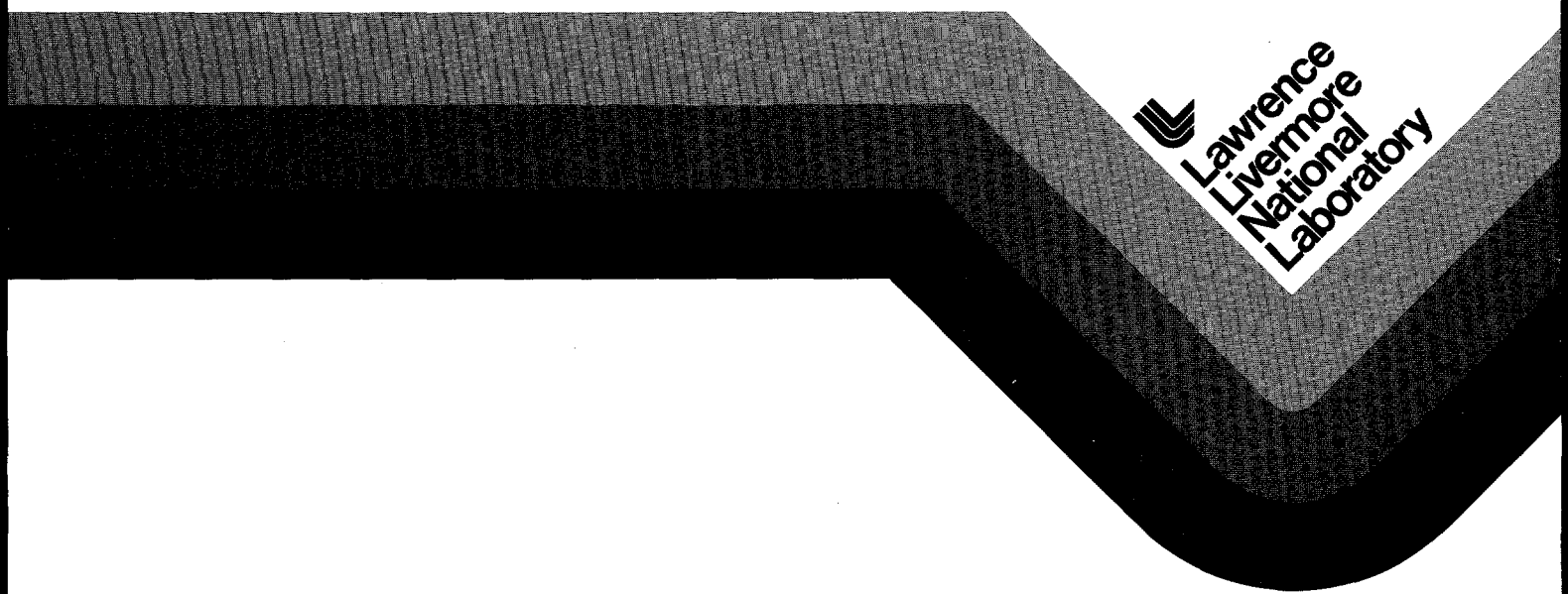
UCRL-LR-120452

An Object-Oriented Extension for DeBugging the Virtual Machine

Robert G. Pizzi, Jr.

December 1994



Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

CONF-

UCRL-LR-120452
Distribution Category UC-705

An Object-Oriented Extension for DeBugging the Virtual Machine

Robert G. Pizzi, Jr.
University of California, Davis

Master of Science Thesis

Manuscript date: December 1994

LAWRENCE LIVERMORE NATIONAL LABORATORY
University of California • Livermore, California • 94551

An Object-Oriented Extension For Debugging The Virtual Machine

By

Robert G. Pizzi, Jr.
BS, University of California at Irvine, 1992

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

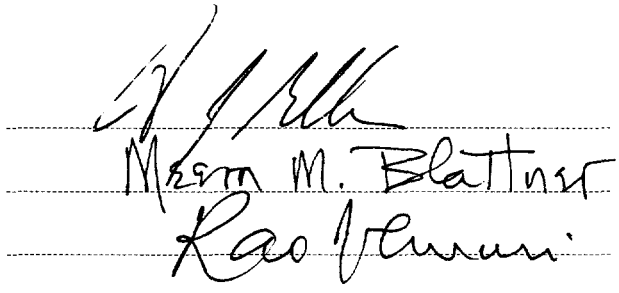
OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:



Committee in Charge

December 1994

Robert G. Pizzi, Jr.
December 1994
Computer Science

An Object-Oriented Extension For Debugging The Virtual Machine

Abstract

A computer is nothing more then a virtual machine programmed by source code to perform a task. The program's source code expresses abstract constructs which are compiled into some lower level target language. When a virtual machine breaks, it can be very difficult to debug because typical debuggers provide only low-level target implementation information to the software engineer. We believe that the debugging task can be simplified by introducing aspects of the abstract design and data into the source code. We introduce OODIE, an object-oriented extension to programming languages that allows programmers to specify a virtual environment by describing the meaning of the design and data of a virtual machine. This specification is translated into symbolic information such that an augmented debugger can present engineers with a programmable debugging environment specifically tailored for the virtual machine that is to be debugged.

Acknowledgments

Dr. Patrick Miller, my thesis advisor, who is the source of a plethora of great ideas, but he says that this system "...is the best idea I've ever had." Also, Pat, thanks for the many diverting office conversations which included many "hard-sell" speeches to motivate me on my quest.

Dr. Meera Blattner and Dr. Rao Vemuri for agreeing to be on my committee as chairperson and signatory, respectively.

Grace M. Hopper, for her revolutionary contributions to the field of computer science and for discovering the first "bug" in a computer (literally), thus making this thesis possible.

Contents

List of Figures

Chapter 1

Introduction

When software engineers design and implement computer programs, they are in fact creating virtual machines. In essence, the computer program is the expression and realization of the abstract ideas of the virtual machine. This expression is a concrete, rigorous form that allows automatic translation into an executable program. Unfortunately, as abstractions become more concrete, vital abstract design information is filtered away. In the final form of the virtual machine, the executable image encodes little of the original design information making it very difficult to debug. Typically, a debugger presents information in a raw form (i.e. integers, characters, and pointers) and not in the higher level constructs described by the program design. For example, a value that the programmer thinks about as a linked list is represented as a pointer or single node and not as a coherent structure. What a programmer considers a bitmap, typical debuggers present only as a list of integers. The lack of abstract information during debugging is particularly problematic for high level translators. For instance, `yacc` translates into a low-level programming language, typically C. In that case, the problem is that the user programmed at a high level of abstraction (e.g. LALR productions) but is forced to debug the low-level implementation rather then the productions. We believe the solution to these problems is to carry high level design information down into the debugger, thus allowing the programmer to debug the abstractions as well as the implementation.

1.1 An Abstract View of Software Engineering

Software engineering defines a software development process containing three phases: definition, development, maintenance.[12] From these general phases, specific process models such as the waterfall or spiral model[2] are derived. Each phase of the process model generates information that is carried into the next phase for further refinement. The type of information is essentially specification documents (i.e. requirements, testing plan, design) written in prose. At some point in the lifecycle, the specifications are converted to code. Unfortunately, it is not possible to place the abstract ideas of the documents into the source code. The abstract ideas must be filtered and transformed into a rigorous form that can be processed by a compiler. This form does not allows for the adequate expression of abstract ideas and therefore those ideas are not explicitly present in the code.

In Figure 1.1, we illustrate our abstract view of the software engineering process and focus on the steps that create an executable program. The *Analysis layer* at the top of the cone shows a software engineer using the problem description to produce a high level design specification. This specification is used in the *Implementation layer* to generate source code. The *Compiler layer* generates object files from the source code and the *Linker layer* creates an executable program image. Each stage of the cone distills the data from the previous stage. The software engineer at the bottom of the cone receives information from the *Debugger layer* about the program in the form of lines of source code and memory locations. Yet, the program image itself is devoid of the abstraction of the original virtual machine. When using a debugger, the necessary abstract information can only be found within the prose of the design documents.

1.2 Debugging the Virtual Machine

During debugging, the software engineer is limited by how much abstract information about the virtual machine is output from the debugger. For example, Figure 1.2[1] lists

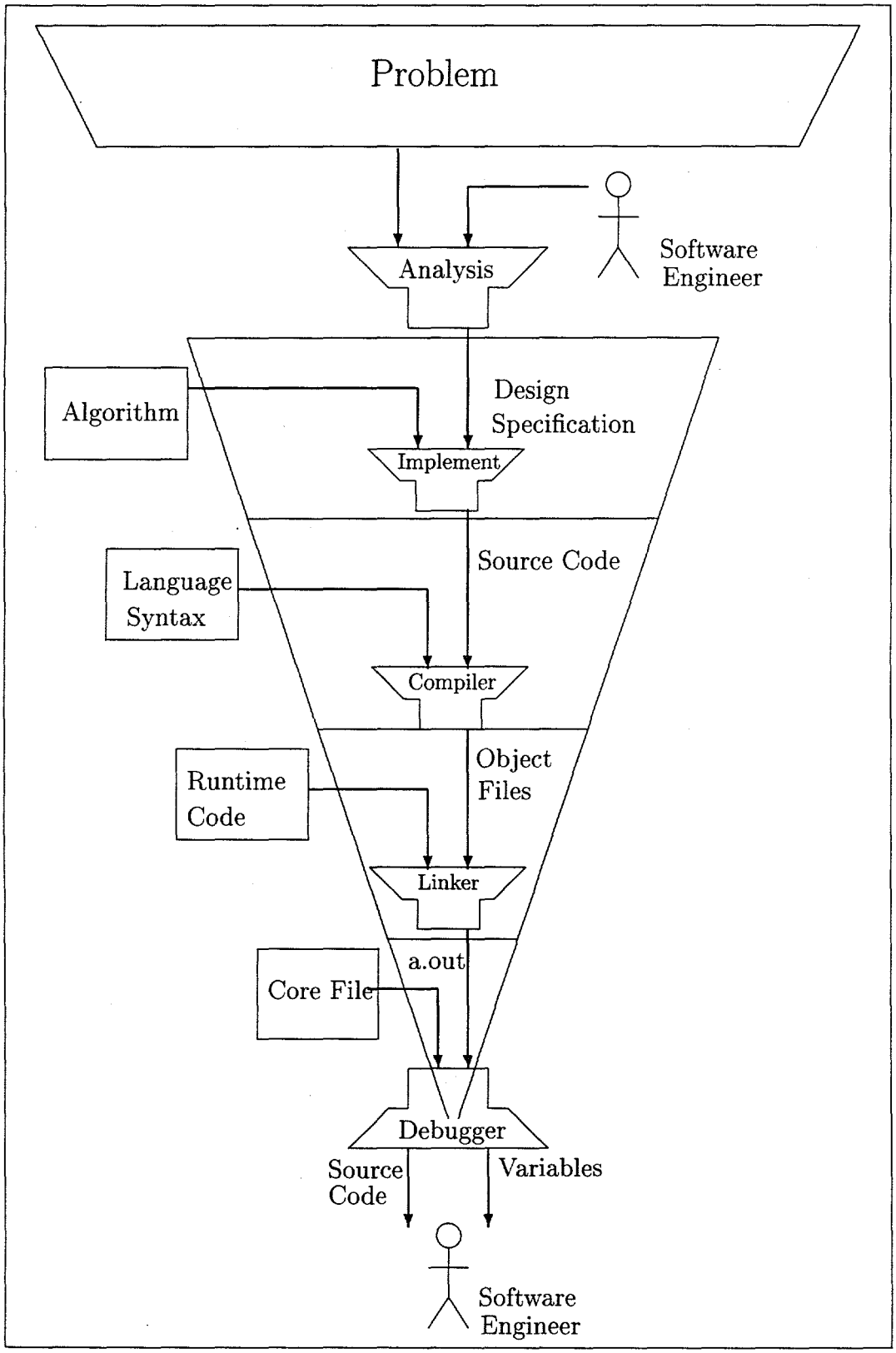


Figure 1.1: "Cone of Debugging Knowledge"

Requirements Document

3.1.1.3 Processing

The filename inputs shall be sequenced internally in any fashion deemed implementationally efficient. This may involve some type of sorting or table mechanism. Enforcement of the filename modification policy of write one, read many shall be handled.

Design Document

3.1.1 Design

Filename management shall be implemented as a linked list. Each node of the linked list shall contain a pointer to dynamically allocated storage for the contents of the filename. The nodes in the linked list shall always be added to the end of the list to preserve ordering requirement.

Source Code

```
void FM_add(FManager fm, char *fname)
{
 FManagerNode node=(FManagerNode)malloc(sizeof(struct _FManagerNode));

 node->fname =  strcpy(malloc(strlen(fname)), fname);
 node->next = NULL;

 if (fm->count) {
    fm->tail = fm->tail->next = node;
 } else {
    fm->head = fm->tail = node;
 }
 fm->count++;
}
```

Figure 1.2: Excerpts From Sample Process Documents

```
Breakpoint 1, main () at main.c:120
120 printf("last name:  %s", FM_get(fm, argc-1));
(gdb) print fm
$1 = (struct _FManager *) 0x100116c4

(gdb) print *fm
$2 = {count = 3, head = 0x100116d4, tail = 0x100116d4}

(gdb)print *fm→head→next
$3 = {fname = 0x10011718 "bob.c", next = 0x10011724}

(gdb)
```

Figure 1.3: Debug Transcript

excerpts from the sample documents and code produced by a software engineering process. The requirement and design specifications specify the use of a linked list. However, when examining the implementation excerpt, the linked list abstraction is muddied by details of pointers and memory allocations. An experienced software engineer will recognize these details as properties of a linked list, but it is the programmer's responsibility to make the connection between the design characteristics and the implementation details.

The burden placed upon the practitioner is problematic when you consider issues of "programming in the large". Some of these issues include: quantity of code (50,000+ lines); multiple engineers; multiple vendors; long lifecycle (5+ years); legacy codes. Mistakes during debugging can occur if the engineer makes incorrect assumptions about the current design of a program. This is a result of a lack of design information explicit within the source code. For example, Figure 1.3[2] shows a typical transcript from a debugging session of the code as found in Section A.3. The engineer is attempting to view the linked list. However, the abstraction of a linked list is lost and the debugger can only present the data as specified by the implementation. Thus,

[1]More complete samples can be found in Appendix A.
[2]The version of gdb used is: 4.12-mips-sgi-irix4.0.5

the engineer can only view the data as it is implemented and not at the level of the abstraction, a linked list.

In our view, current debuggers perceive a program in a way that is similar to a person's perception of a novel. For example, a novel maybe decomposed into parts: novel; chapters; paragraphs; sentences; words; letters. The words are formed by grouping letters, sentences are formed by grouping words, and so on, until the novel is completed. If one could only read the novel one letter at a time, it would be nearly impossible to understand the *meaning* of the novel. Unfortunately, current debuggers manipulate the equivalent of only the letters and words. Debugging is currently the equivalent of trying to understand a novel by reading individual sentences without any context about the story. The abstract structure and meaning of both the novel and the program are lost when being examined at such a low level. With only variables, expressions and source code, the software engineer at the bottom of the cone (Figure 1.1) must attempt to reconstruct the abstract concepts of the program or find them within the design documentations. Since all the important design information is external to the source code, the software engineer may make assumptions about the program structure. These assumptions may appear correct, but on further review are incorrect and thus cause additional software failures later in the software's lifecycle.

1.3 Proposed Solution

The remainder of this thesis will concentrate on our solution to the problem of debugging virtual machines. Our system, Object-Oriented Debugging Interface Extensions, **OODIE** provides a mechanism to specify "virtual debugging environments". These environments allow a programmer to specify a virtual machine's design and abstractions directly into the source code. An augmented debugger can then use the additional information to present higher-level abstract information about the original machine.

We first discuss the foundations of the OODIE debugging environment. Following

that, we present the design and implementation of a prototype OODIE system. We then present some example programs that use the OODIE extensions. Finally, we discuss our conclusions and suggest future lines of research within OODIE.

Chapter 2

Problem and Solution Outline

Generally speaking, debugging is a very time consuming and difficult task forming a "love-hate" relationship with programming. Until programmers can take a perfect specification and produce perfect code using a perfect compiler under a perfect operating system, debugging will always be a part of programming. Currently, debugging is difficult because of its low-level approach to debugging *all* programs. Most programs are eventually compiled down into raw machine instructions, but it is unlikely that an engineer wants to debug the program's execution at that level. Ideally, a programmer would like to debug a program at a higher, more abstract level, which increases comprehension about the overall function of the program. The next generation of debuggers must allow the programmer to debug at whatever level of detail is required to resolve the bug, from raw assembler, up to source code, and up to program design.

This chapter will outline our view of the current state of debugging and the "tricks of the trade" used to work around the deficiencies of a typical debugger. Finally, we itemize some requirements for the next generation of debugging tools.

2.1 Motivation

Debugging occurs as a consequence of a software failure. This difficult task is dreaded by most software engineers because of the time and effort required to diagnosis and correct the software failure. As noted in [13] and [17], human psychological conditions affect one's ability to debug any given program. Fortunately, debugging tools are available to aid the software engineer in overcoming the numerous barriers in solving a software failure.

Typical tools available to an engineer include: test data generators, execution flow summarizers, file comparators, simulators, symbolic dumps, trace packages, source level debuggers.[18] Despite theses tools, the process of debugging is still a very low level activity. At a fundamental level, debugging consists of "stop, print, and go". This assessment is obvious upon further examination of the information content produced by the tools. Below, we will discuss some standard tools and what types of information are available to the user.

Symbolic Debugger One standard Unix debugging tool is `adb` [1] and is best used with a `core` file. A core file is a snapshot of the process's memory at the time the program stopped. The available information consists of program symbols, memory, machine registers, and assembly instructions. Essentially, one is able to examine the data that resulted in the software failure. Access to data is made through the core file where it is represented in hexadecimal or octal format. The engineer is exposed to all of the machine details and must access the program using the paradigm of core memory.

Trace A trace tool provides information about program execution order. However, to do existing machine performance, the number of executed instructions is very large resulting in a large trace. This overload of information will not be useful as the offending instructions can not be distinguished from correct instructions.

Source-Level Debugger A source-level debugger is a synthesis of a trace

log and a symbolic debugger augmented with program source information. It provides a dynamic between the user and the program by providing step-wise control of the execution. The only significant advantage offered by this type of debugger is to provide automated methods to gain information about the program that one can otherwise obtain using separate tools. Essentially, a source-level debugger simply provides a glossy interface to a symbolic dump.

These represent the fundamental tools that the software engineer has to correct a software failure. Each of these tools provide low-level detail about the machine and the program state. The engineer is exposed to all of the detail at once and it is difficult to filter out which information is important and which is just noise. When programs were small, this problem was not unsurmountable. Today, you can not generate an executable program under 100 kilobytes or a core image under 500 kilobytes. As machines get larger memories and more complicated programming paradigms are used, program sizes can only increase. Given existing tools which expose every detail of the machine, information overload occurs during debugging. As a result, the useful information needed to solve the software failure is indistinguishable within the trace output.

2.2 Common Debugging Practices

Debugging is not as impossible as the previous section might suggest. Debugging a high-level program requires that the *debugging paradigm match the programming paradigm.* Software engineers have developed programming styles that maximize the information content when using a debugger and learn which styles provide the best information at debug time. Essentially, the engineer must generate useful program information that does not get filtered out by the compilation process. In the next sections, we will discuss a few common debugging practices that help the software engineer get more out of their debugger.

2.2.1 Documentation

Documentation is one of the products in a software engineering software process. Considerable effort is invested into providing accurate and detailed documentation for a project. The minimum required documentation for a fully matured software process includes: a project plan, project estimates, high-level requirements, detailed requirements, high-level analysis, detailed analysis, detailed design, a test plan, a maintenance plan, and a quality assurance plan. The quality of the information within each of these documents varies depending on the project and personnel composing the data. It is beyond the scope of this thesis to assess the quality of documents produced by a software process, however, a few important observations are relevant. The presentation of the vast quantities of data is key to the understanding of that data. But, the current form of the data lacks a linkage between the other process documents and the source code. Therefore, the software engineer must refer to information relying on the table of contents and index to determine where information is located. The software engineer must also decide which documents will be relevant during debugging. To access the process documents, the software engineer must refer back to the hard copy documents to find the relevant information related to the design and implementation. This requires the engineer to read and comprehend the appropriate documentation "off-line" before starting to debug. It is clear that the current form and usage of the process documents provides no integration with the source code and debugger.

It is possible to increase the integration between source code and documentation. Consider the `WEB` system[7] which is essentially, a combination of a program's design and implementation into one document. The design portion is tailored to act as a running commentary on the implementation of a given part of the code. Cross-references between the design and implementation can be made within other sections of a `WEB` document. For an example of a `WEB` document, see Appendix B. When a `WEB` document is woven[1] into source code, all of the descriptions found in the document

[1]The `WEB` term to generate source code from the document.

are lost. Thus, debugging WEB code is not any easier then debugging code that is not part of a WEB document. In fact, the process of weaving source code may introduce extra code not explicitly specified in the document to provide certain implementation details. These details are exposed by the debugger without any documentation to help the software engineer understand the purpose or origin of those codes, making it much harder to debug. The WEB system takes a step in the right direction by integrating code with design, but important abstract information is lost to the debugger.

Documentation is generally considered useful by the software engineering community, but it is unclear how to use it effectively during debugging. The common knowledge that software engineers despise documentation leads us to conclude that the documentation itself is not useful because it lacks a tight integration with the actual work done by software engineers, specifically, coding and debugging.

2.2.2 Coding Practices

A seasoned software engineer will have most likely experienced difficulties in debugging a particular piece of software. As a result, engineers have changed their coding practices to accommodate their environment. The purpose of these changes is to provide additional abstract information about the design and data of a program such that the information is not filtered by the compilation process and accessible using a typical debugger. In essence, the engineer is using the constructs of the given programming language to express the high-level details of a program. The difficulty is that a typical programming language[2] is not designed for high level abstractions, but rather for higher-level control over implementation decisions. As a result, developing a debugging harness[3] is an *ad hoc* process that is not portable between *different engineers.* Specifically, when multiple engineers are cooperating on a large software project, each engineer may have their own "system" of debugging harnesses. These harnesses are routinely inserted then deleted to improve program efficiency.

[2]Traditional 3GL and 4GL languages, Pascal, C, C++, Ada, FORTRAN, Assembler, etc.
[3]Any code used strictly for debugging purposes

The absence of a standardize debugging methodology prevents other engineers from exploiting the debugging harnesses introduced by another simply because they are ignorant of the fact that harnesses exist and unaware of which data instances are applicable for use with a particular harness.

The discipline of software programming may be more of a trade skill then it is a rigorous engineering exercise. Analyzing different programming styles is beyond the scope of this thesis. However, there some common debugging techniques used to augment a poor debugging environment. The following list describes a few of those techniques.

Print Statements (I/O) This is the most commonly used of all debugging techniques as print statements benefit from their ease of use and customizable data reporting capabilities. However, print statements do have a drawback. Using the statements require the engineer to enter a edit-compile-debug cycle in which the engineer discovers where to place the print statements, what data to print, and what format to print the data. A common solution is to output as much data as possible when first creating the statements. Since the output will necessarily contain a lot of data the engineer must manually filter out what is unimportant.

Debug routines This technique uses one or more functions to perform a task during program execution that requires more advanced processing than a single print statement can provide. An example would be a routine that prints the contents of a linked list by traversing the links. Some debuggers allow the software engineer to manually invoke the debug routine from the debugger command line. This eliminates the need for populating the source code with calls to the debug routines. Essentially, debug routines are a grouping of print statements, therefore the routines have the same drawbacks as do individual print statements. Also, the presence of the routines may not be known to other engineers debugging the code and the effort used to implement the debugging routines

is wasted and may be duplicated if the routines are prematurely removed from the source code.

Debug code This technique employs the use of code statements that alter the execution of the code for the purpose of identifying a bug. This type of code manifests itself as some type of conditional test[4] that guards a print statement. This technique is most useful in filtering the amount of data obtained from print statements. It may be difficult to insert code at the correct locations and if the inserted code does not test the right condition or output the data in a useful form, the data will be meaningless.

Modularization This is a technique taught to engineers learning a top-down approach to programming. Modularization compartmentalizes the program into logical blocks in which each ideally performs a unique and cohesive task. While modularization provides a structural decomposition of a program that allows easier comprehension by the engineer, the implementation performance is degraded by routine call overhead[5]. It is possible to over-modularize a program to the point of unacceptable performance. The situation is more acute when dealing with time-critical real-time programming. As a result, modularization must be optimized by removing the routine call over head. This has the effect of breaking down the modularization hierarchy and flattening the abstract structure of a program's implementation. This process is more commonly called **inlining**[6]. As a reflection of the demand for code inlining, some programming languages use a pragma[7] to specify when inlining should be used and the C++ language specifies inlining within its language definition.[19] When a programming language does not support inlining, engineers must manually inline code

[4]The test could be in an IF statement, or `#ifdef __DEBUG`

[5]Register save/restore, indirect jump, stack management.

[6]A compiler optimization that eliminates the function call overhead by emitting the code for the called function at the point in the code where the function is called.

[7]Meaningful comment used as a directive to the compiler.

either by replicating code or using a macro language. As a result, structural information expressed by inlined routines is lost during debugging as the debugger examines the implementation *after* all of the inlining is complete.

The key to debugging is the ability to control the execution of a program while obtaining useful information about data of the program. Engineers can use the above techniques to gain control and information at the cost of changing the design and structure of the program. All of the debug code must be removed for the production version of the software. Removing this code may cause or eliminate side-effects which are significant to the correct execution of the software. This causes a condition affectionately known as a *Heisenbug.*[8][9][4] Ultimately, the demands of greater performance usually outweigh the benefits of debug-time information therefore debugging harnesses are constantly being created and removed as the maintenance effort continues. This represents a duplication of work and increases the cost of maintaining the software.

2.3 Debugging Requirements

Debugging requirements specify the functionality of the interface between the executable program and the debugger. A listing of these requirements can be found in [15] which describes most of the debugging features currently found in a typical debugger, such source-level debugging, breakpoints, etc. These requirements, however, only address low-level aspects of the debugging environment such as requirements to be able to access operating system calls, symbolic references, and core memory. These requirements define a debugging paradigm that characterizes all programs in terms of low-level, implementation dependent details. Examining a program at such a microscopic level of detail may, at times, be required to track down errors in low-level components of a program and system, such as device drivers, interrupt handlers, kernel code, etc. Those types of programs are only a small fraction of the software

[8] In other words, a bug that disappears when using a debugger.

base. Therefore, engineers must be able to debug at a higher level of abstraction to comprehend the abstractions within a program in a form other than core memory.

Clearly, programming language development has outstripped advances in debugging technique and tools. Programming languages can now describe abstract data types and data inheritance hierarchies[16], but typical debuggers can only manipulate them at a low-level, specifically, core memory. The use of abstraction is very useful in comprehension of fine-grained details.[20]. Programming languages provide hierarchical abstractions to separate the implementation details from the higher-level language constructs. However, a typical debugger forces *all* programs to use a debugging paradigm which does not use any hierarchical abstractions. It is our view that eliminating the abstract hierarchy increases the difficulty in debugging a program.

The fundamental premise of this thesis is that compilation systems must preserve the program's abstract information for later use in a debugger. The following list describes the requirements for such a system.

Unified Debugging Environment It is clear that when a program and a debugger are tightly coupled, the debugger becomes a more useful tool. For example, enhanced symbol table and profiling data may be used within a debugger to automatically generate graphical displays of the program variables. In addition, coupling a program's abstract design and data with a debugger will allow consistent manipulations of those abstractions throughout the debugging process. We believe that providing a unified debugging environment will simplify the maintenance of software as the environment can be customized for each virtual machine. All engineers working on a particular software product will be more effective and productive as they all will have a unified set of tools to aid the debugging process.

Multi-paradigm Debugging Currently, multi-lingual debuggers can debug multiple programming languages. In other words, the debugger recognizes many language syntax forms and can express the program in terms of the appropriate language definition. We would like a system that can express the programming

paradigm down to the level of the debugger forming a consistent debugging environment that matches the programming paradigm. For example, imperative programs describe a flat, in core memory hierarchy while distributed programs use a hierarchy of remote memory across a network. A debugger needs to be able to understand the meaning and implications of the general and specific programming paradigm that a given program uses and present the appropriate view during debugging.

Mixed-paradigm Debugging This is a logical consequence of multi-paradigm debugging. A single program may be compiled from a mix of programming languages. A multi-paradigm debugger is required to recognize the context of the program and switch to the appropriate source programming language paradigm as the program executes code generated by different source programming languages.

Meaningful Source Commentary Programmers write comments to provide additional information about the program. Comments are also read by the engineer when viewing the source code. If comments were important during the design and implementation of a program, it is logical that comments are equally important during debugging. But, the static nature of comments and the lack of an automated tool to process comments[9] limit their usefulness. If an engineer was able to use comments to program an augmented debugger with the meaning of the elements of a program's abstract design and data, the debugger is more useful tool and source code commentary becomes dynamic. This dynamic element provides additional incentive to thoroughly document code as there is a measurable benefit to their usage. The benefit is a tighter coupling between the program and the debugger.

Data Abstraction Abstraction is a concept that is often used in computer science. It is used to comprehend complicated concepts by using simple ideas.[20]

[9]For instance, cross-referencing comments to a particular data structure.

Object-oriented languages have shown the benefits of defining high-level abstraction within the language definition.[14] We would like a system that can provide the same data abstraction qualities used during design time to be available during debug-time. Allowing the engineer access to the abstractions that form the basis of a particular data structure or program is the key to understanding that program and the key to properly maintaining that program throughout its lifecycle.

Clearly, this list is not a complete description of all debugger features, but it does provide the conceptual foundation for the development of a new debugging system. The system takes the approach of integrating design elements into the debugging process by defining a new debugging paradigm that explicitly combines abstraction with implementation.

Chapter 3

Design and Implementation

This chapter presents the design and implementation of our prototype OODIE environment. We believe that the expression of the abstract objects within a typical program are comprehended by the engineer in an object-oriented fashion in which abstraction and hierarchy are the keys to understanding. Therefore, it seemed obvious that our system be hierarchical and based upon object-oriented principles. This approach requires a debugger programming language to provide the expressions of the abstract data and design of a program and provides a tighter coupling between the compiler and debugger. The coupling of the compiler and debugger for the abstract elements of a program provides for a unified debugging paradigm.

We begin by discussing the object-oriented foundation of our system followed by the requirements for our debugging environment. Then, we outline the features of our prototype and the debugger programming language. We conclude by examining the overall design and implementation details of the compiler and debugger.

3.1 Object-Oriented Foundation

The use of the debugging techniques described in Section 2.2 creates a clean hierarchy of abstract data which provides orthogonality throughout the implementation of the code. In most cases, debug routines are used to output the meaning of abstract

data types while using a debugger. The debug routines are essentially functional methods attached to abstract data such that the routines describe the data in a form consistent with the virtual machine. The engineer has implicitly defined an object class with methods that act upon the data. In this case, the routines are specifically for debugging the abstract data type.

The concept of attaching methods to abstract data objects is one found in object-oriented programming environments. Analysis of the requirements drawn up in Section 2.3 leads us to believe that our unified debugging environment should be built on an object-oriented philosophy. This conclusion is made possible by recognizing certain aspects of the actual debugging process used by software engineers. This process is based on some of the fundamental principles of object-oriented programming and are described below.

Inheritance This is the guiding design principle for objects within a object-oriented programming language. The purpose of inheritance is to form a hierarchy of similar objects which share functionality and data. The hierarchy defines classes of objects that refine the meaning of base class objects. The refined classes (subclasses) add additional meaning by overriding the existing meaning of functions and providing new functionality.

Encapsulation Object-oriented programming languages let programmers collect similar data and functionality into one object allowing the design of cohesive objects within the data hierarchy. In an object-oriented programming language, encapsulation is supported by language constructs. Yet, within a language which does not directly support such concepts, the engineer may employ such techniques as a matter of programming style. Our system requires encapsulation of typical debugging techniques and the data instances within a program. By grouping data in this fashion, our debugging environment will have a uniform definition of the abstractions within a program. With a uniform environment, the expression of the abstract meaning of data will be portable among different engineers.

Inheritance and encapsulation are only two of the many properties one can use to define an object-oriented system. We strongly believe these properties convey the *meaning* of data. Given a typical debugger, the typical engineer must comprehend the meaning manually, by either a deep understanding of the code or debugging techniques to maximize information presented in the debugger. Our debugging environment provides mechanisms that assist the engineer and provide consistent expression of the abstractions such that all engineers working on the software will comprehend the data using the same abstraction.

3.2 Abstract Debugging Environment

An object-oriented system requires a model that is the basis for all abstractions used in the system. Our model is based on the composition of the abstractions within the implementation of a program and is illustrated in Figure 3.1. From the bottom up (inside out), each abstract data type has a name and a type. This type is a concrete representation with a scope within the source code module. That module is programmed in a specific programming language and it is compiled into the virtual machine. From the top down (outside in), any virtual machine can be built from different programming languages. Each language uses its specific constructs to specify source code modules which express the scope of the type of a named abstract data type instance. This model is the guide to the object-oriented structure of our debugging environment by defining the abstract meaning of data with respect to the whole program. Therefore, the abstract objects for our environment must be able to express the abstract meaning of the data.

3.2.1 Objects

Objects are the core of any object-oriented system. For our debugging environment, we do not need a general purpose object system. Rather, we see the need for a restricted set of object classifications for debugging purposes. The following is a list

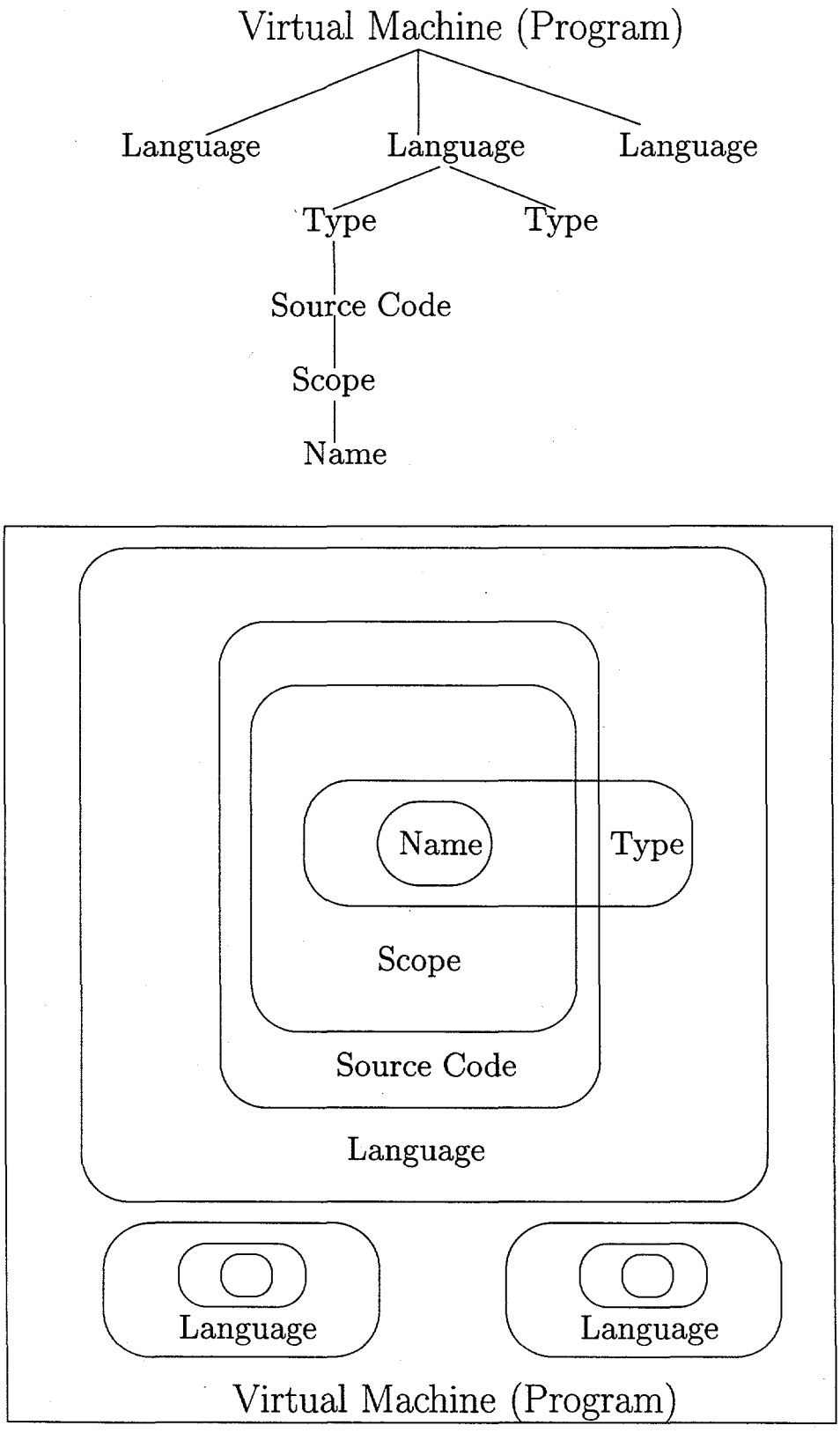


Figure 3.1: Abstract Data Type Hierarchy Model

of the debugging objects for our environment.

Abstract Data Type (ADT) ADTs allow the engineers to describe abstractions so that the debugger can manipulate them. However, there needs to be an association between the actual implementation and the abstraction to support non-object-oriented based languages. By specifying this association, the engineer may overload a data instance of any ADT at any time. The object provides specification all of the abstract meaning of the data in a form understood by our debugger.

Abstract Scope Sometimes the conceptual scope of a design does not align with the physical scope of the code. Abstract scoping objects allow for explicit modularization during debug-time without the performance penalty. In that respect, abstract scoping is very similar to the C++ inline function concept. Additionally, our debugger retrieves structural information about the code layout contained in abstract scopes. Also, an abstract scope may transcend normal language scoping boundaries. This means engineers may define scopes appropriate to the design because they are less confined by the implementation language of efficiency concerns.

Abstract Break/Watch Points These are standard debugging features extended for our unified debugging environment. An abstract breakpoint may be placed in the source code at unique and interesting locations within the code. Similarly, an abstract watchpoint can be used to gather metrics about the program, report data values, etc... These abstract points should allow logical groups of similar points to be jointly activated or deactivated. This control could be specified by using the debugger or by the program. Abstract point allow the engineer to define known points of interest as the code is being developed. These points represent common points that will be frequently activated during debugging and therefore are a convenience to the software engineer.

Abstract Module An abstract module is a grouping of larger code blocks into a single abstract grouping. This is similar in concept to an abstract scope, however, an abstract module may transcend function and source file boundaries.

Abstract Source Language The abstract source language is a specification of the programming language used to implement modules of code. This specification helps the debugger correctly parse source programming language specific syntax. This is important so as to allow the debugging environment to have the same syntax as was used in the programming environment. This eliminates the need to learn a different syntax for accesses within the debugger.

Virtual Machine This is the abstraction of the whole program, which includes all of the abstract objects mentioned above, plus it includes the grouping of all source programming languages to form the virtual machine. It is through this object that the "greater meaning" of the program is described.

The above lists only the types of abstract objects that we feel should exist in our unified debugging environment. The emphasis has been to specify a regular and structured method to communicate data to the debugger that normally is implicitly carried by the engineer when debugging a program. The regular specification of the meaning of the data allows for an automated tool to process the information for the benefit of the original engineer or any additional engineers that debug the program.

3.2.2 Methods

The second principle of an object oriented system is the association of functions to data. A function of this sort is called a *method* of that object. To accomplish encapsulation of standard debugging techniques with our debugging environment, we have decided on a set of pre-defined methods for our debugging objects. By pre-defining these methods, we have restricted the meaning of each of the methods to a specific purpose as indicated by the method name. Unlike a typical object-oriented system, our debugging objects do not allow for general purpose methods to be defined

for a debugging object, nor can the parameters of the methods be altered. As a result of these imposed restrictions, we use the term *feature function* to describe these restricted methods. The following is a list of the features we feel are needed for our debugging environment.[1]

Printer This feature is responsible for displaying the value (meaning) of the abstract object. Exactly how this feature accomplishes that task is user-defined. Multiple printer features could be defined to support polymorphism or other high-level features of the object.

Description A description is a constant text string that describes an object. In other words, this feature provides a way to supply comments about the object so that the debugger can present this data when the user demands a description of the object.

Validator This feature is intended as a way to check the integrity of an abstract object. The validator feature's definition and action are under programmer control. Validation can be placed anywhere in the control flow of a program.

Watcher This feature provides a check-pointing facility on the data of an abstract class. Statistics about the data can be gathered or the value of the object could be output just like a print statement. The watcher feature is executed at a watchpoint placed within the control flow of the program.

Operators Operations can be defined for objects using a set of operators. This is similar to operator overloading in C++. Operations are relational (equal, less than, greater than, etc.), arithmetic (addition, subtraction, etc.), or conversion (typecasting). These operations are inherently polymorphic as the meaning of the operators will depend upon the operands. The operations are used within expressions entered in at the command line of the debugger.

[1] One could debate whether a particular feature has a valid meaning for a particular object. We will assume that this is an implementation decision, therefore all object could have all of the features.

Access history This feature provides a trace history of the accesses done upon a specific instance of an abstract object. Such a history could contain the location in the code that last modified the object, or the sequence of functions that used the instance as a parameter. Whatever activity that requires a history of an instance's run-time usage can be specified.

Initialization This feature provides the facility to seed an instance of the abstract object with valid data. This can be used as an initializer, or could be used to facilitate a save/restore/undo feature.

Clearly, the above list does not encompass all debugging techniques and aids. For example, we have avoided aspects of parallel and distributed debugging. Although the techniques we have listed are valid for that type of debugging, additional features could be defined such as: synchronization, processor/network topology, event sequencing, latency, message contents, etc... The object-oriented design of the abstract objects allow for easy addition to the set of feature functions.

3.3 Interface Extensions

We have developed a prototype system that demonstrates our object-oriented debugging environment. It is an integrated system with a large subset of the required features. This section will briefly outline the features currently supported in our system.[11] Currently, our system supports the source programming language of ANSI C. The choice of this source programming language was an implementation detail and does not impact upon the design concepts or features of our environment.

The first step in realizing our system is to decide on the form of the expression of the abstract objects. Our expressive form goes by the generic name of "Interface Extensions". The design must meet the following requirements: have a simple grammar and syntax structure; be largely source programming language independent; and be easy to remove for production compiles. The syntactical structure must be simple and concise. Also, we require a structure such that high-level language translators could

generate our extensions in its generated code. As mentioned previously, our perspective about the source programming language is that of an implementation detail and thus only one modular component of the whole debugging environment. Thus, it did not matter which source programming language we chose for our prototype system. Most important, however, is the requirement that the extensions be easily removed when not compiled for a debugging session. We achieve this by placing the extensions within comment block structures. Our extensions should be considered compiler **pragmas**.[2] When compiling for a production version of the code, the pragmas are ignored.[3]

The existing prototype system does not implement all of the objects or features as described in Section 3.2. The representative set of features that are supported is as follows:

Abstract Data Type Essentially, this definition forms a class-like data structure for the purposes of encapsulating the implementation of debugging routines. Additionally, the abstraction may be defined as a subclass from an existing abstraction.

Mapping The programmer may annotate the code with a mapping between the a variable name or type and the abstraction it represents.

Abstract Scope Abstract scoping is limited to the scoping rules within a source programming language's function boundary.

Abstract Break/Watch Points These features are fully supported, except the points can not be activated under program control.

Abstract Source Language The abstract source language can only be defined by specifying delimiting tokens and it is only supported within the parsing of commands entered at the command line of the debugger.

[2] A pragma is defined as a "meaningful comment" or compiler directive.
[3] See Appendix C for a summary of the extensions.

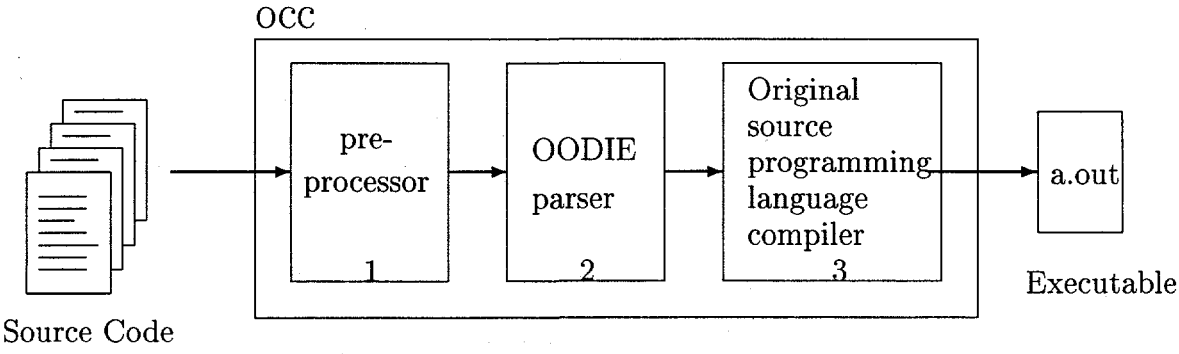


Figure 3.2: Phases of the OODIE Prototype Compiler

Virtual Machine Multi-language support is not implemented. However, some of this functionality is already present within the debugger.

Feature Functions The supported feature functions include: printer (one only), description, watcher, and validator.

3.4 OODIE Compiler

One of the requirements of the interface extensions is to be source programming language independent. Effectively, this creates another language that is specified along with the normal programming language statements such that placement of the extensions will not break normal compilers for that source language. The solution is to place pragmas within the comment block structure of the source programming language so that the pragmas are completely ignored by a normal compiler. However, there must be a mechanism to extract the pragmas from the source code. Ideally, one would modify the compiler to recognize the new pragmas. This was not practical in the context of our prototype. With all of that to consider, we have decided on a three-phase compiler for OODIE which is designed as a substitute for the normal source programming language compiler. The phases are diagrammed within Figure 3.2. Phase 1 is a preprocessor used to handle macro expansion and header file inclusion, if appropriate. For C and C++, this is simply the cpp program. Phase 2

takes the pre-processed source code and extracts the pragmas from the source code. Essentially, it acts as a filter between the preprocessor and the source programming language compiler such that the Phase 2 parser passes through normal source code and extracts the OODIE extensions. Phase 2 is also responsible for generating the symbolic information required to support the OODIE abstract objects. Phase 3 is the original source programming language compiler that normally is used to compile the source code. This phase executes the compiler such that debugging symbols are generated and an executable file is produced.

The modular design of the OODIE compiler encourages its implementation for many source programming languages. By separating the compilation of the extensions from the source programming language, only small modifications need be done on the OODIE parser to recognize the language and extract the pragmas.

3.4.1 Symbolic Information

Phase 2, the OODIE parser, must generate the additional symbolic information required to support the OODIE feature set. The key barrier is the source programming language compiler. Since that compiler does not recognize OODIE extensions, it must essentially be tricked into passing symbolic information into the executable and then into the debugger. The best solution would be to add extensions into the symbol table that is placed into the executable by the compiler. Unfortunately, adding extensions to the normal symbol table format is not possible since these formats are not standardized across many platforms.

The decision was made to encode the necessary symbolic information for OODIE features within the syntax of the source programming language. Therefore, Phase 2 outputs additional source code for each source file parsed. This additional information is then compiled by Phase 3 which generates symbols that pass into the debugger. Fortunately, the additional code can be easily hidden from the user when debugging. Additionally, great effort was spent on maintaining the source line-number information between the input source code and the code that the OODIE compiler actually

compiles. A illustration of the structure of this symbolic information is shown in Figure 3.3 and a sample of the Phase 2 generated code is listed in Appendix D.

3.5 OODIE Debugger

Essentially, the OODIE compiler generates additional symbol tables for the abstract objects of the program. Since it was not possible to integrate these additional symbols into existing symbol tables, an existing debugger required modification. We decided upon the GNU project's `gdb`[4] debugger to be the basis for the OODIE debugger. This allowed us to immediately gain the portability and functionality of `gdb`. The necessary modifications were actually quite minor. Due to the extensible design of the debugger[10], it was very easy to add new commands to support our new features. The new commands provide the interface to the new symbolic information provided by the OODIE symbol tables. For more details about the added commands, see [11] or Appendix E for a command summary.

The only real issue about the implementation of the debugger was how to access the new symbolic information. Since it is not currently part of the normal symbol table, some other mechanism has to be used to extract the data. First, the location of the data must be known and this is relevant to the general type of debugger being used. The `gdb` debugger is called a *separate process* debugger in that the executable being debugged is run as a child process of the debugger under the control of either the `ptrace` or `/proc` mechanism.[5] Each mechanism provides methods to access the child process' memory image, but only a byte at a time[6] or via `ioctl` calls[7]. Either method is slow, but it is the only way to transfer data safely[8] and bidirectionally

[4] Version 4.12 source is used as the baseline.

[5] `/proc` is Unix System V, `ptrace` is BSD Unix

[6] `ptrace`

[7] `/proc`

[8] Safely with respect to an operating system point of view. Essentially, a debugger program needs to have complete control of the child process. Specifically, control of the execution of the child and access to the memory space of the child will ensure accurate display of program data with the debugger program.

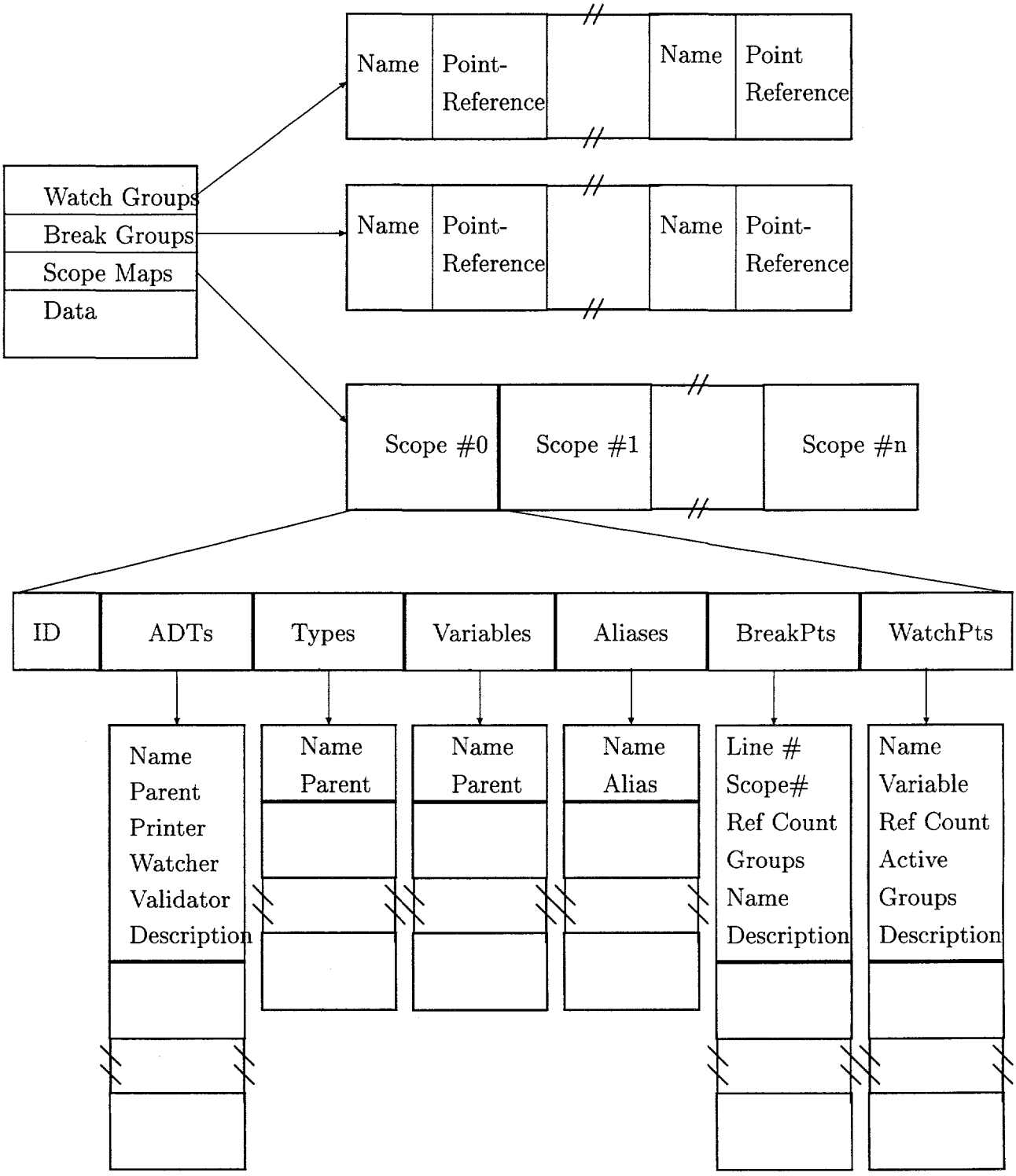


Figure 3.3: OODIE Symbol Table Structure

between the child and the debugger. However, to implement the abstract break/watch points these mechanisms must be able to activate, deactivate and keep track of group reference counting. If the debugger maintained this data, the child would have to signal the debugger when these points were reached to check its active status. This would significantly increase the execution time of the child process as the operating system mechanisms to do this are expensive. As a result, the performance penalty would lessen the usefulness of the break and watch points. Also, the requirement for the abstract data type object assumes one would like to access the contents of the object from the debugger's command line. As required, a printer feature function will be executed. Since at this point the child process is halted, it must be restarted and set to execute the routine but only after the OODIE tables were examined to search for the printer feature function. The bottom line is that access between the child process and the debugger is expensive and slow and it seemed obvious that our debugging features require extensive access to the child's core memory. Ultimately, we believed the performance penalty would be significant with small programs and unbearable with medium or large codes.

Our solution to our performance problem takes advantage of a feature found in the gdb debugger. It is possible to invoke a function existing in the program directly from the command line of the debugger. When analyzing the amount of symbol table searches and function invocations required to support our environment, the most efficient system is to let the child process execute all of the table searching and function execution. This is accomplished by linking a library of "OODIE helper" functions into the executable. Under this scheme, the debugger simply invokes a known library helper routine and allows the routine to execute within the child process. This simplifies the OODIE symbol tables in that pointers to the data structures are directly traversed and memory locations are modified by the child itself. With the OODIE library functions, a robust interconnected symbol table is easy to construct, maintain and efficiently search because the debugger/child process communication overhead is kept at a minimum.

As it turned out, the debugger aspects of the environment were simplified by using a library routine approach. The design and content of the OODIE symbol table reflects the decision that the child process will be examining the data rather than the debugger process. In addition, the library routines are source programming language independent as it is only the debugger that is aware of the routines and the proper procedure to invoke them. This is illustrative that tighter coupling between the debugger, compiler and abstract objects of the program allows the debugger to capitalize on the increased information about a program and provide the user with a customizable environment for the virtual machine being debugged.

Chapter 4

OODIE Examples

This chapter contains example codes that use the OODIE extensions to program the debugger about the program's abstract data type and design. We feel that the chosen examples are representative of general classes of data structures and they show the utility of having additional information about the data accessible in the debugger.

Except when noted, the examples will be coded in C. Also, we will present figures that show a sample transcript of a debugging session using the gOODIE debugger. Throughout the examples, OODIE extensions in the source code will be highlighted using **bold** type. For the debugger transcripts, user inputs are indicated by *italic* type. For a summary of the OODIE extensions see Appendix C and Appendix E for gOODIE debugger commands or [11] for a more complete description of both. Some of our examples are condensed for the figures, but full listings are given in Appendix F.

The next three sections present examples that are representative of general classes of data structures. These classifications are based upon the use and structure of the abstract data elements within a program. First, globally structured data type examples are presented. Second, examples that use data which have a hidden meaning known only to the programmer are shown. Third, examples of how a high-level programming language can be extended with OODIE extension to allow easier debugging are discussed.

4.1 Global Structure

Sometimes, data objects are not self contained. They may, for instance in a linked list, use pointers to reference other structures. It is very difficult for a debugger to print or manipulate such a structure without help. These structures may have a clear representation, but the original type information specified in the source code is not sufficient to describe it fully to the debugger. In most cases, it is the structure that makes the abstraction useful. For example, a binary tree has a uniform structure such that the orientation of data within the structure carries a meaning. The general indicator for a global structured abstract data type is one where a picture of the structure is extremely useful comprehending the data type. If the debugger could print the data structure in its natural graphical form, comprehension of the data would be easier. Unfortunately, a typical debugger can only display the component parts of the data structure as specified by the source code. This requires the software engineer to recognize similar patterns of implementation to identify the global structure. The meaning of the data can only be comprehended once the global structure is understood.

This section contains two code examples that use globally structured data types. Essentially, these examples will show the utility of providing a unified debugging interface for a "pretty-print" mechanism for abstract data types. As a matter of style, an engineer could easily duplicate the results of the example without the use of OODIE, but that implementation would not be portable between different engineers. Meaning, another engineer may not be aware of the special debugging functions and not use them or recreate the equivalent routines. With a standardized interface for manipulating abstract data types, this problem is eliminated.

4.1.1 Linked List

This example implements a simple linked list structure that manages a list of indexes into a global array of names. The implementation is listed in Figure 4.1 and uses

```
typedef struct _list {
   unsigned char id;
   struct _list *next;
} LIST, *LISTPTR;

/*pragma defabstract LinkedList // #1
      printer LinkPrinter
      description "A simple linked list of names"
   end */

static char gNames[][8] = {
        "Samuel", "Hugo", "John", "Trevor", "Robert"
};

#ifdef __OODIE
void LinkPrinter (LISTPTR data, char *params)
{ printf("\n");
  for( ; data != NULL; data = data->next)
        printf("%s --> ", gNames[data->id]);
  printf(" NULL\n");
}
#endif /*__OODIE*/

#define NEWNODE (LISTPTR)malloc(sizeof(LIST))

int main() {
   /*pragma defvar head isa LinkedList // #2 */
   LIST head; LISTPTR other;
   head.id = 2; other = head.next = NEWNODE;
   other->id = 0; other = other->next = NEWNODE;
   other->id = 4; other->next = NULL;
   other = NULL;
}
```

Figure 4.1: Linked List Implementation

```
Breakpoint 1, main () at linkedlist.c:56
56 other = NULL;
(gdb) print (LIST)head
$2 = {id = 2, next = 0x10011064}
(gdb) devar head
head isa LinkedList
      "A simple linked list of names."
(gdb) aprint head
John --> Samuel --> Robert --> NULL
(gdb)
```

Figure 4.2: Debug Transcript For The Linked List Example

common coding styles to implement the linked list. The interesting items are the OODIE extensions used within the code. The first extension (as indicated by #1 within the code listing) defines an abstraction named `LinkedList` with two features, a printer and description. The printer feature requires a function name, and the description is just a string. The printer function, LinkPrinter, defines the meaning of the abstract data and global structure of the linked list. The second extension (#2) defines the mapping between the variable instance of the linked list (`head`) and the abstraction named `LinkedList`.

Figure 4.2 shows a transcript of a sample debugging session using our OODIE debugger. This representation presents the data exactly how the linked list is implemented. The problem is that we feel that what the programmer wants to see is the abstraction of the linked list and not necessarily the implementation details. The `devar head` command displays a description of the variable instance and the name of its abstraction. The `aprint` command prints the abstract value of the linked list, and this meaning is what the engineer wants to see. This value is produced by the user supplied printer feature function.

Admittedly the code is a trivial example, however it quickly illustrates how a few simple definitions and annotations can provide a lot of abstract information to the debugger. With this information, the debugger can present the data in the form consistent with the original abstract design. The new representation of the *meaning* of

the data leads to *comprehension* of the abstract data without having to comprehend the details of the implementation.

4.1.2 Hash Table

This examples shows a hash table implementation. The hash table is of interest because it composes two implementation forms into one abstraction. The table uses the first letter of the input string as a hashing key. The hashing function takes the ASCII value of the character and computes an index into an array. This implementation uses an un-sorted chaining policy to handle collisions, thus, each element in the array is a linked list of similarly hashed strings. Figure 4.3 is a partial listing of the code showing the data structures and the test harnesses.[1]

The hash table abstraction is another example of how a graphic representation can lead to comprehension of the meaning of the data and also the implementation. A sample transcript of a typical debugging session of this hash table code is listed in Figure 4.4. The first few commands examine the details of the implementation. Normally, this is all that would be available and the engineer would have to trace pointers to view the data. The key question to ask is, "Where has the clean abstraction of the hash table gone?" The answer is that the abstraction has been codified and filtered away by the programming language, compiler and the linker, and been reduced to pointers to core memory. With a few simple OODIE extensions, the engineer can now see the data in the context of the abstraction. The last few debugger commands used in the transcript show the utility of our OODIE system and finally, present the hash table in a form very close to the abstract design.

4.2 Abstract Data

The software engineer may use data structures that have a hidden meaning. The meaning is hidden because the engineer is encoding an abstract meaning into an

[1] The complete source is listed in Section F.1

```
typedef struct _TableEntry { /* define table entry */
    char *name;
    struct _TableEntry *chain;
} TableEntry;

#define MAXENTRIES 26

typedef struct {
  TableEntry *baseTable[MAXENTRIES];
} HASHTABLE;

/*pragma defabstract HashTable
      printer HashPrint
      description
         "Hash table using first character for hash value\n  \
         and using chaining to resolve collisions."
   end */

int main()
{
   /*pragma defvar ht isa HashTable */
  HASHTABLE ht;
  INITHASHTABLE(&ht);

  ADDTOTABLE("Hello"); ADDTOTABLE("JOHN");
  ADDTOTABLE("Mike");  ADDTOTABLE("Zebra");
  ADDTOTABLE("Jackson"); ADDTOTABLE("mother");
  ADDTOTABLE("heather"); ADDTOTABLE("sandra");
  ADDTOTABLE("Victor");  ADDTOTABLE("Victoria");
  ADDTOTABLE("michelle");
}
```

Figure 4.3: Partial Hash Table Code

```
Breakpoint 1, main () at hash.c:101
101   ADDTOTABLE("michelle");
(gdb) next
102 }
(gdb) whatis ht
type = HASHTABLE
(gdb) print ht
$1 = {baseTable = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x10010fa4, 0x0,
   0x10010fb0, 0x0, 0x0, 0x10010fbc, 0x0, 0x0, 0x0, 0x0, 0x0,
   0x10010ff8, 0x0, 0x0, 0x10011004, 0x0, 0x0, 0x0, 0x10010fc8}}
(gdb) print ht.baseTable[7]
$2 = (TableEntry *) 0x1001fa4
(gdb) print *ht.baseTable[7]
$3 = {name = 0x100015ac "Hello", chain = 0x10010fec}
(gdb) print *ht.baseTable[7]->chain
$4 = {name = 0x100015dc "heather", chain = 0x0}

(gdb) devar ht
ht      isa HashTable
      "Hash table using first character for hash value
      and using chaining to resolve collisions."

(gdb) aprint ht
[ 7] : --> Hello --> heather
[ 9] : --> JOHN --> Jackson
[12] : --> Mike --> mother --> michelle
[18] : --> sandra
[21] : --> Victor --> Victoria
[25] : --> Zebra
(gdb)
```

Figure 4.4: Debug Transcript For The Hash Table Example

optimized implementation form. For example, an integer may represent a string that is found within a table. The structure of the data is not important to the meaning, but rather the structure is simply a container of the information. Unfortunately, when debugging such an abstraction, the debugger is unaware that the data is a container and that there is a hidden meaning to the value of the data. The reason this occurs is that the programming language, compiler and linker have completely filtered out the abstract meaning and transformed the structure into a form recognizable by the computer. It is left up to the engineer to understand the compiler transformation and manually convert the data from the optimized form to that of the abstract meaning.

The next two examples illustrate this situation and the difficulty in debugging this type of data instance. Unlike the global structure examples (Section 4.1) which at least presents the component parts of the abstraction, the debugger as no additional information about the data and can only present the data exactly how it was specified in the implementation. In most cases, the container is complete, but is in an unreadable form. In these examples, we will show how the OODIE extensions can be used to express the explicit meaning of the data instance without relying on the engineer to know how the data is contained.

4.2.1 Bitflags

This example shows a common problem when debugging a concrete representation of an abstraction. Consider a technique to codify a set of state values within a 32-bit integer. Specifically, if the nth bit has a value of 1, then the nth state is present or active. The code in Figure 4.5 represents this type of codification. In the example, an integer is representative of which signal interrupts are active within a UNIX program. In this case, if the nth bit is set, then the nth signal within a set of 32 signals is active. The compression of the boolean state of each of the 32 signals provides the best implementation performance and is concise, but it is difficult to debug. The debugger "sees" the state space as just an integer and not 32 distinct values.

Unfortunately, debugging such an efficient implementation does not provide any

```
/*pragma defabstract SigMask
      printer SigPrint
      description "Signal mask"     end */

/* printer method function omitted */

int main() {
   /*pragma defvar mask isa SigMask */
     int mask;

     mask = sigsetmask(0x0de30347);
     mask = sigsetmask(mask);
}
```

Figure 4.5: Bitflags Example

clues to the meaning of the data, especially when the form of the data is essentially atomic to the debugger. This is exactly the case in our example; the representation of all of the signal states is an integer. From the perspective of the debugger, it can only present the data as an integer and the abstract meaning can not be represented. In Figure 4.6, the transcript shows the situation just described and shows the use of the OODIE printer feature to completely express the meaning of the abstraction. There is no possible way a debugger could present this data in a form consistent with the abstraction without the additional symbolic information provided by the OODIE extensions.

4.2.2 Bitmap

This example is very similar to the bitflags example. However, a bitmap is not just a container of data, rather, the bitmap, *as a whole* carries the meaning of the abstraction. The individual components of the abstraction have no useful meaning, yet a typically debugger can only express the bitmap by displaying the component parts. Using the OODIE extensions, the true meaning of the data is expressed. In Figure 4.7[2], the transcript shows how using the standard debugging commands are

```
Breakpoint 1, main () at bitflags.c:24
24 mask = sigsetmask(mask);
(gdb) next
25 }
(gdb) print mask
$1 = 98763335
(gdb) print /x mask
$2 = 0x5e30247

(gdb) aprint mask
SIGHUP, SIGINT, SIGQUIT, SIGEMT, SIGBUS, SIGUSR2, SIGCLD, SIGPOLL,
SIGIO, SIGURG, SIGWINCH, SIGPROF
(gdb)
```

Figure 4.6: Debug Transcript For Bitflags Example

not effective when dealing with a bitmap and how the OODIE command set does exactly what is desired by the engineer. In this case, the bitmap data is imaged using a textual representation of the data. It is important to note that the printer feature can be as sophisticated as necessary. For this example a graphical display tool could have been used to view the bitmap.

Considering the type of data presented in the previous examples, the engineer could make due with the representation given by the typical debugger. However, what about data that has no textual form? For example, an audio clip has a machine representation similar to that of a bitmap, but it can not be "printed" by a debugger in any meaningful form. An OODIE version of a printer could easily output the audio clip to the speaker.

4.3 High Level Language Translators

A common technique in the design of a new programming language is the use of an existing programming language as an intermediate form. This allows the reuse of

[2]Complete code can be found in Section F.2

```
Breakpoint 1, main () at bitmap.c:90
90 i=i*9+3; /* just here for a break point */
(gdb) print xlogo16_bits

$1 = "\017\200\036\200<@x x\020\b\t\005\002@\a \017 \036\020\036\b<\004x\002"

(gdb) print /x xlogo16_bits

$2 = {0xf, 0x80, 0x1e, 0x80, 0x3c, 0x40, 0x78, 0x20, 0x78, 0x10, 0xf0, 0x8,
      0xe0, 0x9, 0xc0, 0x5, 0xc0, 0x2, 0x40, 0x7, 0x20, 0xf, 0x20, 0x1e,
      0x10, 0x1e, 0x8, 0x3c, 0x4, 0x78, 0x2, 0xf0}

(gdb) devar

Abstract Variables:
Name                 Abstraction          Description
xlogo16_bits         BitMap16             16 x 16 image
xlogo32_bits         BitMap32             32 x 32 image

(gdb) info alias
Visible Aliases:
      x16=xlogo16_bits
      x32=xlogo32_bits
(gdb) ap x16

XXXX            X
 XXXX           X
  XXXX         X
   XXXX       X
   XXXX      X
    XXXX    X
     XXXX   X
      XXX X
      XX X
      X XXX
     X  XXXX
     X   XXXX
    X    XXXX
   X      XXXX
  X        XXXX
 X          XXXX
```

Figure 4.7: Debug Transcript For Bitmap Example

existing compiler technology to be easily applied to the new language. The leveraging off of existing tools expedites the development of the new language. In this case, the new programming language's compiler acts as a high level language translator as it compiles its language and generates a "more common" source programming language. However, using the existing debugging tools are not as useful as tools designed specifically for the new programming language. The effort to develop new debugging tools is substantial for the new programming language and defeats the purpose of using a common language as an intermediate form. Therefore, new debugging tools are generally not built for new programming languages.

Although the translated language benefits from reusing existing software tools, with respect to debugging, the use of a programming language hierarchy makes debugging this type of program very difficult. When attempting to debug a program generated by this system, all of the abstractions one found in the high level programming language will have been filtered *twice*, once by the translator and once by the back-end compiler. Most likely, the debugger to be used will be one that can debug the back-end source code and the engineer will be exposed to **all** of the implementation details generated by the translator. The abstractions as well as the source code of the high level language are not accessible. In effect, the virtual machine created for the high level programming language is buried in back-end implementation details.

What is desired is a debugger modified for use with the high-level language. This assumes one has access to the source code of a debugger and it has been designed such that adding support for a new source language is not impossible. The GNU debugger, `gdb` is one such debugger, however, one still must apply significant effort in understanding the existing `gdb` code to decide what to add and maintain. Basically, it is a significant effort to rewrite a debugger for the high level language, especially as the purpose of using the back-end source programming language was to provide reuse of existing tools. Therefore, programmers of the high level language are sometimes left with no choice but to learn the details of the implementation and use the debugger to debug the implementation. Once the bug is found, one hopes to be able to figure

out the cause by manually translating the implementation back into the high-level programming language.

One solution is to program a debugger with a description of the virtual machine generated by the translator. Our unified debugging environment provides this ability. The advantage in using OODIE extensions is that the translator can automatically generate the extensions required to support the high-level programming language. Also, as the virtual machine changes, the OODIE extension would change along with it, allowing updates of the high-level language to be fully supported by the debugger.

The next two examples use two high-level programming languages that use a translator to generate a common back-end source programming language. We will show how extensions added by the programmer combined with automatically generated extensions by the translator can describe the virtual machine of the high level language.

4.3.1 Yacc

For this example, we will examine the `yacc` programming language[6]. A `yacc` program describes a compiler by specifying a LALR grammar augmented with action routines. The `yacc` compiler takes the grammar and generates a LALR virtual machine by generating C code. As expected, if one needs to debug the grammar, one must learn the implementation details of the machine generated code and figure out its meaning relative to the grammar rules that compose the `yacc` source code.

In Figure 4.8, the production rules are listed while the complete `yacc` intermediate code is found in Section F.3. The `yacc` program is straightforward with respect to describing the grammar rules. However, Figure 4.9 illustrates how all of the implementation details of the generated code must be understood to debug the grammar productions.[3] A seasoned `yacc` programmer will have learned a few tricks to help debug the machine generated code. First, the programmer realizes that the virtual

[3]The generated code is listed in Section F.3.2

```
%%
E : E '+' E
  {
      /*pragma describe it "Plus Rule" */
     $$ = Binop($1, $2, $3);
  }

  | E '*' E
  {
      /*pragma describe it "Multiply Rule" */
    $$ = Binop($1, $2, $3);
  }
```

Figure 4.8: Yacc Productions

machine is a finite-state automaton that is table driven. Second, all of the production rules are located within a `switch` statement near the end of the code. Based on that, the engineer can find the source line where the production rule of interest exists within the generated code and set a breakpoint on that line. Third, the generated code uses `#line` directives to match the source line numbers to the generated C code. Once the engineer has learned the implementation of the virtual machine, productive debugging can take place. Needless to say, debugging `yacc` grammars can be difficult because the code is machine generated. However, if the `yacc` compiler was modified to emit OODIE extensions, it could use them to provide extremely useful information to the engineer.[4]

Figure 4.10 is another transcript of debugging the `yacc` grammar, but it shows the additional information supplied by the extensions. The second transcript shows the usage of gOODIE debugger commands to obtain the abstract structural information

[4]The `yacc` generated code listed in Section F.3.2 contains the OODIE extensions as if the `yacc` compiler generated them. Note, that the OODIE extension in the source for the `yacc` grammar is also present. The OODIE extensions are at the end of the listing where the production rule code exists.

```
Breakpoint 1, main () at y.tab.c:23
23   main() { yyparse(); }
(gdb) step
yyparse () at y.tab.c:169 169   yypv = &yyv[-1];
(gdb) list

166             /*
167             ** Initialize externals - yyparse may be called more than once
168             */
169             yypv = &yyv[-1];
170             yyps = &yys[-1];
171             yystate = 0;
172             yytmp = 0;
173             yynerrs = 0;
174             yyerrflag = 0;
175             yychar = -1;
176
177             goto yystack;
178             {
179                     register YYSTYPE *yy_pv;    /* top of value stack */
180                     register int *yy_ps;        /* top of state stack */
181                     register int yy_state;      /* current state */
182                     register int  yy_n;    /* internal state number info */
183
(gdb)
```

Figure 4.9: Debug Transcript For yacc Code

```
Breakpoint 1, main () at y.tab.c:23
23   main() { yyparse(); }
(gdb) info scope

OODIE abstract scopes:
Global File Scope
    yyparse
       E : E '+' E
         | E '*' E

(gdb) info bpt *

Scope: Global File Scope                **Current Scope**
  none

Scope: yyparse
  none

Scope: E : E '+' E
Name                Description                    Line Refs Groups
Plus                Plus rule                       518    0

Scope:    | E '*' E
Name                Description                    Line Refs Groups
Multiply            Multiply rule                   529    0

(gdb) mark b Multiply
Breakpoint 2, at 0x400c4c:  file y.tab.c, line 529
(gdb) cont
a*i

Breakpoint 2, yyparse () at y.tab.c, line 529
529   yyval = Binop(yypvt[-2], yypvt[-1], yypvt[-0]);
(gdb) aprint $2
$1 = 42 '*'
(gdb)
```

Figure 4.10: Debug Transcript (using gOODIE) For yacc Code

about the code. The first command, `info scope` displays all of the abstract scopes within the code. If one examines the C generated source, one will noticed that the compiler has added the extensions needed to create an abstract scope for each production rule. The compiler named the abstract scopes the same as the production rules. The `yacc` compiler also added abstract breakpoints at the beginning of each action routine. The second debugger command, `info bpt *`, lists all of the abstract breakpoints in the program. With this information, the engineer can easily set the breakpoint for the multiply production by using the `mark b Multiply` command. Then, the engineer continues the execution of the program and enters sample text for the grammar to parse (`a*i`). At this point, the execution halts as it has reached the abstract breakpoint. The source line listed is the code within the multiply production, but after it has been translated by `yacc` and the convenience variables (`$$, $1, $2`) are missing. Another OODIE feature is to provide for source code aliases and the `yacc` compiler described the convenience variables within a sourcealias extension. The aliases are now accessible using the abstract print command. By using `aprint $2` command, the engineer discovers that the multiply operator is being used.[5,6]

In this example, we have shown how a modified `yacc` compiler can simplify the task of debugging its code by emitting OODIE extensions along with its generated C code. In effect, the `yacc` compiler could program the gOODIE debugger with abstract information about its virtual machine, which is an LALR grammar parser. This abstract information is the same information used by the engineer when programming the grammar. The engineer can now see the same abstractions during debugging as were used during the design and coding of the grammar. As a result, the implementation details are hidden and the engineer can debug the grammar without understanding the those low-level details. We claim that debugging with the OODIE extension will be significantly easier and more productive as the engineer is debugging the grammar and not the implementation.

[5]Referring back to the `yacc` grammar, the $2 variable is representative of the 2nd token in the grammar rule, which is the * character.

[6]The $1 on the left-hand side of the output value is a `gdb` convenience variable.

4.3.2 Sisal

This examples illustrates the problem of debugging code written for the Sisal[8] programming language. Sisal is a functional programming language designed for efficient and portable scientific programming on parallel machines. Figure 4.11 shows an implementation of Simpson's Rule[3] programmed in Sisal. The Sisal compiler acts as a high-level translator. It compiles a Sisal program into a more common programming language. Figure 4.12 is a extremely condensed listing of the generated C code for the Simpson's Rule program. Sisal represents an interest problem for programmers. Sisal C code is complicated by a number of implementation details. For example, the language is tuned for high-performance on a shared memory multi-processor environment. Additionally, Sisal is a language with functional semantics and it is translated into a procedure language for compilation. This is a fundamental transformation of the programming paradigms. Unfortunately, Sisal does not have a fully integrated debugging tool. Therefore, the programmer must understand all implementation details before beginning to debug a Sisal program using standard debugging tools. Needless to say, debugging Sisal is a very difficult task.

The generated C code from the Sisal compiler is shown with OODIE extensions. To fully express the functional programming paradigm of Sisal into the level of the debugger, extensive use of abstract scopes and abstract breakpoints must be used. By defining an abstract scope enclosing all of the C statements that represent a single Sisal source statement and defining an abstract breakpoint to represent a Sisal source statement, debugging Sisal becomes easier. Also, most of the symbolic variable names have been changed between the Sisal code and the generated C code. The `sourcealias` and `devar` extensions create the mapping between the optimized implementation and what was programmed in the original Sisal source. All of these abstractions must be used to present a Sisal program in the debugger that is consistent with the Sisal programming paradigm.

Figure 4.13 is a sample transcript of a debugging session without the use of the gOODIE commands. The transcript shows the process of viewing elements of a Sisal

```
define main

global sin( x : real returns real )

function vectorsum(a : array[real] returns real)
  for element in a
     returns value of sum element
  end for
end function

function main( a, b : real; n : integer returns real )

  let
    Delta := abs( b - a ) / real( n );
    odd,even :=
    for i in 1, n
      fi    := sin( a + real(i) * Delta )
    returns
      array of fi when ~ mod(i,2) = 0  %collect odd panels
      array of fi when   mod(i,2) = 0  %collect even panels
    end for;
    odd_sum := vectorsum(odd);
    even_sum := vectorsum(even);
    F_1 := sin(a);
    F_n := sin(b);
  in
      ( F_1 + F_n + 4.0 * odd_sum + 2.0 * even_sum ) * Delta / 3.0
  end let

end function
```

Figure 4.11: Sisal Code Implementing Simpson's Rule

```
#include "OODIE.h"

static void OODIE_RealVector(valp,parms);

/*pragma defabstract realvector
      printer OODIE_RealVector
      description "Array[real]"
end */

static void InitGlobalData();
static void _VECTORSUM( args );

static void _MAIN( args ) FUNCTION args;
{
  register POINTER tmp10;
  register POINTER tmp9;

 /*pragma defvar tmp9 isa realvector */
 /*pragma sourcealias odd = tmp9; */
 /*pragma defvar tmp10 isa realvector */
 /*pragma sourcealias even = tmp10; */

 /*pragma scope
  "simp.sis 30:(F_1+F_n+4.0 * odd_sum+2.0 * even_sum ) * Delta/3.0"
*/

 /*pragma defbreak _x line30 "line 30 of simp.sis" */

     Times( tmp18, ((struct Args13*)cframe19)->Out1, (2.0) );
     OptPlus( tmp6, tmp6, tmp18 );
     OptTimes( tmp6, tmp6, tmp7 );
     OptTimes( tmp6, tmp6, (3.3333333333333310000e-01) );
     ((struct Args21*)args)->Out1 = tmp6;

 /*pragma scope end */

static void InitGlobalData();
void SisalMain( args );
```

Figure 4.12: Condensed C Code Generated By Sisal Compiler

```
Breakpoint 2, _MAIN (args=0x1021f330 "") at simp.c:184
184 Call( _VECTORSUM, cframe15 );
(gdb) print tmp9

$1 = (unsigned char *) 0x1021f2a0 "\020!L"

(gdb) print *(struct Array *)tmp9

$2 = {Base = 0x1021f04c "", LoBound = 1, Size = 5, Phys = 0x1021f030,
Mutex = 0 '\000', Mutable = 0, RefCount = 1}

(gdb) print ((float*)((struct Array *)tmp9)→Base)[1]
$5 = 0.308865547
(gdb) print ((float*)((struct Array *)tmp9)→Base)[2]
$6 = 0.808736086
(gdb)
```

Figure 4.13: Debug Transcript For Sisal Program

array in the debugger. First, it is assumed that you are knowledgeable about the implementation details of the Sisal compiler. If you know these details, you can *begin* to debug the program. Without deep understanding of the implementation details, you can not debug the high-level Sisal program. Assuming one has deep knowledge of the Sisal implementation, it is still very difficult to access the data. As mentioned previously, the implementation is tuned for maximum efficiency and most, if not all of the abstract data structures are referenced as `char` pointers and require typecasting to access the real meaning of the pointer. Also, all of the Sisal variable names are gone and the appropriate temporary variable name that represents the Sisal variable can be found by analyzing the whole function.[7] Access to the elements of the array requires another step of dereferencing data through another hidden data structure. In other words, even with deep knowledge of the implementation details, debugging Sisal is a laborious, inefficient task that is not very productive in fixing a bug in the original Sisal source code.

Figure 4.14 is a sample transcript of a debugging session using the gOODIE com-

[7]Sisal implementation creates a significant number of anonymous temporary variables all of the form: `tmp`*n*. Also, the temporary variables are reused during different parts of the implementation.

```
Breakpoint 1, _MAIN (args=0x1021f330 "") at simp.c:162
162      register FUNCTION cframe19 = (FUNCTION) &CallFrame1;
(gdb) info scope


OODIE abstract scopes:
Global File Scope
  _MAIN
    simp.sis 30: ( F_1 + F_n + 4.0 * odd_sum + 2.0 * even_sum ) * Delta / 3.0

(gdb) info bpt *

Scope: Global File Scope
  none

Scope: _MAIN     **Current Scope**
  none

Scope: simp.sis 30: (F_1+F_n+4.0 * odd_sum + 2.0 * even_sum ) * Delta / 3.0
Name             Description           Line Refs Groups
_x               line 30 of simp.sis    230    0 line30

(gdb) mark bg line30
Breakpoint 2 at 0x400d04:  file simp.c, line 230.
(gdb) continue
Breakpoint 2, _MAIN (args=0x1021f9a0 "") at simp.c:232
232      Times( tmp18, ((struct Args13*)cframe19)->Out1, (2.0) );
(gdb) info alias

Visible Aliases:
        odd=tmp9
        even=tmp10

(gdb) aprint odd

[ 1,5: # DRC=1 PRC=1
 3.088655e-01
 8.087361e-01
 9.999997e-01
 8.096718e-01
 3.103797e-01

(gdb)
```

Figure 4.14: Debug Transcript (using gOODIE) For Sisal Program

mands. The data presented by the debugger is not a perfectly glossy representation of Sisal paradigm. Specifically, implementation source code is still display by gOODIE. However, through the use of abstract scopes, abstract breakpoints and source aliases, access to the elements of the array is significantly easier. Sisal source line information is available as well as the Sisal variable names. The OODIE extensions are able to express the Sisal programming paradigm so that within the debugger, a Sisal programmer can debug the Sisal implementation *without* deep understanding of the implementation.

Chapter 5

Summary and Conclusions

In this final chapter of our thesis, we present a summary of our unified debugging environment as well as conclusions about its implications towards debugging. We also outline alternative research topics based upon our prototype environment.

5.1 Summary

The preliminary research on software engineering and the role of debugging in a typical software process indicated that a holistic solution was required to increase integration between the design and maintenance phases of a software process. We began with an investigation into common debugging practices that are currently used to help debug programs. These practices vary wildly between programmers but are extremely useful in presenting the abstract meaning of data when using a debugger. The problem is there is no standard form for the insertion and deletion of techniques within the source code. Our solution is to provide a unified debugging environment that encapsulates important aspects of these techniques into a standard form.

For our environment, we developed a debugger programming language that is placed into the source code as pragmas inside comment blocks. It seemed logical to design the form to support inheritance, encapsulation and overriding by using some of the principles of object-oriented programming. These principles allow the definition

of the abstract data and design of a virtual machine. It is the abstractions of the virtual machine that describe aspects of the "higher" meaning of the virtual machine, i.e the design in the source code that would survive into the debugger phase. This meaning is what is truly required by the programmer when using a debugger.

OODIE, Object Oriented Debugging Extensions, describes the abstract data and design of the virtual machine such that an augmented debugger can access the abstractions. The benefit of designing, programming *and debugging* a virtual machine with the same abstractions, regardless of implementation detail, should improve the maintenance of a software system throughout its lifecycle.

5.2 Conclusions and Expectations

The purpose of our environment is to help the engineer debug their program. As a result of developing a prototype environment, we arrived at the following conclusions.

Programmer knows best The only one that carries the true meaning of a program and its data is the programmer. Therefore, it make sense that the programmer be able to express the meaning such that an automated tool can help process and manipulate the information. For example, in the bitflags example (Section 4.2.1), only the programmer knows about the meaning of each bit in the integer data. The debugger alone can not possibly present the integer in any other form other than a number.

Object Oriented This follows from acknowledging that the programmer knows best and currently the best way for the programmer to express this knowledge is through an object-oriented paradigm consisting of objects, inheritance, encapsulation and overrides. The bitmap example in Section 4.2.2 shows that the base class abstract definition of a bitmap image can be subclassed to represent exact sizes of an image.

Abstraction and Hierarchy Comprehension is easier when some type of abstraction is used to hide the details and focus on what is important. When abstraction is used, a hierarchy of abstractions is formed as the level of detail increases as the abstractions are made concrete. This hierarchy must be maintained during debugging so that both the implementation details and the abstraction are available to the engineer. The linked list example in Section 4.1.1 uses OODIE to express the data in the exact form as specified by the abstract design of the data structure.

Software Process Ignores Debugging The typical software engineering process lumps debugging into the maintenance phase of the software lifecycle. The process fails to describe a rigorous programming practice to aid in debugging a large software product. The only item the current process offers to the engineer when debugging is a voluminous set of printed documentation that may or may not reflect the latest modifications. On a large scale programming effort with many different engineers working on the same system, it is imperative that they all see the same abstractions so as not to assume too much about the design and data and introduce latent programming bugs. Paper documents are not sufficient as a program is a dynamic entity and it requires a dynamic environment that paper documents can not provide. The OODIE extensions are dynamic as they are a part of the source code and thus integrates the code with the abstract specification.

Active Comments Rigorous commenting practices are required to provide sufficient information about the program. However, updates to the comments may not occur as the code changes. If the comments were "active", meaning, that the comments had a purpose that directly effects how an engineer debugs a program, there is more incentive to comment and to maintain those comments as the source code changes. In effect, using a standard debugging programming language with active comments provides a means to manipulate the comments and to present them in a regular form while debugging.

Unified Environment The benefit of a unified environment is that the programmer will be provided a consistent programming paradigm through design, implementation and debugging phases of software product. This represents a tighter coupling of the phases and may increase productivity of software engineers.

Virtual Machines Existing debuggers expose low-level implementation details of the source code. For the next generation of programming languages, this detail is too microscopic to be useful or productive. As have programming languages used abstraction to allow expression of "higher meaning", so to must a debugger use those same abstractions to describe the virtual machine. An engineer must be able to debug the virtual machine and not just the implementation to be productive and to not introduce other bugs as the code matures.

5.3 Future Work

The goal of OODIE is to provide a customizable debugging environment for software engineers. Based on the prototype, further development of OODIE should be refinement of the abstract objects to provide a debugging environment that is more extensible for any virtual machine. However, a richer thread of research opportunity is in the application of OODIE. The software engineering community desires any tool that can help aid in the task of fixing a bug. This includes a "user-friendly" debugger that is aware of the meaning of the program. OODIE can provide this and more as the potential of OODIE is realized through it use by software engineers in production software.

5.3.1 Development

There are three major paths of development for the OODIE environment. First, all of the elements of the debugging environment (Section 3.2) should be implemented.

This will require enhancement to the Interface Extensions to account for the new abstract objects. Consequently, the gOODIE debugger must support the new objects by providing new commands to access the additional abstract information. The second path of development is to support additional source programming languages. This will require obtaining a grammar for the language and adding the appropriate code to support the OODIE functionality. The choice of source programming language will influence the implementation of some of the abstract objects, especially the abstract scope object. The difficulty is that the code must still be compiled by the normal compiler for that language. Third, automatic analysis of the meaning of data structures would allow for automatic generation of the OODIE extensions and feature functions. This is equivalent to the theoretical problem of program decidability. Therefore, the automation of the creation of feature functions for all possible abstract data types within a programming language can not be done. However, patterns may be recognizable for a small subset of abstract data types and routines could be formed from generic templates.

Eventually, a standard OODIE language definition will be defined and will lead compiler writers to directly support the extensions. This implies that the standard symbol tables integrate the OODIE abstract data which then allows demands that the standard debugger support the extensions. Ultimately, programming the debugger along with the software will be a standard practice in the software engineering development process.

5.3.2 Applications

The greatest benefit to be realized is the application of OODIE within the software engineering process. Specifically, OODIE will effect how software engineer develop software as more abstract information is present at all phases of development and maintenance. Besides the general software process benefits, OODIE will be most beneficial to high-level language translators. As shown in Section 4.3, the expression of the programming paradigm used to design the high-level program enables the

debugger to present a consistent view of the program. This is done by emphasizing the important details and ignoring the implementation details. The key for the high-level translators is to automatically generate the extensions within its generated code. This frees the programmer from manually adding the "standard" abstractions for the virtual machine. However, the programmer could extend those standard abstractions by using the subclass mechanism as well as create new abstractions for the specific virtual machine being programmed. This unifying environment enables the software engineer to program and debug using the abstractions of high-level language rather than debug at the low-level of the implementation.

Programming in a medium-level programming language will also benefit by using the OODIE extensions. Specifically, the OODIE extensions could be placed in standard header files and feature function implementations could be present in library archives. Operating system vendors could supply programmers OODIE-fiend versions of standard O/S system calls and data structures. For example, the bitflags example in Section 4.2.1 could be provided within the standard header files as the signal mask is an operating system defined abstraction. Having a predefined set of OODIE abstractions that express the programming environment will greatly aid the software engineer when developing software for the environment.

Another goal of the OODIE environment is to provide increased integration between program design, implementation and debugging. Unfortunately, the OODIE extensions are specified in the implementation and there is still an amount of conversion from the design into the extensions. To increase the integration, modification to the `WEB` system (Section 2.2.1) is a possible solution. In a sense, the `WEB` system is a high-level translator. It generates either source code or a specification-like document from a source text. The `WEB` system could be modified to automatically generate OODIE extensions when the source code is generated. The combination of the `WEB` system and the OODIE extensions would provide a tightly integrated software engineering development environment.

OODIE is a tool to help develop and maintain software throughout its lifecycle.

When the extensions are used and applied to different application, it will then be clear what are the exact weaknesses and strengths of the environment. We foresee that with routine use of extensions will drive the demand for standardization of OODIE. This will lead to better native compiler support and greater acceptance of OODIE in the software engineer community.

Bibliography

[1] 4.2 Berkeley Software Distribution. *Unix Reference Manual.* Man Pages Section 1.

[2] B. Boehm. A sprial model for software development and enhancement. *Computer*, 21(5):61–72, May 1988.

[3] Curtis F. Gerald and Patrick O. Wheatley. *Applied Numerical Analysis.* Addison-Wesley, third edition, 1984.

[4] W.C. Gramlich. Debugging methodology (session summary). In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, volume 18(8), pages 1–3. SIGPLAN Notices, 1983.

[5] IEEE. *IEEE Guide To Software Requirements Specifications.* ANSI/IEEE Std 830-1984.

[6] Stephan C. Johnson. Yacc: Yet another compiler-compiler. In *Unix Programmer's Manual Supplementary Documents.* University of California, 1984.

[7] Donald E. Knuth. *Literate Programming.* Center for the Study of Language and Information (CSLI), 1991.

[8] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas. SISAL: Streams and iteration in a single assignment language: Reference manual version 1.2. Manual

M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[9] Vern Paxson. A survey of support for implementing debuggers. Technical report, 1990. Obtained as an electronic publication.

[10] Robert Pizzi. Gnu debugger internal architecture. Technical Report UCRL-MA-115656, UC/Lawrence Livermore National Laboratory, December 1993. Electronically available at anonymous `ftp://sisal.LLNL.gov/`.

[11] Robert Pizzi. Occ/goodie user manual. Technical Report UCRL-MA-118895, Lawrence Livermore National Laboratory, November 1994. Electronically published via anonymous `ftp://sisal.LLNL.gov/`.

[12] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*, chapter 1. McGraw-Hill, third edition, 1992.

[13] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*, chapter 19. McGraw-Hill, thirdq edition, 1992.

[14] James Rumbaugh and Michael Blaha. *Object-Oriented Modeling and Design*, chapter 1. Prentice Hall, 1991.

[15] Rich Seidner and Nick Tindall. Interactive debug requirements. *Association of Computing Machinery*, 1983.

[16] Mary Shaw. Abstraction techniques in modern programming languages. *IEEE Software*, pages 10–26, October 1984.

[17] Martin Sitt. *Debugging: Creative Techniques and Tool for Software Repair*, chapter 1. Wiley Professional Computing. Wiley, 1992.

[18] Ian Sommerville. *Software Engineering*, chapter 23. Addison-Wesley, thrid edition, 1989.

[19] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 1992.

[20] Andrew S. Tanenbaum. *Structured Computer Organization.* Prentice-Hall, Inc, second edition, 1984.

[21] Ross N. Williams. *FunnelWeb User's Manual*, 1.0 edition, 1992. FunnelWeb V3.0.

Appendix A

Specification Document Samples

This appendix contains samples of software engineering specification documents. Due to space limitations, the complete documents are not listed. The purpose is to illustrate the content of information in each of the the different specifications. Section A.1 shows a sample portion of a requirements document following IEEE standards.[5]. Section A.2 details some design decisions based upon the requirements specification. Section A.3 illustrates a possible implementation of the requirements and design.

A.1 Requirement Specification

```
3.1.1  Filename Management Requirement
3.1.1.1  Introduction

This requirement manages the input source files as specified on the
command line.  All filenames must be available throughout the
execution of the program.  Management of the filenames shall allow for
random access to the names.  Filename modification rights shall be
write once, read many.

3.1.1.2  Inputs

A single input shall consist of unbounded length, null terminated C
string.  The filenames shall be assumed to be UNIX filenames.  There
is no maximum bound on the total number of inputs.

3.1.1.3  Processing

The filename inputs shall be sequenced internally in any fashion
```

deemed implementationally efficient. This may involve some type of sorting or table mechanism. Enforcement of the filename modification policy of write one, read many shall be handled.

3.1.1.4 Outputs

Filenames shall be output in the order in which the filenames was input. The relative ordering between the filenames based upon their input order shall be preserved.

A.2 Design Specification

3.1 Filename Management
3.1.1 Design

Filename management shall be implemented as a linked list. Each node of the linked list shall contain a pointer to dynamically allocated storage for the contents of the filename. The nodes in the linked list shall always be added to the end of the list to preserve ordering requirement.

3.1.2 Access functions
3.1.2.1 Creation

The linked list may be statically or dynamically allocated through the declaration of the filename management linked list structure.

3.1.2.2 Adding Names

A valid C string is passed to a function which adds the string to the end of the linked list. No validation of the string is performed.

3.1.2.3 Deleting Names

Not supported

3.1.2.4 Reading Names

Given a discrete integer value, x, in the range 0..n-1, where n is the total number of filename currently in the list, retrieve the x-th

filename entered into the linked list.

A.3 Code Implementation

```
#include <stdlib.h>
#include <stdio.h>

typedef struct _FManagerNode {
        char *fname;
        struct _FManagerNode *next;
} *FManagerNode;

typedef struct _FManager {
        int     count;
        FManagerNode head, tail;
} *FManager;

FManager FM_init()
{
  FManager fm = (FManager)malloc(sizeof(struct _FManager));
  fm->count = 0;
  fm->head = fm->tail = NULL;
}

void FM_add(FManager fm, char *fname)
{
 FManagerNode node=(FManagerNode)malloc(sizeof(struct _FManagerNode));

 node->fname =  strcpy(malloc(strlen(fname)), fname);
 node->next = NULL;

 if (fm->count) {
    fm->tail = fm->tail->next = node;
 } else {
    fm->head = fm->tail = node;
 }

 fm->count++;
}
```

```
char *FM_get(FManager fm, int idx)
{
  register int i;
  register FManagerNode node;

  if ((idx < 0) || (idx >= fm->count)) return NULL;
  for(node=fm->head, i=0;
      ((i<idx) && (node != NULL));
      node=node->next, i++) ;

  return (node != NULL ? node->fname : NULL);
}

main(int argc, char *argv[])
{
  FManager fm;
  register short i;

  fm = FM_init();
  for(i=0; i<argc; i++)
    FM_add(fm, argv[i]);

  /* do something interesting */

  printf("last name: %s\n", FM_get(fm, argc-1));
}
```

Appendix B

WEB Sample

This appendix contains a sample WEB document. This example is taken directly from [21]. The example is a document that describes and implements a listing of consecutive numbers raised to a power. The implementation is Ada.

```
@!-------------------------------------!
@!  Start of FunnelWeb Example .fw File !
@!-------------------------------------!

@t vskip 40 mm
@t title titlefont centre "Powers:"
@t title titlefont centre "An Example of"
@t title titlefont centre "A Short"
@t title titlefont centre "FunnelWeb .fw File"
@t vskip 10 mm
@t title smalltitlefont centre "by Ross Williams"
@t title smalltitlefont centre "26 January 1992"
@t vskip 20 mm
@t table_of_contents

@A@<FunnelWeb Example Program@>

This program writes out each of the first @{p@} powers of the first
@{n@} integers.  These constant parameters are located here so that
they are easy to change.

@$@<Constants@>==@{@-
n : constant natural := 10;       --How many numbers? (Ans: [1,n]).
p : constant natural := 5;        --How many powers? (Ans: [1,p]). @}
```

```
@B Here is the outline of the program.  This FunnelWeb file generates
a single Ada output file called @{Power.ada@}.  The main program
consists of a loop that iterates once for each number to be written
out.

@O@<Power.ada@>==@{@-
@<Pull in packages@>

procedure example is
  @<Constants@>
begin    --example
  for i in 1..n loop
    @<Write out the first p powers of i on a single line@>
  end loop;
end example;
@}

@B In this section, we pull in the packages that this program needs to
run.  In fact, all we need is the IO package so that we can write out
the results.  To use the IO package, we first of all need to haul it
in (@{with text_io@}) and then we need to make all its identifiers
visible at the top level (@{use text_io@}).

@$@<Pull in packages@>==@{with text_io; use text_io;@}

@B Here is the bit that writes out the first @{p@} powers of @{i@}.
The power values are calculated incrementally in @{ip@} to avoid the
use of the exponentiation operator.

@$@<Write out the first p powers of i on a single line@>==@{@-
declare
  ip : natural := 1;
begin
  for power in 1..p loop
     ip := ip*i;
     put(natural'image(ip) & " ");
  end loop;
  new_line;
end; @}
```

Appendix C

OODIE Interface Extension Summary

This appendix will briefly summarize the grammar and meaning of the current extensions supported by our OODIE compiler. The following typographic conventions will be used: `typewriter` font and all lowercase indicates a terminal keyword; *italic* and starting with a capital letter indicates a non-terminal; asterisk (*) indicates zero or more repetitions of the non-terminal; plus (+) indicates one or more repetitions of the non-terminal; brackets ({}) indicate optional tokens; vertical bar (|) indicates OR selection; double-colon (::) is the non-terminal definition operator. For a more complete syntax description, see [11].

C.1 Pragma Syntax Forms

`/*pragma` *Definition* `*/`

Pragma statement, only one definition allowed.

`/*pragma begin` *Definition*`+` `pragma end */`

Pragma block, multiple definitions can be specified.

C.2 Definitions

`defabstract` *Name* { : *Name* } *AbstractSpec*`*` `end`

Define a new abstraction with a given name. This new abstraction maybe subclass of an existing abstraction.

`deftype` *Name* `isa` *Name*

Define a mapping of a source programming language data type to an abstraction.

`deftype` *Name AbstractSpec** `end`

Define an anonymous abstraction for a named source programming language data type.

`defvar` *Name* `isa` *Name*

Define a mapping of a named variable instance to an abstraction.

`defvar` *Name AbstractSpec** `end`

Define a mapping of a named variable instance to an anonymous abstraction.

`scope` *String*

Define the beginning of an abstract scope. Must following the scoping rules of the programming language.

`scope end` {*String*}

Define the end of the last defined abstract scope.

`watch` *Name Name* {*GroupList*} *Description*

Define a watch point for the specified variable at the current location in the source code.

`defbreak` *Name* {*GroupList*} *Description*

Define an abstract break point at the current location in the source code.

`sourcealias` *AliasDefinition*

Define a name to be used as an alias of another name.

C.3 Other Definitions

Name :: *C string*

An ASCII string with a maximum length of 31 characters.

String | *Description* :: *C string*

A quoted, variable length C string.

AliasDefinition :: *Name* = *Name* ;

Definition of an alias.

AbstractSpec :: *Feature Name*

Specification of a list of feature functions defined for an abstraction.

Feature :: `printer | watcher | validator | description`

Names of supported feature functions.

GroupList :: *Name* | *Name* , *GroupList*

A list of names that indicate different logical groups of break or watch points.

Appendix D

Sample OODIE Generated Code

The following code is a sample of that which is generated by phase 2 of the OODIE compiler. Note, this code contains the original source, preprocessor included files, and OODIE symbol information. The sample has been slightly edited to reduce its size by removing included header file code and some whitespace.

```
# 1 "bob.c"
# 1 "/usr/local/lib/gcc-lib/mips-sgi-irix4.0.5/2.5.8/include/stdlib.h" 1 3
/* header deleted */

# 1 "bob.c" 2

# 1 "OODIE.h" 1

# 2 "bob.c" 2

typedef struct list {   /* list element structure */
    char a;
    struct list *next;
} LIST;
typedef LIST *LISTPTR;


void LinkPrinter (LISTPTR data, char *params)
{
static char _OOSCOPE[]="LinkPrinter";static short _OOSCOPENUM=1;
  printf("\n");
  while (data) {
        printf("%c --> ", data->a);
```

```
        data = data->next;
  }
  printf(" NULL\n");
}


int main()
{ static char _OOSCOPE[]="main";static short _OOSCOPENUM=1;
  /*pragma d*/
  LIST head; LISTPTR other;
  head.a = 'a';
  other = head.next = (LISTPTR)malloc(sizeof(LIST)) ;
  other->a = 'b';
  other = other->next = (LISTPTR)malloc(sizeof(LIST)) ;
  other->a = 'c';
  other->next = 0 ;
}

/* beginning of OODIE generated code */
#if 1
#define NOPARENT ((_OOadt *)0L)
#define NOFUNCT  ((_OOFunct)0L)
#define NONAME   ((char *)0L)

#define NOINTFUNCT ((_OOIntFunct)0L)
#ifndef NULL
#define NULL (0)
#endif

static _OOadt _OOOADT[] = {
        {"LinkedList", NOPARENT, (_OOFunct)LinkPrinter, NOFUNCT, NOINTFUNCT,
         "A simple linked list containing letters."},
        {NONAME, NOPARENT, NOFUNCT, NOFUNCT, NOINTFUNCT, NONAME}
};

static _OOvars _OO1VAR[] = {
        {"head", {"", &_OOOADT[0], NOFUNCT, NOFUNCT, NOINTFUNCT, ""}},
        {NONAME, {NONAME, NOPARENT, NOFUNCT, NOFUNCT, NOINTFUNCT, NONAME}}
};
```

```
static short _OO_OODIETABLE=0;
static char _OOSCOPE[] = "Global File Scope";
static short _OOSCOPENUM = 0;
static _OOMAPPING _OOMAP[] = {
{
        {"Global File Scope",NONAME, -1},
        _OOOADT,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
},
{
        {"main",NONAME, 0},
        NULL,
        NULL,
        _OO1VAR,
        NULL,
        NULL,
        NULL,
        NULL,
},

};

static struct _OOBGROUP _OOBPTGROUPS[] = {
        {NULL, NULL}
};

static struct _OOWGROUP _OOWATGROUPS[] = {
        {NULL, NULL}
};

static _OOINFOMAP _OOINFO = {
        &_OOMAP[0],
        &_OOWATGROUPS[0],
        &_OOBPTGROUPS[0],
        NULL,
```

```
        __FILE__,
        0, 2, 0,
        {'\000'}
};

static char _OOLANGUAGE[] = "C";
static char _OODELIMITERS[] = "\t.[]()+-%/~'=<>!@#$^|`{};:,?";
extern void (*_OOLIB()); static void *_OOlib = _OOLIB;

static _OOINFOMAP *_OOGETMAP() { return &_OOINFO; }

#endif /* __OODIE */
```

Appendix E

OODIE Debugger Command Summary (gOODIE)

This appendix will briefly summarize the new commands added to a GNU gdbdebugger to support the OODIE feature set. The following typographic conventions will be used: `typewriter` font and all lowercase indicates a command name; *italic* and starting with a capital letter indicates a command parameter; brackets ({}) indicate optional commands; vertical bar (|) indicates OR selection. For a more complete syntax description, see [11].

`aprint` *Variable*

Print the abstract value of a variable.

`alocation`

Display the current abstract scope name.

`info scope` *Scopename*

Display descriptive information about all of the current abstract scope.

`info alias` *Aliasname*

Display true symbolic name of an alias.

`mark {w|wg|wa|b|bg|ba}` *Name*

Set either abstract break or watch points by name or by group or all.

`unmark {w|wg|wa|b|bg|ba}` *Name*

Clears either abstract break or watch points by name or by group or all.

`info watchers` {*Name* | * }

Display information about a named watchpoint or points in the current abstract scope or all.

`info bpt` {*Name* | * }

Display information about a names breakpoint or points in the current abstract scope or all.

`devar` *Name*

Give a full description of a named OODIE variable.

`detype` *Name*

Give a full description of a name OODIE type.

Appendix F

Complete Example Code Listings

This appendix contains the complete source listings for some of the example codes found in Chapter 4. These code listings will be coded in C, except when noted and OODIE extensions will be highlighted by using **bold** type.

F.1 Hash Table

```
#include <stdio.h>
#include <stdlib.h>

#ifdef __OCC
#include "OODIE.h"
#endif

typedef struct _TableEntry { /* define table entry */
    char *name;
    struct _TableEntry *chain;
} TableEntry;

#define MAXENTRIES 26

/*pragma defabstract HashTable
      printer HashPrint
      description
         "Hash table using first character for hash value\n  \
         and using chaining to resolve collisions."
   end */
```

```
typedef struct {
  TableEntry *baseTable[MAXENTRIES];
} HASHTABLE;


#ifdef __OODIE

void EntryPrint(TableEntry *entry, char *params)
{
  if (params == NULL)
      printf("%s\n", entry->name);
  else
      printf("%s", entry->name);
}

void HashPrint(HASHTABLE *ht, char *params)
{
  short i,flg=0;
  TableEntry *entry;
  for(i=0;i<MAXENTRIES;i++) {
    if ((entry = ht->baseTable[i])) {
        printf("[%2d] :", i);   flg=1;
                for(; entry != NULL; entry = entry->chain) {
                    printf(" --> ");
                        EntryPrint(entry, (char *)1);
                }
        printf("\n");
    }
  }
  if (!flg) printf("\n**EMPTY**\n");
}

#endif


/* compute hash function for data */
#define HASHFUNCTION(data)  (toupper(*(data)) - 'A')
#define INITHASHTABLE(ht) \
        { short i; for(i=0;i<MAXENTRIES;i++) (ht)->baseTable[i]=NULL; }
```

```
void AddEntry(HASHTABLE *hash, char *data)
{
  TableEntry *entry;
  TableEntry *newentry = (TableEntry *)malloc(sizeof(struct _TableEntry));
  int index = HASHFUNCTION(data);

  /* fill-in entry */
  newentry->name = data;
  newentry->chain = NULL;

  /* time to insert into table */
  if ((entry = hash->baseTable[index])) {
      /* use chaining */
     for( ; entry->chain != NULL; entry = entry->chain) ;
     entry->chain = newentry;
  } else {  /* no entry at table entry */
    hash->baseTable[index] = newentry;
  }
}


/*specific macro for main function */
#define ADDTOTABLE(str) AddEntry(&ht, str);

int main()
{
   /*pragma defvar ht isa HashTable */
  HASHTABLE ht;
  INITHASHTABLE(&ht);

  ADDTOTABLE("Hello"); ADDTOTABLE("JOHN");
  ADDTOTABLE("Mike");  ADDTOTABLE("Zebra");
  ADDTOTABLE("Jackson"); ADDTOTABLE("mother");
  ADDTOTABLE("heather"); ADDTOTABLE("sandra");
  ADDTOTABLE("Victor");  ADDTOTABLE("Victoria");
  ADDTOTABLE("michelle");
}
```

F.2 Bitmap Code

```
#include <stdlib.h>
#include <stdio.h>
#ifdef __OCC
#include "OODIE.h"
#endif

#define xlogo16_width 16
#define xlogo16_height 16
static unsigned char xlogo16_bits[] = {
   0x0f, 0x80, 0x1e, 0x80, 0x3c, 0x40, 0x78, 0x20, 0x78, 0x10, 0xf0, 0x08,
   0xe0, 0x09, 0xc0, 0x05, 0xc0, 0x02, 0x40, 0x07, 0x20, 0x0f, 0x20, 0x1e,
   0x10, 0x1e, 0x08, 0x3c, 0x04, 0x78, 0x02, 0xf0};


#define xlogo32_width 32
#define xlogo32_height 32
static char xlogo32_bits[] = {
   0xff, 0x00, 0x00, 0xc0, 0xfe, 0x01, 0x00, 0xc0, 0xfc, 0x03, 0x00, 0x60,
   0xf8, 0x07, 0x00, 0x30, 0xf8, 0x07, 0x00, 0x18, 0xf0, 0x0f, 0x00, 0x0c,
   0xe0, 0x1f, 0x00, 0x06, 0xc0, 0x3f, 0x00, 0x06, 0xc0, 0x3f, 0x00, 0x03,
   0x80, 0x7f, 0x80, 0x01, 0x00, 0xff, 0xc0, 0x00, 0x00, 0xfe, 0x61, 0x00,
   0x00, 0xfe, 0x31, 0x00, 0x00, 0xfc, 0x33, 0x00, 0x00, 0xf8, 0x1b, 0x00,
   0x00, 0xf0, 0x0d, 0x00, 0x00, 0xf0, 0x0e, 0x00, 0x00, 0x60, 0x1f, 0x00,
   0x00, 0xb0, 0x3f, 0x00, 0x00, 0x98, 0x7f, 0x00, 0x00, 0x98, 0x7f, 0x00,
   0x00, 0x0c, 0xff, 0x00, 0x00, 0x06, 0xfe, 0x01, 0x00, 0x03, 0xfc, 0x03,
   0x80, 0x01, 0xfc, 0x03, 0xc0, 0x00, 0xf8, 0x07, 0xc0, 0x00, 0xf0, 0x0f,
   0x60, 0x00, 0xe0, 0x1f, 0x30, 0x00, 0xe0, 0x1f, 0x18, 0x00, 0xc0, 0x3f,
   0x0c, 0x00, 0x80, 0x7f, 0x06, 0x00, 0x00, 0xff};
/*pragma begin
   defabstract BitMap
      printer DRAWMAP
      description "ASCII image of an X bitmap"
   end
   defabstract BitMap16 : BitMap
      printer DrawMap16
      description "16 x 16 image"
   end
   defabstract BitMap32 : BitMap
```

```
      printer DrawMap32
   description "32 x 32 image"
   end
   defvar xlogo16_bits isa BitMap16
   defvar xlogo32_bits isa BitMap32
   sourcealias x16 = xlogo16_bits; x32 = xlogo32_bits;
 end */

#ifdef __OODIE

void DRAWMAP(char *image, char *params)
{
  int w,h;
  register short i,k;
  sscanf(params, "%d %d", &w, &h);
  if ((w==0) || (h==0)) {
    printf("Expected width height parameters to be non-zero\n\n");
    return;
  }

  k=(w/8)*h;
  for(i=0;i<k;i++, image++) {
    register short m;
    for(m=0;m<8;m++) {
      if ((*image) & (1<<m)) printf("X");
      else printf(" ");
    }
    if (((i+1) % (w/8)) == 0) printf("\n");
  }
}

void DrawMap16(char *image, char *params)
{
  char *bfr = "16 16";
  DRAWMAP(image, bfr);
}

void DrawMap32(char *image, char *params)
{
  char *bfr = "32 32";
  DRAWMAP(image, bfr);
```

```
}

#endif


int main()
{
  int i;
  i=i*9+3;  /* just here for a break point */
}
```

F.3 Yacc Code

F.3.1 Yacc Source Code

```
%token id
%left '+'
%left '*'
%{
#include <stdio.h>
#include <ctype.h>

#ifdef __OCC
#include "OODIE.h"
#endif

int
yylex()
{
  extern int yylval;
  int        c;

  do { c = getchar(); } while (isspace(c));
  yylval = (isupper(c))?(tolower(c)):(c);
  return (isalpha(c))?(id):(c);
}

main() { yyparse(); }
yyerror(s) char *s; { puts(s); }
```

```
int T = 'A';
Binop(a,op,b)
{
  printf("%c = %c %c %c\n",T,a,op,b);
  return T++;
}
%}
%%

E : E '+' E
  {
     /*pragma describe it "Plus production rule" */

     $$ = Binop($1,$2,$3);
  }

  | E '*' E
  {
     /*pragma describe it "Multiply production rule" */

     $$ = Binop($1,$2,$3);
  }

  | id
  ;
```

F.3.2 Yacc Generated C Code

```
# define id 257

#include <stdio.h>
#include <ctype.h>

#ifdef __OCC
#include "OODIE.h"
#endif

int
yylex()
{
```

```
  extern int yylval;
  int           c;

  do { c = getchar(); } while (isspace(c));
  yylval = (isupper(c))?(tolower(c)):(c);
  return (isalpha(c))?(id):(c);
}
```

/*pragma language YACC " !@#&*()[]" */

```
main() { yyparse(); }
yyerror(s) char *s; { puts(s); }

int T = 'A';
Binop(a,op,b)
{
  printf("%c = %c %c %c\n",T,a,op,b);
  return T++;
}
#define yyclearin yychar = -1
#define yyerrok yyerrflag = 0
extern int yychar;
extern int yyerrflag;
#ifndef YYMAXDEPTH
#define YYMAXDEPTH 150
#endif
#ifndef YYSTYPE
#define YYSTYPE int
#endif
YYSTYPE yylval, yyval;
typedef int yytabelem;
# define YYERRCODE 256
yytabelem yyexca[] ={
-1, 1,
        0, -1,
        -2, 0,
        };
# define YYNPROD 4
# define YYLAST 10
yytabelem yyact[]={
```

```
     2,     4,     3,     4,     1,     0,     0,     0,     5,     6 };
yytabelem yypact[]={

  -257,   -41, -1000,  -257,  -257,   -39, -1000 };
yytabelem yypgo[]={

     0,     4 };
yytabelem yyr1[]={

     0,     1,     1,     1 };
yytabelem yyr2[]={

     0,     7,     7,     2 };
yytabelem yychk[]={

 -1000,    -1,   257,    43,    42,    -1,    -1 };
yytabelem yydef[]={

     0,    -2,     3,     0,     0,     1,     2 };
typedef struct { char *t_name; int t_val; } yytoktype;
#ifndef YYDEBUG
#       define YYDEBUG  0       /* don't allow debugging */
#endif

#if YYDEBUG

yytoktype yytoks[] =
{
        "id",   257,
        "+",    43,
        "*",    42,
        "-unknown-",    -1      /* ends search */
};

char * yyreds[] =
{
        "-no such reduction-",
        "E : E '+' E",
        "E : E '*' E",
        "E : id",
```

```
};
#endif /* YYDEBUG */
/*
 *      Copyright 1987 Silicon Graphics, Inc. - All Rights Reserved
 */

/* #ident       "@(#)yacc:yaccpar       1.10" */
#ident  "$Revision: 1.5 $"

/*
** Skeleton parser driver for yacc output
*/

/*
** yacc user known macros and defines
*/
#define YYERROR         goto yyerrlab
#define YYACCEPT        return(0)
#define YYABORT         return(1)
#define YYBACKUP( newtoken, newvalue )\
{\
        if ( yychar >= 0 || ( yyr2[ yytmp ] >> 1 ) != 1 )\
        {\
                yyerror( "syntax error - cannot backup" );\
                goto yyerrlab;\
        }\
        yychar = newtoken;\
        yystate = *yyps;\
        yylval = newvalue;\
        goto yynewstate;\
}
#define YYRECOVERING()  (!!yyerrflag)
#ifndef YYDEBUG
#       define YYDEBUG  1       /* make debugging available */
#endif

/*
** user known globals
*/
int yydebug;                    /* set to 1 to get debugging */
```

```
/*
** driver internal defines
*/
#define YYFLAG          (-1000)

/*
** global variables used by the parser
*/
YYSTYPE yyv[ YYMAXDEPTH ];      /* value stack */
int yys[ YYMAXDEPTH ];          /* state stack */

YYSTYPE *yypv;                  /* top of value stack */
int *yyps;                      /* top of state stack */

int yystate;                    /* current state */
int yytmp;                      /* extra var (lasts between blocks) */

int yynerrs;                    /* number of errors */
int yyerrflag;                  /* error recovery flag */
int yychar;                     /* current input token number */



/*
** yyparse - return 0 if worked, 1 if syntax error not recovered from
*/
int
yyparse()
{
        register YYSTYPE *yypvt;        /* top of value stack for $vars */

        /*
        ** Initialize externals - yyparse may be called more than once
        */
        yypv = &yyv[-1];
        yyps = &yys[-1];
        yystate = 0;
        yytmp = 0;
        yynerrs = 0;
```

```
        yyerrflag = 0;
        yychar = -1;

        goto yystack;
        {
                register YYSTYPE *yy_pv;    /* top of value stack */
                register int *yy_ps;        /* top of state stack */
                register int yy_state;      /* current state */
                register int  yy_n;         /* internal state number info */

                /*
                ** get globals into registers.
                ** branch to here only if YYBACKUP was called.
                */
        yynewstate:
                yy_pv = yypv;
                yy_ps = yyps;
                yy_state = yystate;
                goto yy_newstate;

                /*
                ** get globals into registers.
                ** either we just started, or we just finished a reduction
                */
        yystack:
                yy_pv = yypv;
                yy_ps = yyps;
                yy_state = yystate;

                /*
                ** top of for (;;) loop while no reductions done
                */
        yy_stack:
                /*
                ** put a state and value onto the stacks
                */
#if YYDEBUG
                /*
                ** if debugging, look up token value in list of value vs.
                ** name pairs.  0 and negative (-1) are special values.
```

```
                ** Note: linear search is used since time is not a real
                ** consideration while debugging.
                */
                if ( yydebug )
                {
                        register int yy_i;

                        printf( "State %d, token ", yy_state );
                        if ( yychar == 0 )
                                printf( "end-of-file\n" );
                        else if ( yychar < 0 )
                                printf( "-none-\n" );
                        else
                        {
                                for ( yy_i = 0; yytoks[yy_i].t_val >= 0;
                                        yy_i++ )
                                {
                                        if ( yytoks[yy_i].t_val == yychar )
                                                break;
                                }
                                printf( "%s\n", yytoks[yy_i].t_name );
                        }
                }
#endif /* YYDEBUG */
                if ( ++yy_ps >= &yys[ YYMAXDEPTH ] )    /* room on stack? */
                {
                        yyerror( "yacc stack overflow" );
                        YYABORT;
                }
                *yy_ps = yy_state;
                *++yy_pv = yyval;

                /*
                ** we have a new state - find out what to do
                */
        yy_newstate:
                if ( ( yy_n = yypact[ yy_state ] ) <= YYFLAG )
                        goto yydefault;         /* simple state */
#if YYDEBUG
                /*
```

```
                ** if debugging, need to mark whether new token grabbed
                */
                yytmp = yychar < 0;
#endif
                if ( ( yychar < 0 ) && ( ( yychar = yylex() ) < 0 ) )
                        yychar = 0;             /* reached EOF */
#if YYDEBUG
                if ( yydebug && yytmp )
                {
                        register int yy_i;

                        printf( "Received token " );
                        if ( yychar == 0 )
                                printf( "end-of-file\n" );
                        else if ( yychar < 0 )
                                printf( "-none-\n" );
                        else
                        {
                                for ( yy_i = 0; yytoks[yy_i].t_val >= 0;
                                        yy_i++ )
                                {
                                        if ( yytoks[yy_i].t_val == yychar )
                                                break;
                                }
                                printf( "%s\n", yytoks[yy_i].t_name );
                        }
                }
#endif /* YYDEBUG */
                if ( ( ( yy_n += yychar ) < 0 ) || ( yy_n >= YYLAST ) )
                        goto yydefault;
                if ( yychk[ yy_n = yyact[ yy_n ] ] == yychar )  /*valid shift*/
                {
                        yychar = -1;
                        yyval = yylval;
                        yy_state = yy_n;
                        if ( yyerrflag > 0 )
                                yyerrflag--;
                        goto yy_stack;
                }
```

```
        yydefault:
                if ( ( yy_n = yydef[ yy_state ] ) == -2 )
                {
#if YYDEBUG
                        yytmp = yychar < 0;
#endif
                        if ( ( yychar < 0 ) && ( ( yychar = yylex() ) < 0 ) )
                                yychar = 0;             /* reached EOF */
#if YYDEBUG
                        if ( yydebug && yytmp )
                        {
                                register int yy_i;

                                printf( "Received token " );
                                if ( yychar == 0 )
                                        printf( "end-of-file\n" );
                                else if ( yychar < 0 )
                                        printf( "-none-\n" );
                                else
                                {
                                        for ( yy_i = 0;
                                                yytoks[yy_i].t_val >= 0;
                                                yy_i++ )
                                        {
                                                if ( yytoks[yy_i].t_val
                                                        == yychar )
                                                {
                                                        break;
                                                }
                                        }
                                        printf( "%s\n", yytoks[yy_i].t_name );
                                }
                        }
#endif /* YYDEBUG */
                        /*
                        ** look through exception table
                        */
                        {
                                register int *yyxi = yyexca;
```

```
                        while ( ( *yyxi != -1 ) ||
                                ( yyxi[1] != yy_state ) )
                        {
                                yyxi += 2;
                        }
                        while ( ( *(yyxi += 2) >= 0 ) &&
                                ( *yyxi != yychar ) )
                                ;
                        if ( ( yy_n = yyxi[1] ) < 0 )
                                YYACCEPT;
                }
        }

        /*
        ** check for syntax error
        */
        if ( yy_n == 0 )        /* have an error */
        {
                /* no worry about speed here! */
                switch ( yyerrflag )
                {
                case 0:         /* new error */
                        yyerror( "syntax error" );
                        goto skip_init;
                yyerrlab:
                        /*
                        ** get globals into registers.
                        ** we have a user generated syntax type error
                        */
                        yy_pv = yypv;
                        yy_ps = yyps;
                        yy_state = yystate;
                        yynerrs++;
                skip_init:
                case 1:
                case 2:         /* incompletely recovered error */
                                /* try again... */
                        yyerrflag = 3;
                        /*
                        ** find state where "error" is a legal
```

```
                                ** shift action
                                */
                                while ( yy_ps >= yys )
                                {
                                        yy_n = yypact[ *yy_ps ] + YYERRCODE;
                                        if ( yy_n >= 0 && yy_n < YYLAST &&
                                              yychk[yyact[yy_n]] == YYERRCODE)
                                                /*
                                                ** simulate shift of "error"
                                                */
                                                yy_state = yyact[ yy_n ];
                                                goto yy_stack;
                                        }
                                        /*
                                        ** current state has no shift on
                                        ** "error", pop stack
                                        */
#if YYDEBUG
#       define _POP_ "Error recovery pops state %d, uncovers state %d\n"
                                        if ( yydebug )
                                                printf( _POP_, *yy_ps,
                                                        yy_ps[-1] );
#       undef _POP_
#endif
                                        yy_ps--;
                                        yy_pv--;
                                }
                                /*
                                ** there is no state on stack with "error" as
                                ** a valid shift.  give up.
                                */
                                YYABORT;
                        case 3:         /* no shift yet; eat a token */
#if YYDEBUG
                                /*
                                ** if debugging, look up token in list of
                                ** pairs.  0 and negative shouldn't occur,
                                ** but since timing doesn't matter when
                                ** debugging, it doesn't hurt to leave the
                                ** tests here.
```

```
                                */
                                if ( yydebug )
                                {
                                        register int yy_i;

                                        printf( "Error recovery discards " );
                                        if ( yychar == 0 )
                                              printf( "token end-of-file\n" );
                                        else if ( yychar < 0 )
                                                printf( "token -none-\n" );
                                        else
                                        {
                                                for ( yy_i = 0;
                                                     yytoks[yy_i].t_val >= 0;
                                                     yy_i++ )
                                                {
                                                        if ( yytoks[yy_i].t_val
                                                                == yychar )
                                                        {
                                                                break;
                                                        }
                                                }
                                                printf( "token %s\n",
                                                        yytoks[yy_i].t_name );
                                        }
                                }
#endif /* YYDEBUG */
                                if ( yychar == 0 )          /* reached EOF. quit */
                                        YYABORT;
                                yychar = -1;
                                goto yy_newstate;
                        }
                }/* end if ( yy_n == 0 ) */
                /*
                ** reduction by production yy_n
                ** put stack tops, etc. so things right after switch
                */
#if YYDEBUG
                /*
                ** if debugging, print the string that is the user's
```

```
                ** specification of the reduction which is just about
                ** to be done.
                */
                if ( yydebug )
                        printf( "Reduce by (%d) \"%s\"\n",
                                yy_n, yyreds[ yy_n ] );
#endif
                yytmp = yy_n;                   /* value to switch over */
                yypvt = yy_pv;                  /* $vars top of value stack */
                /*
                ** Look in goto table for next state
                ** Sorry about using yy_state here as temporary
                ** register variable, but why not, if it works...
                ** If yyr2[ yy_n ] doesn't have the low order bit
                ** set, then there is no action to be done for
                ** this reduction.  So, no saving & unsaving of
                ** registers done.  The only difference between the
                ** code just after the if and the body of the if is
                ** the goto yy_stack in the body.  This way the test
                ** can be made before the choice of what to do is needed.
                */
                {
                        /* length of production doubled with extra bit */
                        register int yy_len = yyr2[ yy_n ];

                        if ( !( yy_len & 01 ) )
                        {
                                yy_len >>= 1;
                                yyval = ( yy_pv -= yy_len )[1]; /* $$ = $1 */
                                yy_state = yypgo[ yy_n = yyr1[ yy_n ] ] +
                                        *( yy_ps -= yy_len ) + 1;
                                if ( yy_state >= YYLAST ||
                                        yychk[ yy_state =
                                        yyact[ yy_state ] ] != -yy_n )
                                {
                                        yy_state = yyact[ yypgo[ yy_n ] ];
                                }
                                goto yy_stack;
                        }
                        yy_len >>= 1;
```

```
                        yyval = ( yy_pv -= yy_len )[1]; /* $$ = $1 */
                        yy_state = yypgo[ yy_n = yyr1[ yy_n ] ] +
                                *( yy_ps -= yy_len ) + 1;
                        if ( yy_state >= YYLAST ||
                             yychk[ yy_state = yyact[ yy_state ] ] != -yy_n )
                        {
                             yy_state = yyact[ yypgo[ yy_n ] ];
                        }
                }
                                        /* save until reenter driver code */
                yystate = yy_state;
                yyps = yy_ps;
                yypv = yy_pv;
        }
        /*
        ** code supplied by user is placed in this switch
        */
        switch( yytmp )
        {

case 1:
{

   /*pragma scope "E : E '+' E" */
   /*pragma describe it "Plus Rule" */
   /*pragma defbreak Plus "Plus rule" */
   /*pragma sourcealias $$=yyval;$1=yypvt[-2];$2=yypvt[-1];$3=yypvt[-
0]; */

                yyval = Binop(yypvt[-2],yypvt[-1],yypvt[-0]);

   /*pragma scope end */

} break;
case 2:
{

   /*pragma scope " — E '*' E" */
   /*pragma describe it "Multiply Rule"*/
   /*pragma defbreak Multiply "Multiply rule" */
   /*pragma sourcealias $$=yyval;$1=yypvt[-2];$2=yypvt[-1];$3=yypvt[-
0]; */
```

```
                yyval = Binop(yypvt[-2],yypvt[-1],yypvt[-0]);
   /*pragma scope end */
} break;
        }
        goto yystack;           /* reset registers in driver code */
}
```