system will automatically assign a <label-id> to it; this label is used to cancel it. The system function DEFERRED_ACTION_LIST (cf., Appendix A) can be used as a <variable> to also identify actions to be canceled.

The <display-action> causes one or more pieces of information to be displayed on the output display device, or on some auxiliary file or device.

<display-action> ::= **display** <what-list> [**on** <file-name>]
<what-list> ::= <expression> [**as** <type>] [, <what-list>]

When the <display-action> is initiated, the designated <expression>s are printed on the file <file-name>, which defaults to the output display device. The syntax of <file-name> is operating-system-dependent. Unless indicated otherwise, an <expression> will be displayed in a format suitable to its type attributes; the <type> clause can be used to override this default. Numerous examples of the <display-action> are given in the next section. The <display-action> works by maintaining a display buffer of all <expression>s to be displayed. This buffer is presented to the user either when one of the format functions PAGE or LINE is **display**ed (cf., Appendix A) or when the interactive request mode is entered.

The <execute-action> causes the user's program to be executed, leaving the interactive request mode of RAIDE.

<execute-action> ::= **execute** [<expression> [<segment-generic>]]
| **execute while** <condition>

execute without an argument causes the program to be resumed at its point of suspension (i.e., the default reference point), or at its beginning if there is no suspension point. The other forms of the <execute-action> are the same except that execution limits are imposed. These execution limits can be thought of as transient deferred actions which cause a **break** when the appropriate event occurs. The first alternative of the <execute-action> causes a particular number of segment-generics to be executed. For example, "**execute** 3 STATEMENTS" causes the user's program to be resumed for the execution of exactly three source-level statements of the current procedure. <segment-generic> defaults to the lowest-level segment-generic defined for the host source language. This form allows the user to step thru execution of the program. The other alternative of the <execute-action> causes the program to be resumed until the indicated <condition> becomes false. It should be noted that resumption of the user's program at an arbitrary location is not allowed in RAIDE. It is felt that such a facility is not well structured and is error-prone, and that it should be avoided in the debugging phase just as unrestricted transfers of control should be avoided in source language programs. With the addition of the <skip-action>, the user should find this constraint tolerable.

The <for-action> is a control structure which allows repetitive initiation of some <action>.

<for-action> ::= **for** <specific-list> -> <specific-id>
do <action>

The <specific-id> identifies a specific variable which has a scope encompassing the repetitive <action>. It is implicitly defined to be of type **specific**. The <specific-list> identifies a set of specifics, the members of which are successively assigned to the <specific-id>. Each time the repetitive <action> is initiated, <specific-id> will identify a different member of the set. The programmer cannot explicitly change this association within the repetitive <action>.

The <if-action> is a control structure which selects one of two possible actions for initiation based on some conditional expression.

<if-action> ::= **if** <condition> **then** <action> [**else** <action>] **fi**

If the indicated <condition> is true, the <action> following **then** is initiated; otherwise the <action> following **else**, if present, is initiated.

The <input-action> causes a series of <utterance>s to be read from some file or device. Its syntax is:

<input-action> ::= **input** [<file-name>]

<file-name> is the name of a file which contains a collection of <utterance>s which are initiated. The syntax of <file-name> is operating-system-dependent, and it defaults to some operating-system-dependent value. The <input-action> is most useful for

specifying libraries of canned debugging procedures.

The <quit-action> causes the debugging system to be exited and control returned to the operating system. Its syntax is:

<quit-action> ::= **quit** [<message>]

If present, the <message> is displayed on the output display device before termination of the debugging session.

The <reference-action> establishes a reference point which is different from the default.

<reference-action> ::= **reference** [<variable>]

If a <variable> is indicated, it is used to disambiguate references to specifics and generics until the next <reference-action> or <execute-action> is initiated. **reference** without an argument causes the reference point to revert to the default reference point (i.e., to the reference point when the interactive request mode was entered most recently).

The <restore-action> reestablishes an environment which was previously saved by a <save-action>.

<restore-action> ::= **restore** <file-name> [**saving** <file-name>]
<save-action> ::= **save** <file-name>

Once an environment has been named and saved in a file, RAIDE can restore that environment using the <restore-action>. The **saving** clause of the <restore-action> is a shorthand for specifying a <save-action> immediately preceding a <restore-action>. The syntax of <file-name> is operating-system-dependent.

The <set-action> changes the value of some user program or debugging variable.

<set-action> ::= **set** <variable> **to** <expression>

If a user program or debugging <variable> is designated, its value is changed to that of the <expression>. If the <variable> designates a <specific-id> (which must have been declared with a <specific-declaration>), the variable is made to reference the specific indicated by the <expression>.

The <skip-action> causes the value of the default reference point to change by skipping execution of the remainder of some segment of code.

<skip-action> ::= **skip** [<segment-generic>]

The <skip-action> affords the user the opportunity to abort execution of a segment of the currently suspended program. This is especially useful when the user discovers a program error, but desires to continue execution to discover other possible errors. For example,

skip PROCEDURE.
execute.

will cause the user's program to resume execution in the procedure which called the one in which execution was originally suspended. Any deferred actions associated with the segment's exit will be initiated as if the segment terminated by normal program execution.

The <system-action> allows the execution of RAIDE to be temporarily suspended to allow control to pass to the operating

system.

<system-action> ::= **system** [<system-command>]

RAIDE will subsequently be reentered with the environment it had prior to initiation of the <system-action>. If the host operating system allows such a facility, the <system-command>, if present, is the single command executed before RAIDE is automatically reentered.

The <while-action> is another control structure which allows repetitive initiation of some <action>.

<while-action> ::= **while** <condition> **do** <action>

Until the indicated <condition> becomes false, the specified <action> will be initiated repeatedly.

All of the primitive actions of RAIDE have now been outlined. The omission of a trace primitive action, present in virtually all previous debugging systems, should be noted. In Dispel a trace primitive is unnecessary: tracing can be implemented using the <when-clause> and the <display-action>. Examples of tracing debugging procedures are given in the next section.

C. Examples

This section contains numerous examples of RAIDE debugging requests coded in the debugging system language Dispel. The examples are designed to show the extent and power of RAIDE as well as to demonstrate Dispel. In the examples below, it will be assumed that the host source language is block-structured and

contains the segment-generics PROCEDURE, BLOCK, and STATEMENT and the data-generics VARIABLE, PARAMETER, and CONSTANT. Furthermore, the existence of three segment-generic related system functions is assumed. They are CURRENT_PROCEDURE, CURRENT_BLOCK, and CURRENT_STATEMENT, yielding specifics indicating the procedure, block, or statement most recently active when the function is invoked. Like the generics, these generic-related functions are language-dependent and must be supplied for each language interfaced to the system. The language-independent RAIDE system functions are described as needed below, and are listed in Appendix A.

The remainder of this section contains examples in this form: a description of the debugging request desired, the Dispel code corresponding to the request, and an explanation and comments concerning the code.

1. Change the value of the variable 'var' in the procedure 'foo' to the value of 'n'.

set var **in** foo **to** n.

This command is given when execution of the program has been suspended and RAIDE is in the interactive request mode. If the program has been interrupted during execution of 'foo' (i.e., the reference point is 'foo'), then the segment-qualifier "**in** foo" is optional since it will default.

2. List the names and current values (if any) of all variables declared in the currently executing procedure which have not been accessed more than 'n' times.

```
for each VARIABLE in CURRENT_PROCEDURE -> var do
  if (#ACCESSES(var) ≤ n)
    then display LINE, var, " = ", VALUE(var)
  fi.
```

#ACCESSES is a system function yielding the number of times which the indicated data-specific has been accessed during total program execution. LINE is a system function causing the items following to be displayed starting on a new line of the output display device. Notice that displaying the specific variable 'var' causes the source-level name of the variable indicated by the specific to be printed. The current value of some specific is obtained by the VALUE system function. The value of an uninitialized variable is displayed as a question mark.

3. Set a breakpoint when statement 'm' in the currently executing procedure has been executed 'n' times.

```
after STATEMENT:m in CURRENT_PROCEDURE
  if (#ENTRIES(m) = n)
    then begin
      cancel ;
      break "n-th execution of statement m"
    end
  fi.
```

This is an example of a deferred action. It sets a trap after statement 'm' in the currently executing procedure. #ENTRIES is a system function which yields the number of times the indicated segment-specific has been entered during program execution. The prefix "STATEMENT:" is appended to 'm' in the first line since the generic type of 'm' may be ambiguous. It could be a VARIABLE or PROCEDURE, and the system may be unable to determine which 'm' is being referenced.

4. List the source for statements 'm' thru 'n' of the procedure 'foo'.

```
for each STATEMENT in RANGE(foo,m,n) -> stmt do
  display LINE, VALUE(stmt).
```

"STATEMENT:" is not necessary here unless the host language numbers segments other than STATEMENTs (e.g., BLOCKs). RANGE is a system function which yields a generic value identifying some particular subrange of another generic value.

5. List the names of all procedures declared in the procedure 'main' which have not been executed more than 'n' times.

```
for each PROCEDURE in main -> proc do
  if (#ENTRIES(proc) ≤ n)
    then display LINE, #ENTRIES(proc), TAB(10), proc
  fi.
```

TAB is a system function which causes the items following to be displayed at the indicated column on the output display device. Since the specific 'proc' will indicate only those procedures which are declared in the procedure 'main', the request above applies only to the top-level procedures in 'main'.

6. Extend the preceding example to produce a complete procedure execution profile and indent the output to reflect the logical structure of the program.

```
define execution_profile (specific major_proc ;
    integer indent) as
  for each PROCEDURE in major_proc -> proc do
  begin
    display LINE, TAB(indent), #ENTRIES(proc),
      TAB(indent+10), proc ;
    call execution_profile(proc,indent+5)
  end.

call execution_profile(main,0).
```

This is the first example using a debugging procedure. Notice in particular how the procedure is called recursively to handle

the logical nesting of the program. This example demonstrates the utility of debugging variables (viz., 'major_proc', 'indent', and 'proc').

7. Write a debugging procedure to trace all calls of a particular subroutine within a procedure, indicating the location of the call, the name of the subroutine called, and the names and values of all of its formal parameters.

```
define trace_proc_calls (specific subr) as
  begin
    subr_entry_trace:
    before subr ENTRY
      display LINE, "Trace at statement ", CURRENT_STATEMENT,
        " in ", CURRENT_PROCEDURE ;
    subr_entry_trace:
    after subr ENTRY
    begin
      display LINE, CURRENT_PROCEDURE,
        " entered with the following parameters:" ;
      for each PARAMETER in CURRENT_PROCEDURE -> parm do
        display LINE, TAB(10), parm, " = ", VALUE(parm)
    end
  end.
```

This example demonstrates another procedure of sufficient utility to merit inclusion within a debugging procedure library. The definition establishes two deferred actions, one which is initiated in the environment of the calling procedure (**before** subr ENTRY) and one which is initiated in the environment of the called subroutine (**after** subr ENTRY). Notice how the procedure is capable of setting many traps, all of which are identified by only one label.

8. Disable all of the tracing described in the preceding example.

```
cancel subr_entry_trace.
```

This one **cancel** action removes all of the traps set by the procedure 'trace_proc_calls'.

9. Write a debugging procedure to produce a DO-trace of a GOTO-less subroutine [Foul 75].

```
define do_trace (specific subr) as
  for each BLOCK in subr -> block do
    do_block_trace:
    after block ENTRY
      display " ", CURRENT_STATEMENT.
```

In a GOTO-less subroutine, tracing each block rather than each statement produces a more compact trace. No essential information is lost because, once the block is entered, it is guaranteed that each statement in that block will be executed since arbitrary GOTOs are not permitted. 'do_trace' works by simply printing the number of the first statement in each block which is entered.

10. Extend the preceding example to trace all blocks of all subroutines declared within some user procedure.

```
define do_trace_all (specific proc) as
  for each PROCEDURE in proc -> subr do
  begin
    do_proc_trace:
    after subr ENTRY
      display LINE, CURRENT_PROCEDURE, ":" ;
    call do_trace(subr)
  end.
```

Since statement numbering may be done at the procedure level, it is necessary to identify the name of the procedure entered before listing statement numbers within that procedure.

11. Define a mechanism for determining dynamically the nesting depth of an active program block [Math 75].

```
define block_depth_counter (specific proc) as
  for each PROCEDURE in proc -> subr do
  begin
    for each BLOCK in subr -> block do
```

```
    begin
      block_depth_level:
      after block ENTRY
        set block_level to block_level + 1 ;

      block_depth_level:
      before block EXIT
        set block_level to block_level - 1
    end ;
    call block_depth_counter(subr)
  end.

% Define and initialize the global block level counter. %
integer block_level.
set block_level to 0.
```

Invoking 'block_depth_counter' with the name of the user's main routine as an argument will establish breakpoints at the beginning and ending of each block of the user's entire program. The purpose of these breakpoints is to increment and decrement the variable 'block_level'. Thus once 'block_depth_counter' is invoked, the value of 'block_level' during execution indicates the nesting depth of the CURRENT_BLOCK. With this mechanism it is possible to establish various breakpoints based on the dynamic depth of program execution; see the following two examples.

12. Using the mechanism described in the preceding example, produce a DO-trace of a program only when the block nesting level is greater than some constant 'n'.

```
when (block_level > n)
  display CURRENT_BLOCK.
```

The <when-clause> will be evaluated each time the value of 'block_level' is changed, which occurs both at the beginning and ending of each block. Therefore, tracing will occur twice for each block, at entry and exit.

13. Write a debugging procedure to break on the 'n'-th recursive call of some user procedure.

```
define recursion_break (specific proc ; integer n) as
begin
  recursion_break_label:
  after proc ENTRY
    set recursion_level to recursion_level + 1 ;

  recursion_break_label:
  before proc EXIT
    set recursion_level to recursion_level - 1 ;

  recursion_break_label:
  when recursion_level = n
  begin
    cancel recursion_break_label ;
    set recursion_level to 0 ;
    break
  end
end.

% Define and initialize the global recursion counter. %
integer recursion_level.
set recursion_level to 0.
```

The action "**call** recursion_break(foo,10)" causes a break when 'foo' is entered recursively for the tenth time.

14. Write a debugging procedure to produce the Algol-W postmortem dump [Site 71:125-127].

```
define postmortem as
begin
  specific segment, caller ;
  display PAGE, "=> Postmortem dump of active segments" ;
  set segment to CURRENT_BLOCK ;

  while DEFINED(segment) do
  begin
    display LINE(2), "=> Segment name: ", segment, LINE(2),
      " value of local variables:", LINE ;
    for each PARAMETER in segment -> parm do
      call print_parameter_value(parm) ;
    for each VARIABLE in segment -> var do
      call print_variable_value(var) ;
    set caller to CALLER(segment) ;
```

```
    if DEFINED(caller)
      then display LINE(2), segment, " was activated from ",
             caller, ", near coordinate ",
             CURRENT_STATEMENT in caller
    fi ;
    set segment to caller
  end ;

  display LINE(2), "=> End of postmortem dump"
end.
```

Unlike the previous debugging procedures, 'postmortem' is concerned primarily with the dynamic execution structure of the program. Using the **while** action and appropriate system functions, it is possible to trace back thru execution of the program. The system function DEFINED yields a true value if the variable indicated by the argument has a value and yields false otherwise. CALLER is a system function which accepts a segment-specific as an argument and yields a specific indicating the segment which called the argument specific. The procedure above assumes two debugging procedures 'print_parameter_value' and 'print_variable_value' have been defined elsewhere. They handle the special formats in which the Algol-W translator displays variables and parameters, based on their attributes.

Any debugging system must provide a mechanism for interrogating the current state of not only the executing host program, but of the debugging system itself. For example, the user may wish to know the names of all debugging procedures which have been defined, what breakpoints are currently in effect, and what the value of the current reference point is. In RAIDE, these requests are facilitated by introducing system

functions to extract status information and by viewing Dispel simply as another "host" source language. Thus, it is possible to **display** segments of Dispel procedures and to write debugging procedures which affect other debugging procedures. In the examples to follow, it will be assumed that Dispel contains the segment-generics COMMAND and ACTION and the data-generics VARIABLE and PROCEDURE.

15. Write a debugging procedure to print the names of all actions on the deferred action list.

```
define print_deferred_actions as
begin
  integer counter ;
  specific deferred_action ;
  set counter to 1 ;
  set deferred_action to DEFERRED_ACTION_LIST(counter) ;
  display LINE ;

  while DEFINED(deferred_action) do
  begin
    display deferred_action, SPACE(3) ;
    set counter to counter + 1 ;
    set deferred_action to DEFERRED_ACTION_LIST(counter)
  end
end.
```

This example uses the system function DEFERRED_ACTION_LIST to access the labels associated with each of the currently deferred actions. Each label is either the one specified, or the default label assigned to it by the system. Notice that DEFERRED_ACTION_LIST returns a segment-specific just as if it was a user program variable. By changing the principal **display** action of 'print_deferred_actions' to a **cancel**, a debugging procedure to 'cancel_deferred_actions' can be defined.

16. Write a debugging procedure to list the source of some debugging procedure.

```
define print_debug_proc (specific proc) as
  for each ACTION in proc -> action do
    display LINE, VALUE(action).
```

Initiating the action "**call** print_debug_proc(print_debug_proc)" will cause the **for** action which constitutes the body of 'print_debug_proc' to be printed.

D. Discussion

The purpose of this section is to answer two questions: "How well does Dispel fulfill its design criteria?" and "What are its shortcomings and how can it be improved?".

The following shows how each of the design criteria of Table VII is fulfilled:

1. That Dispel is interactive-oriented is supported by its similarity to other interactive command languages. For example, each <utterance> and <action> begins with a keyword and is terminated by the full-stop symbol (.). The full-stop symbol initiates immediate incremental execution of the utterance. Altho the control structures (e.g., **for** and **while**) give Dispel the flavor of a batch-oriented language, these constructs are used primarily in debugging subroutines, which the ordinary user is unlikely to define online.

2. The set of primitive Dispel <action>s is small. Excluding the control structure <action>s, there are only thirteen primitives. Furthermore, there is a one-to-one correspondence

between these <action>s and the primitive RAIDE actions.

3. The basic concepts of RAIDE (cf., Section II.B) are assimilated into Dispel. This is evidenced by the close correspondence between the syntactic entities of the language and the basic system concepts (e.g., specific variables, the construct <specific>, and the RAIDE concept of a specific; the <when-clause> of a command and a RAIDE event; and the **reference** <action> and the RAIDE reference point).

4. The syntax chart of Appendix B attests to the uniformity of Dispel. The complete syntax is formally specifiable in just a few pages. Further, there are no "semantic" qualifications; a <variable> is a variable in any context.

5. The claim that Dispel is simple to use is subjective. Ultimately, this claim must be substantiated by each user individually.

6. The language has been designed to be "English-like" in many respects and, therefore, easy to read and understand. Pronouncing a debugging command should be the same as explaining it. For example, the <utterance>:

```
before each STATEMENT in foo
  display CURRENT_STATEMENT.
```

has the pronunciation:

"Before executing each statement in the subroutine 'foo', display its statement number."

This in itself is an explanation of the <utterance>.

7. The Dispel syntax discourages voluminous output by requiring explicit specifications and assuming defaults which result in the smallest amount of information. For example:

```
for each VARIABLE in main -> var do
  display var, " = ", VALUE(var).
```

displays only the variables which are declared locally to 'main', not all variables accessible to it. To get a display of all user variables requires a recursive debugging procedure, rather than extending the display action to "**display** ALL.".

8. Extendability is afforded in Dispel thru the definition of debugging procedures by the user and thru the definitions of language-dependent entities under the control of the language interfacer. The numerous examples of the preceding section evidence this. Provided the interfacer identifies the proper generics for a particular language and the primitive RAIDE actions are mathematically "sufficient", Dispel's procedures provide adequate extendability.

Altho Dispel is believed to be a reasonably good debugging language, experience with its use has shown where it can be improved. Specifically:

1. It would be convenient to include initialization as part of the declarations of debugging variables. The sequence:

```
integer foobar.
set foobar to 0.
```

is so common as to suggest an extension such as:

integer foobar **initially** 0.

The problem with doing this is maintaining uniformity with debugging procedure parameter declarations. The phrase:

define foo (**integer** bar **initially** 0) **as** ...

is clearly to be avoided.

2. It would be convenient to label a group of deferred actions without repeating the label. The sequence:

<label-id>: **before** ... <action> ;
<label-id>: **after** ... <action>

is common in debugging procedures (e.g., example 7 of the preceding section). Having the same label on several deferred actions makes **cancel**ing them easier. An extension such as:

<label-id>: (**before** ... <action> ;
after ... <action>)

may be called for.

3. Debugging procedures should be allowed to return values (i.e., there should be debugging functions). For example, it is currently impossible to define a procedure similar to the built-in system function CALLER which returns the 'n'-th caller of some subroutine. The straightforward definition would be as follows.

```
define callern (specific proc ; integer level) : specific as
if (level ≤ 1)
  then return CALLER(proc)
  else return callern(CALLER(proc),level-1).
```

4. The current structure of Dispel places too much dependence on global debugging variables. As an example, the procedure

'block_depth_counter' of the preceding section requires the global variable 'block_level'. Altho it is generally transparent to the user (who should need only to know about 'block_depth_counter'), what if the user inadvertently redefines 'block_level' in some other context? Clearly, debugging variable and procedure names need to be protected at times.

5. Having all deferred action labels known globally likewise creates problems. For example, it might be useful to have two invocations of 'trace_proc_calls' (example 7) to trace procedure calls of the user subroutines 'foo' and 'bar'. This can be done, but it is only possible to **cancel** both or neither since 'subr_entry_trace' is a global deferred action label. Clearly, such labels should be associated with the invocation of the debugging variable, not its definition. This suggests the addition of debugging variable labels. Then the tracing routine could be defined as:

define trace_proc_calls (**specific** proc ; **label** trace_label) **as**
...

and invoked as "**call** trace_proc_calls(foo,foo_trace)".

6. Debugging variables can currently be of only two types: **specific** and **integer**. This is expressively limiting; at least **boolean** and **character** need to be added. But rather than adding more declaration types, what may be needed is a typeless debugging variable declaration. For example, a general-purpose procedure to scan a particular user subroutine for the names of all variables with a certain value could be defined as:

```
define variable_scan (declare proc, value) as
  for each VARIABLE in proc -> var do
    if (var = value)
      then display var.
```

Similarly, a debugging procedure which "remembers" all **set** actions could be defined as:

```
define super_set (declare target, source) as
begin set max_set_list_index to max_set_list_index + 1 ;
  set set_targets(max_set_list_index) to target ;
  set set_sources(max_set_list_index) to source ;
  set target to source
end.
```

Then procedures to print out the values of 'set_targets' and 'set_sources' could be defined.

7. When **display**ing a variable, it would be more natural for its value, rather than its name, to be printed by default. This is similar to **set**, which assumes that the right-hand expression yields a value. Thus, if 'foo' is a program variable with the value 10,

```
display foo.
```

would display "10" rather than "foo". The VALUE system function could then be discarded and a system function NAME could be added to handle those cases when the variable's name is desired.

8. The choice of the keyword **input** is unfortunate; **library** would be more accurate and less confusing.

9. There is need for at least one new system function, OFFENDER, which returns a segment- or data-specific value identifying the cause of an exception. For example, the deferred action:

```
on OVERFLOW
  begin
    display "Overflow detected at statement ",
      CURRENT_STATEMENT, " in ", CURRENT_PROCEDURE, LINE,
      "Variable ", OFFENDER,
      " has the value ", VALUE(OFFENDER), LINE ;
    break
  end.
```

pinpoints precisely the location of an overflow exception. OFFENDER is defined only when there is an outstanding exceptional condition.

Chapter IV. The Virtual Debugging Machine

The virtual machine on which translated programs using the RAIDE debugging system run is called SPAM (Specialized Prodebugging Abstract Machine). Unlike the other canned product sharing its name, SPAM's composition is not of concern to the user, due to its imperceptibility. The first section of this chapter describes SPAM's design criteria. This is followed by a description of the major components of the virtual machine, its internal operation, and an outline of its instruction repertoire. The chapter is concluded with a discussion of how SPAM aids the debugging process and how well it meets its design criteria.

A. Design Criteria

/* Regrettably, most programs are required to execute with a maximum of speed and a minimum of caution; it is small wonder, therefore, that we characterize the common result as a "crash".

-- John R. Ehrman, [Ehrm 72:20] */

The primary purpose of SPAM is to provide the user's program with a machine environment that is not hostile to the debugging process [Ehrm 72, Glas 68, Pyle 71, Zelk 71]. Most existing computers have been designed with little consideration for how the machine itself might aid in the debugging of programs; execution speed and efficiency have always been the primary design criteria.

The design of SPAM is influenced by the B6700 [Orga 73], several recent virtual machines designed to aid in code generation [Boul 72, Pask 73, Wort 72], and a previous virtual machine oriented toward debugging [Pull 69]. All of these are high-level machines directed toward Algol-like languages. Likewise, SPAM is oriented toward block-structured, procedural languages of the Algol family. It is stated in previous chapters that the debugging system is not inherently directed toward one particular class of languages. Nevertheless, since SPAM is restricted to Algol-like languages and since RAIDE can be used only to debug programs which run on SPAM, the virtual machine effectively limits the scope of the entire debugging system. This may at first appear to be a severe restriction of system design criteria 1 and 3, which state that the system should be source language independent and usable on multilingual collections of programs. Nonetheless, the class of Algol-like languages is large and diverse. A debugging system which is usable on any and all languages within this class deserves the attribute "language-independent". Until the design and implementation of a truly universal computer (the machine equivalent of the proverbial UNCOL), all software systems will of necessity be less than totally language-independent.

It must be remembered that SPAM is a tool used in the implementation of RAIDE. As such, the existence of SPAM is not vital to the health and well-being of the debugging system. SPAM could, in fact, be replaced by some other machine, such as

the IBM System/360/370. This was not done for two reasons. First, the use of a virtual machine should facilitate portability of the system. To move RAIDE to another real machine, the language translators need not be remodified; only the SPAM interpreter and RAIDE need be bootstraped over to the new machine. Also, since the virtual machine is oriented toward debugging, the total implementation effort should be less since much of the error detection will be done at the machine level. This reduces the implementation burden of both RAIDE, which needs not perform as much error checking, and any translators interfaced to RAIDE, since they need not generate as much run-time error checking code.

Nevertheless, there are two disadvantages to the virtual machine approach. First, since the code generated during a debugging run will likely be different from that normally generated, it is not possible to use RAIDE to track down code generation errors and other language translator bugs. Also, rewriting or modifying the code generation phase of most translators is a nontrivial undertaking. But as a translator writing system resource, RAIDE is oriented more toward future translator implementations than current ones.

B. Machine Architecture

/* spam, Spam, SPAM, SPAM, Lovely Spam, Wonderful Spam!
-- Monty Python's Flying Circus */

The basic architecture of SPAM is illustrated by Figure IV-1. There are eight major components in the machine:

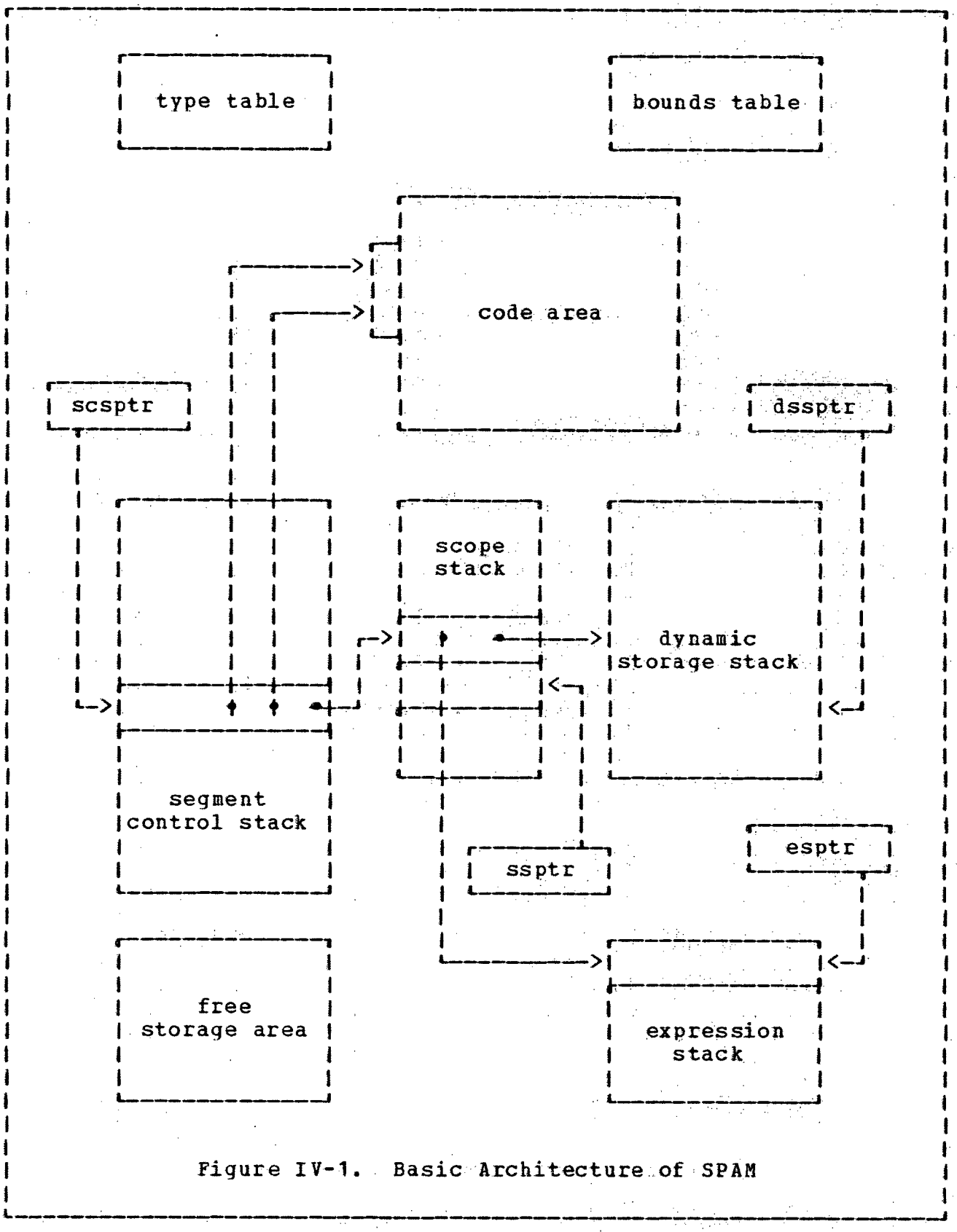


Figure IV-1. Basic Architecture of SPAM

1. The type table contains information concerning the types of variables, the arguments of procedures and functions, and the values yielded by functions. This information is necessary to carry out complete run-time type-checking of variable assignments, and procedure and function invocations. The type table is supplied to SPAM by the language translators.

2. The bounds table contains the lower and upper bounds of all arrays, and the bounds of integer subranges (e.g., 1..10 in Pascal) and scalar variables (e.g., color = (red, white, blue) in Sue). This information is necessary to carry out run-time range checking of variable values. The bounds table is supplied by the language translators.

3. The code area contains only SPAM machine code, which cannot be modified during execution. The basic unit of machine code is a syllable; an instruction consists of one or more syllables. A unit of executable machine code related together by some common purpose is called a segment. SPAM segments are much like the segments of RAIDE (e.g., procedures, loops, and assignment statements). All references to code are made thru the segment control stack and segment descriptors. The code area is supplied by the language translators.

4. The entries of the segment control stack identify a code segment and reference the environment accessible by that segment. Since the complete state of an executing code segment is contained within one segment control stack entry, a new

segment is executed by merely pushing an entry onto this stack. Likewise, segment exit results in an entry being popped from the stack. The currently active segment is pointed to by the segment control stack pointer (scsptr).

5. The entries of the scope stack identify the beginning of the range of variables and values accessible by each code segment. The top entry of this stack is pointed to by the scope stack pointer (ssptr).

6. The dynamic storage stack contains the values of all variables and constants, except for those which are too long to fit conveniently within it. Altho it is called the dynamic storage stack, its first part contains information which is constant during execution of the user program. This static part contains templates for all of the code segments and data items of the program. It must be supplied by the language translators. All variables and constants are accessed by a (level,order) pair. The level indicates the lexic level of the item being referenced and corresponds to the logical nesting of procedures and blocks within the source program. The order indicates which item within the level is desired. The order is an offset into the dynamic storage stack; the level identifies a scope stack entry which refers into the dynamic storage stack. The last valid entry of the dynamic storage stack is pointed to by the dynamic storage stack pointer (dssptr).

7. The expression stack contains the values of all intermediate calculations and is the stack on which most of the arithmetic machine instructions operate. As with the dynamic storage stack, ranges of accessible values within the expression stack are delimited by entries of the scope stack. The last valid entry of this stack is pointed to by the expression stack pointer (esptr).

8. The free storage area contains the values of all strings, all values which will not fit conveniently within the dynamic storage stack, and all values which are allocated outside the normal scope conventions. Garbage collection is necessary only within this area.

The basic unit of information within SPAM is the descriptor. Descriptors contain not only values, but also extensive information to aid in run-time debugging. There are five kinds of descriptors: segment, data, array, segment control stack entry, and scope stack entry. Each descriptor is diagrammed in detail in Appendix C. Segment descriptors delimit any section of machine code which the language translators desire to identify; they model the segment-specifics of RAIDE. Data descriptors describe all program variables and constants. Array descriptors are similar to data descriptors except that they contain an offset into the bounds table. Altho arrays can only be one-dimensional, an element of an array can itself be another array. Thus, multidimensional arrays are possible.

The formats of the entries in the bounds and type tables are diagrammed in detail in Appendix D. The bounds table format is straightforward and should require no explanation. The type table identifies eight basic classes: primitive (viz., void, integer, real, boolean, and string), subrange (e.g., 1..10), scalar (e.g., (apples, oranges)), reference (i.e., pointers), structure, procedure, array, and union (in the Algol 68 sense).

Given the architecture illustrated by Figure IV-1 and the descriptor and table formats contained in Appendixes C and D, it is possible to describe the actions of the virtual machine and its instructions using a microprogramming language called Spamdol (SPAM Descriptive Object Language). Spamdol is modeled after the machine description language of [Wort 72]. The notation below should be clear; the only unusual feature is that Spamdol statement bracketing is accomplished using paragraphing (i.e., program indenting) rather than more syntactically refined devices such as **begin-end** or **do-od** pairs. As an example of Spamdol, the following is the basic instruction execution cycle of SPAM:

```
while (scsptr ≠ null) do
   local instr_length, instr_offset
   with scs[scsptr] do
      if (segtype = procedure) and (lc = external)
         then execute_external_procedure(addr)
              pop_scs
         else if (offset ≥ length)
                 then pop_scs
                 else instr_offset := cao + offset
                      instr_length :=
                         execute_instruction(instr_offset)
                      offset +:= instr_length
```

Notice the declaration of local Spamdol variables (e.g.,

instr_length and instr_offset) and the use of Spamdol procedures (e.g., execute_instruction). As an example of a Spamdol procedure, the following is the routine which pops an entry from the segment control stack:

```
pop_scs:

local old_scse
old_scse := scs[scsptr]
if old_scse.dm and old_scse.interrupt.bx
   then interrupt "before segment exit"
if (scsptr ≠ null) and (old_scse.sso ≠ scs[scsptr-1].sso)
   then pop_ss
decr scsptr
if (scsptr ≠ null) and (old_scse.segtype ≠ procedure)
   then scs[scsptr].offset +:= old_scse.length
if old_scse.dm and old_scse.interrupt.ax
   then interrupt "after segment exit"
```

The instruction set of SPAM is described in Appendix E. All instructions have either zero, one, or two operands. Most instructions reference either the top of the segment control or expression stack. The majority of these instructions are straightforward. Since segment control instructions are the only ones which differ substantially from those of other block-structured, Algol-like machines, an outline of the Spamdol code defining each of these instructions is contained in the appendix. Since the SPAM segment is a generalization of the control structures of previous language-directed machines, segment control instructions have necessarily been generalized. Thus, for example, there are no IF and FOR instructions; these can be constructed from the instructions provided, along with the definitions of suitable segment types. Owing to the author's bias toward structured programming, SPAM does not

contain a GOTO instruction.

Appendix F contains a complete example of a SPAM object program, including both the machine code and the descriptors needed for its execution. By following thru this example carefully and relating it to the information contained in Appendixes C thru E, the astute reader will appreciate the ways in which SPAM's design facilitates the debugging process.

C. Discussion

This section discusses the ways in which SPAM's design aids the debugging process and identifies some of the shortcomings of the design.

The primary way in which SPAM's design aids debugging is thru the use of highly typed segments and data (i.e., a tagged architecture). Each segment of SPAM code has an associated descriptor which indicates its segment type (e.g., procedure or statement) and its parameter types as an offset into the type table. This information is used to type-check segment invocations: agreement in number and type of actual and formal parameters, and the validity of values yielded by segments. Each data value has an associated descriptor which indicates its type as an offset into the type table, whether or not it is a reference value (i.e., pointer), whether or not its current value is defined, and whether or not its value is a constant (i.e., unalterable). This information is used to type-check assignments, to insure proper accessing and longevity of

pointers, to prevent accessing undefined values, and to prevent inadvertently changing constant values. Array descriptors are like data descriptors except they contain an additional bounds table offset field which is used for run-time array bounds checking.

Another feature of the SPAM descriptors which aids debugging is the interrupt information field. Flags are used to implement the segment- and data-incidents of RAIDE (viz., entry, exit, access, and update). The descriptors thus provide a convenient method for placing traps at desired segment locations and data values. SPAM detects an enabled interrupt and signals RAIDE, which finds and executes the appropriate deferred action.

Some of the descriptor information is defined solely to aid in identifying the state of program execution (e.g., determining the name of the current procedure). The identifier field of the segment descriptor, and the level and order fields of the data and array descriptors serve this purpose.

Several of the components of SPAM's architecture are designed specifically to aid in the detection of run-time errors. The type table is used to type-check variable assignments and procedure and function invocations. The bounds table is used for array bounds checking and range checking of enumerated user types. The separation of the code area from the various data areas prevents the ubiquitous "executing data" and "manipulating code" errors which are often difficult to diag-

nose. The segment control, scope, and dynamic storage stacks and their associated pointers aid in interrogating the state of user program execution: the invocation history is indicated by the segment control stack, and all accessible values are indicated by the scope and dynamic storage stacks.

SPAM's design has several inherent shortcomings, and further deficiencies are evident from experiences with using the machine. The known failings and suggestions for overcoming them follow:

1. Altho it is possible to have multiple allocations of a particular array at any one time in SPAM, it is not possible for each instantiation to have different bounds. This is due to the bounds table being fixed in size at translation-time. The bound values themselves can be set at run-time (using the SETBT instruction), but there is no mechanism for dynamically allocating a bounds table entry. Simply defining a new storage management instruction NEWBT which allocates and initializes a new bounds table entry, and which automatically sets the bounds table offset of the associated array descriptor, should alleviate this problem.

2. Similarly, an array component of a structure cannot have flexible bounds due to the constraints of the type table format. As with the bounds table problem, this may be overcome by allowing the type table to grow at run-time. Nevertheless, problems with array bounds suggest possibly that the actual

bound values should be associated with the array itself (e.g., in its descriptor) rather than in a separate bounds table. Altho this would complicate the array descriptor format, the elimination of the bounds table may be adequate compensation.

3. SPAM has no primitive bits or powerset types nor set operators. Their inclusion is necessary to conveniently implement the corresponding features in languages such as Pascal and Algol 68. But rather than adding more primitive types and data classes, it may be more beneficial to provide some means of dynamically extending the type facilities of SPAM. It is not apparent how this might be done, whether it is possible or even whether it is desirable.

4. In some ways SPAM is too strongly typed. Many applications validly require treating something which was once data as code and vice versa (e.g., translate-and-go translators such as Watfiv, and dynamically evaluatable languages such as Lisp and Snobol4). This suggests combining the code and free storage areas and tagging each syllable as either data or code. Then a privileged instruction can be added to SPAM which explicitly resets these tags. It must also be possible then to create segment and data descriptors dynamically, which requires the addition of several other instructions.

5. At times it would be convenient to temporarily ignore or override the type of some value (e.g., while performing binary transput, and for implementing typeless or weakly typed

languages such as BCPL). What may be needed is a privileged instruction (like the PL/I UNSPEC function) to explicitly override type-checking or a special mode of execution during which type-checking is not performed.

Chapter V. The Implementation

/* Let him choose out of my files,
his projects to accomplish.
-- William Shakespeare, Coriolanus */

This chapter will discuss in general terms the UBC implementation of RAIDE, including its scope, the internal design of RAIDE, how languages and translators are interfaced to it, and some reflections on the various tools used in the implementation. In an attempt to keep this thesis down to a manageable size and to promote its longevity, the nitty-gritty details of the implementation have been avoided here. The interested reader is referred to [John 78] for the minutiae.

A. Scope of the Implementation

The preceding chapters describe RAIDE as a complete system to aid in the run-time analysis and debugging of computer programs. This section describes the pragmatic scope of the actual implementation undertaken.

The **explain** and **inquire** actions have not been implemented. Altho they are important parts of any debugging system, they deal with topics not of primary relevance to this thesis. Implementation of the **explain** action is a fairly straightforward, but tedious, exercise in interactive documentation. It is basically the operating system's responsibility to provide this facility. The **inquire** action is in the realm of artificial intelligence and techniques for its implementation are a subject

of continuing study in that sector of computer science.

Of the forms of the <when-clause> of a <command>, the **when** <condition> form has not been implemented due to its ramifications and since it can be simulated using the **before** and **after** forms, altho not as simply from the user's viewpoint. Likewise, the **while** <condition> form of the **execute** action has not been implemented.

Altho the debugging language Dispel constitutes a convenient and clean interface between the user and RAIDE, its implementation is not of paramount importance to this thesis. Thus, user communication with RAIDE has been accomplished by extending the macroprocessor TOSI [Vene 76] thru the definition of macros and TOSI "events". Altho TOSI is a fairly conventional macroprocessing language [Brow 74], the important semantic effects of function invocations are not output strings, but rather the activation of "events". Each of the primitive RAIDE actions is associated with a TOSI event which implements that action. Since it is possible using the TRUST translator writing system [Vene 76] to translate Dispel programs into TOSI strings, the current interface can be considered a "low-level" implementation of the debugging language.

To demonstrate the debugging system, it is necessary to modify or write at least one language translator to produce the required SPAM code and RAIDE specifications. Since writing a translator or even changing the code generation phase of an

existing one is generally a nontrivial task, and since the translator is necessary only to demonstrate RAIDE and is not central to the implementation itself, it is desirable to choose a language which is simple enough that a reasonable translator can be provided, yet which is complete enough to be representative of existing "real" languages. The language chosen which suits these criteria is Asple [Marc 76]. Asple is a block-structured, Algol-like language containing integer, boolean, and reference variables; the control structures: assignment, if-then, if-then-else, input, output, and while; and simple arithmetic expressions. The principal features missing from Asple are procedure declarations and invocations, and parameter passing. Since it was felt these are necessary for a truly representative language, a dialect of Asple containing procedures (called Aspro) was defined.

Providing a translator from Aspro to SPAM might easily have become a substantial project unto itself. Fortunately, a translator for Asple was already available [Appe 78]. It translated Asple source programs into an intermediate graphic representation (called GRAIL) which was then toured by a code generator to produce IBM System/360/370 machine code. Thus, producing the required Aspro translator involved extending the Asple to GRAIL translator into an Aspro to GRAIL translator, and modifying the graph tourer to produce SPAM code and the appropriate RAIDE specifications.

To verify design criterion 1 (viz., source language independence), it is necessary to interface more than one language to RAIDE. The second language chosen was BCPL [Rich 69]. The readily available implementation of BCPL is a locally adapted version of the standard portable translator called BCPL-V [Rich 77]. The original translator produces a parse tree which is toured by semantic routines to output a symbolic assembler program for a mythical machine called MINICODE. Code generation routines then translate this intermediate representation into the machine code of the host computer. BCPL-V was modified to interface with RAIDE by writing semantic routines which generate SPAM code and RAIDE specifications directly from the parse tree. Altho not all constructs of the original language are implemented in the modified translator, most of those which have not been are extensions to the standard language anyway.

B. Internal Design of RAIDE

The basic information structure of RAIDE is illustrated by Figure V-1. It has eight major components:

1. The type name table parallels SPAM's type table, that is, an offset into the type table also indexes the type name table. The value of a type name table entry is an offset into the string index table. Thus, given a type table offset, the name of that type as a character string is obtained by referencing indirectly thru the type name and string index tables.

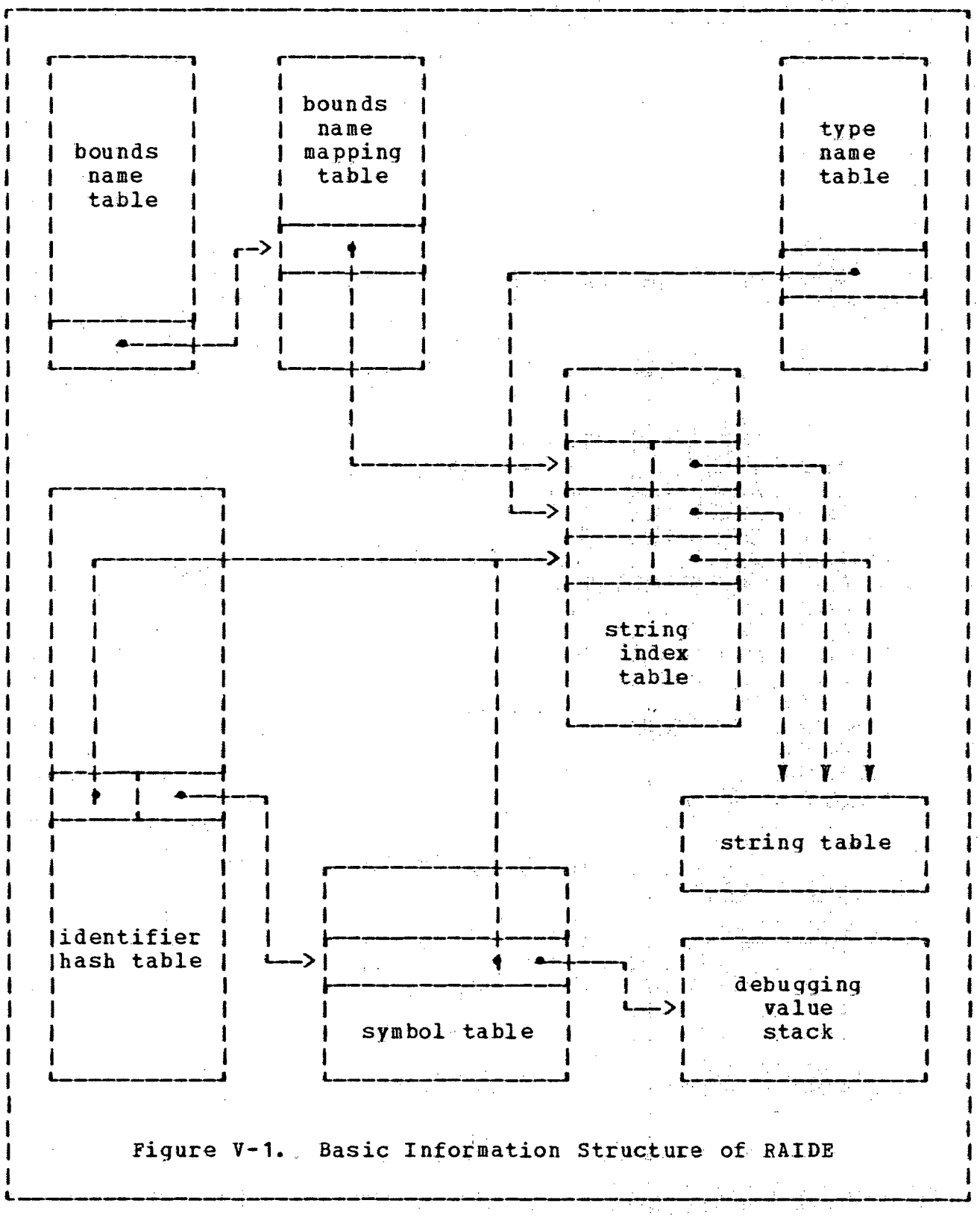


Figure V-1. Basic Information Structure of RAIDE

2. The bounds name table parallels SPAM's bounds table. Unlike the type name table, its values do not reference the names of bounds, but rather the names of the values of bounds. For example, the scalar (red, white, blue) may be stored internally as a subrange of integer type (e.g., 0, 1, and 2). Nevertheless, for printing purposes, the name of some bounds value must be output rather than its internal value (e.g., 'red' rather than 0).

3. Since the mapping of bounds table offsets and bounds names is not one-to-one, the bounds name mapping table is needed to effect the proper mapping. For example, the scalar (red, white, blue) occupies one entry in both the bounds and the bounds name tables. But since three different names must be accessible, the value of the bounds name table entry is an offset into the first of three bounds name mapping table entries which are used like the type name table entries to reference indirectly thru the string index table. If scalar types are not supported by the implementation, both the bounds name and bounds name mapping tables are superfluous.

4. The entries of the string index table identify the strings representing identifiers and the source-level code associated with each program segment. All identifiers (e.g., type names, variable names, and the names of generics) are referenced thru the string index table. An entry contains two fields: the length of the string and an offset into the string area to the start of the character string.

5. The string area is an array of characters containing the strings representing all identifiers as well as the user source program code.

6. The symbol table contains entries for all entities which are accessible by RAIDE, including all user program data- and segment-specifics, generics, events and deferred actions, system functions, and debugging variables and procedures. The formats of the entries of the symbol table are diagrammed in detail in Appendix G. The linkage of symbol table entries is complex, owing to the number of ways in which the table must be accessed. Examples of RAIDE symbol table entries are therefore presented in Appendix H.

7. The debugging value stack contains the values of all debugging variables during execution. An entry of this stack contains two fields: a value-defined flag and the value itself. For an integer variable, the stack contains its actual integral value; for a specific variable, the stack contains an offset into the symbol table for the user program entity associated with the variable.

8. To access symbol table entries by name, it is necessary to hash identifiers thru the identifier hash table. Since one identifier can represent several different entities in RAIDE (e.g., 'foo' may be a user program variable in two different scopes and 'line' may be a user variable as well as a system function), the value of an identifier hash table entry identi-

fies the first of a series of "homonymic" symbol table entries. A homonym chain is then used to determine the appropriate entry.

Appendix I contains a complete example of a RAIDE program specification, including both the data- and segment-specifics of the user source program. The next two sections below discuss how the information supplied by a language interfacer and a translator are integrated into RAIDE's tables.

C. Interfacing a Language

Table III summarizes the information which a language interfacer must supply for each language interfaced to RAIDE. This section will describe in greater detail how the interface is accomplished.

The first requirement of the language interfacer is to identify itself to RAIDE by supplying a pair of the form (lc,id), where 'lc' is the language code of the language named 'id'. 'lc' is used in SPAM descriptors (cf., Figure C-1) and in RAIDE symbol table entries (cf., Figure G-5) to identify the source language in which a particular program segment was written. RAIDE maintains a table of all language name pairs internally.

For a particular host language, the segment- and data-generics are defined as a series of triples of the form (gentype,code,id), where 'gentype' is a flag indicating whether the generic name 'id' is a segment- or data-generic, and 'code'

is the type code associated with the generic. For example, the triples for a block-structured language might be:

```
(segment,1,PROCEDURE)     (data,1,VARIABLE)
(segment,2,BLOCK)         (data,2,CONSTANT)
(segment,3,STATEMENT)     (data,3,PARAMETER)
```

The definition of these triples results in the construction of generic symbol table entries (cf., Figure G-3). 'code' is the segment- or data-generic type code used in SPAM descriptors (e.g., the 'segtype' field of Figure C-1) and in RAIDE symbol table entries (e.g., the 'datatype' field of Figure G-1).

The language-oriented run-time error messages are supplied to RAIDE as an indexed file, where the indexes correspond to error numbers. SPAM and RAIDE dictate the ordering of error numbers; the language interfacer must associate a message with each error number. For example, when SPAM detects an array subscript range error, it reports this to RAIDE using a particular error number. RAIDE uses this number to locate the appropriate error message for the language in which the offending program was written.

The identification of language-dependent exceptions is accomplished as a series of pairs of the form (code,id), where 'code' is the exception number associated with the exception named 'id'. For example, (10,FAILURE) might be defined for some language having statement failure. 'code' is the exception number used in the SPAM SIGNAL instruction (cf., Appendix E). For each language-dependent exception identified by the language interfacer, an event symbol table entry (cf., Figure G-4) is

created.

Generic-related and language-dependent system functions are supplied as a series of pairs of the form (id,routine-number), where 'id' is the function's name and 'routine-number' is a number which associates the function with an internal list of routines. For each function defined by the language interfacer, a system function symbol table entry (cf., Figure G-5) is created.

D. Interfacing a Translator

Tables IV and V summarize the information which a translator must supply for a source program to interface to RAIDE. This section will describe in greater detail how the interface is accomplished.

An interfaced translator's most basic requirement is to generate a SPAM code version of a user source program. As exemplified by Figure F-2, this consists of a sequence of operation and operand codes, where each code occupies one syllable in SPAM's code area. As part of the SPAM code, the descriptors associated with all segment- and data-specifics referenced in the object program must also be supplied. These descriptors constitute the initial dynamic storage stack. The other essential information which a translator must always provide is the initial state of the bounds and type tables. Appendix D diagrams the required formats of these tables. A translator must supply quintuples of the form (tto,d,c,lb,ub) to

define the bounds table and must supply triples of the form (class,arg1,arg2), where 'arg1' and 'arg2' are dependent on the value of 'class', to define the type table. If a translator only supplies SPAM code, SPAM descriptors, and the bounds and type tables, then RAIDE executes at the basic machine level of debugging support.

To execute at the simple symbolic level of support, a translator must supply enough translation-time symbol table information to enable RAIDE to construct a skeletal run-time symbol table for the purposes of reporting errors symbolically and executing simple interactive commands. Specifically, pairs of the form (dsso,id) must be supplied for all user data-specifics (e.g., variables and symbolic constants), where 'dsso' is the SPAM dynamic storage stack offset to the template descriptor associated with the specific named 'id'. These pairs are used to construct data-specific symbol table entries (cf., Figure G-1). Additionally, a translator must supply pairs of the form (tto,type-id), (bto,bnmto), and (bnmto,bounds-id) to facilitate construction of the type name table, the bounds name table, and the bounds name mapping table, respectively. Since at the simple symbolic level of debugging only identifiers are supplied, RAIDE can only report errors symbolically and perform interactive requests which are dependent solely on identifier names (e.g., "**display** x.", but not "**display** x **in** foo.").

If a translator provides sufficient information to allow construction of complete data-specific symbol table entries

(viz., the 'level', 'order', 'datatype', and 'link' fields of Figure G-1), then RAIDE can operate at the symbolic debugging level. An example of a such a description is presented by Figure I-1. It is then possible to respond to requests such as "**set** x **in** foo **to** 10." and "**display each** VARIABLE.". Since no information is yet available concerning the static structure of a user program, requests such as "**for each** STATEMENT **in** foo ...." cannot be honored.

Segment-specifics are represented to RAIDE as quadruples of the form (id,segtype,dsso,link), as diagrammed in Figure G-2. They result in the construction of segment-specific symbol table entries, which are RAIDE's way of storing the description of the static structure of a user source program. This corresponds to the full debugging level of support. An example of such a description is presented by Figure I-2. With such information it is possible to respond to interactive requests involving the static program structure, such as the various trace routines of Section III.C. If a translator additionally supplies RAIDE with the source program code (viz., the 'phrases' field of Figure G-2), then the deluxe debugging level of support is entered.

E. Reflections on the Implementation Tools

The first component of the system to be implemented was a simulator for SPAM. PL/I [IBM 70] was initially chosen as the implementation language, but was abandoned after a few weeks for the following reasons:

1. In PL/I the definition of structured types and the declaration of variables are combined. Thus, to declare two arrays containing the same structures (e.g., the dynamic storage stack and the expression stack), either the structure elements must be written out twice or the LIKE declaration attribute must be used. The former is clearly undesirable since it is error-prone and makes program modification more difficult. The numerous restrictions on the use of LIKE [IBM 70:345-346] make it undesirable as well. Type definitions, in the Pascal sense, can be simulated with a CONTROLLED "variable" declaration, but this technique also suffers from seemingly unnecessary restrictions besides being an example of tricky programming (i.e., say one thing, but mean another).

2. Structure declarations in PL/I cannot contain variant parts, in the Pascal sense. The SPAM descriptor formats make considerable use of this feature (cf., Figure C-2). The DEFINED declaration attribute can be used to simulate variants, but this requires allocating all space to accommodate the largest variation and requires the programmmer to insure proper storage alignment of values. This, too, is both tricky and error-prone.

3. There is no mechanism in PL/I for declaring arrays of heterogeneous element types. This is needed, for example, with the dynamic storage stack, which is composed of segment, data, and array descriptors. As with variant parts, an array of union type can be simulated with the DEFINED declaration attribute. This overlaying resulted in numerous inexplicable errors under

the translator used [Mill 76a], which was the primary motivation for the abandonment of PL/I as the systems implementation language.

4. The implementation of a machine simulator inherently involves the use of case statement constructs. Altho an indexed case statement can often be simulated not too opaquely in PL/I using a LABEL array and a GOTO, the need for an else clause forces the use of strung out if-then-else statements instead. The latter clearly obscures the fundamental structure of the machine simulator.

5. PL/I contains no provision for "named" (i.e., manifest) constants. Their use increases the modifiability of the system tremendously. Named constants can be introduced using a preprocessor or can be simulated thru the declaration of STATIC INITIAL "variables". The locally available PL/I preprocessor is notorious for its expense and especially for its tendency to unprettyprint (uglyprint?) programs. Using constants disguised as variables can introduce needless run-time overhead, besides being tricky programming.

All of the above criticisms of PL/I point to the need for a language which handles types more elegantly. Thus, PL/I was discarded in favor of Pascal [Jens 74]. Both the machine simulator and RAIDE were implemented using the locally available Pascal translator [Poll 77], which accepts a somewhat deviant dialect of standard Pascal. Altho an attempt was made to adhere

to the standard language to facilitate portability, the insidious lure of the bell and the whistle became overpowering.

```
/* Temptation is an irresistible force
   at work on a movable body.
                        -- Henry Louis Mencken */
```

Overall, Pascal proved to be a useful systems implementation language. Nevertheless, it suffers from numerous minor defects and omissions which were a continual source of frustration. The deficiencies of Pascal have been well documented in the literature [Conr 76, Habe 73, Knob 75, Leca 75]. The first reference relates best the problems with using Pascal as a systems implementation language. Most of the problems described in [Conr 76] were verified by the experiences of implementing RAIDE. Some of these shared experiences are presented here:

1. Pascal has been criticized most severely for its lack of dynamic array allocation and the difficulties arising from including array dimensions directly into the array type definition [Conr 76:sec. 3.2]. SPAM's tables and several of its stacks would best be allocated early during execution. Since this simply was not possible, large chunks of memory were tied up needlessly to avoid retranslating all subroutines frequently.

2. Syntactically, a Pascal constant cannot be a translation-time expression [Conr 76:sec. 3.3.1]. In several instances it was found that one constant's value was a function of one or more others (e.g., length = max_index - min_index + 1). Either a variable had to be used or the constant defined with a literal

value along with a comment explaining the origin of the value. Neither solution is satisfactory.

3. Pascal lacks authentic block-structure [Conr 76:sec. 4.3]. All variables must be declared at the procedure level, not at the level to which they are most appropriately local. Altho a minor annoyance, Sue's [Clar 74] separation of identification and storage allocation is more desirable.

4. The addition of Algol 60 own (or PL/I STATIC) variables would, in some instances, reduce parameter passing and better facilitate information hiding [Conr 76:sec. 3.7.1]. Again, a separation of identification and storage lifetime is desirable.

5. Procedure variables [Conr 76:sec. 3.7.1] were needed in the implementation of RAIDE since, for example, generic-related functions (e.g., CURRENT_PROCEDURE) need not be specified until run-time. This had to be simulated using operating-system-dependent routines, thus reducing further the portability of the implementation.

6. Programs would be more concise and clearer if values could be assigned to the fields of a record variable in one statement, as in Algol-W and Algol 68 [Conr 76:sec. 4.2].

The next few criticisms are aimed specifically at standard Pascal since PASCAL/UBC has been extended to deal with them.

7. In Pascal a function cannot yield a value of record type or an array [Conr 76:sec. 2.1]. This seemingly nonuniform

restriction increases the use of var parameters.

8. The case statement of standard Pascal does not contain an else clause [Conr 76:sec. 2.4]. This often leads to strung out if-then-else statements, with a resulting loss of clarity. An else clause in the case statement is essential for software which purports to be fail-safe, as is expected of a debugging system.

9. Nesting of if-then statements could be reduced if boolean expressions are evaluated in the McCarthy manner [Conr 76:sec. 2.3].

10. Provision should be made for initializing all values statically, including arrays [Conr 76:sec. 4.2]. Applications using decision tables, which are often the clearest means of structuring an algorithm, require "constant" arrays.

11. Separate compilation of procedures is essential for the implementation of a large software system such as RAIDE [Conr 76:sec. 3.7.1]. The expense of retranslation becomes intolerable otherwise. Even Fortran is more sophisticated than Pascal in this regard.

The following criticisms are presented in more depth since they point to problems which have not been dealt with adequately in the literature.

12. Assuming that separate procedure compilation is available, a related extension which is needed is global variables (i.e.,

EXTERNAL in the PL/I sense). Altho the extensive use of global variables is currently being contested among language designers [Wulf 73], their restrained use does reduce the need for parameter passing and can aid information hiding. Altho PASCAL/UBC has been extended to provide global variables, their implementation is primitive. To make even a minor change in the definition of one global variable may require retranslation of all procedures, even those which contain no references to the modified variable.

13. String handling is very poor in Pascal, primarily since strings are treated as arrays of characters. There are no variable length strings, no substring pseudovariable to assign characters to a segment of a string, and no incore transput. Altho the RAIDE implementation is not particularly string-oriented, these features are especially useful during debugging. Even a debugging system has to undergo debugging.

14. Pascal's transput facilities are austere and difficult to extend. The provision that the first record of a file must be read when the file is opened may be acceptable in a batch-oriented environment, but it makes interactive processing unnaturally difficult. The handling of nontextfiles is likewise too restrictive. It is sometimes convenient to place records of various types together in one file or to easily read and write portions of arrays (e.g., to implement the **save** and **restore** actions of RAIDE). The syntax of the Pascal file declaration makes doing so painfully arduous.

15. Pascal should provide lower and upper bound functions for all datatypes. Given the scalar type "color = (blue, red, yellow)", it is the programmer's responsibility to remember that the upper bound of 'color' is 'yellow'. The 'pred' and 'succ' functions cannot even be applied effectively to scalar types without knowing this. If constants are allowed to be translation-time expressions, then bound functions will also be essential for subrange types, such as the bounds of an array, to reduce the number of constant definitions necessary.

16. In the PASCAL/UBC implementation, identifiers are only significant to the first ten characters. Worse still, the language standard [Jens 74] suggests for portability that identifiers be unique within their first eight characters. This is far too restrictive for systems programming applications since it encourages the use of cryptic abbreviations and can lead to difficult-to-detect errors if two identifiers are not unique within their first ten characters. A limit of ten characters is not a significant improvement over six, as in Fortran.

The length of the above comments concerning Pascal may lead the reader to believe that its choice as the systems implementation language was poor. The criticisms are meant to be complete, not scathing. Altho Pascal was not designed for systems implementation, it has proved to be a useful tool.

Even tho both the SPAM simulator and RAIDE were implemented in Pascal, Dispel was provisionally implemented by embedding Dispel-like commands in the macroprocessor TOSI [Vene 76]. For example, the Dispel utterance "**display** x **in** foo." has the TOSI transliteration "(DISPLAY,(IN,(ID,x),(ID,foo)))". TOSI macros were written for each of the primitive actions of Dispel as well as for most of Dispel's nonterminal symbols. Altho not elegant from the user's point of view, this method of interfacing with RAIDE proved expeditious. Since the macroprocessor interprets strings directly, there is no need to translate utterances into an intermediate form. Thus, reinvocation of a debugging routine merely involves reinterpreting its string definition. Further, TOSI automatically handles many of the laborious details, such as providing the primitive arithmetic operators and reporting undeclared debugging variables.

The primary difficulty with using TOSI was the incompleteness of its documentation [Vene 76]. Several primitive functions were not described, undocumented implementation limitations were unearthed painfully, and there were some inaccuracies in the documentation. Furthermore, due to limited use and a low level of support, the implementation was somewhat shaky. Executing TOSI also proved expensive, due to its implementation using the TRUST translator writing system.

The Aspro translator was also implemented using TRUST, together with an extension of TRUST called LANCE [Appe 78]. Experiences using TRUST are described elsewhere [Abra 77]. The

most significant criticism of TRUST is that it is poorly documented. As with TOSI, implementation limitations were not documented, resulting in many hours of frustrating debugging.

The best aspect of the Aspro translator is the code generator routine. It is an Algol-W program which tours an intermediate graphic representation of the user source program. Because of the convenience of the representation and since the tree tourer is a well-written, readable program, modifying the Aspro translator to generate SPAM code was moderately easy, requiring only about two weeks of effort.

The last phase of the implementation was modifying the BCPL-V translator to interface with RAIDE. The translator, which itself is written in BCPL, is well structured in terms of modular composition. Unfortunately, its coding is poor and implementation documentation is nearly nonexistent.

Despite the bad example presented by the translator itself, BCPL proved a good systems implementation language. Altho syntactically peculiar, the control structures are natural and convenient for systems applications. The typelessness of BCPL was not a negative feature (as had been expected), but a record type definition facility was missed. The moral is that the syntax of a language is not as important a consideration in its use for systems software as is the way in which the language is used. In other words, human factors are more significant than the choice of language.

Chapter VI. Summary and Conclusions

The research covered by this thesis involves many facets. The following are the major accomplishments of this research:

1. The current state of interactive, run-time program debugging was extensively surveyed.

2. A new debugging tool was conceived in response to perceived deficiencies with existing ones. The system, RAIDE, was based on the abstraction of debugging concepts. Except for some minor features, RAIDE was completely implemented.

3. A debugging command language, called Dispel, for interacting with RAIDE was designed and evaluated. Numerous examples of its use to express debugging requests were presented. Due to time constraints however, Dispel was not implemented as specified; a low-level, transliteral implementation of Dispel using an interactive macroprocessor was substituted.

4. A virtual machine to facilitate run-time debugging was designed and evaluated. A simulator for it was implemented.

5. Two language translators, for Aspro and BCPL, were modified to interface with RAIDE to demonstrate its language-independence.

6. Reflections on the tools used in the various phases of the implementation were presented.

This chapter continues with an exposition of the importance of RAIDE, a discussion of its shortcomings, some thoughts on possible future extensions to RAIDE and other areas in debugging worthy of exploration, and some suggestions based on the experiences of the implementation for how translators should be designed to facilitate run-time debugging.

A. Importance

The debugging system RAIDE and its language Dispel are original and significant contributions to the state of the art of program debugging for the following reasons:

1. The kernel of RAIDE contains a small set of primitives sufficient to implement all the traditional debugging actions. Unlike previous debugging systems which have been designed rather haphazardly, RAIDE's design is based on the concept of minimum sufficiency.

2. The traditional debugging primitives (e.g., traces, dumps, and traps), have been generalized in RAIDE. An example of this generalization is the lack of a primitive trace action. All traditional debugging aids are available to the user thru debugging procedures. This generalization of debugging concepts should allow for the easy inclusion of future debugging aids, and should allow the system to be extended to deal with languages different from the algebraic procedural languages on which most previous systems have been based. RAIDE is a fresh approach to interactive, high-level symbolic debugging.

3. RAIDE is one of the few debugging systems to have language independence as a primary design criterion. Virtually all previous system have been language-dependent by design or implementation.

4. The RAIDE debugging language is more extensive and uniform than that of any previous debugging system. Dispel represents a compromise between an interactive command language and a special-purpose programming language. This compromise results from system design criterion 4 (viz., that the system be usable both in batch and interactively).

5. RAIDE is the first interactive debugging system designed specifically as a translator writing system resource. Such off-the-shelf design should reduce the implementation burden of future language implementors.

6. RAIDE potentially provides more run-time and analysis debugging information than most previous systems. It has been designed to filter, not mask, this information so that the user can obtain maximum benefit from the debugging environment.

7. RAIDE is one of the few systems which can be used to debug multilingual collections of programs. Several preceding systems have provided an interface to machine-language subroutines; RAIDE enables subroutines to be written in any high-level language for which an interface has been defined.

8. Language translators which are interfaced to RAIDE can supply information in varying levels of detail. Previous debugging systems have generally required complete cooperation from translators.

B. Shortcomings

/* When the learned man errs, he errs in a learned way.
-- Arabic proverb */

Nothing is perfect, and since RAIDE is something, it must be imperfect. This section will describe limitations in the design and implementation of RAIDE, and shortcomings which have become apparent during its development.

1. Whereas language-independence is an advance over previous debugging systems, this approach does have several disadvantages. Foremost among these is that run-time changes to correct the source program are difficult in a noninterpretive environment. Either the translators must be very cooperating (i.e., available on demand at run-time), or temporary patches to the translated program could be made at run-time using some "universal" programming language (for which RAIDE has access to a translator). Nevertheless, the value of nonsource-language changes is dubious since the user is unlikely to remember to modify the source program to reflect the run-time changes.

2. Another disadvantage of the language-independent approach is that using one or more of the host source languages to specify debugging actions may foster confusion. It is thus

desirable to provide a separate debugging language which the user must learn. Besides the initial overhead in learning a debugging language, many users appear to inherently prefer debugging using the notation of the host source language. Nevertheless, no existing debugging system uses exactly the source language for debugging commands at run-time. Invariably an extended subset is used. Since there is already deviation, some users' desire for "compatibility" must be seen as a subjective bias.

3. A third disadvantage, which is inherent in any language-independent system, is that it may not always be possible to cater to the peculiarities of particular source languages, either existing or future. It appears impossible to answer the criticism "How do you know that language X can <u>really</u> be interfaced to RAIDE?" without actually attempting to define the interface.

4. The implementation has effectively excluded interpreted languages from the class of RAIDE debuggable languages. Using a virtual machine thus limits the scope of the entire implementation.

5. The decision to implement RAIDE using a virtual machine has limited slightly the class of bugs which the system can detect. Specifically, translator and loader errors cannot be detected with RAIDE. It is not clear, however, that existing systems based on "real" machines address this area adequately, or even

that such errors are significant in comparison to user errors. The virtual machine implementation of RAIDE may also prevent its use in detecting machine-dependent bugs, such as character collating sequence dependencies, and in detecting implementation dependencies, that is, the gray areas of a language's design which are left formally undefined.

6. Interfacing an existing translator to RAIDE is nontrivial. In fact, if the structure of the translator is poor, interfacing it may be impossible or even more difficult than providing a simple language-dependent run-time debugging package instead. The translators chosen for this thesis were selected primarily for their modifiability. Such modifications are considered necessary, but not fundamental, for demonstrating the feasibility of the system. Nevertheless, RAIDE is not aimed at existing translators. It has been designed primarily as a tool for use with implementing future translators. As such, it is much like any other translator writing system resource. When a new parser generator is implemented, it is not criticized for being difficult to incorporate into existing translators.

7. There is no obvious way of dealing with the question "Can RAIDE do x?" without actually producing Dispel utterances to accomplish the desired goal. In other words, how can it be "proved" in some rigorous way that RAIDE's primitive actions are sufficient to carry out all conceivable debugging requests? Many of the traditional debugging requests have been shown doable by construction thru the examples of Section III.C.

Thus, the question becomes one of proving that the primitives are sufficiently general and extendable to implement all "computable" debugging requests. Since no existing debugging system has been shown to fulfill this theorem, in some sense RAIDE is at least as well designed as other debugging systems.

8. The most serious shortcoming of the work reported in this thesis is that it is not user tested. The implementation has been primarily undertaken to demonstrate the feasibility of the system design concepts. This leaves open questions concerning how pleasant the system is to use, which can be answered only by field testing and obtaining the views of actual users. Insufficient feedback has also left open the question as to what is the implementation overhead in using the system [Lest 71a, Lest 71b]. Also, is it possible to evaluate the system apart from user satisfaction and aesthethic considerations?

C. Future Directions

/* Excelsior!

-- motto of New York state */

The purpose of this section is two-fold: to describe possible extensions to RAIDE and its implementation, and to suggest other directions in high-level debugging which might be fruitful areas for exploration.

There are numerous ways in which RAIDE can be extended:

1. RAIDE can be extended to handle parallel processing [Hadj 76, Vict 77]. The only limiting factor at present is

SPAM. By introducing a process control stack and a process descriptor, and by spaghetti-stacking the segment control, scope, dynamic storage, and expression stacks, SPAM can be extended to support parallel processing. It should then be straightforward to add a process segment-generic and a generic-related system function CURRENT_PROCESS to RAIDE.

2. It would be interesting to produce a microprogrammed implementation of SPAM. This might appease the criticism that run-time debugging is too expensive. Much of the current system overhead is in the simulation of SPAM, not within RAIDE proper.

3. Producing an implementation of RAIDE which is operating-system- and machine-independent [Vict 77], but retaining RAIDE's high-level and symbolic nature, would be worthwhile. Altho the design of RAIDE conforms with this goal, the current implementation is both operating-system- and machine-dependent.

4. It may be possible and desirable to extend RAIDE to include more program testing aids [Itoh 73, Panz 76], such as test procedures. A test procedure is a routine whose sole intent is to verify the correctness of another routine by exercising all its control paths and testing it out on various input data values. Since testing and debugging are closely related activities, it is reasonable to expect that one software tool might be used to aid both.

5. RAIDE is a debugging aid, not a diagnostic system. It makes no attempt to analyze the cause of an error. Some work

has been done on automating the process of tracing an error to its cause [Davi 75]. It may be possible to add such a capability to RAIDE.

6. Adding an **undo** command to RAIDE, which reverses the effect of the previously entered command, may be a useful addition. The major problem with its implementation is determining exactly how much and what information must be saved to enable an **undo** to be accomplished.

7. Transput is an area in which RAIDE could successfully be expanded. The **display** action is deficient. There is a need for reading and writing, not only user values, but information concerning the state of the debugging system. With such facilities, it may be possible to implement **save** and **restore** as debugging procedures and thus to remove them from the list of primitive system actions. It should also be possible then to implement an **unexecute** action to reverse execution of the user program. Such a capability has been shown to be valuable [Zelk 71, Zelk 73]. Beefing up the transput will also make it possible to keep program execution statistics between runs, thus enabling cumulative statistics to be maintained.

Besides various extensions to the existing debugging system, this work and related observations have pointed out other areas in the realm of debugging which might be pursued:

1. There is a definite need for empirical studies to validate the claim that debugging systems actually do aid significantly

in the program development process. Some work has been done in this area [Bier 75, Goul 75, Gran 66], but much more is needed.

2. It is apparent that debugging techniques are not being taught adequately to novice programmers. More research is necessary on effective ways of conveying debugging methods [Math 74]. Debugging is largely an intuitive process at present. It needs to be systematized more adequately.

3. The RAIDE approach to debugging is to provide one tool which is useful to users of various source languages. Another approach is to design a debugger generator system which, given a language definition, produces a language-dependent debugger for programs written in that language. This is analogous to the table-driven parser versus parser generator approaches in translator writing systems research.

D. Suggestions for Translator Design

In light of the experiences gained in the implementation of RAIDE, this section will discuss the various ways in which translator implementors can facilitate run-time debugging. Whether the implementor intends to use RAIDE for debugging or to provide a language-dependent debugging package, or if the implementor wishes only to provide adequate hooks for someone else to add such a facility, the following points should be kept in mind:

1. do not throw away translation-time information too readily;

2. structure the translator such that information is easy to obtain;
3. code the translator so that it is easy to read and to modify;
4. if possible, choose to generate code for a hospitable object machine;
5. treat run-time error conditions in a uniform manner;
6. write run-time library routines with debugging and error handling in mind;
7. generate object code which is easy to understand and to modify; and
8. document!

Some of these points have been treated previously to some extent in [Ashb 73]. They will all be discussed in detail below.

There is a tendency among translator implementors to discard translation-time information as soon as there is no further apparent need for its retention. For example, source program code is generally eliminated once it has been syntactically analyzed and, for block-structured languages, variable symbol table information is often discarded once the scope block of the variable has been translated. Undoubtedly, information abandonment has been motivated by valid implementation constraints: often there has been insufficient memory for total retention. Nevertheless, the premature discarding of translation-time information is a major difficulty with interfacing a translator to a run-time debugging system. Further, efficiency and

resource constraint arguments are becoming more difficult to sustain in view of the "software crisis" and continuing advances in hardware technology. Even on minicomputer systems, limited memory is rapidly being relegated to the history books. Also, translator execution speed is becoming of less concern in view of the overall efficiency of the entire software development cycle.

There are basically three classes of information which a translator should provide to a debugging system: descriptions of each symbolic data item, descriptions of each important segment of object code, and the source code broken into lexic tokens. For each symbolic data item of the user source program, the translator should provide its:

a. identifier name;

b. data type (e.g., variable or constant);

c. lexic (level,order) pair, or other address specification;

d. scope with respect to procedures and blocks;

e. type, including range of values for scalar and enumerated types;

f. dimension and subscript bounds information, where known, for an array; and

g. initial or constant value, if known.

For each important code segment of the user source program, the translator should provide its:

a. label name, if present;

b. segment type (e.g., statement or procedure);

c. scope (i.e., its relationship to other code segments);

d. parameter and result types, if any; and

e. bounds within the object code area or file.

It is convenient, altho not essential, for the translator to supply the debugging system with the user source program code. It should be decomposed into lexic tokens and associated with the corresponding object code segment descriptions. For RAIDE, the information listed above is provided as part of the SPAM descriptors and the RAIDE program specifications.

When designing the internal structure of a language translator, it is important to bear in mind how difficult it will be for the implementor, or someone else, to provide the translation-time information which a debugging system will need. Three areas of design are of special importance: the structure of the symbol table, translator modularization, and the intermediate user program representation.

The translator symbol table should contain all symbol information in a centralized, easy to access, and understandable format. Ideally, it should be possible to write one module which, given the symbol table as an argument, generates a file describing all program data items in a format acceptable to the run-time debugging system. The decentralization of symbol table information was a problem encountered in interfacing the BCPL-V translator to RAIDE. Separate BCPL vectors are used to store

different symbol attributes. Undoubtedly it was done this way due to the absence of record types in BCPL, the language in which the translator itself is written.

The decomposition of the translator into modules is an important consideration. The traditional translation phases (viz., lexic scanning, syntactic analysis, semantic processing, and code generation) should be represented by separate translator routines. Such separation makes understanding the translator easier and, hence, makes modifying it to interface to a debugging system easier. Both the BCPL and Asple translators benefit greatly from this organization. In fact, several other candidate translators were rejected because of their poor internal organization. The locally available Pascal translator, for instance, is a one-pass translator whose spaghetti-like intermingling of scanning, analysis, semantic processing, and code generation make it virtually unintelligible to all but hearty souls.

Constructing an intermediate user program representation during translation greatly facilitates interfacing to a debugging system. Both the BCPL and Asple translators transform programs into graphic intermediate representations on which both semantic routines and code generators operate. Ideally, it should be possible to write one module which, given the intermediate representation as an argument, generates a file describing all important program code segments in a format acceptable to the run-time debugging system. A nongraphic intermediate

representation is less desirable since it obscures the static structure of the source program.

Altho it should go without saying, a translator should be coded in a readable and, therefore, easy-to-modify manner. In particular, variable names should be mnemonic, the code should be logically indented, and adequate internal documentation should exist. It is lamentable indeed that there are so many poorly coded translators in existence still. It is indefensible that the translators for a systems implementation language (viz., BCPL) and a student-oriented, "structured" language (viz., Pascal) should be poorly coded. Regrettably, such was found to be the case with the BCPL and Pascal translators locally available. In contrast, the code generator for the Asple translator was well coded, which explains why it was possible to completely change the object machine code generated in only two weeks!

If at all possible, the translator implementor should generate object code for a machine which is hospitable to run-time debugging. SPAM was introduced into the implementation of RAIDE for just this reason (cf., Section IV.C). If the object machine is inhospitable, the implementor should not hesitate to simulate on the host machine a pseudomachine more concordant with the goals of debugging, in particular the detection of run-time errors. Arguments that interpreting virtual machine code is inefficient are rapidly losing ground with the advent of new hardware technologies such as micropro-

gramming. A translator implementor's best efforts at providing run-time debugging aids can be thwarted by an object machine which is antagonistic toward debugging.

To facilitate run-time debugging, the implementor should treat all run-time errors detected in a uniform manner, for example, thru a common error handler. Additionally, a reasonable recovery should be instituted for each error condition so that execution can proceed logically once the error has been corrected or patched by the user at run-time. The presence of terminal (i.e., nonrecoverable) errors should be avoided since they frustrate run-time debugging. The implementor should also insure that an executing program is never left in an unstable state due to a run-time error (e.g., partially computed values should not be left in registers). It can be extremely difficult for a debugging system to deal with an unstable machine state. Ideally, error breaks should occur only at points corresponding to breaks between statements in the source program code. It is preferable that the machine state be such that the offending statement can be reexecuted at run-time following error correction or recovery.

Similarly, any run-time library routines which the implementor provides should interface cleanly with the debugging system. Specifically, library routines should detect and treat run-time errors in the same manner as error checks compiled directly into the object program. It is also convenient if the interface between the object code and library routines is the

same as that between the object code and user subroutines. This avoids special cases which only complicate the debugging system.

How the code generators go about producing object code also affects the ease of interfacing with a debugging system. It is preferable that all data values be accessed uniformly to avoid having to handle numerous special cases at run-time. For example, the BCPL global vector and Fortran COMMON variables may present difficulties if code for them is generated differently from other program values such as local variables and constants. The code generators should also avoid optimizing the object program such that there is no longer an obvious correspondence between the source and object versions of the program. If too much rearrangement occurs, it is difficult for the user to understand the behavior of the executing program. The goals of run-time debugging and optimization conflict. To facilitate debugging, it is often necessary to supply more information at run-time than is normally provided, whereas the goal of optimization is to provide less information than usual.

Last, but certainly not least, a translator implementor should not underestimate the value of well-written and accurate implementation documentation. With both the BCPL and Asple translators, documentation was nonexistent at best and inaccurate at worst! This sorry state greatly hindered interfacing the translators to RAIDE. Any programmer who implements software and fails to adequately document it should be shot (i.e., fired in an industrial environment, or failed in an academic

environment). No one should assume that a program will never be modified, or at least looked at, by someone else.

It is hoped that this advice on how translators should be constructed will not only benefit run-time debugging, but also benefit the overall quality of user-oriented software.

Glossary

For the convenience of the reader, definitions are provided below for the debugging terminology used in this thesis. Altho an attempt has been made to conform to the standard information processing definitions, inevitably some variations in usage have arisen.

ANALYSIS INFORMATION: information used to examine the state of program execution, for example, variable or execution profiles.

BREAKPOINT: a particular location in a program or a particular time during execution at which some debugging action is to be initiated.

BUG: any error which is associated directly or indirectly with a computer, a computer program, information entering or leaving a computer, a computer operator or programmer, or anyone related to or anything associated with any person engaged in an occupation which uses computers.

DEBUGGING: the process of recognizing, isolating, and correcting mistakes in some computer program. In this thesis the term is restricted to the detection of logic and semantic errors. The detection of syntactic errors is <u>not</u> referred to as debugging.

DEBUGGING SYSTEM: a collection of software tools which aid in the debugging process.

DUMP: a display of some aspect of the state of a program. The two classes of dumps are: memory (or "core") dumps and variable dumps. The latter are of primary concern in high-level debugging systems and are of two types: snapshot and postmortem.

ENTOMOLOGY: the study of bugs by observation [VanT 74:154].

EXECUTION PROFILE: a profile of the frequency of execution of statements or procedures within a program.

FLOW TRACE: a trace of the program labels or source-level statement numbers encountered during execution of a program.

LOGIC ERROR: a deficient implementation of an algorithm.

POSTMORTEM DUMP: a dump of a program at its point of abnormal termination.

PROFILE: a postexecution display of the activity of some aspect of a program, primarily beneficial for testing and optimization purposes. The two classes of profiles are: variable profiles and execution (or flow) profiles. Profiles can either be complete for the entire program or selective within a portion of the program.

SEMANTIC ERROR: a misunderstanding in the use of some construct in the source language.

SNAPSHOT DUMP: a dump of an executing program at a specified point or time.

SUBROUTINE TRACE: a trace of the subroutines invoked (possibly including their parameter and result values) during execution of a program.

SYMBOLIC DEBUGGING: the debugging of a program in terms of the source-level names and constructs of that program.

TESTING: the process of verifying that a computer program behaves according to its specifications.

TRACE: a display of the execution path of a program. The three classes of traces are: flow traces, variable traces, and subroutine traces. Traces can either be complete for the entire program or selective within a portion of the program.

TRACEBACK: a trace of the procedure invocation state of a program at a particular point in time.

VARIABLE PROFILE: a profile of the number of accesses of or changes to the variables in a program.

VARIABLE TRACE: a trace of the names and values of each variable accessed or changed during execution of a program.

References and Annotated Bibliography

[Abra 77] Abramson, Harvey David; Appelbe, William Frederick; and Johnson, Mark Scott. "Assaulting the Tower of Babel: Experiences with a Translator Writing System". Technical Report 77-12, Department of Computer Science, University of British Columbia, 1977 November. 11pp.

[Appe 78] Appelbe, William Frederick. A Semantic Representation for Translation of High-Level Algorithmic Languages. Doctoral Dissertation: Department of Computer Science, University of British Columbia, 1978 August. 144pp.

[Ashb 73] Ashby, Gordon; Salmonson, Loren; and Heilman, Robert. Design of an interactive debugger for FORTRAN: MANTIS. Software -- Practice and Experience, 3:1 (1973 January-March), 65-74.
Describes an interactive debugging system interfaced to the Fortran language, in which no modifications are needed to the source program at translation-time; all debugging code is added by the debugging system at run-time. Nevertheless, the language translator was modified to provide extensive run-time information. The authors give some guidelines to language implementors on what information is needed by a debugging system.

[Bair 75] Baird, George N. Program debugging using COBOL '74. AFIPS Conference Proceedings, 44 (NCC 1975), 313-318.
Describes the language-provided debugging facilities incorporated into the 1974 ANSI Cobol Standard. These include the capability to suppress the effects of added debugging statements at either translation- or run-time, special debugging SECTIONs which act as interrupt procedures, and a global DEBUG-ITEM which contains information concerning the state of program execution.

[Ball 77] Ballard, Alan John. "UBC DEBUG: The Symbolic Debugging System". Computing Centre, University of British Columbia, 1977 September. 126pp.
Describes the interactive Symbolic Debugging System (SDS) available under the MTS operating system to aid in the debugging of IBM System/360/370 machine-language programs. It has been extended to handle Fortran, PL/I, and PL360 source programs in a somewhat symbolic manner. The usual machine-level facilities, as well as a rudimentary programmable breakpoint mechanism, are provided.

[Balz 69] Balzer, Robert M. EXDAMS -- EXtendable Debugging and Monitoring System. AFIPS Conference Proceedings, 34 (SJCC 1969), 567-580.
Describes the EXDAMS online debugging system for PL/I programs. No modifications are made to the language translator; instead, a preprocessor inserts into the source code calls to a run-time monitoring package. The run-time package keeps a history file which enables reverse program execution. The package also provides run-time display of the source code, flow analysis, and motion-picture display of variables. All these facilities, however, are available only after the program has executed and the history file has been constructed; the system does not support interactive debugging.

[Balz 74] Balzer, Robert M. A language-independent programmer's interface. AFIPS Conference Proceedings, 43 (NCC 1974), 365-370.
Describes a programmer's environment which can be interfaced to any interpretive, interactive language which can perform single statement evaluation (e.g., the EVAL function of Lisp). The system provides an editor with prettyprinting and undoing capabilities, the ability to edit previous commands, and programmable breakpoints. The system is written in INTERLISP, using the protocol of the ARPANET to initially interface with the ECL programming system.

[Baue 73] Bauer, Friedrich Ludwig (editor). Advanced Course on Software Engineering. Springer-Verlag, 1973. 545pp. [ISBN 3-540-06185-1]

[Baye 67] Bayer, Rudolf; Gries, David; Paul, M.; and Wiehle, H.R. The ALCOR Illinois 7090/7094 post mortem dump. Communications of the ACM, 10:12 (1967 December), 804-808.
Describes an Algol 60 postmortem dump facility which incurs almost no run-time overhead unless a program failure occurs. The language translator, however, has been changed substantially to produce information accessible to the run-time dump system.

[Bern 68] Bernstein, William A.; and Owens, James T. Debugging in a time-sharing environment. AFIPS Conference Proceedings, 33:1 (FJCC 1968), 7-14.
Discusses the need for a "debugging support system" independent of an operating system for use by systems programmers in debugging an executing time-sharing operating system. The authors also discuss the possibility of a debugging environment which is heuristic and, thereby, self-debugging.

[Bier 75] Bierman, A.W.; Baum, R.I.; and Silverman, M. Trace information as an aid to debugging. SIGCSE Bulletin, 7:3 (1975 September), 44-49.
Reports the results of an experiment supporting the claim that interpretive execution of machine code which enables tracing information at abnormal termination is a better tool for debugging than normal execution without traces.

[Blai 71] Blair, James Curtis. Extendable non-interactive debugging. In [Rust 71], 93-115.
Describes the commands and internal organization of the PEBUG debugging system. The system contains three basic components: a breakpoint/interpreter, a command scanner, and command processing routines.

[Bobr 72] Bobrow, Daniel Gureasko. Requirements for advanced programming systems for list processing. Communications of the ACM, 15:7 (1972 July), 618-627.
Expounds the virtues of the facilities provided by the INTERLISP system (formally known as BBN-LISP) to provide the programmer with an environment conducive to program construction, debugging, testing, and editing.

[Boul 72] Boulton, Peter I.P.; and Jeanes, David L. The structure and performance of PLUTO, a teaching oriented PL/I compiler system. INFOR, 10:2 (1972 June), 140-153.
Briefly describes a student-oriented PL/I translator which generates code for a PL/I-compatible virtual machine whose execution is interpreted. The authors stress the advantages for run-time debugging of interpreting such an intermediate code.

[Brad 68] Brady, Paul T. Writing an online debugging program for the experienced user. Communications of the ACM, 11:6 (1968 June), 423-427.
Stresses the desirability of minimizing typing, correcting errors, and producing terse error messages in typewriter-oriented, machine-language debugging systems.

[Brow 73] Brown, Arthur Robert; and Sampson, W.A. Program Debugging. Computer Monographs: Macdonald and Company Limited, and American Elsevier Publishing, 1973. 166pp. [ISBN 0-356-04267-7 and 0-444-19565-3]
Contains a superficial description of debugging techniques as related to business applications and environments.

[Brow 74] Brown, Peter J. Macro Processors and Techniques for Portable Software. Wiley Series in Computing: John Wiley and Sons Limited, 1974. 244pp. [ISBN 0-471-11005-1]

[Bull 72] Bull, G.M. Dynamic debugging in BASIC. Computer Journal, 15:1 (1972 February), 21-24.
Describes a rudimentary debugging facility added to Basic on the ICL 803.

[Chea 72] Cheatham, T.E., Jr.; and Wegbreit, Ben. A laboratory for the study of automating programming. AFIPS Conference Proceedings, 40 (SJCC 1972), 11-21.
Discusses the need for providing programming-oriented systems which can assist substantially in the development and maintenance of programs.

[Clap 74] Clapp, J.A.; and Sullivan, J.E. Automated monitoring of software quality. AFIPS Conference Proceedings, 43 (NCC 1974), 337-341.
Describes Simon, an automated aid integrated into an operating system to monitor the overall progress of a programming project's development. The system automatically retains a history of all programming bugs by category.

[Clar 74] Clark, B.L.; and Ham, F.J.B. "The Project SUE System Language Reference Manual". Technical Report CSRG-42, Computer Systems Research Group, University of Toronto, 1974 September. 97pp.

[Conr 70] Conrow, Kenneth; and Smith, Ronald. G. NEATER2: a PL/I source statement reformatter. Communications of the ACM, 13:11 (1970 November), 669-675.
Describes a PL/I prettyprinter which has an option to insert code into the source to provide a run-time frequency count.

[Conr 76] Conradi, Reidar. Further critical comments on PASCAL, particularly as a systems programming language. SIGPLAN Notices, 11:11 (1976 November), 8-25.

[Conw 73] Conway, Richard Walter; and Wilcox, Thomas R. Design and implementation of a diagnostic compiler for PL/I. Communications of the ACM, 16:3 (1973 March), 169-179.

Presents an overview of the design criteria and implementation structure of the PL/C diagnostic translator and briefly describes the diagnostic extensions to the PL/I language implemented in PL/C.

[Conw 77] Conway, Richard Walter; Moore, Charles, Jr.; and Worona, Steven L. An interactive version of the PL/C compiler. Proceedings of the ACM Annual Conference, Seattle, Washington (1977 October), 308-314.
Discusses the conceptual model of a terminal as an "internal procedure" in an interactive system for a block-structured language. Specifically, debugging commands are viewed as an extended subset of the host language, which is PL/C in this case. PL/C procedures illustrate the actions of the debugging environment.

[Cres 70] Cress, Paul H.; Dirksen, Paul H.; and Graham, J. Wesley. FORTRAN IV with WATFOR and WATFIV. Prentice-Hall, 1970. 447pp. [ISBN 13-329433-1]

[Cris 65] Crisman, P.A. (editor). The Compatible Time-Sharing System: A Programmer's Guide. The MIT Press, 1965. Section AH.8, 39pp.

[Cuff 72] Cuff, R.N. A conversational compiler for full PL/I. Computer Journal, 15:2 (1972 May), 99-104.
Describes some of the capabilities of the IBM PL/I Checkout Compiler and its implementation. It is interpretive, provides numerous run-time checks and diagnostics in source language terms, can set checkpoints and restart execution, and allows source-level program modifications at run-time. The system is highly language-dependent.

[Davi 75] Davis, Alan Mark; Tindall, Michael H.; and Wilcox, Thomas R. Interactive error diagnostics for an instructional programming system. SIGCSE Bulletin, 7:1 (1975 February), 168-171.
Briefly describes a doctoral project by Davis consisting of "an analysis system for execution-time errors". The system executes programs interpretively; when an error is detected, an analysis system attempts to diagnose the source of the error by reverse program execution and interaction with the user.

[Ehrm 72] Ehrman, John R. System design, machine architecture, and debugging. SIGPLAN Notices, 7:8 (1972 August), 8-23.

Places much of the blame for the present "software crisis" on poor system designs and machine architectures which do not enhance program debugging. The author presents his ideas on how machines should be designed to facilitate debugging.

[Evan 65] Evans, Thomas G.; and Darley, D. Lucille. DEBUG -- an extension to current online debugging techniques. Communications of the ACM, 8:5 (1965 May), 321-326.
Describes the DEBUG typewriter-oriented, machine-language debugging system for the Univac M-460. The system is noteworthy for two reasons: patches cause dynamic run-time program relocation and patches to the machine code cause automatic updating of the source code.

[Evan 66] Evans, Thomas G.; and Darley, D. Lucille. On-line debugging techniques: a survey. AFIPS Conference Proceedings, 29 (FJCC 1966), 37-50.
Surveys current (1966) online debugging techniques for both machine-language and high-level language systems. The authors stress the importance of the user having full flexibility, the user being able to make modifications in the notation of the language of the program, and automatic source program modification in parallel to modifications of the translated program.

[Ferg 63] Ferguson, H. Earl; and Berner, Elizabeth. Debugging systems at the source language level. Communications of the ACM, 6:8 (1963 August), 430-432.
Describes the BUGTRAN debugging system, a predecessor of EXDAMS, which uses a preprocessor to make modifications to Fortran programs.

[Ferl 71] Ferling, H.-D.; Schmitt, H.; Strelen, Chr.; Thielmann, H.; and Waldschmidt, H. Objektzeitfehlerprüfung bei einem Testlaufcompiler für Algol 60. Angewandte Informatik, 4 (1971 April), 165-168.
Describes the techniques used in the ALCOR Algol 60 translator to detect at run-time subscripts out of range, undefined variables, recursive procedure calls, and incompatible parameters. The system uses both translation-time generated tables and information stored within the machine code, and takes advantage of the peculiarities of the IBM 7090 hardware.

[Fong 73a] Fong, Elizabeth N. "A Set of Debugging and Monitoring Facilities to Improve the Diagnostic Capabilities of a Computer". Technical Note TN-763, National Bureau of Standards, Washington, D.C., 1973 March. 20pp. [(NTIS) COM-73-50314]

Lists and describes a number of diagnostic facilities which should be available at translation-time, link/load-time, and run-time to facilitate debugging.

[Fong 73b] Fong, Elizabeth N. Improving compiler diagnostics. Datamation, 19:4 (1973 April), 84-86.
A condensation of [Fong 73a].

[Foul 75] Foulk, Clinton R. The DO trace: a simple and effective method for debugging GOTO-free programs. SIGPLAN Notices, 10:9 (1975 September), 11-18.
Describes a form of program tracing, called the DO-trace, which is claimed to be simpler and more effective than full or selective tracing of goto-less programs.

[Gain 69] Gaines, R. Stockton. The Debugging of Computer Programs. Doctoral Dissertation: Department of Electrical Engineering, Princeton University, 1969 August. 170pp. [UMI order number 70-14,209]
Gives a historical overview of debugging methods, primarily as related to machine-language debugging systems. The author stresses the importance of certain methods and suggests new directions, most of which have now been implemented in various systems.

[Gall 74] Galley, S.W.; and Goldberg, Robert P. Software debugging: the virtual machine approach. Proceedings of the ACM Annual Conference, New York, New York (1974), 395-401.
Discusses the use of virtual machines for debugging operating systems and other privileged software at the machine language level without the necessity of a dedicated physical machine.

[Gell 75] Geller, Dennis P. Debugging other languages in APL. Software -- Practice and Experience, 5:2 (1975 April-June), 139-145.
Suggests that debugging might feasibly be done using APL as a pseudocode. In this process, a program written in some source language is translated manually into APL and debugging is carried out using the APL interpreter.

[Glas 68] Glass, Robert L. SPLINTER -- a PL/I interpreter emphasising debugging capability. Computer Bulletin, 12:5 (1968 September), 180-185.
Describes a noninteractive interpreter for a subset of PL/I written in Fortran on the Univac 1108 under EXEC II. The system depends on language extensions to provide flow and variable

traces. Interpretation of an intermediate code facilitates extensive run-time error detection and correction; postmortem dumps and frequency counts are also available.

[Goul 75] Gould, John D. Some psychological evidence on how people debug computer programs. International Journal of Man-Machine Studies, 7:2 (1975 March), 151-182.
Describes a controlled experiment designed to provide evidence on how trained programmers debug computer programs. Interactive debugging aids were available to the subjects, but few of them used the facilities due, probably, to the artificial constraints imposed on the experiment.

[Gran 66] Grant, E.E. "An Empirical Comparison of On-Line and Off-Line Debugging". Report SP-2441, System Development Corporation, Santa Monica, California, 1966 May. 16pp. [(NTIS) AD-633 907]
Describes an experiment to compare online and offline program debugging with respect to person-hours and computer time. The results are fairly inconclusive and highly dated.

[Gris 70] Grishman, Ralph. The debugging system AIDS. AFIPS Conference Proceedings, 36 (SJCC 1970), 59-64.
Describes an interactive debugging system for use on the CDC 6600 with Fortran and assembler programs. No modifications to the language translators are necessary; the debugging system accepts source listing files as input and extracts relevant information from these. Translated programs can be either simulated or directly executed and traps are not implemented by modifying the machine code. The system keeps a history file and has backup capabilities.

[Gris 71a] Grishman, Ralph. Criteria for a debugging language. In [Rust 71], 57-75.
Suggests that a debugging language contain event-driven procedures and that the debugging language closely resemble the host language. The author lists criteria for the comparison of debugging systems and briefly describes the AIDS system, giving a sample terminal session and an analysis of the system from the user's viewpoint.

[Gris 71b] Griswold, Ralph E.; Poage, J.F.; and Polonsky, I.P. The SNOBOL4 Programming Language. Prentice-Hall, 1971, 149-163. [ISBN 13-815373-6]
Describes the language-provided debugging aids of the Snobol4 programming language, which include function and variable tracing, and snap and postmortem dumping.

[Gris 75] Griswold, Ralph E. A portable diagnostic facility for SNOBOL4. Software -- Practice and Experience, 5:1 (1975 January-March), 93-104.
Describes a language extension to the macro implementation of Snobol4 which allows the user to access the underlying virtual machine. Such access can be used to aid in debugging the implementation, to write subroutines to analyze internal data structures for pedagogic reasons, and to provide capabilities which are illegal under normal circumstances.

[Habe 73] Habermann, Arie Nicolass. Critical comments on the programming language Pascal. Acta Informatica, 3 (1973), 47-57.

[Hadj 76] Hadjioannou, Michael. Debugging of parallel programs. Proceedings of the Ninth Hawaii International Conference on System Sciences, Honolulu, Hawaii (1976 January), 20-22.
Proposes an extension to EXDAMS [Balz 69] supporting the debugging of programs containing parallel control structures.

[Hall 75] Hall, Wayne; and Davidson, James Edward. "The LISP Library User's Manual". Department of Computer Science, University of British Columbia, 1975 August, 35-56.
Describes the debugging package available in LISP/MTS at the University of British Columbia. The primary facilities provided are conditional breakpoints on function calls, display and modification of the evaluation stack, the ability to change the debugging reference point, and the evaluation of any Lisp form including calling the Lisp structure editor. The package is interactive and language-dependent.

[Halp 65] Halpern, Mark. Computer programming: the debugging epoch opens. Computers and Automation, 14:11 (1965 November), 28-31.
Discusses the need for bound checks on all numeric variables, limits on transfers to prevent interrupts, checks on time constraints, and source-level dumps. The author theorizes that debugging is the next major hurdle to be surmounted in computer science.

[Hans 75] Hanson, David R. "Additions to the SITBOL Implementation of SNOBOL4 to Facilitate Interactive Program Debugging". Technical Report S4D51, Department of Computer Science, University of Arizona, 1975 April. 26pp.

Describes extensions to the Sitbol implementation of Snobol4 which facilitate interactive debugging. Using these few language additions, the author was able to write an interactive debugging command package for Snobol4 in Snobol4 itself. Great reliance is placed on the dynamic evaluation capabilities of the language (e.g., the CODE function and computable branching).

[IBM 70] International Business Machines Corporation. "IBM System/360 Operating System: PL/I (F) Language Reference Manual". Order Number GC28-8201, Systems Reference Library, New York, New York, 1970 June. 445pp.

[IBM 72] International Business Machines Corporation. "IBM OS Full American National Standard COBOL". Order Number GC28-6396-3, Systems Reference Library, New York, New York, 1972 May, 326-330.
Describes extensions to ANSI standard Cobol to allow programmers to incorporate debugging statements directly into their Cobol source programs. The features provided include paragraph tracing, symbolic tracing of program identifiers, and a primitive conditional breakpoint facility.

[Inga 72] Ingalls, Daniel H.H. The execution time profile as a programming tool. In [Rust 72], 107-128.
Stresses the importance of profiles to aid in program debugging, testing, and optimization (with emphasis on the last). The author lists seven reasons why all language processors should contain a profile facility.

[Itoh 73] Itoh, Daiju; and Izutani, Takao. FADEBUG-I, a new tool for program debugging. Proceedings of the IEEE Symposium on Computer Software Reliability, New York, New York (1973 May), 38-43.
Describes a module-testing package which analyzes all logical execution paths in a program, accepts test data and anticipated results to test each path, and reports any discrepancies between the actual and expected results.

[Jens 74] Jensen, Kathleen; and Wirth, Niklaus. PASCAL User Manual and Report. Lecture Notes in Computer Science: Springer-Verlag, 1974. 167pp. [ISBN 0-387-90144-2 and 3-540-90144-2]

[John 78] Johnson, Mark Scott. "An Implementor's Guide to RAIDE". Technical Manual TM-20, Department of Computer Science, University of British Columbia, 1978 August. 46pp.

Contains details of the UBC implementation of RAIDE which are too technical for this thesis.

[Jöns 68] Jönsson, Sven Ingvar. On-line program debugging. BIT, 8 (1968), 122-127.
Describes a graphic-display online debugging system designed for machine-language programs using a dual-processor dedicated computer. The system can switch readily from interpretive to noninterpretive execution.

[Jose 69] Josephs, William H. An on-line machine language debugger for OS/360. AFIPS Conference Proceedings, 35 (FJCC 1969), 179-186.
Describes a rather standard machine-language debugging system named DYDE for use under OS/360 with an IBM 2260 graphic display terminal.

[Kemm 76] Kemmerer, Richard A. A SIMULA 67 debugging system. Proceedings of the Fourth International Conference on the Design and Implementation of Algorithmic Languages, New York, New York (1976 June), 28-46.
Describes the design of extensions to the Simula 67 language to facilitate debugging. These include exception handling, tracing, execution profiles, postmortem dumps, and run-time assertions.

[Kirs 74] Kirsch, Barry M. "An Improved Error Diagnostics System for IBM System/360-370 Program Dumps". Master's Thesis: Department of Computer and Information Science, Ohio State University (Columbus), 1974 June. 60pp.
Describes a modified IBM System/360/370 core dump using ad hoc analysis of program interrupts to suppress irrelevant dump information.

[Knob 75] Knobe, Bruce S.; and Yuval, Gideon. Some steps towards a better PASCAL. Computer Languages, 1:4 (1975), 277-286.

[Knut 71] Knuth, Donald E. An empirical study of FORTRAN programs. Software -- Practice and Experience, 1:2 (1971 April-June), 105-133.
Presents the results of a study to determine the programming habits of Fortran programmers. The author uses the data obtained to stress the importance of program execution profiles in debugging, testing, and optimization.

[Koch 69] Kocher, Wallace. "A Survey of Current Debugging Concepts". NASA Contractor Report CR-1397, Goddard Space Flight Center, Greenbelt, Maryland, 1969 August. 91pp. [(NTIS) N69-35613]
Surveys the debugging aids available in several systems, with only short mention of online debugging techniques.

[Kuls 69] Kulsrud, Helene E. HELPER: an interactive extensible debugging system. Proceedings of the Second Symposium on Operating Systems Principles, Princeton University (1969 October), 105-111.
Describes HELPER, a machine-language interactive system used under the IDA CDC 6600 operating system. The system itself contains four parts: an incremental command translator, a simulator, debugging routines, and a communicator. Language translators must be modified to produce relevant information. The system keeps a history file and has backup capabilities.

[Kuls 71] Kulsrud, Helene E. Extending the interactive debugging system HELPER. In [Rust 71], 77-91.
Reviews the material presented in [Kuls 69] and describes two recent improvements in the system: an online instruction system and an interface enabling Fortran programs to be debugged using the system.

[Lamp 65] Lampson, Butler W. Interactive machine language programming. AFIPS Conference Proceedings, 27:1 (FJCC 1965), 473-481.

[Leca 75] Lecarme, Olivier; and Desjardins, Pierre. More comments on the programming language Pascal. Acta Informatica, 4 (1975), 231-243.

[Ledg 75] Ledgard, Henry F. Programming Proverbs. Hayden Book Company, 1975. 134pp. [ISBN 0-8104-5522-6]
Contains a short section on debugging techniques: top-down debugging and language- and system-provided debugging aids.

[Ledg 76] Ledgard, Henry F.; Singer, Andrew; and Hueras, Jon. "A User's Guide to the PASCAL Assistant". Technical Report, University of Massachusetts, 1976 June. 39pp.
Describes an environment in which Pascal programs can be developed, tested, and maintained.

[Leed 66] Leeds, Herbert D.; and Weinberg, Gerald M. Computer Programming Fundamentals. McGraw-Hill Book Company, 1966, 358-395.
Describes the machine-language and Fortran debugging facilities under the SHARE operating system for the IBM 7090. For both languages, the programmer must specify debugging actions at translation-time thru language extensions.

[Lest 71a] Lester, Bruce P. The cost of debugging. Proceedings of the Fourth Hawaii International Conference on System Sciences, Honolulu, Hawaii (1971 January), 713-715.
A condensation of [Lest 71b].

[Lest 71b] Lester, Bruce P. "Cost Analysis of Debugging Systems". Project MAC Report TR-90, Massachusetts Institute of Technology, 1971 September. 112pp.
Develops a method for predicting the cost of eight interactive debugging system features in an implementation-independent manner. The author uses the Vienna Definition Language to define an abstract machine on which the features analyzed are run. To do this he defines a set of primitive actions needed by an interactive debugging system.

[Mann 73] Mann, George A. A survey of debug systems. Honeywell Computer Journal, 7:3 (1973), 182-198.
Identifies five approaches to debugging: batch, interactive, internal, external, and playback. The author lists the desirable features of a language-independent "debugging support system" and proposes a debugging language. A brief discussion of an implementation strategy for such a system is presented.

[Marc 76] Marcotty, Michael; Ledgard, Henry F.; and Bochmann, Gregor V. A sampler of formal definitions. Computing Surveys, 8:2 (1976 June), 191-276.

[Math 74] Mathis, Robert F. Teaching debugging. SIGCSE Bulletin, 6:1 (1974 February), 59-63.
Describes a course in debugging techniques with a proposed syllabus, reading list, and projects list.

[Math 75] Mathis, Robert F. Flow trace of a structured program. SIGPLAN Notices, 10:4 (1975 April), 33-37.
Suggests that program execution flow tracing for structured programs can be prettyprinted to clarify the logical flow of execution and that variable traces can be paragraphed to display the scope of variables.

[Mill 76a] Miller, Alan. "UBC PL/1: Using PL/1 at UBC". Computing Centre, University of British Columbia, 1976 June. 80pp.

[Mill 76b] Mills, Harlan D. Software Development. IEEE Transactions on Software Engineering, SE-2:2 (1976 December), 265-273.

[Moor 75] Moore, C.G., III; Worona, Steven L.; and Conway, Richard Walter. "PL/CT -- A Terminal Version of PL/C". Technical Report 75-259, Department of Computer Science, Cornell University, 1975 September. 7pp.
Describes a version of PL/C designed for use on an IBM 2741 typewriter terminal running under the operating system CMS. A limited subset of PL/C-like statements can be entered at run-time as commands; source program changes require retranslation.

[Morg 71] Morgan, Howard L.; and Wagner, Robert A. PL/C: the design of a high-performance compiler for PL/I. AFIPS Conference Proceedings, 38 (SJCC 1971), 503-510.
Describes the design criteria and implementation of the diagnostic PL/I translator PL/C.

[Moul 67] Moulton, P.G.; and Muller, M.E. DITRAN -- a compiler emphasizing diagnostics. Communications of the ACM, 10:1 (1967 January), 45-52.
Describes the DITRAN noninteractive debugging system, which aims to detect all nonlogical errors within Fortran programs. This is accomplished by maintaining run-time control blocks for each variable to check initialization, type errors, and range errors, and by maintaining run-time tables of source program statement offsets and identifiers. The system contains a use monitor and some facilities to enhance student-oriented computing.

[ORei 76] O'Reilly, Dennis. "UBC IF: The Interactive FORTRAN Manual". Computing Centre, University of British Columbia, 1976 September. 121pp.
Describes an interactive, interpretive Fortran language processor available under the MTS operating system on an IBM System/360/370. Extensive language-dependent debugging aids are provided, including run-time source statement modification, variable and parameter checks, breakpoints, and subroutine invocation traces. Programmable breakpoints provide some degree of system extendability.

[Orga 73] Organick, Elliot Irving. Computer System Organization: The B5700/B6700 Series. ACM Monograph Series: Academic Press, 1973. 132pp. [ISBN 0-12-528250-8]

[Palm 77] Palme, Jacob. SIMDDT -- for conversational debugging of SIMULA programs. Simula Newsletter, 5:2 (1977 May), 13-16.
Briefly describes the DECsystem-10 Simula debugging system (SIMDDT). Altho the system requires the translator to generate special tables, little run-time overhead is required unless an error is detected.

[Panz 76] Panzl, David J. Test procedures: a new approach to software verification. Proceedings of the Second International Conference on Software Engineering, San Francisco, California (1976 October), 477-485.
Presents the concept of a test procedure, a formal specification of test cases to be applied to one or more target program modules. A Test Procedure Language (TPL) is defined and an implemented system employing it for Fortran programs is described.

[Pask 73] Pasko, Henry John. "A Pseudo-Machine for Code Generation". Technical Report CSRG-30, Computer Systems Research Group, University of Toronto, 1973 December. 91pp.

[Peck 75] Peck, John E.L. "The Essence of Computer Science". Technical Manual TM-7, Department of Computer Science, University of British Columbia, 1975 October. 47pp.

[Pier 74] Pierce, R.H. Source language debugging on a small computer. Computer Journal, 17:4 (1974 November), 313-317.
Describes the DDS (Dynamic Debugging-Source) interactive system for the Coral 66 programming language on the Argus 700 small-scale computer. Using translated code, the system can set breakpoints, resume execution, examine and alter variables and absolute locations, and alter the source program. Alterations can be done using only a subset of Coral since the full translator is not available during debugging.

[Poll 77] Pollack, Bary William; and Fraley, Robert A. "PASCAL/UBC User's Guide". Technical Manual TM-2, Department of Computer Science, University of British Columbia, 1977 November. 54pp.

[Pool 73] Poole, P.C. Debugging and testing. In [Baue 73], 278-318.
Presents a short tutorial on program debugging and current debugging techniques. A list of desirable features of an interactive debugging system is presented.

[Pull 69] Pullam, John M. "An Object-time Diagnostic Facility for High-level Languages". Master's Thesis: Department of Electrical Engineering, University of Toronto, 1969. 155pp.
Describes a noninteractive debugging system for PL/I programs implemented by simulating a debugging-oriented virtual machine named POODL. The system depends on language extensions to provide flow and variable tracing.

[Pyle 71] Pyle, I.C.; McLatchie, R.C.F.; and Grandage, B. A second-order bug with delayed effect. Software -- Practice and Experience, 1:3 (1971 July-September), 231-233.
Relates an experience demonstrating the difficulty of tracking down a software bug when the underlying machine architecture does not provide sufficient built-in operand error detection.

[Rain 73] Rain, Mark. Two unusual methods for debugging system software. Software -- Practice and Experience, 3:1 (1973 January-March), 61-63.
Describes two techniques for testing software: the bug farm -- a program which accepts a valid program for the system being tested as input and which outputs this program in a randomly scrambled form, and the bug contest -- paying users a bounty to discover and report bugs in a new system.

[Reis 75] Reiser, John F. "BAIL -- A Debugger for SAIL". Technical Report STAN-CS-75-523, Computer Science Department, Stanford University, 1975 October. 24pp.
Describes BAIL, a language-dependent, interactive debugging system for Sail programs running under either the TENEX or the TOPS-10 operating system on a DECsystem-10. When a breakpoint occurs, any of the following actions is permitted: a simple Sail expression can be evaluated, a user or system procedure can be called, an assignment can be made, or a BAIL command can be executed. Only a few system commands are supported; tracing and the setting of breaks are handled by procedure calls. Altho code generation changes are not required, the translator must supply BAIL with descriptions of all variables, procedures, and statements.

[Rich 69] Richards, Martin. BCPL: a tool for compiler writing and system programming. AFIPS Conference Proceedings, 34 (SJCC 1969), 557-566.

[Rich 77] Richards, Martin; Peck, John E.L.; and Manis, Vincent Stewart. "The BCPL Programming Manual". Technical Manual TM-10, Department of Computer Science, University of British Columbia, 1977 December. 62pp.

[Rust 71] Rustin, Randall (editor). Debugging Techniques in Large Systems. Prentice-Hall, 1971. 148pp. [ISBN 0-13-197319-3]
Contains a number of papers related to debugging which were presented at a Courant Institute symposium in 1970. Especially relevant are [Gris 71a], [Kuls 71], and [Blai 71].

[Rust 72] Rustin, Randall (editor). Design and Optimization of Compilers. Prentice-Hall, 1972. 141pp. [ISBN 0-13-200204-3]

[Sack 68] Sackman, H. Time-sharing versus batch processing: the experimental evidence. AFIPS Conference Proceedings, 32 (SJCC 1968), 1-10.
Summarizes the arguments for and against time-sharing and batch processing. By comparing five previous studies, the author concludes that time-shared processing results in greater programmer productivity and program quality.

[Satt 72] Satterthwaite, Edwin Hallowell, Jr. Debugging tools for high level languages. Software -- Practice and Experience, 2:3 (1972 July-September), 197-217.
Describes a modification to the Algol-W translator which provides run-time symbolic traces, frequency count information, and a postmortem dump. These facilities must be requested at translation-time and are language-dependent. The paper describes the implementation and contains statistics concerning the overhead of the debugging facilities.

[Satt 75] Satterthwaite, Edwin Hallowell, Jr. Source Language Debugging Tools. Doctoral Dissertation: Computer Science Department, Stanford University, 1975 May. 338pp. [UMI order number 75-25,602 and Technical Report STAN-CS-75-494]
Elaborates on the material presented in [Satt 72] and provides excruciating details of the implementation.

[Schw 71] Schwartz, Jacob T. An overview of bugs. In [Rust 71], 1-16.
Gives a broad overview of program bugs and the debugging process. The author advocates debugging languages which are "event-oriented", that is, which consider time to be an important component of the debugging process (e.g., What was the value of z the first time that x was greater than y?).

[Scow 72] Scowen, R.S. "Debugging Computer Programs -- A Survey with Special Emphasis on ALGOL". Report NAC-21, National Physical Laboratory, Teddington, England, 1972 June. 36pp. [(NTIS) N73-11189]
Briefly discusses principles of debugging, but is primarily concerned with benchmarking the error diagnostic facilities of many Algol 60 translators. The author distinguishes between active debugging (requiring user intervention) and passive debugging (automatic intervention by the system).

[Seid 68] Seidel, Kenneth P. Debugging past and present. Software Age, (1968 August), 22,24,26-28.
Surveys the debugging capabilities under OS/360 for programs written in assembly language, Cobol, Fortran, and PL/I.

[Site 71] Sites, Richard L. "ALGOL-W Reference Manual". Technical Report STAN-CS-71-230, Computer Science Department, Stanford University, 1971 August. 169pp.

[Step 74] Stephens, P.D. The IMP language and compiler. Computer Journal, 17:3 (1974 August), 216-223.
Briefly mentions that the EMAS translator for IMP contains a debugging option which causes code to be generated to check for such run-time conditions as undefined variables and assignment truncation, and special code to facilitate a postmortem dump in source language terms.

[Stoc 65] Stockham, Thomas G., Jr. Some methods of graphical debugging. Proceedings of the IBM Scientific Computing Symposium on Man-Machine Communication, Yorktown Heights, New York (1965 May), 57-71.
Describes an early machine-language graphic display debugging system which can produce core dumps in octal and can display the dynamic execution flow of programs as either graphs or flowcharts. Nongraphic aspects of the system are not discussed.

[Stoc 67] Stockham, Thomas G., Jr. Report on the working conference on online debugging. Communications of the ACM, 10:9 (1967 September), 590-592.

Advocates the development of "integrated programming systems". The conference centered on how machine-language debugging techniques can be applied to high-level languages.

[Stru 74] Strunk, William, Jr.; and White, E.B. The Elements of Style. Macmillan, 1959. 71pp.

[Teit 69] Teitelman, Warren. Toward a programming laboratory. Proceedings of the International Joint Conference on Artificial Intelligence, Washington, D.C. (1969 May), 1-8a.
Describes the PILOT system at BBN which contains an automatic spelling correction system (DWIM), an editing package, a breakpoint package, and an "advising facility" which allows the interfaces between modules to be modified easily. The author stresses the importance of keeping an execution history, programmable breakpoints, and the user interface.

[Text 78] Texture Support Group. "Texture User's Manual". Technical Manual TM-8, Department of Computer Science, University of British Columbia, 1978 January. 59pp.

[Thom 76] Thomson, C.M. Error checking, tracing, and dumping in an ALGOL 68 checkout compiler. Proceedings of the Fourth International Conference on the Design and Implementation of Algorithmic Languages, New York, New York (1976 June), 93-98.
Describes the run-time checks required of Algol 68 programs and the facilities provided by the FLASC diagnostic translator for full-language Algol 68, which include all of the standard capabilities except variable tracing.

[UWM 73] University of Wisconsin at Madison. "OLDS Reference Manual for the 1110". Academic Computing Center, University of Wisconsin at Madison, 1973 February. 20pp.
Describes an online debugging system for the Univac 1110. Altho designed for debugging machine-language and Fortran programs, the system itself provides a debugging language allowing for function definitions, list manipulations in the style of Lisp, and the usual breakpoint and machine-level debugging facilities.

[Vand 74] van den Bosch, Peter Nico. "The Design and Implementation of a Document Processor". Master's Thesis: Department of Computer Science, University of British Columbia, 1974 October. 157pp.

[VanT 74] Van Tassel, Dennie L. Program Style, Design, Efficiency, Debugging, and Testing. Prentice-Hall, 1974, 117-165. [ISBN 0-13-729939-7]
Presents a general discussion of debugging, ways of detecting and preventing bugs, and a taxonomy of bugs and their causes. Only a superficial discussion of automated debugging aids is presented.

[VanW 76] van Wijngaarden, Aad; Mailloux, Barry J.; Peck, John E.L.; Koster, Cornelis H.A.; Sintzoff, Michel; Lindsey, C.H.; Meertens, Lambert G.L.T.; and Fisker, R.G. (editors). Revised Report on the Algorithmic Language Algol 68. Springer-Verlag, 1976. 236pp. [ISBN 3-540-07592-5 and 0-387-07592-5]

[Vene 76] Venema, Tjeerd. "TRUST User's Guide". Department of Computer Science, University of British Columbia, 1976 August. 65pp.

[VerS 64] ver Steeg, R.L. TALK -- a high-level source language debugging technique with real-time data extraction. Communications of the ACM, 7:7 (1964 July), 418-419.
Briefly describes the TALK (Take A LooK) debugging system which is executed as a four step process: translation of the user program, translation of TALK commands, program execution, and TALK debugging output editing. Implemented for the CS-1 programming language, no special code is produced by the translator altho a symbol table must be generated. The system allows only for conditional variable traces.

[Vict 76a] Victor, Kenneth E. "The Design and Implementation of DAD, a Multi-Process, Multi-Machine, Multi-Language, Interactive Debugger". Augmentation Research Center, SRI International, Menlo Park, California, 1976 August. 51pp.
Describes a debugging system design which supports the debugging of multiprogrammed software, the components of which may be written in different high-level languages and translated and running on machines of varying architectures. DAD is being implemented in the National Software Works network environment. Altho designed as a high-level, symbolic debugging system, the debugging commands themselves are fairly low-level and resemble those of machine-language debugging systems. This report also contains a brief but useful history of debugging.

[Vict 76b] Victor, Kenneth E. "User's Guide to DAD". Augmentation Research Center, SRI International, Menlo Park, California, 1976 September. 74pp.

Explains the primitive commands of DAD. The command language is not uniform due to the large number of options applicable and specific to each command. The language is also low-level in that it makes extensive use of special symbols and uses a positional, assembler-like command format. The ability to "program" the special symbols allows the user to tailor somewhat the command language to resemble a particular host source language.

[Vict 77] Victor, Kenneth E. The design and implementation of DAD, a multiprocess, multimachine, multilanguage interactive debugger. Proceedings of the Tenth Hawaii International Conference on System Sciences, Honolulu, Hawaii (1977 January), 196-199.
A condensation of [Vict 76a].

[Watt 74] Watt, J.M.; Peck, John E.L.; and Sintzoff, Michel. Revised ALGOL 68 syntax chart. SIGPLAN Notices, 9:7 (1974 July), 39.

[Wein 71] Weinberg, Gerald M. The Psychology of Computer Programming. Computer Science Series: Van Nostrand Reinhold Company, 1971. 288pp. [ISBN 0-442-29264-3]

[Wexe 76] Wexelblat, Richard L. Maxims for malfeasant designers, or how to design languages to make programming as difficult as possible. Proceedings of the Second International Conference on Software Engineering, San Francisco, California (1976 October), 331-336.

[Wilc 74] Wilcox, Bruce; Hafner, Carole; Friedman, Paul; Hall, Wayne; McDonald, David Blair; and Davidson, James Edward. "The LISP/MTS User's Guide". Technical Manual TM-16, Department of Computer Science, University of British Columbia, 1974 September, 65-74.
Describes the language-provided debugging aids of the LISP/MTS system at the University of British Columbia, which include an evaluation stack dump, function tracing, step execution, a form of reverse program execution, and the evaluation of arbitrary Lisp forms.

[Wilc 76] Wilcox, Thomas R.; Davis, Alan Mark; and Tindall, Michael H. The design and implementation of a table driven, interactive diagnostic programming system. Communications of the ACM, 19:11 (1976 November), 609-616.

Provides an overview with examples of CAPS, an interactive diagnostic translator/interpreter that allows beginning programmers to prepare, debug, and execute simple programs at a PLATO IV display terminal. CAPS automatically diagnoses (but does not correct) errors both at translation- and run-time by reverse program execution and interaction with the user. The system supports programs written in Fortran, Cobol, and PL/I.

[Wino 75] Winograd, Terry. Breaking the complexity barrier again. SIGPLAN Notices, 10:1 (1975 January), 13-30.
Proposes that what is needed to break the present "complexity barrier" in software development is an integrated programming system capable of relieving many of the programmer's tedious burdens, such as debugging.

[Wolm 72] Wolman, B.L. Debugging PL/I programs in the Multics environment. AFIPS Conference Proceedings, 41:1 (FJCC 1972), 507-514.
Mentions the language extensions and system facilities of Multics PL/I for program debugging. The system provides, without the use of a special debugging translator, patching in source language terms, conditional breakpoints, breakpoint subroutines, tracing of procedure calls, and profiles of program execution.

[Wort 72] Wortman, David Barkley. "A Study of Language Directed Computer Design". Technical Report CSRG-20, Computer Systems Research Group, University of Toronto, 1972 December. 207pp.

[Wulf 73] Wulf, William; and Shaw, Mary. Global variable considered harmful. SIGPLAN Notices, 8:2 (1973 February), 28-34.

[Zelk 71] Zelkowitz, Marvin V. Reversible Execution as a Diagnostic Tool. Doctoral Dissertation: Department of Computer Science, Cornell University, 1971 January. 149pp. [UMI order number 71-17,676]
Describes an extension to the PL/C language and translator which allows batch programs to be executed in reverse when an error is detected. Altho the author summarily dismisses online environments, he advocates the use of extensive interrupt facilities to aid in program debugging and proposes hardware extensions to aid debugging.

[Zelk 73] Zelkowitz, Marvin V. Reversible execution. Communications of the ACM, 16:9 (1973 September), 566.
A condensation of [Zelk 71].

[Zimm 67] Zimmerman, Luther L. On-line program debugging -- a graphic approach. Computers and Automation, 16:10 (1967 November), 30-31,34.
Briefly describes GBUG, a graphic display online debugging system for machine-language programs. The author claims that speed of information display is the primary advantage of a graphic debugging system.

Appendix A. RAIDE System Functions

1. Status Functions

CALLER(foo)
: returns a specific value identifying the segment which caused entry into the segment-specific 'foo'.

COMMENT(n,foo)
: returns the 'n'-th comment of the segment-specific 'foo' as a character string. The null string is returned if there is no 'n'-th comment.

CURRENT_EXCEPTION
: returns the name (as a character string) of the currently active exception. The null string is returned if there is no currently active exception.

DEBUG_LEVEL
: returns an integer indicating the current level of debugging support in effect. (See Table VI for the range of result values.)

DECLARATION_LIST(n)
: returns a data-specific value identifying the 'n'-th most recently declared top-level debugging variable or procedure.

DEFERRED_ACTION_LIST(n)
: returns a segment-specific value identifying the 'n'-th most recently deferred action. n=0 identifies the currently active deferred action, if there is one.

DEFINED(foo)
: returns logical true if the data-specific 'foo' currently has a value; false is returned otherwise.

ENVIRONMENT_LIST(n)
: returns the name (as a character string) of the 'n'-th most recently saved environment. n=0 returns the name of the currently active environment, if there is one. The null string is returned if there is no 'n'-th accessible environment.

LANGUAGE(foo)
: returns the name (as a character string) of the source language in which the segment-specific 'foo' was written.

RANGE(foo,m,n)
: returns a generic value identifying a subrange of the generic 'foo' between 'm' and 'n'.

REFERENCE_POINT
returns a segment-specific value identifying the current reference point.

TYPE(foo)
returns the type (as a character string) of the data-specific 'foo'.

VALUE(foo)
returns the current value of the segment- or data-specific 'foo'. A question mark is returned if 'foo' is currently undefined.

2. Display Format Functions

LINE(n)
causes the output display device to be advanced 'n' lines. (The argument is optional and defaults to 1.)

PAGE
causes the output display device to be advanced to the beginning of a new page.

SPACE(n)
causes the output display device to be spaced over 'n' spaces. If spacing fills the display buffer, it is displayed and the output display device is advanced one line. (The argument is optional and defaults to 1.)

TAB(n)
causes the output display device to be tabbed to column 'n' of the current (or subsequent) line.

3. Analysis Functions

#ACCESSES(foo)
returns the number of times the data-specific 'foo' has been accessed since system initialization.

#ALLOCATIONS(type)
returns the number of times dynamic allocations of type 'type' have been made since system initialization.

#ENTRIES(foo)
returns the number of times the segment-specific 'foo' has been entered since system initialization.

#EXITS(foo)
returns the number of times the segment-specific 'foo' has been exited since system initialization.

#UPDATES(foo)
returns the number of times the data-specific 'foo' has been updated since system initialization.

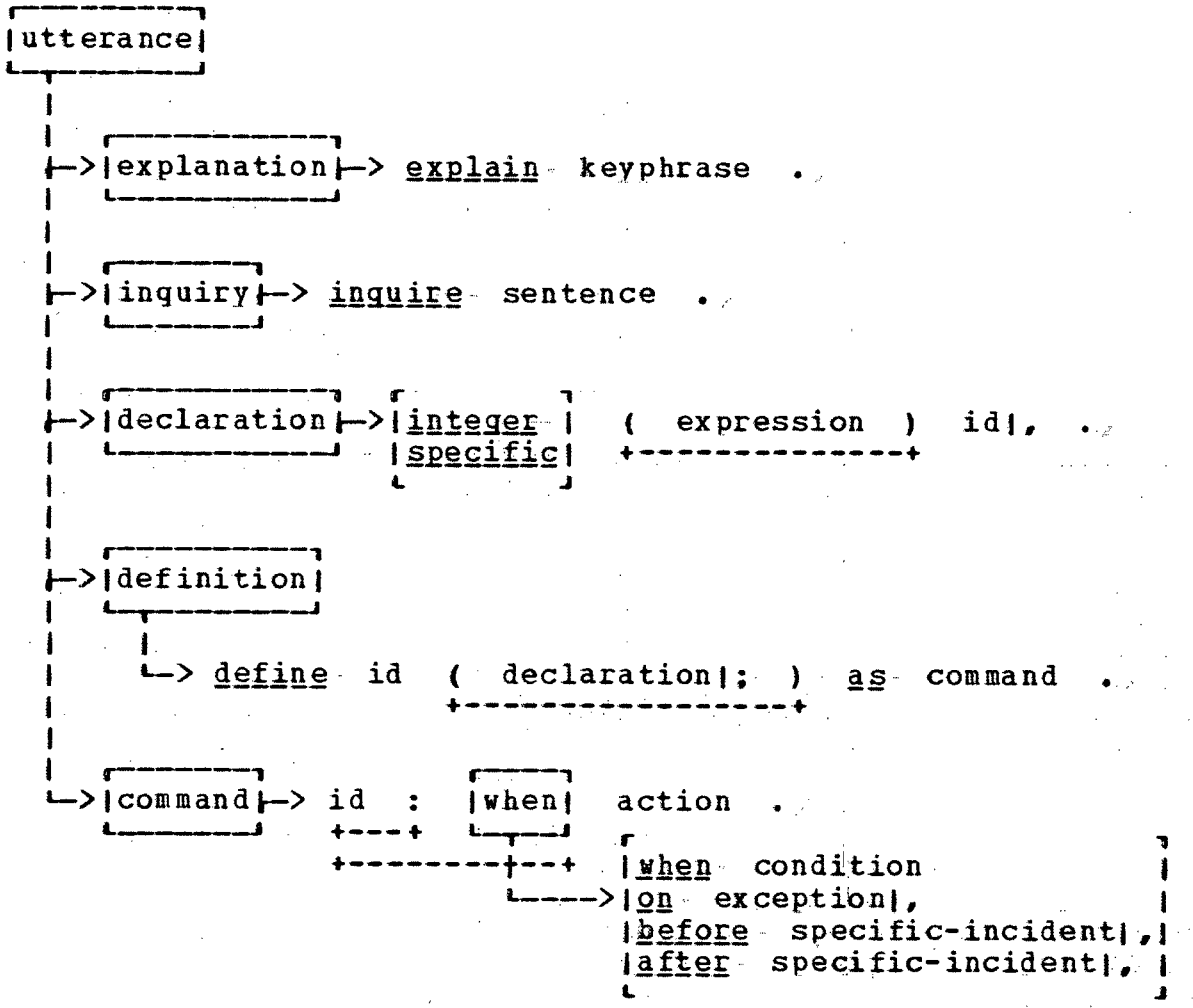
4. Language-Dependent System Functions

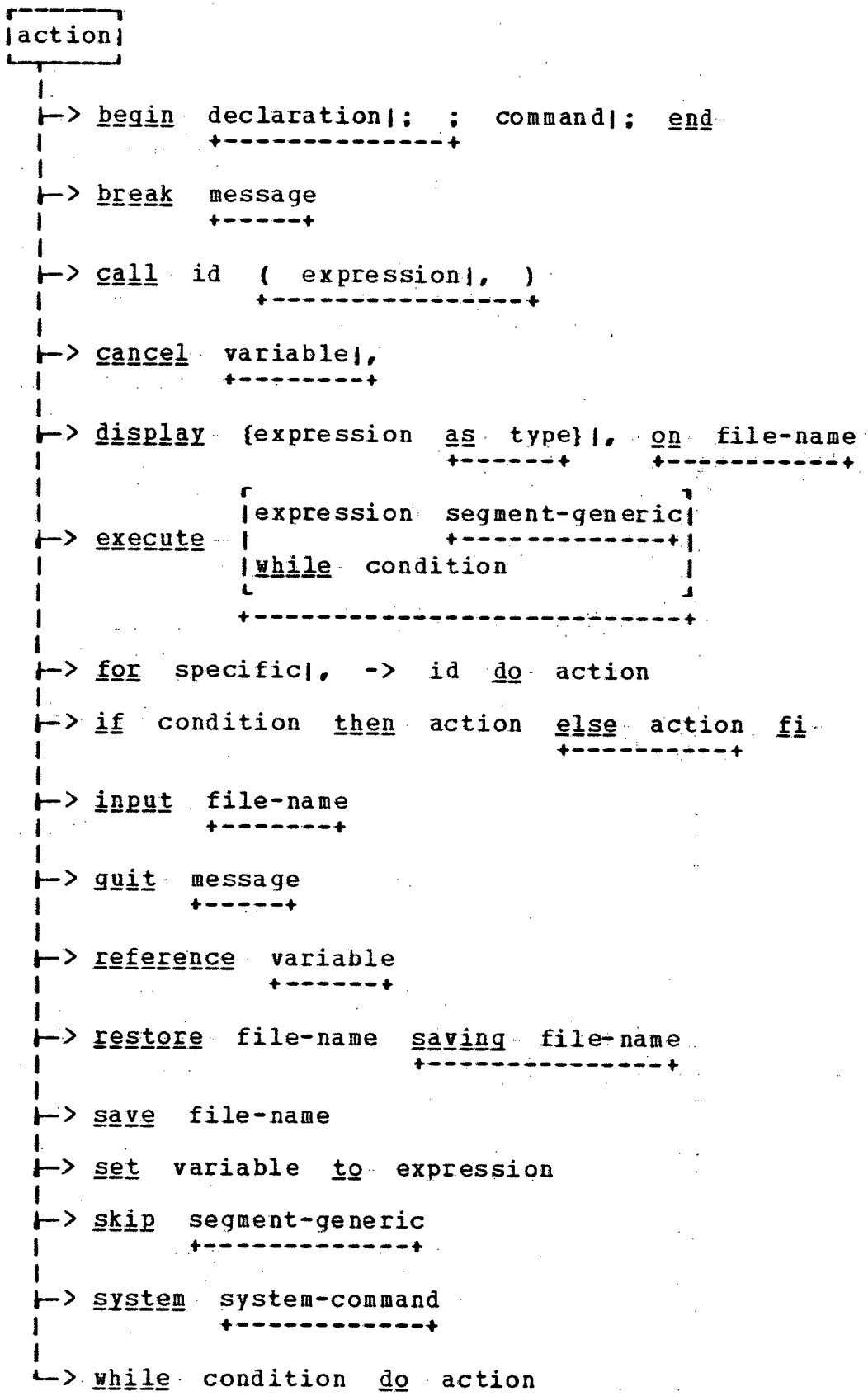
CURRENT_segment-generic
returns a specific value identifying the currently executing segment of type 'segment-generic'.

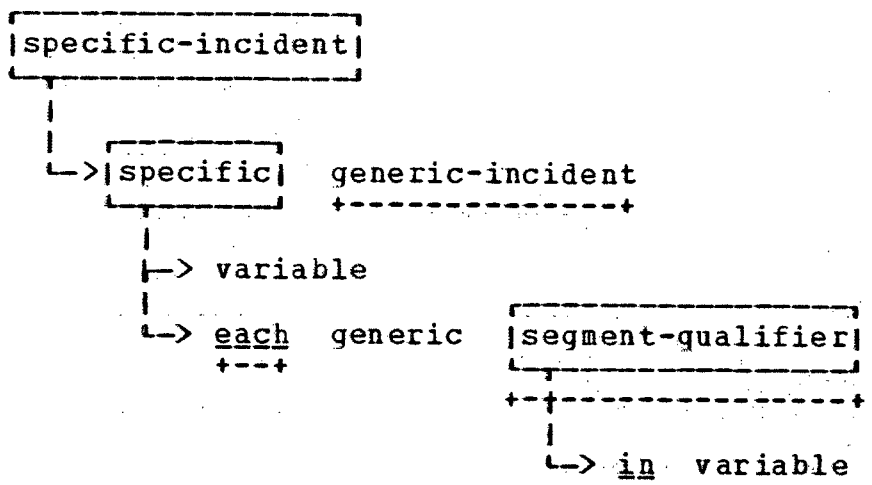
Appendix B. Dispel Syntax Chart

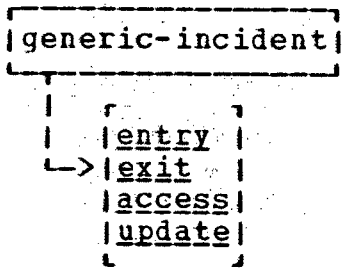
Legend:

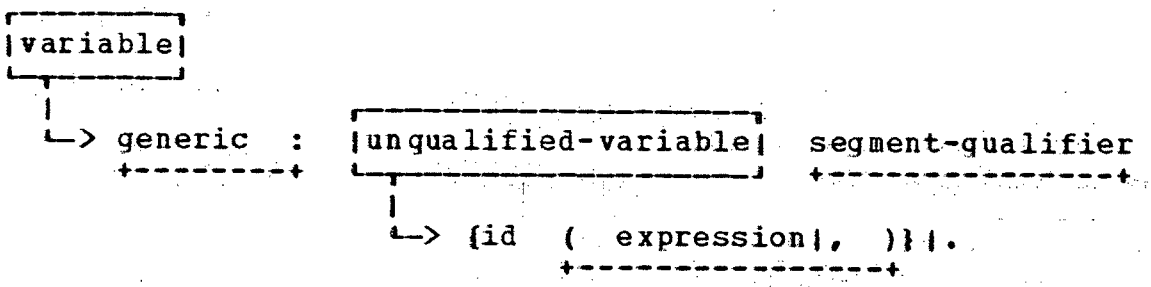
A ⊢→ B ⊢→ C └→ D	A is defined as either B, C, or D	[A] [B] [C]	choose one of A, B, or C
A|B	A, ABA, ABABA, ...	abc	abc is optional
{A B}	treat A and B as one construct		




action
begin declaration ; command ; end
break message
call id (expression ,)
cancel variable ,
display expression as type , on file-name
execute expression segment-generic
while condition
for specific , -> id do action
if condition then action else action fi
input file-name
quit message
reference variable
restore file-name saving file-name
save file-name
set variable to expression
skip segment-generic
system system-command
while condition do action



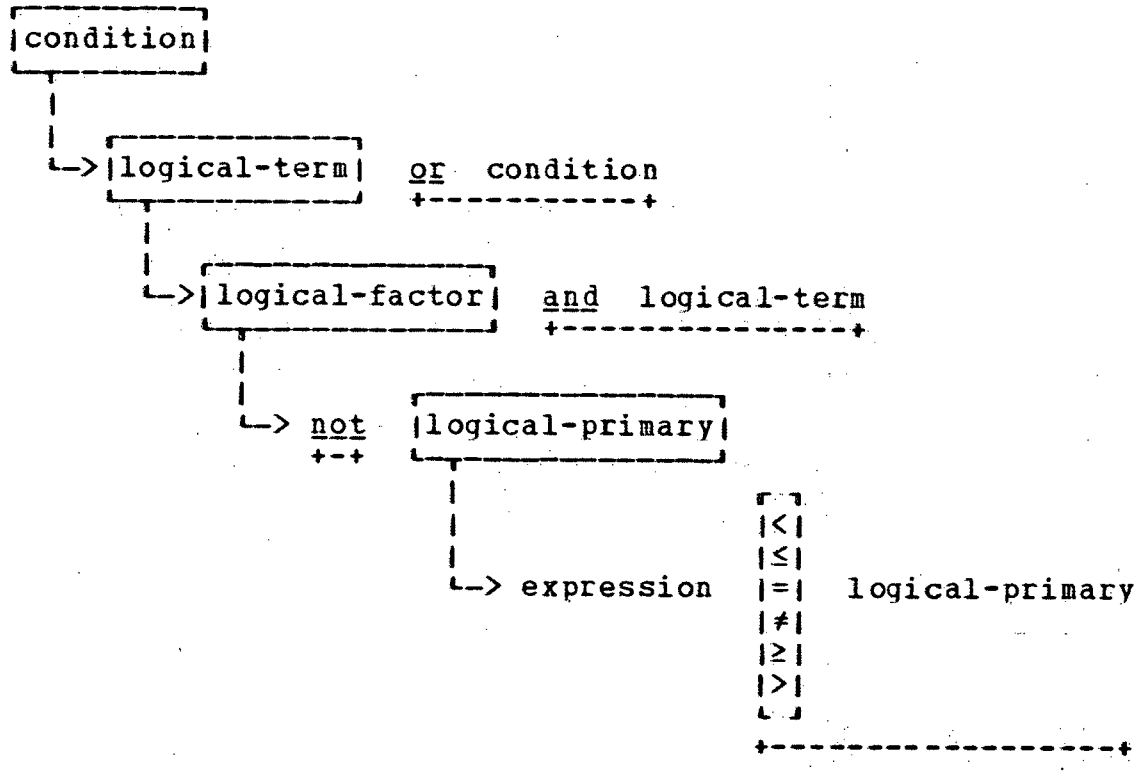


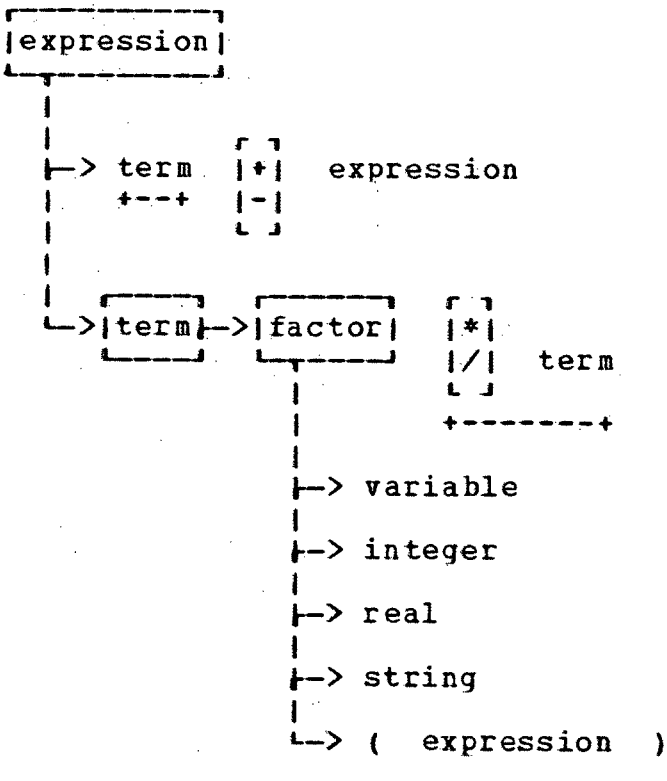


nonterminals represented by string literals:
file-name, keyphrase, message, sentence, system-command

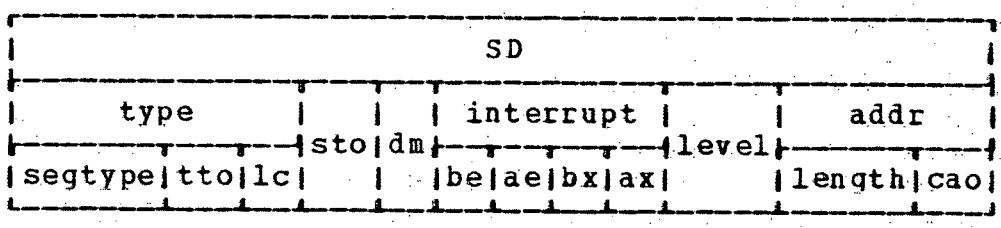
nonterminals represented by identifiers:
exception, generic, segment-generic, type

lexically primitive nonterminals:
id, integer, real, string


condition
logical-term
or condition
logical-factor
and logical-term
not
logical-primary
expression
<
≤
=
≠
≥
>
logical-primary


expression
term
+
-
expression
term
factor
*
/
term
variable
integer
real
string
(expression)

Appendix C. SPAM Descriptor Formats



type = type information field

- segtype = segment type code (e.g., procedure or block)
- tto = the type table offset to the type describing the segment (e.g., **proc (int) real**)
- lc = if segtype = procedure, a code indicating in which source language the procedure was written {The special code "external" indicates that the procedure is defined externally to the machine.}

sto = the RAIDE symbol table offset to the entry associated with this item

dm = a flag to indicate that this segment is executing in debug mode

interrupt = interrupt information field

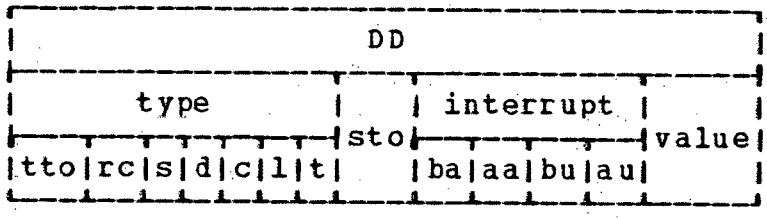
- be = before entry interrupt enabled flag
- ae = after entry interrupt enabled flag
- bx = before exit interrupt enabled flag
- ax = after exit interrupt enabled flag

level = the lexic level of this segment

addr = if lc = external, the address of the external routine

- length = if lc ≠ external, the length of the segment in syllables
- cao = if lc ≠ external, the code area offset to the first instruction of the segment

Figure C-1. Segment Descriptor Format



type = type information field

- tto = the type table offset to the type describing the data item (e.g., integer, real, scalar, or structure)
- rc = the number of levels of indirection associated with the data item (e.g., if rc = 1, the item is a reference to some other value)
- s = subrange of integer or scalar flag (i.e., the bounds table contains the bounds for the integer values which this data item can possess)
- d = value field defined flag
- c = constant value flag (i.e., the value of this data item cannot be altered)
- l = long value flag (i.e., the actual value of this item is in the free storage area, the value field of this descriptor contains an offset into the free storage area)
- t = temporary value flag (used to facilitate garbage collection)

sto = the RAIDE symbol table offset to the entry associated with this item

interrupt = interrupt information field

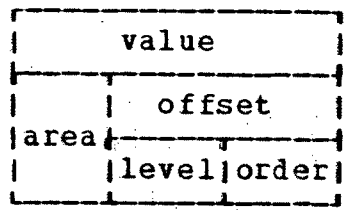
- ba = before access interrupt enabled flag
- aa = after access interrupt enabled flag
- bu = before update interrupt enabled flag
- au = after update interrupt enabled flag

[continued on next page]

Figure C-2. Data Descriptor Format

value = value field {The format of this field is dependent on the type field, and is defined as follows:}

a. if rc > 0:



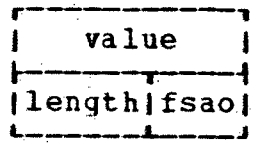
area = a code for the area to which the reference refers (e.g., dynamic storage stack or free storage area)

offset = the offset in the indicated area to the data item

level = if area = dynamic storage stack, the lexic level of the item

order = if area = dynamic storage stack, the order of the item

b. if l = true:



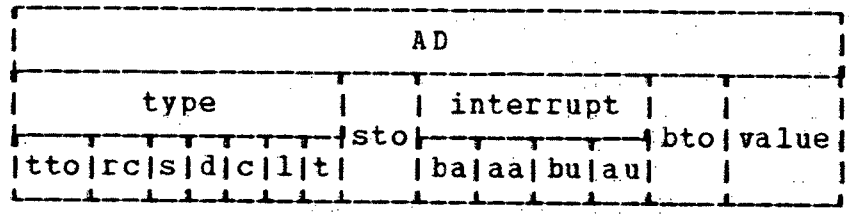
length = the length of the data item in the free storage area

fsao = the offset in the free storage area to the first syllable of the data item

c. otherwise:

value = the actual value of the data item

Figure C-2. Data Descriptor Format (continued)



type = type information field {This field has the same format as that of a data descriptor, but with special meaning for the following subfields:}

d = a flag to indicate that storage for the array has been allocated

t = a flag to indicate that the storage referenced by this descriptor is also referenced by another descriptor and that it must not be deallocated

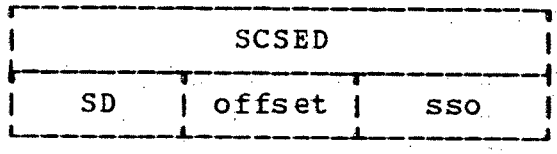
sto = the RAIDE symbol table offset to the entry associated with this item

interrupt = interrupt information field {This field has the same format as that of a data descriptor.}

bto = the offset in the bounds table to the bounds of the array

value = value field {This field has the same format as that of a data descriptor for a reference (i.e., when rc > 0). If area = dynamic storage stack, the (level,order) pair addresses the first element of the array. The subsequent elements are addressed (level,order+1), (level,order+2),....}

Figure C-3. Array Descriptor Format

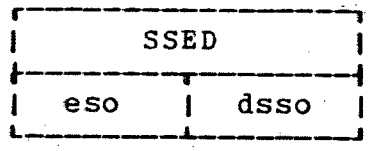


SD = a field containing the same subfields as that of a segment descriptor

offset = the offset in the code area (relative to the first instruction of this segment) to the next instruction to be executed

sso = the offset in the scope stack to the scope which is local to this segment

Figure C-4. Segment Control Stack Entry Descriptor Format

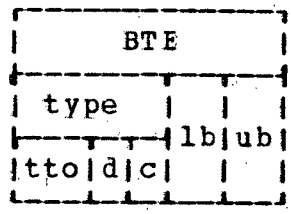


eso = the offset in the expression stack to the first entry available to the segment(s) which index(es) this scope stack entry

dsso = the offset in the dynamic storage stack to the first entry available to the segment(s) which index(es) this scope stack entry

Figure C-5. Scope Stack Entry Descriptor Format

Appendix D. SPAM Table Entry Formats



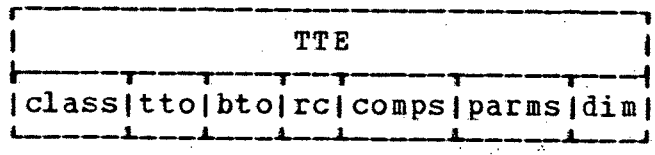
type = type information field

- tto = the type table offset to the subrange of integer or scalar type for which this entry defines the bounds
- d = bounds fields defined flag
- c = constant bounds value flag

lb = lower bound value

ub = upper bound value

Figure D-1. Bounds Table Entry Format



class = class code {The format of the remaining fields is dependent on the class field, as follows:}

a. primitive

tto = a self-referent type table offset

b. subrange

tto = the type table offset to the base type of the subrange

bto = the bounds table offset defining the subrange

c. scalar

tto = the type table offset to the base type used to implement the scalar values

bto = the bounds table offset defining the subrange used to implement the scalar values

d. reference

tto = the type table offset to the base type of the reference

rc = the number of levels of indirection associated with the reference (i.e., reference count)

e. structure

comps = the number of components constituting the structure {The components themselves are described by the succeeding 'comps' type table entries.}

[continued on next page]

Figure D-2. Type Table Entry Format

f. procedure

tto = the type table offset to the type yielded by the procedure (which may, of course, be void)

parms = the number of parameters which the procedure accepts {The parameters themselves are described by the succeeding 'parms' type table entries.}

g. array

tto = the type table offset to the base type of the array

dim = the dimensionality (i.e., number of subscripts) of the array {The subscripts themselves are described by the succeeding 'dim' type table entries.}

h. union

comps = the number of components constituting the union {The components themselves are described by the succeeding 'comps' type table entries.}

i. deferred {to indicate that the actual entry description is elsewhere in the type table}

tto = the type table offset to the actual entry description

Figure D-2. Type Table Entry Format (continued)

Appendix E. SPAM Instruction Repertoire

/* Bloody instructions which, being learned,
return to plague the inventor.
-- William Shakespeare, Macbeth */

The instructions of SPAM have been divided into seven classes based on their use. For each instruction, a mnemonic is given. This is followed by a list of its operand(s), if any, and an explanation of its execution, enclosed in braces. Also, for some of the instructions, the associated Spamdol code is presented in outline form. Necessary consistency and error checks have been suppressed to improve overall readability.

1. Segment Control Instructions

MARKSS {Push a new entry onto the scope stack to establish a new scope.}

```
push_ss(esptr+1,dssptr+1)
```

DEFSEG level, order
{Push a new entry onto the segment control stack to define a segment whose descriptor is addressed by the pair (level,order).}

```
push_scs(dss[level,order],0,ssptr)
```

DEFDATA n, loc
{Push n new entries onto the dynamic storage stack and allocate storage if necessary. The n descriptors starting at location (0,loc) are templates for the entries to be made.}

```
for i := loc to loc+n-1 do
   push_dss(dss[0,i])
   if dss[dssptr] is array_descriptor
      then allocate_array
```

```
PARM    loc
        {The top of the expression stack must contain a
           parameter to the procedure whose segment descriptor
           must be second from the top. Type-check the parame-
           ter, pop it from the expression stack, and push it
           onto the dynamic storage stack. (0,loc) addresses
           the template for the procedure's formal parameter,
           which is necessary to properly associate formal and
           actual parameters.}

        local parm_no
        parm_no := dssptr - ss[ssptr].dsso + 2
        if ¬scs[scsptr].dm or check_parameter_type(parm_no,loc)
           then push_dss(es[esptr])
                dss[dssptr].sto := dss[0,loc].sto
                pop_es
           else interrupt "parameter type mismatch"


CALL    {The top of the expression stack must contain the
           descriptor for a segment which is to be called. The
           parameters to the segment are all on the dynamic
           storage stack and have been type-checked. The
           segment descriptor is popped and a new entry is
           pushed onto the segment control stack, thus initi-
           ating execution of the segment.}

        local no_parms
        no_parms := dssptr - ss[ssptr].dsso + 1
        if ¬scs[scsptr].dm or
              check_number_of_parameters(no_parms)
           then push_scs(es[esptr],0,ssptr)
                pop_es
           else interrupt "wrong number of actual parameters"


EXITSEG segtype
        {Pop entries from the segment control stack up thru the
           first whose segment type is segtype. If the segment
           is returning a value, it must be moved onto the top
           of the expression stack. None of the intermediate
           segments popped can return a value.}

        local result
        result := es[esptr]
        while (scs[scsptr].segtype ≠ segtype) do
           pop_scs
        if (tt[scs[scsptr].tto].tto ≠ void)
           then if ¬scs[scsptr].dm or
                        check_result_type(result.type)
                   then push_es(result)
                   else interrupt "result type mismatch"
```

```
CYCLE    segtype
         {Pop entries from the segment control stack until the
            first segment whose type is segtype is encountered.
            Reset this segment so that it begins reexecuting
            from its beginning. None of the intermediate seg-
            ments popped can return a value.}

         while (scs[scsptr].segtype ≠ segtype) do
            pop_scs
         scs[scsptr].offset := 0


SKIP     segtype, offset
         {The top of the expression stack must contain a boolean
            value. If this value is false, the first segment
            whose segment type is segtype continues executing at
            the specified offset. The boolean is popped from
            the expression stack. SKIP cannot be used to exit
            thru a segment which returns a value; EXITSEG must
            be used in such circumstances.}

         if es[esptr].value
            then pop_es
            else pop_es
                 while (scs[scsptr].segtype ≠ segtype) do
                    pop_scs
                 scs[scsptr].offset := offset


CASE     {The top of the expression stack must contain an
            integer value and the next entry must contain an
            array descriptor. After popping the top two stack
            entries, the array is indexed to obtain the offset
            from which the current segment is to continue
            executing.}

         if check_array_subscript
            then scs[scsptr].offset := index_array
                 pop_es
                 pop_es
            else interrupt "case range error"
```

2. Data Access Instructions

```
PUSHV    level, order
         {Push the value addressed by the pair (level,order)
            onto the expression stack.}
```

PUSHR level, order
{Push a reference to the value addressed by the pair (level,order) onto the expression stack.}

POP {Pop the top entry from the expression stack.}

STORE {The top of the expression stack must contain a value which is to be stored into the location indicated by the reference which is next from the top. These two stack entries are then popped. STORE must check for scope violations involving references; it is not allowed to assign a value to a reference variable whose lifetime exceeds that of the value itself. STORE must also check for size errors, check for assignment to a constant, and process the before and after update interrupts.}

COPY {Duplicate the entry on the top of the expression stack.}

SWAP {Swap the top two entries on the expression stack.}

SLICE {The top of the expression stack must contain the value of a subscript into the array described by the entry second from the top. (This entry must either be an array descriptor or a reference to an array descriptor.) After popping the top two stack entries, SLICE returns on top of the expression stack one of the following:
1. the value of an element of the array,
2. a reference to an element of the array,
3. a descriptor for a subarray of the array, or
4. a reference to a subarray descriptor.}

SELECT n
{The top of the expression stack must contain a descriptor for a structured value or a reference to such a value. This descriptor is replaced by either the value of the n-th subfield of the structure or by a reference to the same.}

DEREF {The top of the expression stack must contain a reference value. This entry is replaced by the value referenced.}

3. Arithmetic and String Instructions

None of these instructions operate on arrays. Also, no automatic type conversions are performed on operands (i.e., the operand(s) must be of the correct type).

a. binary instructions {The top two entries of the expression stack must contain the operands.}
   i. integer or subrange of integer operands
      ADD SUB MUL DIV MOD LT LE EQ NE GE GT
   ii. real operands
      ADD SUB MUL DIV LT LE EQ NE GE GT
   iii. boolean operands
      AND OR
   iv. string operands
      CONCAT
   v. reference operands
      EQ NE

b. unary instructions {The top of the expression stack must contain the operand.}
   i. integer or subrange of integer operand
      ABS NEG SUCC PRED
   ii. real operand
      ABS NEG
   iii. boolean operand
      NOT

SUBSTR {The top of the expression stack must contain either two or three operands. If three operands are supplied, the top two entries must contain integer values which represent the length (top entry) and starting position (next entry) of the substring to be obtained, and the third entry from the top must contain a string value or a reference to a string from which the substring will be taken. If only two operands are supplied, the length is assumed to be the remainder of the string and only the starting position and base string need to be specified. After popping its operands, SUBSTR returns on top of the expression stack either a string or a reference to a string which is the substring desired.}

4. Transput Instructions

PUT {The top of the expression stack must contain a string value which is output to the primary output device. This string is then popped from the expression

stack.}

GET {The next record from the primary input device is read into a newly allocated string in the free storage area and a descriptor of this string value is pushed onto the expression stack. A null string indicates that the end of the file has been encountered.}

5. Storage Management Instructions

SETBT bto
{The top two entries of the expression stack must contain integer or subrange of integer values. These two entries are popped and their values placed into the bounds table at offset bto. The top entry defines the dynamic upper bound which is to be placed into the table; the next entry defines the lower bound.}

ALLOC tto
{Storage for a structure of type tto is allocated in the free storage area and a data descriptor of this type is pushed onto the expression stack.}

6. Status Instructions

TESTYPE mask
{The type field of the descriptor on top of the expression stack is checked against the mask and the entry is replaced by a boolean value which indicates whether or not the masked bits of the type field are set.}

LENGTH {The top of the expression stack must contain a string value. This entry is replaced by the length of the string.}

LBOUND {The top of the expression stack must contain an array descriptor or a value of scalar type. This entry is replaced by the lower bound, obtained from the bounds table, for the item.}

UBOUND {The top of the expression stack must contain an array descriptor or a value of scalar type. This entry is replaced by the upper bound, obtained from the bounds table, for the item.}

7. Miscellaneous Instructions

NOOP {Execution continues with the next instruction.}

CONVERT tto
{The top of the expression stack must contain a value of primitive, subrange of integer, scalar, or reference type. This entry is replaced with the value converted to the primitive type tto.}

ASSERT {The top of the expression stack must contain a boolean value. This entry is popped and, if its value is false, an interrupt is generated.}

SIGNAL exception
{A user-specified interrupt is generated, signaling the occurrence of some exception.}

STOP {The machine is abnormally terminated.}

DUMP code
{One of the machine areas, stacks, or tables (cf., Figure IV-1) is dumped. The code indicates which is to be dumped.}

Appendix F. An Example SPAM Object Program

```
/* A really great talent finds its happiness in execution.
                                    -- Johann Wolfgang von Goethe */
```

This appendix presents a complete example of a SPAM object program, including both the machine code and the descriptors needed for its execution. The example chosen is Towers of Hanoi [Peck 75:30-32]. A source program for it written in an Algol-like language is contained in Figure F-1, and its SPAM translation is given in Figure F-2. The SPAM code is presented in an assembler-like notation; literal and variable operands must be converted into offsets into the dynamic storage stack. The left-most column of Figure F-2 contains the machine code translation of the SPAM assembler code. The machine code is only presented here for completeness; no further explanation will be given. Refer to Appendix E for explanations of the SPAM instructions. The initial states of the dynamic storage stack, the segment control stack, and the type table are outlined in Figure F-3; only important fields are mentioned. The expression stack and bounds table are initially empty.

```
stmt     lexic
number   level     source code

                    main:
          1         begin
                       proc hanoi (integer value n ;
                          char value me,de,ma) ;
   1        2          if n>0 then
   2                      hanoi(n-1,me,ma,de) ;
   3                      print(" MOVE DISK ",n," FROM PEG ",
                             me," TO PEG ",ma) ;
   4                      hanoi(n-1,de,me,ma)
            2          fi ;
   5                   hanoi(4,"A","B","C")
          1         end
```

Figure F-1. Towers of Hanoi Source Program

```
machine
code          assembler_code     comments

  1 0 20      DEFSEG  STMT5      initial call to 'hanoi'
  0           MARKSS
 32 0 1       PUSHV   HANOI
 32 0 2       PUSHV   4              pass the first parameter
  3 12        PARM    N
 32 0 3       PUSHV   'A'            pass the second parameter
  3 13        PARM    ME
 32 0 4       PUSHV   'B'            pass the third parameter
  3 14        PARM    DE
 32 0 5       PUSHV   'C'            pass the fourth parameter
  3 15        PARM    MA
  4           CALL                   now invoke 'hanoi'

              ORIGIN  51         start 'hanoi' at offset 51
  1 0 16      DEFSEG  STMT1      if statement
  1 0 6       DEFSEG  IF-STMT        define the bounds of the
                                     if statement
 32 1 0       PUSHV   N              evaluate the if expression
 32 0 7       PUSHV   0
 78           GT
  7 64 103    SKIP    ENDIF          skip the then clause if false

  1 0 17      DEFSEG  STMT2      recursive call to 'hanoi'
  0           MARKSS
 32 0 1       PUSHV   HANOI
 32 1 0       PUSHV   N              evaluate 'n-1' and pass it
 32 0 8       PUSHV   1
 65           SUB
  3 12        PARM    N
 32 1 1       PUSHV   ME             pass the second parameter
  3 13        PARM    ME
 32 1 3       PUSHV   MA             pass the third parameter
  3 14        PARM    DE
 32 1 2       PUSHV   DE             pass the fourth parameter
  3 15        PARM    MA
  4           CALL
```

[continued on next page]

Figure F-2. Towers of Hanoi SPAM Code

```
machine
code          assembler code     comments

  1 0 18    DEFSEG   STMT3      transput code
 32 0 9     PUSHV    ' MOVE DISK '
 32 1 0     PUSHV    N
193 5       CONVERT  STRING          convert 'n' to type string
 82         CONCAT                   concatenate the six arguments
 32 0 10    PUSHV    ' FROM PEG '        of 'print' into
 82         CONCAT                       one output string
 32 1 1     PUSHV    ME
 82         CONCAT
 32 0 11    PUSHV    ' TO PEG '
 82         CONCAT
 32 1 3     PUSHV    MA
 82         CONCAT
 96         PUT

  1 0 19    DEFSEG   STMT4      recursive call to 'hanoi'
  0         MARKSS
 32 0 1     PUSHV    HANOI
 32 1 0     PUSHV    N                evaluate 'n-1' and pass it
 32 0 8     PUSHV    1
 65         SUB
  3 12      PARM     N
 32 1 2     PUSHV    DE               pass the second parameter
  3 13      PARM     ME
 32 1 1     PUSHV    ME               pass the third parameter
  3 14      PARM     DE
 32 1 3     PUSHV    MA               pass the fourth parameter
  3 15      PARM     MA
  4         CALL
```

Figure F-2. Towers of Hanoi SPAM Code (continued)

The initial dynamic storage stack entries:

offset	type	important field values	comment
0	segment	segtype=proc tto=1 level=0 length=28 cao=1	main
1	segment	segtype=proc tto=6 level=1 length=109 cao=51	hanoi
2	data	tto=2 l=F value=4	4
3	data	tto=5 l=T length=1 fsao=1	'A'
4	data	tto=5 l=T length=1 fsao=2	'B'
5	data	tto=5 l=T length=1 fsao=3	'C'
6	segment	segtype=if-stmt tto=1 level=1 length=103 cao=57	if stmt
7	data	tto=2 l=F value=0	0
8	data	tto=2 l=F value=1	1
9	data	tto=5 l=T length=11 fsao=4	' MOVE DISK '
10	data	tto=5 l=T length=10 fsao=15	' FROM PEG '
11	data	tto=5 l=T length=8 fsao=25	' TO PEG '
12	data	tto=2 d=F	n
13	data	tto=5 d=F	me
14	data	tto=5 d=F	de
15	data	tto=5 d=F	ma
16	segment	segtype=stmt length=106 cao=54	stmt 1
17	segment	segtype=stmt length=29 cao=70	stmt 2
18	segment	segtype=stmt length=26 cao=102	stmt 3
19	segment	segtype=stmt length=29 cao=131	stmt 4
20	segment	segtype=stmt length=25 cao=4	stmt 5

The initial segment control stack entry:

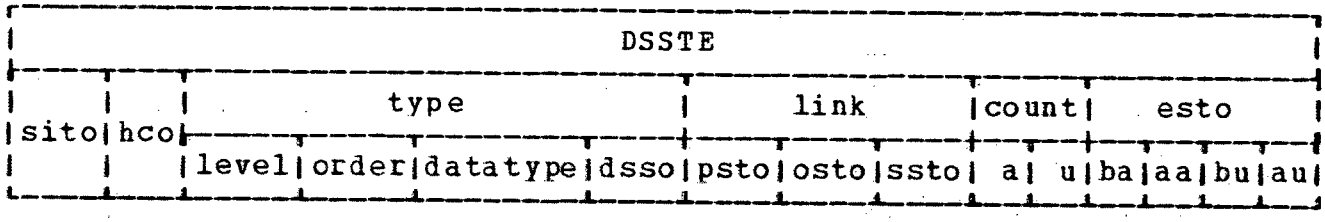
segtype=proc tto=1 length=28 cao=1 offset=0 sso=1

The initial type table entries:

offset	field values	comment
1	class=primitive tto=1	void
2	class=primitive tto=2	integer
3	class=primitive tto=3	real
4	class=primitive tto=4	boolean
5	class=primitive tto=5	string
6	class=procedure parms=4	proc(...)void
7	class=primitive tto=2	integer
8	class=primitive tto=5	string
9	class=primitive tto=5	string
10	class=primitive tto=5	string

Figure F-3. Towers of Hanoi Initial SPAM Machine State

Appendix G. RAIDE Symbol Table Entry Formats



sito = string index table offset

hco = homonym chain offset

type = type information field

- level = the lexic level of the item
- order = the lexic order of the item
- datatype = data type code (e.g., variable or constant)
- dsso = the SPAM dynamic storage stack offset to the "template descriptor" associated with this item

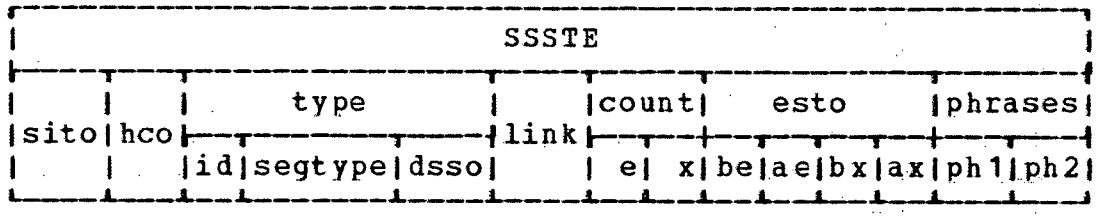
link = linkage information field

- psto = the symbol table offset to the parent at this level
- osto = the symbol table offset to an offspring at this level
- ssto = the symbol table offset to a sibling at this level

count = a count of the number of times the associated data-specific has been accessed (a) and updated (u) since system initialization

esto = the event symbol table offset for the deferred action (if any) associated with the data-incidents before access (ba), after access (aa), before update (bu), and after update (au)

Figure G-1. Data-Specific Symbol Table Entry Format



sito = string index table offset (may be null)

hco = homonym chain offset (may be null)

type = type information field

- id = identification field (e.g., a statement number)
- segtype = segment type code (e.g., procedure or block)
- dsso = the SPAM dynamic storage stack offset to the "template descriptor" associated with this item

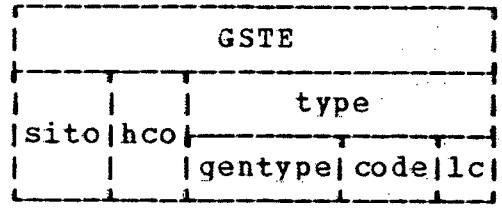
link = linkage information field {This field has the same format as that of a data-specific symbol table entry.}

count = a count of the number of times the associated segment-specific has been entered (e) and exited (x) since system initialization

esto = the event symbol table offset for the deferred action (if any) associated with the segment-incidents before entry (be), after entry (ae), before exit (bx), and after exit (ax)

phrases = an offset into the string index table for the string phrases which constitute the source program code of the associated segment-specific {The order for printing out the code of some node is: ph1, then the code associated with the offspring node, followed by ph2.}

Figure G-2. Segment-Specific Symbol Table Entry Format



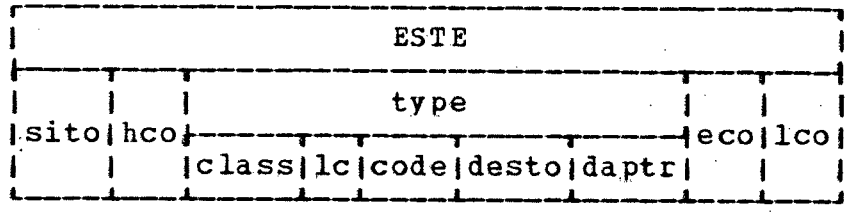
sito = string index table offset

hco = homonym chain offset

type = type information field

- gentype = a flag to indicate whether the generic is a segment- or a data-generic
- code = the segment- or data-generic type code of this generic
- lc = a code indicating for which language this is a generic

Figure G-3. Generic Symbol Table Entry Format



sito = string index table offset (may be null)

hco = homonym chain offset (may be null)

type = type information field

- class = a code to identify the entry as a system default, a language-dependent, or a user-supplied event {The hierarchy of selection is: user-supplied, language-dependent, and system.}
- lc = if type = language-dependent, a code indicating to which language the entry is associated
- code = the error code used by SPAM to identify the event if it is an exception
- desto = a symbol table offset to the deferred action label (if present) associated with the event
- daptr = a pointer to the deferred action (if any) associated with the event

eco = the symbol table offset for the event chain, which is used to chain together all event entries in ascending order of their 'code' field values

lco = the symbol table offset for the label chain, which is used to chain together all event entries having the same deferred action label

Figure G-4. Event Symbol Table Entry Format

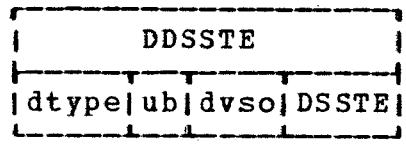
sito = string index table offset

hco = homonym chain offset

lc = a code indicating for which language this is a language-dependent system function {A special code indicates language-independent functions.}

routine_no = a number used to associate the system function with its internal routine equivalent

Figure G-5. System Function Symbol Table Entry Format



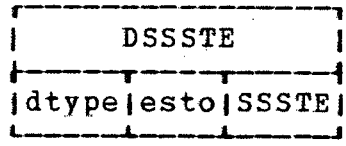
dtype = the type code (viz., integer or specific) of the debugging entity defined by this entry

ub = the upper bound value for debugging entities which are declared as arrays {ub = 0 indicates a simple debugging variable.}

dvso = the debugging value stack offset for the current value (or values, if ub > 0) of the entity defined by this entry {dvso = **null** if the scope of the entity is currently inaccessible. The value of a specific debugging variable is the symbol table offset to the user program entity associated with the specific.}

DSSTE = a field containing the same subfields as that of a data-specific symbol table entry except that the 'dsso' field is unused {The 'datatype' values for Dispel are VARIABLE and PROCEDURE.}

Figure G-6. Debugging Data-Specific Symbol Table Entry Format



dtype = the type code (viz., deferred action label or debugging procedure) of the debugging entity defined by this entry

esto = if dtype = deferred action label, the symbol table offset to an event associated with the label {All deferred actions sharing a label are chained together by the 'lco' field of the event symbol table entry.}

SSSTE = a field containing the same subfields as that of a segment-specific symbol table entry except that the 'dsso' field contains the address of the object code associated with this entry {The 'seg-type' values for Dispel are COMMAND and ACTION.}

Figure G-7. Debugging Segment-Specific Symbol Table Entry Format

Appendix H. Example RAIDE Symbol Table Entries

This appendix presents several examples of RAIDE symbol table entries. The first example demonstrates how user program data-specifics are maintained by RAIDE. An outline of a program in an Algol-like language is contained in Figure H-1. Only variable and procedure declarations are shown since executable statements are unimportant here. User program entities (e.g., variables and procedures) must be accessible in two ways: by their names and by their (level,order) pairs. The former is necessary for interfacing between the user and SPAM (i.e., for mapping between a name in a Dispel utterance and its actual machine-level value). The latter is necessary for interfacing between SPAM and RAIDE (i.e., when SPAM signals an error, RAIDE must determine from the machine state what source-level entities are involved).

Figure H-2 demonstrates how the variables and procedures of Figure H-1 are accessed by their identifier names. Each identifier is uniquely hashed to an offset into the identifier hash table. All symbol table entries with identical identifiers are chained thru the homonym chain. In the figure, each symbol table entry is represented by a node showing only three of its fields: a string index table offset (diagrammed for simplicity as a character string reference), a (level,order) pair, and the homonym chain pointer.

Figure H-3 demonstrates how user identifiers are accessed by their (level,order) pairs. Each symbol table entry is represented by a node showing only three of its fields: a string index table offset (diagrammed for simplicity as a character string reference), an offspring symbol table offset, and a sibling symbol table offset. Given the symbol table offset of 'main' and the (level,order) pairs for accessing some desired identifier, its symbol table entry is located by chaining thru the offspring and sibling symbol table entry fields.

The first example presented in this appendix deals only with user program symbol table entries. It must be remembered that the symbol table also contains entries for generics, events, system functions, and debugging entities (cf., Appendix G). All of these types of entries are connected along the homonym chain for identical identifiers. To resolve ambiguities (e.g., the user can define a procedure called 'line' which is syntactically equivalent to the system function 'LINE'), a hierarchy of access is defined as follows:

1. all user identifiers in order of their static declarations,
2. a generic,
3. an event,
4. a system function, and
5. all debugging variables and procedures.

Thus, if a user has a procedure called 'line', it supersedes the system function 'LINE'. Similarly, the user cannot define a

debugging procedure called 'line' since the system function 'LINE' will supersede it.

```
lexic    lexic
level    order     source code

  0        0       main:
                   begin
  1        0          proc foo
  2        0             (int bell) ;
                      begin
  2        1             real bar ;
                         •
                         •
                      end ;

  1        1          string bell ;

  1        2          proc barbell ;
                      begin
  2        0             bool bar ;
                         •
                         •
                      end ;
                      •
                      •
                   end
```

Figure H-1. Program for Demonstrating the RAIDE Symbol Table

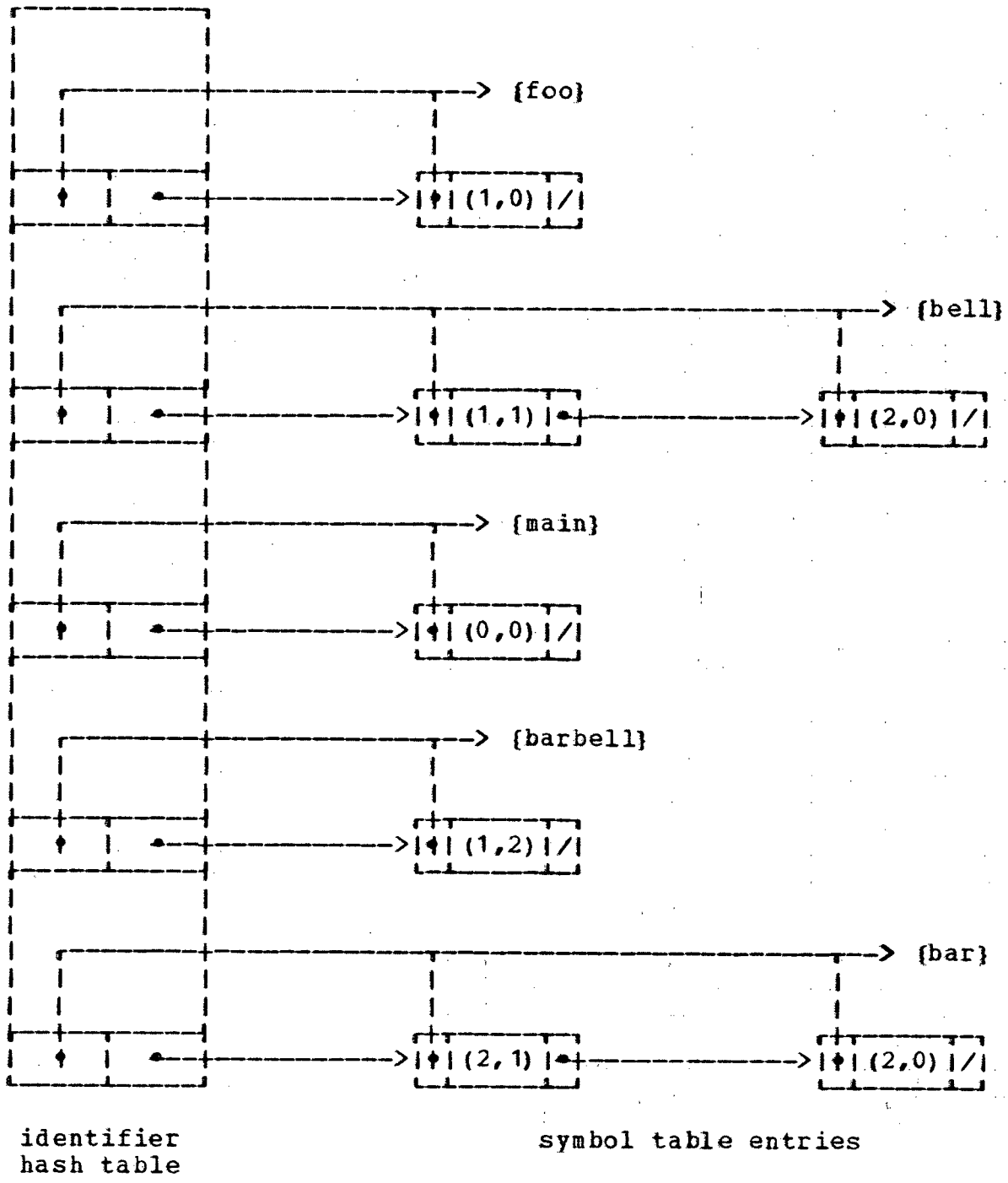
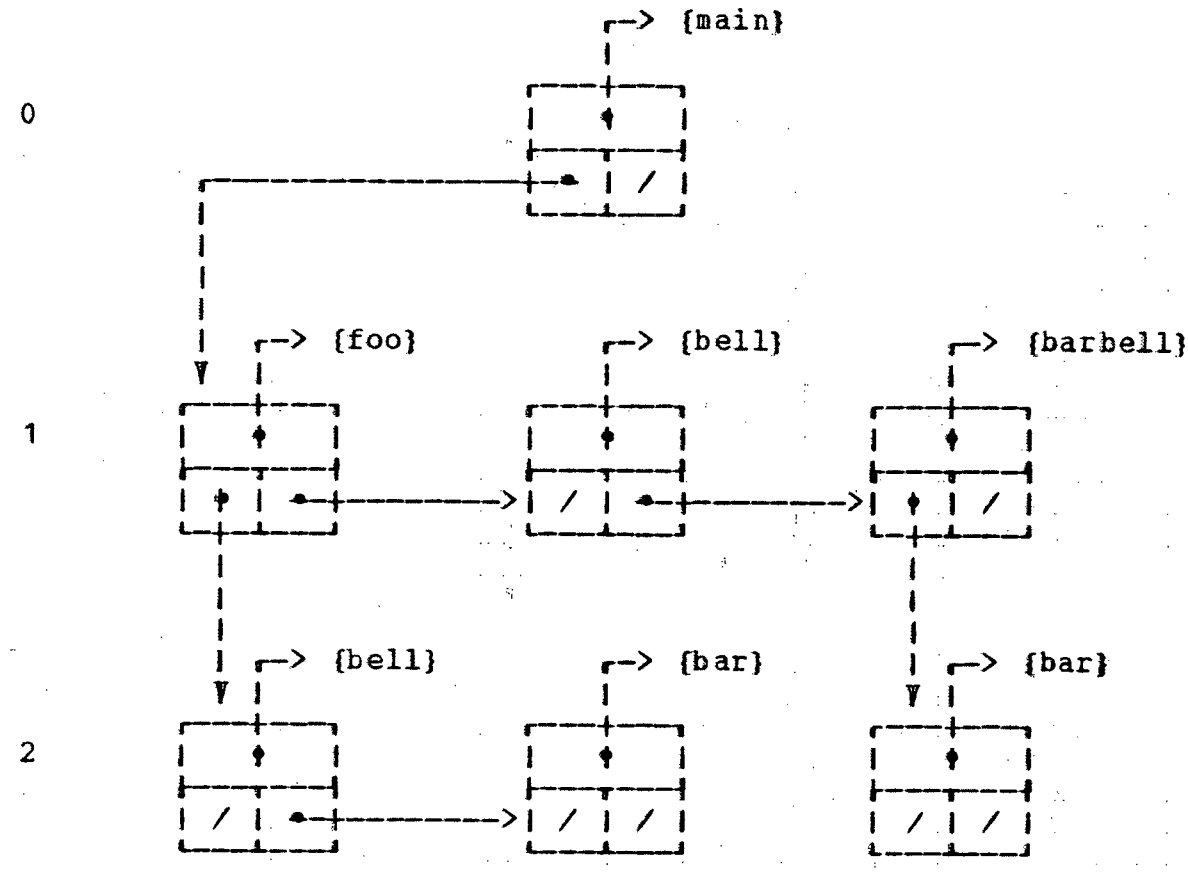


Figure H-2. Accessing a Symbol Table Entry by Identifier



Access (level,order) sequencies:

```
main          (0,0)
   foo        (0,0)   (1,0)
      bell    (0,0)   (1,0)   (2,0)
      bar     (0,0)   (1,0)   (2,1)
   bell       (0,0)   (1,1)
   barbell    (0,0)   (1,2)
      bar     (0,0)   (1,2)   (2,0)
```

Figure H-3. Accessing a Symbol Table Entry by (level,order)

In addition to user program data-specifics, the RAIDE symbol table also contains the description of the static structure of a program. This is done by maintaining symbol table entries for each of the segment-specifics of the source program. Figure H-4 demonstrates the entries for the Towers of Hanoi program of Figure F-1. In the figure, each symbol table entry is represented by a node showing seven of its fields: the identification field (which is either a number or a string index table offset for labeled segment-specifics such as procedures), a segment type code, the SPAM dynamic storage stack offset to the segment's template descriptor, a sibling symbol table offset, an offspring symbol table offset, and two string index table offsets (diagrammed for simplicity as character string references) for the source code phrases.

In general, the 'sito' and 'hco' fields of segment-specific symbol table entries are not used since segments are generally not accessed by name. The major exceptions are procedures. Since each procedure has both a data-specific entry (which links together all data values accessible to the procedure) and a segment-specific entry (which links together all segments subordinate to the procedure), the two entries are connected along the homonym chain. Thus, given a procedure name, it is possible to determine both the variable scope structure and the static program structure of that procedure.

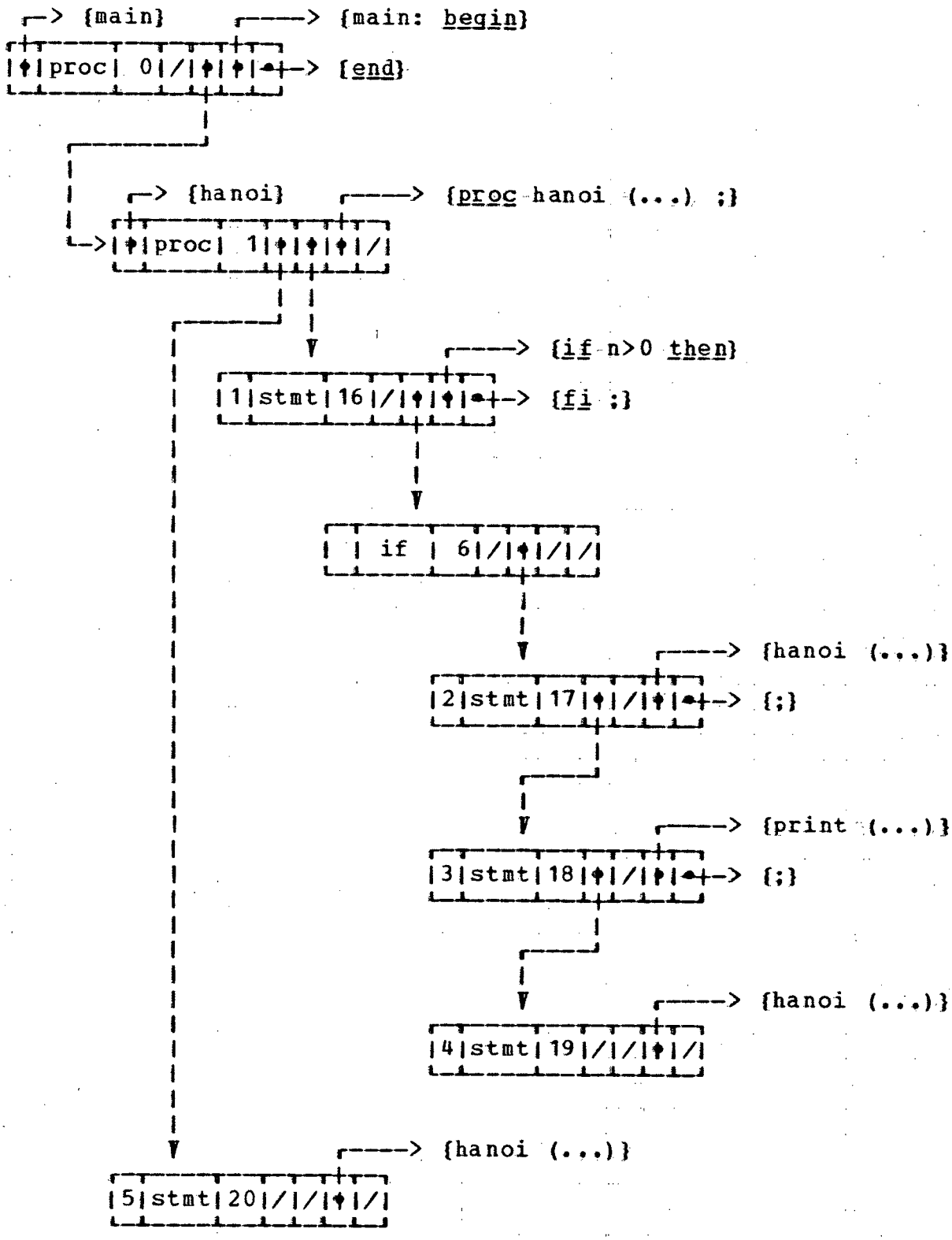


Figure H-4. Example Segment-Specific Symbol Table Entries

Appendix I. An Example RAIDE Program Specification

This appendix presents the RAIDE program specifications for the sample program of Figure F-1, Towers of Hanoi. Figure I-1 outlines the RAIDE specifications of the program's data-specifics and Figure I-2 outlines the same for its segment-specifics. Refer to Appendix G for descriptions of the fields of the data- and segment-specific symbol table entries; only the important fields are mentioned here. See Figure H-4 for a graphic representation of the information contained in Figure I-2. There are two further components of a program specification not presented here: the initial bounds name table and type name table entries. The bounds name table is empty for this example and the type name table simply contains the identifier names of the primitive types, since there are no user-defined types in this example.

offset	(level,order)	datatype	dsso	(sib,off)	identifier
1	0,0	constant	0	0,2	main
2	1,0	constant	1	0,3	hanoi
3	2,0	parameter	12	4,0	n
4	2,1	parameter	13	5,0	me
5	2,2	parameter	14	6,0	de
6	2,3	parameter	15	0,0	ma

Figure I-1. Towers of Hanoi Data Specifications

offset	id	segtype	dsso	(sib,off)	phrases
1	main	proc	0	0,2	{main: begin,end}
2	hanoi	proc	1	8,3	{proc ... ;,}
3	1	stmt	16	0,4	{if n>0 then,fi ;}
4		if-stmt	6	0,5	{,}
5	2	stmt	17	6,0	{hanoi (...),;}
6	3	stmt	18	7,0	{print (...),;}
7	4	stmt	19	0,0	{hanoi (...),}
8	5	stmt	20	0,0	{hanoi (...),}

Figure I-2. Towers of Hanoi Segment Specifications

Bibliographic Reference Index

General Index

This index uses a collating sequence in which special symbols precede alphabetic characters and upper-case letters precede lower-case letters.

<action> 35
<break-action> 36
<call-action> 36
<cancel-action> 37
<cancel-list> 37
<command> 32
<command-list> 36
<compound-action> 36
<data-incident> 34
<declaration> 31
<declaration-list> 32
<definition> 32
<display-action> 37
<exception-list> 32
<execute-action> 38
<explanation> 31
<expression-list> 34
<for-action> 39
<generic-incident> 34
<id-list> 31
<if-action> 39
<input-action> 39
<inquiry> 31
<integer-declaration> 31
<quit-action> 40
<reference-action> 40
<restore-action> 40
<save-action> 40
<segment-incident> 34
<segment-qualifier> 34
<set-action> 41
<skip-action> 41
<specific> 34
<specific-declaration> 31
<specific-incident> 34
<specific-incident-list> 32
<subscripted-variable> 34
<system-action> 42
<unqualified-variable> 34
<utterance> 30
<variable> 34
<what-list> 37
<when> 32
<when-clause> 32
<while-action> 42