

A Knowledge Based Program Diagnostic System

by

Eric J. Byrne

B.S. University of Nebraska - Lincoln, 1983

A MASTER'S THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

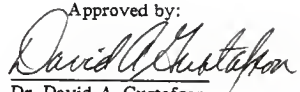
Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1988

Approved by:



Dr. David A. Gustafson
Major Professor

LD
2668
.74
CMSC
1788
1597
C. 2

AL1208 232377

Table of Contents

Table of Contents	ii
List of Figures	iv
Chapter 1 : Introduction	1-1
1.1 Terminology	1-2
1.2 A Knowledge Based Approach	1-4
1.3 Hypothesis	1-6
1.4 Contents of Thesis	1-6
Chapter 2 : Research In Debugging	2-1
2.1 How Programmers Debug	2-1
2.2 Knowledge Based Program Debuggers	2-3
2.3 The Amber Debugger	2-6
Chapter 3 : Requirements for Amber	3-1
3.1 Types of Errors and Faults	3-1
3.2 Knowledge Requirements	3-6
Chapter 4 : Design for the Amber Debugger	4-1
4.1 System Overview	4-1
4.2 Rule Groups	4-5
4.2.1 Symptom Analyser	4-5
4.2.2 Select Error	4-7
4.2.3 Locate Error	4-7
4.2.4 Explain Error	4-10
4.2.5 Locate/Determine Fault	4-11
4.2.6 Explain Fault	4-12
4.3 Supporting Components	4-12
4.3.1 User Interface	4-12
4.3.2 Source Code Reader	4-13
4.3.3 Execution Oracle	4-14
4.4 Fact Sets	4-15
4.4.1 Symptoms	4-15
4.4.2 Suspected Errors	4-15
4.4.3 Possible Error	4-15
4.4.4 Program Knowledge	4-16
4.4.5 Location of Error	4-20

4.4.6 Execution Knowledge	4-20
4.4.7 Location of Fault	4-20
4.4.8 Specific Fault	4-21
Chapter 5 : Introduction to YAPS	5-1
5.1 Facts in YAPS	5-1
5.2 Production Rules in YAPS	5-2
Chapter 6 : Implementation of Amber	6-1
6.1 Starting Amber	6-2
6.2 The User Interface	6-2
6.3 Source Code Reader	6-4
6.3.1 C Scanner	6-5
6.3.1.1 Running the Scanner	6-5
6.3.1.2 Scanner Output	6-8
6.3.1.3 Scanning Procedure	6-9
6.3.1.4 Scanner Components	6-10
6.3.2 Representing Program Knowledge	6-11
6.3.2.1 Program Information File	6-11
6.3.2.2 Representing C	6-13
6.3.2.3 Source Code Fact File	6-18
6.4 Symptom Analyzer	6-19
6.5 Select Error	6-24
6.6 Locate Error	6-27
6.7 Explain Error	6-37
6.8 Locate/Detect Fault	6-38
Chapter 7 : Conclusions	7-1
Bibliography	bib-1
Appendix A : How to Use Amber	A-1
Appendix B : Using Amber - Examples	B-1
Appendix C : YAPS Rules and Lisp Functions	C-1
Appendix D : Scanner Source Code	D-1

LIST OF FIGURES

Fig. 4.1 : Debugging Procedure	4-2
Fig. 4.2 : Components of Amber	4-3
Fig. 4.3 : Components for Locating Faults	4-4
Fig. 4.4 : Abstract Syntax for C.....	4-19
Fig. 5.1 : Example YAPS Rule	5-3
Fig. 5.2 : Another YAPS Rule.....	5-5
Fig. 6.1 : Template for Prompts	6-3
Fig. 6.2 : Example C Program.....	6-13
Fig. 6.3 : Program Information File	6-13
Fig. 6.4 : List Representation of C Statements	6-15
Fig. 6.5 : List Representation of C Expressions	6-16
Fig. 6.6 : Representation of a Statement Block.....	6-17
Fig. 6.7 : Source Code Fact File	6-19
Fig. 6.8 : Symptom Menu	6-20
Fig. 6.9 : Condition Pattern for Rule "sys_abort"	6-21
Fig. 6.10 : List of Symptoms	6-22
Fig. 6.11 : List of Suspected Errors.....	6-23
Fig. 6.12 : Core Dump Messages and Keywords	6-24
Fig. 6.13 : select_error Rule	6-25
Fig. 6.14 : Expression Handling Rule	6-28
Fig. 6.15 : Statement Handling Rule	6-31
Fig. 6.16 : Execution Flow Handling Rule	6-35

Acknowledgements

I would like to thank my parents for the encouragement and support they have given me. Without their encouragement I never could have started my master's.

I would like to thank Dr. David Gustafson who began each of our weekly meetings with the question "Well, what have you gotten done this week?" thus encouraging me to get something, no matter how trivial, done each week.

Finally, I would like to thank my friend Bob Widhalm who told me "If you don't go back in two years, you won't go back." Thanks for being wrong Bob.

Chapter 1

Introduction

"To err is human" is an old adage whose truth is shown throughout the development and life of a computer program. Despite the increasing use of software engineering techniques such as requirements and design specifications, formal specification techniques, walk-throughs, and code-reviews, most programs still contain errors at the end of the development cycle. Testing and debugging are the most time consuming and costly activities of program development [DYER87]. Testing is the process of checking that a program does function correctly. When a test shows that a program is not functioning correctly, then the program must be debugged.

Debugging consists of taking a program that does not run correctly (i.e. has a software failure), locating the cause of the failure (i.e the fault), and then rewriting the program so that it does run correctly. To accomplish this a programmer must identify the symptoms of a software failure and formulate an idea about what error could cause the failure. Next, the programmer must track the error to a specific set of statements and the specific cause of the error (the fault) must be found. The programmer must understand the exact nature of the fault. What can trigger the fault? How is it an error? How does it affect the operation of the program? Finally, the programmer must rewrite the faulty statements so that the program runs correctly. Locating the source of the failure and understanding it can consume as much as 95% of the debugging effort

[MYERS79].

The problem of debugging a program can be viewed as a diagnostic activity, at least to the point of locating the fault. The suitability of expert systems as diagnostic tools has been shown in several domains, such as medicine ([POPLK75], [SHORT76]) and hardware maintenance ([HARTL84], [LAFFE86]).

A knowledge based program debugger (KBPD) can assist a programmer with some or all the tasks involved in debugging. A KBPD can suggest possible errors for a given set of symptoms and can locate the source of an error for certain types of software failure. A KBPD may even be able to determine under what conditions a fault is triggered in a program and suggest possible actions to remove the fault.

This thesis presents the design and implementation of a knowledge based program debugger called Amber. The name Amber was selected for no better reason than the fact that samples of the gemstone amber have been found with "bugs" trapped in the stone. Amber can also be found in the phrase AutoMated deBugGER.

1.1. Terminology

Before continuing, it may be beneficial to define the meaning of terms used in this report. Programmers have been debugging programs for a long time and terms such as error, cause, bug, fault, and failure, are used frequently and often interchangeably. The terms defined below are used to discuss the concept and functionality of a KBPD. Many of these terms are defined in the "IEEE Standard Glossary of Software Engineering Terminology" [IEEE83].

Debugging :

Debugging is an attempt to isolate the source of a failure in a program. It is the process of locating the error and fault that caused a program failure to occur.

Error :

An error is an abstraction of different faults that produce similar results. For example if a program does not terminate (This is a symptom.) then two suspected errors are that the program contains either an infinite loop or a deadlock. The cause of an infinite loop is a fault in the software, such as no exit logic for the loop, or an exit condition that will never be satisfied. These two different faults both produce the same error. Other examples of errors are computational error, domain error, input error, etc. Each error can be produced by a variety of different faults.

Failure :

A software failure is an unacceptable result produced during the execution of a computer program. If a fault is invoked during the execution of a program then a failure will occur. The purpose of software testing is to ensure that a program will not experience a failure.

Fault :

A fault is a specific manifestation of an error. When the processing of some item of data results in a program failure, the presence of a fault becomes known. A fault is the cause of a software failure. For example, a program may produce incorrect output because a print statement has an incorrect format specification. The incorrect format specification is the fault.

Fault Location :

The fault location is the location of a fault in the program source code. This refers to the line or lines of code containing the fault.

KBPD :

Is an abbreviation of the phrase "Knowledge Based Program Debugger." A KBPD

differs from traditional debugging tools by containing knowledge about how to locate a fault in a program. In this report, the term KBPD will be used to refer to knowledge based program debuggers in general and the name Amber will be used to refer to the system implemented as part of this thesis project.

Symptom :

A symptom is a phenomenon or circumstance that suggests the presence of a software failure. Examples of symptoms are incorrect output, non-termination, system abort, etc. A symptom can serve as a clue to the nature of an error.

1.2. A Knowledge Based Approach

A difficult problem in the task of debugging a program is determining what happened during the execution of the program. Where did the error occur? Why did the error occur? What triggered the error? Information about what happened during the execution is often generated by inserting print statements in (hopefully) strategic sections of code. The purpose of a print statement is to display information such as, whether the section of code was entered, the value of variable x, etc. Many systems now have what are called "break-and-see" or run-time debuggers (such as sdb or dbx on UNIX systems). These debugging tools allow a programmer to watch the execution of a program on a statement by statement basis, stop the execution at any time, and display the value of any variable. These tools allow the programmer to collect information about the execution of the program, but the programmer must still do all the thinking. It is the programmer who must analyze the information, develop an idea of what went wrong, trace the source of the error and realize when the problem has been found. Programmers can get on the wrong track and waste much time trying to locate the source of an error when they suspect the wrong thing or fail to understand what is really happening.

An expert system for debugging programs, a KBPD, can assist the programmer in the debugging process. The main purpose of a KBPD is to help analyze information and locate an error for the programmer. This relieves the programmer of much of the burden of the debugging process.

There are several advantages to applying a production system approach to the problem of program debugging. The foremost advantage comes from the standard design of production systems: storing knowledge as rules and facts. This design simplifies the task of adding and modifying knowledge. Encoding knowledge into a set of IF-THEN rules is simpler than encoding knowledge using a procedural language.

A second advantage is in the area of the user interface. All the information gathered or deduced by a production system is stored as facts. These facts can be made available to the user interface for display to the user and allow the KBPD to describe the status of the debugging effort. The user can examine what line of reasoning the KBPD is using, the set of suspected errors, facts known, etc. The user interface can allow the user to control or influence the activities of the KBPD. If the user sees that the debugging is progressing in a direction that is not promising, the user can direct the KBPD's action and thus control the direction or scope of the tool's debugging effort.

A final major advantage is in the area of what we call "expert insight" or "aha" rules. One advantage of experts is that over the years they have solved many problems. When experts encounter a new problem, they compare it with unusual problems they have solved before. They don't waste time with the straightforward approaches when they can identify the specific problem. The expert can recognize when a new problem behaves similarly to a problem that has been solved before. Remembering what caused the earlier problem the expert can begin searching for something similar in the current program. Unusual problems may be hard to solve using straightforward approaches

and thus this expert insight saves much effort. These "expert insight" or "aha" rules are important in debugging and can be implemented easily in a production system.

1.3. Hypothesis

Faults in a program can range from syntax errors, incorrect use of the programming language, subtle typos, to faulty logic or algorithms encoded into the program. Syntax errors are easily detected by compilers. Run time or execution errors, however, may lurk in a program waiting for the correct sequence of events to occur. Syntax errors can be easily corrected given knowledge of the programming language syntax. Some types of run-time errors can be corrected given knowledge of the programming language semantics and common rules of programming, such as "don't divide by zero." Other classes of run-time errors can only be corrected by understanding what task a program is intended to do and determining how its run-time actions differ from the intended actions.

The goal of this research is to develop a knowledge based program debugger for the C program language. The debugger will make use of knowledge about the C language and common programming rules to help a programmer locate an error and sometimes determine the fault causing the error. The debugger will be able to reason about errors that occur when a program violates the semantics of C. The debugger will not be able to reason about what a program is intended to do and it will not be concerned with syntax errors.

1.4. Contents of Thesis

In the next chapter a review of other research efforts into knowledge based program debuggers is given. This chapter discusses the approaches and systems developed in other research projects. The proposed capabilities or requirements for Amber, the

knowledge base debugger built for this effort, are presented in chapter 3. The design for Amber is presented in chapter 4. That chapter discusses the structure of the system and the functionality of the individual components. Chapter 5 is a brief introduction to YAPS (Yet Another Production System) which is the language used to implement the rule base for Amber. This chapter can be skipped if the reader is already familiar with YAPS. Finally, chapter 6 contains a discussion of the implementation of Amber. Conclusions are given in chapter 7. There are several appendices at the end of this work. Appendix A is a user's guide to Amber. Appendix B shows several examples of debugging sessions with Amber. The source code for Amber is contained in appendices C and D.

Chapter 2

Research In Debugging

This chapter provides background information about the development of debugging expertise and knowledge based program debugging systems. The issue of how programmers debug programs has never generated much research interest. However, over the last fifteen years there have been a few studies into this topic. A summary of three of these studies is given first. Debuggers that use knowledge about debugging began to appear around 1980. Over the last eight years papers about several systems have been published. Descriptions of several of these systems is given in the second part of this chapter. A good survey paper about knowledge based debugger systems was written by Seviaora [SEVIO87].

2.1. How Programmers Debug

Despite the large amount of time and effort consumed by debugging a program, little research has been conducted in this area, as compared to other software life cycle tasks. Debugging itself is a skill that a programmer learns and develops over time. There have been several studies about how programmers debug programs. A study done by John Gould from IBM [GOULD75] used ten experienced Fortran programmers and examined how they debugged a set of programs. The study concluded that programmers can and do make use of several sources of information in debugging. Sources are previous

experience, aptitude, motivation, knowledge of the application, input data, output data, meaningful names for variables, comment statements, and values of variables at run time. The study also reported that there tends to be an order in which programmers look for errors: syntactic, grammatical errors that compilers do not detect, and substantive bugs. Programmers seem to use a general strategy of selecting a particular debugging tactic, finding a clue, and developing a hypothesis about a bug. Other results from the study:

- Bugs in assignment statements were the most difficult to debug primarily because programmers usually had to get into the substance of the program, other errors such as over-stepping the bounds of an array were more easily spotted.
- Programmers tended to use an on-line interactive debugger only when they could not find the bug or when they thought the debugger would display certain information.
- Programmers who were fast at debugging tended to find more bugs and make fewer errors than programmers who were slow at debugging.
- Programmers focus their attention on a local region of code, which is defined by both geographic factors (e.g. neighboring statements) and conceptual factors.

Another study on how programmers debug was reported by Mark Weiser [WEISE82]. This study determined that programmers use slicing techniques to debug programs. Program slicing is the process of stripping a program of statements that have no influence on a given variable at a given statement. The result is a slice that is a set of statements related by their flow of data. Weiser cited several other studies to support the idea that programmers start at the point in a program where an error first becomes manifest and then trace the execution backwards from there. The study concluded that slicing has potential in debugging by allowing a programmer to trace the usage of a

variable that has an incorrect value. An interesting point from this study was that programmers do not necessary view a program in a way that conforms to the program's textual or modular structure.

Vessey at the University of Queensland published a study [VESSE85a] that investigated expert and novice debugging processes to determine the relevance of situation-dependent problem solving to debugging expertise. The debugging process was defined as problem determination, gaining familiarity with the program functions and structure, exploring program execution and/or program control, and repairing (and confirming) the error. The conclusion of the study was that expertise is associated with two data-driven or situation-dependent characteristics: breath-first search for the error and the ability to create a model of normal program functioning. The problem with novices was that they became set on a given hypothesis and began ignoring clues that suggested the hypothesis was wrong. Some conclusions from the study are:

- Problem solvers who engage in situation-dependent problem solving tend to spend more time in understanding and formulating a problem than do novices.
- Experts displayed more clue finding activity than novices.
- Whether subjects initially examined the output of the program had no effect on problem solving.
- Programmers who engaged in breath-first search for the error but who did not formulate a model of the program structure and conceive of the error within that context were likely to make mistakes.

2.2. Knowledge Based Program Debuggers

An early KBPD systems was PUDSY (Program Understanding and Debugging SYstem) [LUKEY80]. Lukey proposed a theory of program understanding that included: seg-

mentation of a program, descriptions of the flow of information, recognition of debugging clues, and descriptions of the values of variables. Lukey's theory was incorporated into PUDSY which was able to understand and debug simple PASCAL programs.

PUDSY took the source code of a program and analyzed it to develop an "understanding" or description of the program. The program was segmented into "chunks" of code that formed useful units of analysis. A description of flow between chunks was then generated. Assertions about the purpose of each chunk were derived by using program schema (plans). To a limited extent PUDSY could use meaningful variables names to help match chunks to schema. Debugging was done by comparing the assertions or description of the source built by PUDSY against a specification for a program. Any important discrepancy was considered to be an error. PUDSY could then locate the fault in the code. Sometimes when the chunk containing the fault almost matched a schema PUDSY could propose an "edit" or fix to the program that would remove the fault.

The approach used by PUDSY called program-analysis debugging involves taking a program and its specification and comparing the two. Systems based on this approach tend to be full debugging systems. They strive not only to identify the fault but also to suggest a repair.

A famous KBPD system that uses the program-analysis method is PROUST [JOHNS86]. PROUST uses a description of what a program is intended to do. A description contains a list of goals that must be satisfied and a description of objects the program must manipulate. PROUST uses programming plans that show how programming goals are implemented. To understand a program PROUST carries out analysis by synthesis. PROUST selects one goal from the problem description. It then retrieves a list of plans that implement this goal from its knowledge base and tries to match the individual

plans with the code. If one fits the code, then PROUST adds it to the interpretation that is being built for the program as a whole, and starts work on another goal in the program description. If none of the predicted implementations matches the program, then PROUST must use knowledge about common bugs to try to explain why the implementations fail to match. The choice of implementation model depends on how easy it is to explain the mismatches in terms of errors.

Shapiro in his Ph.D. thesis presented a debugging system for Prolog, based on what he termed "algorithmic program debugging" [SHAPI82]. Shapiro developed a theoretical framework for program debugging and developed diagnostic algorithms that could isolate an erroneous procedure, given a program and an input on which it behaved incorrectly. These algorithms were interactive: they queried a program for the correctness of intermediate results of procedure calls and used these queries to diagnose the error. The debugging system also used inductive inference and program synthesis from examples. The thesis presented a general algorithm that could synthesize logic programs from examples of their behavior. In addition, the system could compare synthesized programs to real programs to detect errors in the real programs.

While the program-analysis approach seems to offer the best chance of detecting a wide range of errors the approach suffers from several drawbacks. First, for larger programs code-driven approaches suffer from the computational cost of detailed analysis. Further, there are problems in comparing specifications to programs in that the two forms must be similar to simplify the comparison process and there is no guarantee that a specification will be any more correct than the program code. In systems like PROUST, if a particular plan is not in its knowledge base, the system is at a loss about how to interpret the corresponding code.

Another approach to KBPDs is the I/O based debugger. Rather than trying to analyze the entire program in detail, these systems concentrate their computational resources on only the suspect sections of the program. An example of this approach is Falosy (Fault Localization SYstem) [SEDL83]. Falosy uses two inputs: the program to be debugged and a manually prepared list of output discrepancies. The system used plans about how tasks related to a particular application area can be carried out and attempted to match its plans to an abstraction of the program to determine where possible errors may be. Falosy could only handle programs that worked in the application area it understood, which is the master file update problem. In addition, Falosy could only localize errors.

Other KBPD systems are WOODPECKER [FOUET84], which could also rewrite programs to clarify their intended purpose by removing overloaded variables and unnecessary statements. Korei presented a system based on program slicing that started with a print statement that printed an incorrect value and then traced backwards in the execution of the system to discover where a fault was located [KOREI86]. Harandi and Lange developed a system to help novice programmers find syntax errors in Pascal programs and described user's reaction to the debugger [HARAN83], [LANGE85]. Currently, all KBPD systems are experimental. There is no commercially available expert system for program debugging.

2.3. The Amber Debugger

The approach taken in this project is to apply knowledge about the programming language being used, its semantics, and common mistakes made with the language. The Amber debugger was developed to show this approach. Amber uses knowledge about the C programming language to attempt to locate errors and determine faults in a program. Amber does not make use of knowledge about what the program is intended to

do. As a result, there are program faults that Amber is not able to detect. The question is what types of faults can be detected using this approach and is it useful to have a tool that can find them?

Chapter 3

Requirements for Amber

A KBPD assists a programmer with the task of locating faults in a program. Two important issues in exploring the concept of a KBPD are: what faults can the KBPD detect and what knowledge does the KBPD need to accomplish this? This chapter describes the capabilities of the Amber debugger and the knowledge that it will need.

3.1. Types of Errors and Faults

The first problem is to specify what types of errors the debugger will be able to reason about and locate. To do this three categories of errors are defined. These errors are described by the effect they introduce into the execution of a program. These are: non-termination errors, system-abort errors, and incorrect output errors. These are broad groups and each contains many possible faults that could cause an error. Non-termination errors are evidenced by a program whose execution never ends. Two possible faults (causes) for this error are infinite loops and deadlock. Deadlock is typically a problem with program logic (synchronization) and is possible to detect by reasoning about function calls. Infinite loops are typically caused by faulty logic for the loop exit condition and can be located by reasoning about program structures. System-abort errors are caused by faults that result in the generation of exceptions that force a program's execution to be stopped by the operating system. Possible faults in this

category are division by zero, over writing the bounds of an array, and copying data to an invalid pointer address. Incorrect output errors is the broadest category. There are many faults that can cause a program to generate incorrect output. Many such faults are caused by incorrect program logic, but incorrect usage of the programming language can also result in incorrect output. Possible programming language faults are formats in print statements that disagree with the print variables or incorrect usage of a pointer construct.

Before continuing with a discussion about how Amber will attempt to handle the different error types it may be helpful to explain briefly the debugging model that Amber uses. This model or structure of activities will be explained in more detail in chapter 4. First, the programmer (the expected user of Amber) must realize that a software failure has occurred in a program. This will typically occur during testing or normal usage of the program. A failure is characterized by symptoms that describe how the program failed. Amber uses these symptoms to develop a set of possible errors that could cause such symptoms. The execution behavior of the program is then studied in an effort to find a section of code whose behavior is predicted by a suspected error. This section of code is then studied in detail with the intention of identifying the software fault that caused the failure.

Non-termination Errors

In general, it is not possible to analyze a program and determine whether it will always terminate. This is the famous "halting" problem in computer science. However, programmers and users of programs can usually determine when a program is taking far too long to execute and deduce that there is something wrong.

When presented with a program that does not terminate Amber will assume three pos-

sible faults: an infinite loop, deadlock, and an impatient user. An impatient user is the easiest of the three suspected faults to deal with, assuming the user is rational but impatient. The question is did the user allow the program enough time to execute before assuming there was a problem? Is the user familiar enough with the program to know what a reasonable amount of time is? Is the system running slowly and if so did the user allow extra time? The point of these questions is to determine if the program was given enough time to run. If the user indicates that he/she might not have waited long enough then Amber will suggest that the user try one more time and wait twice as long as would be expected for the program to complete.

If it seems the program was given enough time for execution then the two remaining suspects are an infinite loop and deadlock. Deadlock is typically not a concern for most application programs and so Amber will attempt to determine how likely deadlock is. If the program does process to process communication or communicates with another program via a network then deadlock is a real possibility. If a program does none of the above then deadlock is not likely. Deadlock itself occurs as a problem either in the logic of a program or in the synchronization protocol between two processes, but it is possible to detect the likelihood of deadlock. When a deadlock occurs a program will call a function and that function will not return. It is not sufficient to conclude that deadlock is occurring just because a function does not return. There is still a possibility that an infinite loop is occurring in the function that was called. However, there are clues that can be used to determine the likelihood that a deadlock has occurred.

- It is likely if the function that does not return is not a function written by the user, but is a system function or a function from some library of functions.
- What is the purpose of the function? Does it do a read operation? Sometimes the name of the function might be useful, such as "read," "fgetc," "fread," "getmsg,"

etc.

If the code for the function is not available, such as a system or vendor supplied function, and if the purpose of the function is to read from somewhere, then it is likely that a deadlock has occurred.

Infinite loops are far more common, particularly in programs written by novice programmers. To detect an infinite loop Amber tries to find a loop that is entered but does not exit. In C programs several loop structures are possible; these are While loops, Do-While loops, and For loops. In addition, loops can be created by "goto" statements that jump backwards in the code and recursive loops created by recursive functions. To detect if a looping statement (For, While, or Do-While) is the source of an infinite loop, Amber applies the following criteria:

- 1) The loop is entered at some point in the execution of the program and the execution of the program never flows out of the loop. The statement immediately following the loop is never reached and the function containing the loop never returns.
- 2) The loop can not contain a call to a function that does not return. This includes functions called in the loop test as well as in the body of the loop.
- 3) The loop can not contain an inner loop that satisfies criteria 1.

If a "goto" statement creates a loop by jumping backwards in the code then this can create an infinite loop. (Naturally, goto's should never be used, but you can never tell about some people.) A "goto" loop is bounded by a labeled statement at the top of the loop and a "goto" statement at the bottom of the loop. The same criteria that is applied to looping statements can be applied to a "goto" loop to determine if the loop is an infinite loop.

Recursive loops are harder to analyze. The boundary of a recursive loop is not clearly defined. A recursive loop is suggested by a function that does not return containing a call to a function that does not return that contains a call to a function that does not return, etc. Further, it does not have to be that function A calls itself repeatedly. It could be that function A calls B calls C calls A calls B calls A. One way of detecting the recursion is to notice that the same function is appearing many times in the stack of function calls that do not return. (At the time of this writing Amber can not detect a recursive infinite loop.)

System-Abort Errors

When the operating system terminates a program for abnormal reasons it generates a core dump and displays a message usually "(core dump)" along with another helpful message such as "bus error," "EMT trapped," or "Illegal instruction." There are other possible messages as well. The causes of such system aborts are usually not logic based but occur because of a violation of the C language semantics.

When Amber is given a program that is terminated by the system it needs to know the termination message displayed by the system. This message is a useful clue about what possible faults could exist in the program. For example, "Illegal instruction" can be generated by over writing the bounds of an array, and "bus error" can be caused by integer division by zero. Given the termination message Amber must next find the statement in the program where the core dump occurs. The statement where the core dump occurs is usually the statement that contains the fault, but this is not always true. If an array bounds is overwritten by only a few positions then a core dump can occur latter in the program execution and not necessarily at the time the overwrite occurs. This is an example of a notoriously hard fault to find. Once, Amber has located the statement where the core dump occurs it must analyze what is happening in

the statement at run-time to determine the fault. (At this time Amber only finds the statement where the core dump occurs.)

Incorrect Output Errors

When presented with a program whose output is incorrect or is not what the programmer expected, Amber will assume that one of three possible faults is occurring: a faulty path decision, a bad calculation, and a bad print statement. Of the three Amber can best deal with a bad print statement. A bad print statement can generate various clues about its existence. For example, if a value between 1 and 200 is expected as output and the value 2039848453 is printed instead, or if no value is printed, or if the string "@!#\$(@*!" is displayed and "fred" was expected. Possible problems with a format specification in a print statement: the data type of a variable does not match the data format specified in the format, or the number of variables specified in the format does not match the number of arguments to the print statement. These types of problems involve understanding the C function printf and is something that Amber will be able to handle. Output errors caused by faulty logic or calculations in the program are difficult to locate when information about the program task is unknown. (At this time Amber can not handle this type of error.)

3.2. Knowledge Requirements

To be able to locate and reason about the types of errors discussed above, Amber will need several different sources of knowledge. Amber will need knowledge about C and how it is used and knowledge about common programming mistakes made in C. It will need to "see" the program being debugged, and it will need to know about the execution or run-time behavior of the program. These different knowledge sources are discussed below.

The C Language

Amber will need to know about the C programming language. To reason about a program a KBPD must understand the meaning of the statements in the program. These "meanings" are best expressed by the semantics of the programming language. A definition of the semantics of a language gives a precise statement of how each operator and statement type in the language behaves when used. Knowledge of the semantics of the language also provides information about what is legal and what is not legal in the language. In other words, it provides information about possible "pitfalls" in the language. For example, an understanding that the operator "=" does an assignment while the operator "==" compares two values can be used to deduce that the statement:

```
speed == distance / velocity ;
```

probably has the wrong operator, whereas a programmer could read the same statement repeatedly and never notice the mistake. Interchanging the operators "=" and "==" is a common mistake in C.

Amber makes use of its knowledge of C to trace through the execution of a program. Amber can realize when a loop is entered or when a statement causes a branch in the execution flow and can determine where the execution will go. Amber can also use its understanding of C to investigate suspected errors. For example, if the suspected cause of a core dump is division by zero, then Amber can examine the statement where the core dump occurs to determine if any division is done.

Another benefit of providing Amber with the ability to understand C is that it can "read" the program to be debugged. It would be possible to translate a program into a generic representation or to abstract the program details from the code, but information particular to the programming language used in the program could be lost, and the information lost could contain the key to a fault.

Errors and Faults

Amber will need knowledge about errors and faults that can occur in C programs. This includes information such as what symptoms can an error produce? What faults can cause an error? How do you locate a particular type of error or fault in a program? How do you recognize a particular occurrence of a fault? This is the expert knowledge that enables a KBPD to reason about symptoms, errors, and faults and debug a program.

This knowledge has several uses in Amber. First, knowledge about symptoms and errors allows Amber to collect information about symptoms shown by a program and reach conclusions about the type of error in the program. Once an error is suspected Amber can use its knowledge to isolate the error in the code. Amber can then generate a list of possible program faults that could cause that error. Given the location of the error Amber may then be able to determine the specific fault in the program. This knowledge about errors and faults provides Amber with much of its debugging ability.

Program Source Code

Amber needs knowledge about the program to be debugged. Unlike other knowledge sources this knowledge is particular to each program and can not be encoded as part of the debugger. Information about a program must be obtained separately. There are two methods for doing this, the first is to ask the user for information about the source code. This approach has several drawbacks. First, the user must answer a potentially large number of questions about the code. Second, the debugger must rely totally on the user for information and any user is likely to supply incorrect information sometime. Thirdly, the information about the program gained from the user will never be as detailed as the information in the code itself. Relying on the user for all program

information will result in either a shallow debugger or a debugger that is difficult to use.

A better approach is to allow the debugger to "read" the source code directly. This will allow the debugger access to all the details of the program source and greatly reduce the volume of information that the user will be expected to supply. By reading the program code the debugger can build a detailed representation of the program and use it as a basis for reasoning about the program and its execution. Also the debugger can examine the code with a "critical eye" and spot details that a programmer might overlook.

Amber is able to read the source code for a C program directly. Amber uses a model of the source code to locate and analyze errors in a program. This ability to read a program greatly enhances the functionality that a KBPD can provide.

Program Execution Information

To locate an error in the source code Amber needs to trace the execution flow of the program. There are several ways that Amber can get this information. The first is to simulate the program execution. Since Amber has access to the source code and understands the semantics of C this is a valid possibility. Several drawbacks to this approach are that the debugger must provide an environment for the simulation. The simulation must be able to interact with data files and an interactive user. Also, there is no guarantee that the simulation environment adequately duplicates the real runtime environment of the program. Thus, a program that fails when executed normally, may function correctly in the simulation. Another approach is to build the KBPD on top of an execution debugger, such as sdb or dbx on Unix. A KBPD that incorporated a break-and-see debugger will be able to run the program in its real environment and also

gain indepth run-time information. The drawback with this approach is that the instrumented program created by the debugger may still function correctly, whereas the uninstrumented program used by the programmer fails. Another drawback is the possibility of overwhelming the debugger with indepth execution information. The advantage of these two approaches is that the debugger can determine much of the program execution information itself and thus kept interaction with the programmer to a minimum.

Another approach is to rely on the programmer for information about the execution of a program. This approach has the drawback that the debugger can deluge the programmer with requests for information and information supplied by the programmer may be incorrect. Amber currently uses this approach and as a result asks a barrage of questions about what a program does while it executes. This puts the burden on the programmer to determine what really happens when the program runs and to supply that information to Amber. The programmer must either use print statements or a tool such as dbx to gather the execution information. More will be said about this later.

Questions about the execution of a program can include a trace of the program. What statements and sections of code were used? Did a function return when called? Did a loop exit? What was the value of a variable at certain times?

Other sources of knowledge

The list of knowledge sources above does not completely cover the range of information that an experienced programmer uses to debug a program. There are other questions and type of information that can prove useful in debugging, such as a history of previous faults and changes to a program. Did a feature work previously? Has the section of code that handles that feature been modified lately? What was modified? Has a

problem similar to the current problem occurred before?

The depth of knowledge and information available to the KBPD determines its power and usefulness. The knowledge encoded into the KBPD is derived from human expertise and program language understanding. The KBPD also needs information about the program source code and how the program behaves when it executes. That knowledge enables the KBPD to relate symptoms to errors, to investigate the possible locations of an error, and determine the specific fault causing the error. The next chapter presents a structure for Amber.

Chapter 4

Design For the Amber Debugger

This chapter describes the structure of Amber and explains how this structure captures the intended debugging capabilities. Amber consists of components that embody expert knowledge about how to debug programs, supporting components for collecting information, and information or facts about the program to be debugged. The next chapter describes the implementation of Amber.

4.1. System Overview

Most of Amber's functionality derives from the knowledge encoded into it. Amber is a knowledge based system and its knowledge is encoded using IF-THEN rules. A major portion of the source code for Amber will therefore be production rules. As a debugging tool, Amber will do several tasks during a debugging session and each task will be performed by a subset of the rules in the system. To aid the clarity of the implementation, a structure is imposed on the rule base. The use of the word "structure" here means that it is possible to group rules logically according to the task that they support. This is equivalent to "modularizing" the rule base.

The first step is to define the tasks that Amber will perform. Figure 4.1 gives a description of the procedure that Amber uses to debug a program. The term "procedure" in computer science implies a sequential style of operation, whereas, expert sys-

tems in general do not operate in a fixed sequential manner. Amber will do its tasks in a given order, but rules within a task will fire when their conditional clauses are satisfied. The shifting of control from one task to another will occur when the rules decide it is appropriate and not at a procedurally preordained time.

The basic purpose of Amber is to help a programmer locate where an error is occurring in a program and determine what the error is. Sometimes Amber can further examine the program and determine what fault in the code is causing the error. (Fault analysis is not implemented at this time. Currently, Amber only locates an error.)

-
- 1) Ask the programmer for the name of the program source code file(s) and "read" the program.
 - 2) Determine what symptoms the program exhibits at run-time.
 - 3) Analyze the symptoms and deduce which errors could be present in the program.
 - 4) Select the most likely error or an error from the set of most likely errors if there is more than one error with the same likelihood.
 - 5) Trace the execution of the program and attempt to locate a section of code whose behavior matches the behavior predicted by the selected error.
 - 6) If an error location is not found then select the next most likely error and re-trace the program execution (go to step 5).
 - 7) If a location for the error is found then show the programmer where the error is occurring and explain the error.
 - 8) Given the location of the error in the source code attempt to determine the fault that causes the error.
 - 9) If a particular fault can be found then explain it to the programmer.

Fig. 4.1 : Debugging Procedure

Given the list of steps shown in figure 4.1 it is possible to define the major components of Amber and explain their functionality. The basic components of Amber are shown in figure 4.2, and the components needed for locating a fault in a program are shown in figure 4.3. In these figures the square boxes represent rules, the rounded boxes represent information or facts, and the fussy square boxes represent components of the debugger that support the debugging task, but do not embody any knowledge.

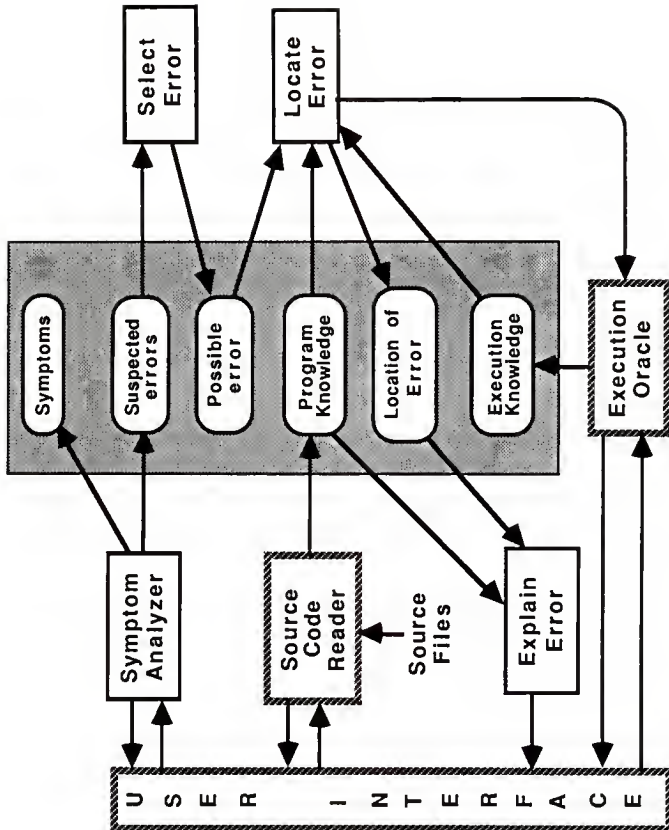


Figure 4.2 : Components of Amber

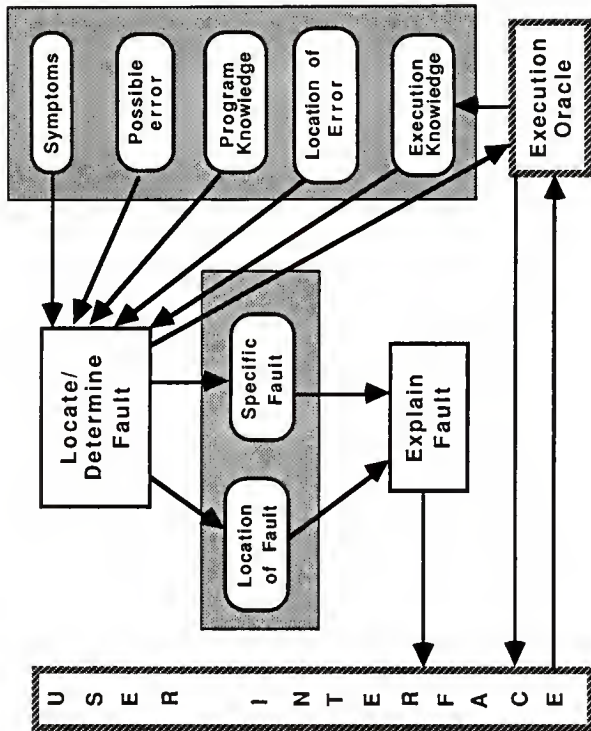


Figure 4.3 : Components for Locating Faults

These two figures show the different rule groups, facts or information that each rule group needs before it can fire, and the facts generated by a group during its run. From these figures can be seen the order in which these groups should fire. The rule groups Symptom Analyzer and Source Code Reader need to fire first and there is no apparent ordering between the two. As was seen in figure 4.1, Amber will read the source code files first.

Figure 4.2 shows that Amber uses two sources of input: the programmer and the program source code files. The programmer whose program does not work is the expected user of Amber. Amber expects the programmer to be able to provide information about how the program's execution behavior differs from the expected behavior, names of the program source code files, and detailed execution information for the program. The usage of this information will be discussed in greater detail in the following sections.

The remainder of this chapter discusses the components shown in figures 4.2 and 4.3. The rule groups are discussed first, followed by the supporting components of the debugger. Finally, the fact sets that hold information about the debugging session are discussed.

4.2. Rule Groups

The rule base or knowledge base embodies the expertise and main functionality of Amber. The rule base contains a collection of rules that can be logically separated by functionality into six groups. These functional groups are explained in the following pages.

4.2.1. Rule Group - Symptom Analyzer

The Symptom Analyzer component interacts with the programmer to build up a collection of symptoms that describes how the execution behavior of the program differs

from the expected behavior. Amber will use a set of menus to collect initial information and then prompt the programmer with specific questions when additional information is needed. For example, if the programmer stated that the program does not terminate, then Amber assumes two possible errors : infinite loop and deadlock. Amber will then try to determine the likelihood of deadlock by asking the programmer questions about what the program does. These are broad questions, such as does the program communicate with another program via a network?

The problem types that Amber will be able to deal with were presented in chapter 3. These are: non-termination, system aborts, and incorrect output. Non-termination has already been discussed. If a program does not terminate then the set of possible errors are infinite loop and deadlock. The symptom analyzer will verify that the programmer believes the program was given enough time to execute. If the programmer indicates that the program was not given enough time then Amber will ask the programmer to try running the program again.

If the program to be debugged is aborted by the system then Amber will need to know the message displayed by the system when the program was aborted. This message will allow Amber to select a set of possible reasons that could explain why the program was aborted. This is will be the set of possible faults.

If the program to be debugged generates incorrect output then Amber will ask "How is the input not correct?" Possibilities are that no output was produced, or too much, or incorrect values were printed. Based on the programmer's responses, Amber can conclude that the program possibly has a bad print statement, contains faulty decision logic, or contains a bad calculation. Of these three possibilities Amber will only be able to handle a bad print statement.

When the symptom analyzer is finished it will have created information about the symptoms of the failure in the program and a set of possible errors. Each possible error will be assigned a weight that specifies its likelihood. The valid range of weights is -2 to 2, where zero means possible, less than zero means not likely, and greater than zero means more likely.

This component of Amber is itself an expert system, though a shallow one. The symptom analyzer was the first component of Amber to be designed and built and has been redesigned several times. The first implementation could collect symptom information and then informed the programmer which errors could cause such symptoms and gave advice about how to locate each possible type of error. The current design of Amber allows Amber to walk the programmer through the process of locating an error.

4.2.2. Rule Group - Select Error

The Select Error component chooses one error from the set of possible errors and directs the debugger to attempt to locate the error in the program source code. To do this, the error with the highest weight is selected. If more than one error has the same weight then one error is selected at random. (In principle that is. In the implementation it is not random at all.) The selected error is designated the "possible error" and Amber will then attempt to locate a section of code whose behavior is predicted by the possible error. For example, if the possible error is an infinite loop then the debugger must try to find a loop that was entered but never exited. (The criteria for identifying an infinite loop was given in chapter 3.) This component is part of the control structure of the debugger and provides the debugger with a specific task to perform.

4.2.3. Rule Group - Locate Error

The Locate Error component embodies the main function of Amber which is the ability to trace through the execution of a program and locate where an error occurs. The rules in this component contain the knowledge required to understand constructs in the C language and how they function. Understanding the statements and expressions in the language gives Amber the ability to follow the program code and compare it to the execution behavior of the program. To do this, information about both the program code and the program execution is required. This component makes heavy use of the Execution Oracle component which is described in a later section.

This component can not proceed until the Possible Error information has been selected by the Select Error component. The possible error determines how the program execution will be traced. Amber has several different techniques for tracing the execution and the possible error dictates which technique will be used. These different techniques are explained below.

If the possible error is an infinite loop then Amber will trace a program as follows:

- 1) Start the trace with the first statement in the function main. Every C program has a function named main that is the top level function in the program.
- 2) If the statement is a loop statement, such as a For, While, or Do-While then determine if that loop exits at any point in the execution. If the loop exits then either the statement following the loop is eventually executed or the function containing the loop eventually exits. The loop could contain a Return statement. If the loop does not exit then go to step 3, otherwise continue the trace with the statement following the loop and proceed to step 4.
- 3) This step has several related parts.
 - 3a) Examine each statement in the loop to determine whether the loop contains an inner loop that is entered but never exits. If such a loop is found then repeat this step for that loop.
 - 3b) Examine each statement in the loop for function calls. If a function call is found then verify that each time the function is called it does return. If a function call is found that does not return then continue the trace with the first statement in that function using step 2.
 - 3c) Check the exit expression of the loop statement for function calls. If a function call is found then use step 3b.
 - 3d) If steps 3a, 3b, and 3c do not find any reason the loop should not exit then conclude that the current loop is the infinite loop. Save information about the loop so that the location of the error remains known.

- 4) If the statement is not a loop statement and is not a Goto statement then check for function calls. For each function call found, verify that the function does return. If a function does not return then continue the trace with the first statement in that function using step 2. If the function does return then continue the trace with the next statement using step 2.
- 5) If the statement is a Goto statement then first scold the programmer for using a Goto, then determine if the Goto does a backwards or forwards jump in the function code. If it is a forward jump, then jump to the correctly labeled statement and continue the trace using step 2. If this is a backwards jump then determine if the execution ever moves to the statement after the Goto statement in the function code. If the execution does reach a point further down the code then continue the trace by jumping to the labeled statement and use step 2. If the execution never reaches past the Goto statement then the possibility exists that this Goto loop is the source of the infinite loop. Currently, Amber is not designed to trace this type of loop, so the programmer will be informed that this is possibly the source of the infinite loop and that it should be checked for inner loops that do not exit or function calls that do not return.

When Amber detects a loop that is entered but does not exit, it stops following the execution flow directly. If it continued to follow the execution statement by statement, then Amber could become caught in the loop. Instead, when such a loop is found Amber scans the statements in the body of the loop ignoring branching caused by If, Switch, or Goto statements, and searches for loops and function calls. For each inner loop found Amber will verify that it does exit when entered and that each function call does return when called.

If the possible error is a core dump, i.e. the system terminates the program, then the program will be traced in a more tedious manner.

- 1) Start the trace with the first statement in the function main.
- 2) If the statement is not a loop statement then determine if the statement executes successfully or not. If not, then determine whether the statement makes a call to any function. If a function call is found then determine if the function returns. If the function does not return then continue the trace with the first statement in the function and use this step. If all functions calls in the statement do return then this is the statement where the core dump occurs. Save information about the statement for later use. If the statement does execute successfully then continue the trace with the next statement in the function and repeat this step.
- 3) If the statement is a loop statement then determine if the loop exits. If the loop does exit then the core dump is not occurring inside the loop so skip the body of the loop and continue the trace with the first statement following the loop and use step 2. If the loop does not exit then continue the trace with the first statement in the loop and use step 2. Step through the execution of the loop using step 2 until

the core dump occurs.

This method is tedious to do because the trace must consult the execution information for each statement and expression to determine if it executed successfully.

If the possible error is a bad print statement then the print statement must be located in the code. This can be done by asking what function the print statement is in and then searching the function for print statements and using the programmer to identify the correct one. Then Amber can study the statement to check for possible problems such as bad data type specified or incorrect number of print variables. (This ability is not currently implemented.)

If the possible error is bad logic for a path decision test or a bad calculation then Amber will inform the user that this type of error can not be located using Amber. It is possible to design and implement a method for attempting to locate a bad decision expression or faulty calculation. Such a method is based on program slices [WEISE82], [KOREI86]. In this method the print statement where the incorrect output is printed is located and the execution of the program is traced backwards from that point, checking at each step for incorrect values of variables and incorrect execution paths. The incorrect path or incorrect variable values are followed in an attempt to find a location where the incorrect value originated or the wrong execution path was entered.

This component of Amber is the largest and its sole purpose is to help the programmer step through the process of locating where in the program the error is occurring and identifying the error.

4.2.4. Rule Group - Explain Error

The Explain Error component is responsible for two tasks: explaining the error that is present in the program and showing the programmer the section of code containing the

error. Explaining the error involves printing a message about the type of error detected. To show the location of the error the statement containing the error is displayed along with the name of the function that contains the statement. Typically, the execution trace done by the Locate Error component stops when the error is located and so the statement containing the error will be the last statement in the trace.

This component contains rules that follow the trace activity done by Locate Error and can realize when the error has been located. This is how control of the debugging session passes from Locate Error to this component.

When this component is done, Amber must determine if it can attempt to locate the specific fault that is causing the error. If it can, then control of the debugging session will pass to the Locate/Determine Fault component. Otherwise, the programmer will be asked to either exit the debugger or start again and work on a different bug.

4.2.5. Rule Group - Locate/Determine Fault

The Locate/Determine Fault component examines the section of code containing the error and the execution information for this section and attempts to locate a specific fault that is causing the error. In addition to using an understanding of the C language, this component makes use of knowledge about different faults that can cause an error and how to detect the fault in code.

This component can not proceed until the Location of Error information has been created by the Locate Error component. In addition, this component will wait until the error has been explained to the programmer and then will proceed to locate the fault if the programmer wants Amber to try. (This component is not implemented.)

4.2.6. Rule Group - Explain Fault

The Explain Fault component is responsible for two tasks: explaining the fault that is causing an error and showing the programmer the section of code containing the fault. Explaining the fault involves printing a message about the fault detected. To show the location of the fault the statement containing the fault is displayed along with the name of the function containing the statement.

This component contains rules that follow the activity done by the Locate/Determine Fault component and can realize when the fault has been located. That is how control of the debugging session passes to this component.

When this component is done the programmer will be asked to either exit the debugger or start again and work on a different bug. (This component is not yet implemented.)

4.3. Supporting Components

4.3.1. Supporting Component - User Interface

The user interface is a collection of supporting functions that allows Amber to maintain a consistent interface with the programmer. All communication between Amber and the programmer will take place through this interface. When any rule needs information that has to be obtained from a user the user interface will be used to issue the question to the programmer and collect the reply.

An important component of the user interface is the help facility. The help facility allows the programmer to request a "help" message for any prompt issued by Amber. Furthermore, after displaying the help message Amber will redisplay the original prompt. In addition, to displaying a help message the programmer can invoke the help facility subsystem or any element of the subsystem from any prompt during the debugging session. The help facility subsystem provides the capability to:

- Terminate a debugging session. This feature can be used if the programmer suddenly "sees" the problem solution and no longer needs the debugger.
- Escape to the shell. This feature allows the user to suspend the debugger and return temporarily to the Unix shell. At any time the user will be able to end the shell and return to the debugger at the point where it was suspended.
- Exit the help facility. If the user invokes the help facility subsystem then it will be necessary to exit the subsystem and return to the debugging session.

After the help facility or a component completes and returns to the debugging session Amber will repeat the prompt that was issued when the facility was invoked.

The help facility functionality listed above is actually just a framework for such a facility. In addition, the help facility could allow the programmer to exercise control over the debugging session. The facility could allow the programmer to view any or all facts in the knowledge base and change these facts to control the debugging process. This allows the programmer to take charge of the debugging session.

The major emphasis of this project is on the structure and capabilities of the debugging process. Because of this the issue of the user interface will not be pursued further. The result of this is a weak and possibly unpleasant user interface in the implementation. Hopefully though, the framework for a more powerful and friendly interface will result from the basic design and capabilities described here.

4.3.2. Supporting Component - Source Code Reader

The Source Code Reader component of Amber is responsible for "reading" the source code of the program to be debugged. Reading C source code involves taking the source and converting it into a form that the rules are able to understand. How to represent the C source code is a separate topic and is discussed later in this chapter. The opera-

tion of the source code reader is given below.

- 1) Prompt the programmer for a name for the program. This is not a source file name, but merely a name that identifies the program. Amber will use this name to create a file that contains information about the source files and functions used in the program. This file is called the program information file and is discussed in chapter 6. The program information file contains a list of source file names and the names of function defined in each file.
- 2) Prompt the user for a list of source code file names. The source code for a C program can be scattered among several different files.
- 3) Read each file and convert the source code into a format understandable to the rules. This step involves scanning each source file and placing a readable representation of the code into a separate file that Amber will use later. For each source code file, a corresponding file called a source code fact file is created. A source code fact file contains a representation of the source code that is understandable by the rules. Source code fact files are discussed in chapter 6. When a source file is read only information about the statement structure is saved. Amber currently does not use information that could be gained by reading declarations.

Step 3 is the major activity of this component and will be performed by a separate tool called the "scanner." The scanner is a separate program that reads C source code and transforms it into the format used by the rules. The scanner is discussed in chapter 6. By transforming the source code into a usable format, Amber can directly study and manipulate the program being debugged.

4.3.3. Supporting Component - Execution Oracle

The function of the oracle is to provide Amber with information about the execution behavior of the program. Amber traces the execution flow in an attempt to locate the source of an error or fault. How the oracle obtains this information is separated from the normal activities of Amber. The task of obtaining run-time information for a program is a project in itself and use of the oracle separates this activity from the activities done by Amber.

In the current system, the source of run-time information is the programmer who uses Amber. This puts a burden on the programmer to collect the necessary information and provide it to the oracle when asked. A future step in the development of the oracle

could be to automate the collection of run-time information by allowing the oracle to run the program and collect information that can be placed in a fact set.

4.4. Fact Sets

The information or data processed by Amber consists of facts. The facts describe information such as the program source code, suspected errors, information deduced by Amber, etc. These facts are logically separated into sets based on the type of information held in the set. The fact sets shown in figures 4.2 and 4.3 are described below.

4.4.1. Fact Set - Symptoms

Symptoms describe how the execution behavior of the program differs from the expected behavior. The Symptom Analyzer produces this information. Information about symptoms is obtained from the programmer and is used by the Symptom Analyzer to deduce a set of suspected errors that could cause the observed symptoms.

4.4.2. Fact Set - Suspected Errors

The Suspected Error facts are created by the Symptom Analyzer component and consists of a set of errors that could cause the observed symptoms of the failure occurring in the program. Each suspected error is given a weight that specifies the likelihood of the error occurring. Weights range from -2 to 2 with negative values indicating less likelihood and positive value indicating more likelihood.

4.4.3. Fact Set - Possible Error

After a set of suspected errors has been generated Amber must select the error that is most likely to be occurring and attempt to locate it in the program. This selection is done by the Select Error component. The fact set created by this component is the Possible Error set. This set contains one error. This error determines how Amber will

trace the program execution when attempting to locate the error. If Amber fails to locate the error currently being investigated, then another suspected error will be selected and will replace this fact set.

4.4.4. Fact Set - Program Knowledge

The Program Knowledge facts are an encoding of the program source code that is understandable by Amber. The scanner, an element of the Source Code Reader component, creates this fact set. Amber may use these facts to step through the program logic, to simulate a portion of the program execution, to display the section of code under discussion, etc. Knowledge of the program structure is important to the debugging process. These facts are not an abstraction of the source code, but represent the structure of the code in the program. An abstraction of the source code structure could result in loss of detail and one the details lost may contain information about the error. How the expressions, statements, and structure of the program are represented has a major effect on the design of Amber. The details of how a program is represented will be discussed in chapter 5, but the design of the representation is given below.

The design of the program representation went through several iterations. In the first design, each statement in a function was represented as a fact. Each statement fact had a unique label that identified the statement and its position in the function. A statement fact also contained information about the type of statement, a label for the statement that preceded it and the statement that followed it in the code, as well as labels for expressions used in the statement. Individual expressions were also represented in separate facts with unique labels that specified the statement that contained the expression, and the position of the expression in the execution order of the statement. This style of representing the structure of the program fits well with the operating style of

the production system. However, a major disadvantage of this representation was that even a short program generated a large volume of facts.

Later, a different style of representing the program structure was selected. This style was based on an abstract syntax representation of C. This style works well for several reasons. First, the scanner does a transformation on the program source code, it reads in a C program and outputs a representation that can be understood by Amber. To do this, the scanner builds a parse tree that represents the structure of the source code. The parse tree contains extra structure that is particular to the parse tree representation of the program and is introduced by the parser grammar. This extra structure is required by the parser to correctly build the parse tree and represent the structure of the code. However, this extra structure in the form of non-terminal nodes, is not necessary for understanding the structure of the program.

An abstract syntax representation is easily created from a parse tree by removing the details that are particular to the tree but not the source code. An abstract syntax representation describes structure, without the extra representational details, and helps relate syntax to semantics more closely [SCHM186]. An understanding of the semantics of C is built into Amber and a style of representing source code that meshes well with the semantic representation is needed. The style of representing the semantics of C is taken from denotational semantics. A denotational specification of the semantics of a language can be used with an abstract syntax representation of a program to simulate the execution of a program or to derive a "meaning" for the program.

Figure 4.4 shows the abstract syntax for C used to represent information about the source code. This style of representation allows the scanner to preserve the structure and sequence of functions defined in a source file. In figure 4.4 a phrase such as $P \in \text{Program}$ means that Program is a syntax domain and P is a nonterminal that represents

an arbitrary member of the domain. The abstract syntax definition shows that a program contains one or more functions. Functions consist of a function name and a block of statements.

This syntax omits any information about declarations defined in the source code. Denotational semantics provides a good method for representing and reasoning about declarations and information about declarations is needed to resolve certain faults. However, Amber currently does not save or make use of such information. Information about declarations is ignored by the scanner though it could easily be saved in a symbol table and transformed into a format compatible with the abstract syntax and C semantics representation. This would be a reasonable enhancement to the design of Amber.

P ∈ Program
 F ∈ Function
 B ∈ Block
 S ∈ Statement
 E ∈ Expression
 I ∈ Identifier
 N ∈ Numeral
 C ∈ Character or String

P ::= F

F ::= I() B | F F

B ::= { S } | { }

S ::= S1 S2 | B |
 case E : S | default : S |
 E ; | I : S |
 if (E) S | if (E) S1 else S2 |
 do S while (E) ; | while (E) S |
 for (E1 ; E2 ; E3) S | for (E1 ; E2 ;) S |
 for (E1 ; ; E2) S | for (; E1 ; E2) S |
 for (E1 ; ;) S | for (; E1 ;) S |
 for (; ; E1) S | switch (E) S |
 goto I ; | break ; |
 ; | continue ; |
 return ; | return E ;

E ::= E1 E2 | E1 = E2 | E1 += E2 |
 E1 -- E2 | E1 *= E2 | E1 /= E2 |
 E1 %= E2 | E1 <<= E2 | E1 >>= E2 |
 E1 &= E2 | E1 ^= E2 | E1 |= E2 |
 E1 ? E2 : E3 | E1 && E2 | E1 || E2 |
 E1 | E2 | E1 ^ E2 | E1 & E2 |
 E1 == E2 | E1 != E2 | E1 < E2 |
 E1 <= E2 | E1 > E2 | E1 >= E2 |
 E1 << E2 | E1 >> E2 | E1 + E2 |
 E2 - E2 | E1 * E2 | E1 / E2 |
 E1 % E2 | (Y) E | sizeof (Y) |
 sizeof E | -E | !E |
 ~ E | &E | *E |
 ++E | --E | E++ |
 E- | (E) | E1 [E2] |
 E.I | E->I | E() |
 E1 (E2) | I | N | C

Figure 4.4 : Abstract Syntax for C

Figure 4.4 shows the statements and expressions for the C language. This abstract syntax does not contain extra details that are imposed by the representation, and the structure of the syntax fits well with the structure of the C semantics.

This abstract syntax style is used for representing information about the source code for a program and this information forms the Program Knowledge fact set and is created by the Source Code Reader component. The program knowledge is used by the Locate Error component to trace the execution of the program and by the Explain Error and Explain Fault components to show the programmer where the error or fault is occurring in the code.

4.4.5. Fact Set - Location of Error

The Locate Error component creates the Location of Error facts. These facts give the name of the function containing the error and the statement where the error occurs. This information is used by the Explain Error component to show the programmer where the error is occurring. It also tells the Locate/Determine Fault component where to search for the fault.

4.4.6. Fact Set - Execution Knowledge

This is information about the execution of the program that is reported by the Execution Oracle component. This information is used by the Locate Error component and the Locate/Determine Fault component. Execution information is knowledge about the path an If statement used, whether a loop exits, did a function return, did a statement execution successfully, etc.

4.4.7. Fact Set - Location of Fault

The Locate/Determine Fault component creates the Location of Fault information.

These facts give the name of the function containing the fault and the statement where the fault occurs. This information is used by the Explain Fault component to show the programmer where the fault occurs. (This fact set is not currently generated because the Locate/Determine Fault is not implemented.)

4.4.8. Fact Set - Specific Fault

This information specifies the specific fault that is causing an error in the program. This information is created by the Locate/Determine Fault component and is used by the Explain Fault component to explain the fault to the programmer. (This fact set is not currently generated because the Locate/Determine Fault is not implemented.)

This concludes the description of the components of Amber. The next chapter describes the implementation of Amber. Examples of debugging sessions using Amber are given in Appendix B.

Chapter 5

Introduction to YAPS

This chapter is an introduction to YAPS [ALLEN83]. The knowledge base for Amber is implemented in YAPS. An understanding of YAPS will be beneficial to the reader while reading chapter 6. If the reader is already familiar with YAPS, then this chapter can be skipped.

YAPS (Yet Another Production System) is used to represent the underlying knowledge of an expert system. YAPS is built on many of the ideas from OPS5, but is designed to allow greater flexibility and readability of production rules than provided by OPS5. YAPS is implemented in Franz Lisp augmented with Maryland's variations and using the University of Maryland flavors. Rules written in YAPS may use Lisp and can call Lisp functions provided by the system or the user. A YAPS environment consists of databases containing sets of facts and production rules. YAPS can maintain and run multiple rule sets and databases. Amber does not make use of this capability of YAPS. YAPS facts and production rules are explained in the following sections

5.1. Facts in YAPS

In YAPS, a fact is an arbitrary Lisp expression containing atoms, integers, and nested Lisp lists of atoms and integers. An example fact is :

```
( knows mary john )
```

Facts are added to a YAPS database using the YAPS function "fact." An example is :

(fact knows mary john)

Once a fact is added to a database it may be used by the rules already active in that database. Facts are used to match patterns in the left hand sides of rules.

Another YAPS function is the "goal" function which also creates a type of fact. A fact loaded into a database by the "goal" function is treated differently from a fact loaded by the "fact" function. A goal is given greater weight than a normal fact when determining which rule to fire next. Goals are used to direct the selection of rules that can fire, while normal facts are used to store information for the rules to act on. The difference between goals and facts will be explained further in the next section.

5.2. Production Rules in YAPS

Production rules in YAPS have an IF - THEN format. The IF portion of a rule (also called the left hand side) consists of a set of patterns that are matched against the current set of facts and goals in the database. The THEN portion of a rule (also called the right hand side) is a set of Lisp expressions that are evaluated when the rule is fired. A rule is fired when all the patterns in the left hand side can be matched to facts in the database. A rule may specify a set of tests that the rule must satisfy in addition to having all the patterns match. A rule from Amber is shown in figure 5.1. This rule fires when a function call is located while stepping through an expression.

```

; This rule handles an expression of the form
; ( E F(expr1) (expr2) ) expr1(expr2) function call

(defp exp_func_call
  (goal (trace -type) )
  ( E F -func -list )
  ->( remove 2 )
  (cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          ; Check for function calls in the parameter list
          ; first and then check this call.
          ( fact FUNC_CALL -func )
          ( make_fact -func ) ; Should just be a string.
          ( make_fact -list ) ; Check this first.
        )
  )
) ; end of rule exp_func_call

```

Figure 5.1 : Example YAPS Rule

The left hand side of the rule shown in figure 5.1 contains two patterns:

```

  ( goal (trace -type) )
  ( E F -func -list )

```

These two patterns must match two facts in the database before the rule can fire. The first pattern must match a goal. In a YAPS pattern a variable is any atom that begins with a hyphen (-). In these two patterns the variables are "type," "func," and "list." These variables can match any value in a fact, but according to these two patterns the goal matched must contain "goal" as the first atom and "trace" as the second atom. The second pattern must match a fact with "E" as the first atom and "F" as the second atom.

The operator "->" is used to separate the left hand side from the right hand side. When this rule fires its first action is to remove the fact that matched the second pattern from the database. This done by using the YAPS function "remove" and specifying the number of the pattern to be removed. The next action is done by the Lisp

"conds" statement. If the value of the variable "type" is "loop," "deadlock," or "coredump" then three new facts are created. The function "make_fact" is part of Amber. It takes an argument and generates a fact. It will be explained in chapter 6.

The rule shown in figure 5.1 is used to trace the execution of a function call. The order that these facts is created is important and exploits the conflict resolution strategy used by YAPS. The first fact created shows that a function call has been found. The second fact records the name of the function. Typically, an identifier is used as the name of a function, but in C a function address expression could also be used. The third fact created contains the parameter list for the function call. Amber will want to check the parameter list for other function calls, before examining the function call.

The rule presented in figure 5.1 makes full use of the YAPS conflict resolution strategy. When the left hand side patterns of more than one rule are satisfied by facts and goals currently in the database then YAPS must decide which rule to fire next. Only one rule may fire at a time. Rules with goal facts patterns are given higher priority than rules with no goal fact patterns. In addition each fact and goal has a unique number associated with it that shows when that fact or goal was added to the database. The rule that matches the most recent goal will fire first. If several rules match the most recent goal then the rule that matches the most recent fact(s) will fire. If two rules match the most recent goals and facts then the rule with the most patterns will be selected.

After the rule in figure 5.1 fires the parameter list will be checked first. Because that fact was added last by the rule. Next, the function name will be checked to see if it is an identifier or an expression. Finally, the function call that was found will be checked to see if it returns.

Another rule from Amber is shown in figure 5.2. This rule is from the Select Error component and determines which suspected error should be selected as the possible

error.

```
; This rule is fired when the debugger needs to take the
; most likely hypothesis and attempt to locate the error
; in the source code.

(defun select_error
  (goal (select error))
  (HYP -suspect -weight) ; Select error with greatest weight
  (~ (HYP -other -mass) with (> -mass -weight)))
->(remove 1)
  (cond (( (= -suspect 'Buf_ovrflw)
           ; Suspected cause is a buffer overflow.
         (goal trace coredump)
        )
        (( (= -suspect 'Deadlock)
           ; Suspected cause is a deadlock.
         (goal trace deadlock)
        )
        (( (= -suspect 'Inf_loop)
           ; Suspected cause is an infinite loop.
         (goal trace loop)
        )
        (( (= -suspect 'Unknown)
           ; The debugger is not prepared to handle the symptoms.
         (goal explain unknown)
        )
      )
  ) ; end of rule select_error
```

Figure 5.2 : Another YAPS Rule

This rule can not fire until the goal to select error is present in the database. In addition, the rule must match the fact for the suspected error (call hypothesis in Amber and denoted with the letters "HYP") with the greatest weight. This requirement is expressed in the patterns:

```
( HYP -suspect -weight ) ; Select error with greatest weight
( ~ ( HYP -other -mass ) with ( > -mass -weight ) )
```

A hypothesis fact has the form:

```
( HYP type-of-error weight )
```

The second pattern in the rule shown in figure 5.2 will match any suspected error fact.

but the third pattern specifies that no other suspected error fact can have a weight greater than the weight for the fact matched by the second pattern. This is specified by the "~~" which is a "not" operator and the "with" operator which specifies a condition. The third pattern reads "A HYP fact whose weight is greater than the weight given to the HYP fact from the second pattern, may not exist in the database."

Chapter 6

Implementation of Amber

The knowledge base for Amber is implemented in YAPS [ALLEN83], a production system developed at the University of Maryland. The knowledge base contains 139 rules, most of which specify knowledge about the C language. Amber is also supported by 12 Lisp functions and a C source code scanner that is written in C. Amber currently runs on a VAX 11/780 operating under Berkeley 4.3 UNIX.

This chapter describes how Amber is implemented. The order in which components are presented will correspond to the order in which the components are used during a debugging session. Discussion of the implementation is intended to point out how the structure presented in the chapter 4 is implemented without going into the details of each rule. The source code for Amber is given in appendix C and D. This chapter is not intended to inform the reader about how to run Amber, that information is given in appendix A.

6.1. Starting Amber

When Amber is loaded [1] it will automatically begin execution. A little welcome message is displayed via the `banner` function in the file "start.l." The function `start` in the same file initializes the YAPS database for the debugging session. The function `start` removes any facts currently loaded in the database and then creates the goal:

```
( goal run scanner )
```

This goal specifies that Amber is ready to "read" the source files for the program to be debugged. By creating this goal control of the debugging session shifts to the Source Code Reader component.

6.2. The User Interface

The user interface is a collection of Lisp functions, not rules. Amber is designed to be used on a standard ASCII terminal with no graphics or flashy interface techniques. The interface consists of special functions for reading in the programmer's response to a question and using the Help Facility. Prompts to the programmer are issued by Amber as the need arises. Most prompts are issued by code in the rhs of a rule. The enforced style of handling a prompt in the Amber code is shown by the code template used during the development of Amber. This template is shown in figure 6.1.

Amber is loaded by executing "mlisp yaps," followed by "(load amber.l)." See appendix A for more information about using Amber.

```

( prog ( ans )
  again

  ( msg N )
  ( msg N T "This is a prompt." )

  ( setq ans ( call_help (lread)
    ("This is a hard coded help message, in case the"
     "programmer whats more information about the"
     "prompt." ) )
  )
  ( cond ( (null ans )
    ; Repeat the prompt
    ( go again )
    )
    ( (equal ans 'valid_answer )
    ; Code for handling a valid reply
    )
    ( t ; User did not select a correct answer
    ( msg N T "*** Please pick a valid letter." )
    ( go again )
    )
  )
) )

```

Figure 6.1 : Template for Prompts

The function `lread` from the file "misc.l" is used to read the programmer's reply to a prompt. The function `callhelp` from the file "help.l" is used to invoke the Help Facility. Both the programmer's reply and a help message for the prompt are passed to `callhelp`. The function `callhelp` will check the first character of the reply. If it is not a question mark then the reply is returned and assigned to the variable `ans`. If the reply is just a question mark then the help message is displayed. After the message is displayed `callhelp` returns a null. This causes the prompt to be re-issued. If the reply is a question mark followed by a valid help command, the help command is executed by the Help Facility. If the reply is a question mark followed by the word "help" then the Help Facility subsystem is entered. Valid Help Facility commands are given in appendix A.

To issue a help command the command must be preceded by a question mark and a space. A help command can be entered at any prompt issued by Amber.

6.3. Source Code Reader

The Source Code Reader component is actually just one rule. This rule fires when the goal "run scanner" is placed in the YAPS database. The rule is named `scan_source` and is in the file "run_scan.1." The first step done is to remove the "run scanner" goal, since that action is about to be carried out. Next, the programmer is prompted to enter a name for the program. The name of the program is not necessarily the name of a source file, but rather a name that is used when referring to the whole program. The scanner will use this name to create a program fact file that contains information about the entire program. Program fact files are discussed a little bit later.

Once, this name has been entered the programmer must supply a list of source file names. For a C program this must include the source code and include files written by the programmer that are used to create the program. Source file names can be entered in two ways:

- 1) Amber can prompt the programmer to type in the name of each file. This can be tedious if there are many files for the program.
- 2) The programmer can enter the name of a file that contains a list of source file names.

At this point Amber is ready to call the C scanner and have the source files read. The scanner is passed the program name and either the list of source file names or the name of a file containing such a list.

When the scanner is finished the program fact file created by the scanner is loaded into the YAPS database. The representation of the source code is not loaded at this time.

That information is contained in source code fact files, which will be loaded as the source code is needed.

At this point the Source Code Reader is finished and Amber is ready to run the Symptom Analyzer. To accomplish this the goal

(goal run symp_anal)

is created by the Source Code Reader component. This goal causes control to shift to the Symptom Analyzer component.

6.3.1. C Scanner

The scanner is a program that parses C source code files and generates a description of the source code. This description is readable by Amber. When the scanner is called, it is given a list of C source code and include files used in a program. Each file is scanned and a parse tree is built that describes the source code. The parse tree is then encoded into a set of facts that are suitable as input to YAPS. The process of parsing and generating facts is repeated for each file. Amber invokes the scanner directly and when the scanner is finished Amber loads the generated facts into the YAPS database.

The scanner is a separate program that consists of a lexical analyzer for C, a C parser, and a fact generator. The scanner itself is written in C and plays a crucial role in providing Amber with the ability to "read" C source code. The scanner contains around 5000 lines of C, Yacc, and Lex code. This section discusses how the scanner is used and outlines how it operates, but the details of its implementation will not be presented. The source code for the scanner is given in appendix D.

6.3.1.1. Running the Scanner

The scanner is invoked directly by Amber. Amber passes information to the scanner through command line arguments or parameters. The scanner is designed based on the

following assumptions about the program source files it is expected to scan. The source code for any program that Amber is used on must meet these assumptions.

- a) Any source code or include file passed to the scanner must be compilable. The C compiler "cc" must be able to successfully operate on the file. The scanner does little error checking and assumes that any file passed to it is syntactically correct. The scanner will detect when a file is not syntactically correct, but will produce no useful error message about the violation.
- b) The scanner currently requires that all source files and include files for a program be located in the same directory.

The scanner can handle C source code files and include files. In addition the scanner can handle files used by Yacc and Lex. Yacc input files should end with the postfix ".y". Lex input files should end with the postfix ".l". System supplied include files (such as `stdio.h`) should not be passed to the scanner.

One possible source code error that the scanner does checks for is comments missing the end-of-comment mark. C does not allow comments to be nested. When a begin-comment mark is found most C compilers will scan until an end-of-comment mark is found. If a comment was missing its end-of-comment mark then the end-of-comment mark found will be for the next comment in the file. The problem is that between the comment with the missing end-of-comment and the next comment there is usually source code that is now treated as part of a comment and therefore "disappears" from the program. This causes the program to run incorrectly.

When the scanner detects the beginning of a comment, it will scan through the program until it reaches an end-of-comment. If a begin-comment mark is detected inside the comment then a warning message will be generated by the lexical analyzer. This warning will inform the user that an end-of-comment may be missing. If the scanner hits

the end-of-file while searching for an end-of-comment mark then an error is generated. To run the scanner, it should be invoked in the directory containing the program files. The files names passed to the scanner should not contain path names (unless full path names are used). Source files that are used as input to the Unix tools Yacc or Lex may be listed in the list file. Source files that are input to Yacc should end with ".y". Source files that are input to Lex should end with ".l". The scanner will pass these files through Yacc or Lex before scanning them.

When the scanner is invoked several parameters must be passed. These parameters are passed as command line arguments and tell the scanner about the program to be scanned. The ordering of parameters is important. The syntax for the scanner invocation is shown below.

```
scanner -p program [-w] { -f filename |  
                    -l filename [ filename ]* }
```

The parameters are :

- f : This parameter is followed by a filename. The file must contain a list of all the source code and include files, listed one file per line. These are the files used by the program to be debugged. The scanner reads this file and scans each file listed. This parameter and the "-l" parameter can not both be used.
- l : This parameter tells the scanner that each source file used in the program is listed on the command line. The list of file names must follow the parameter. Filenames should not be separated by commas. This parameter and the "-f" parameter can not both be used.
- p : This parameter is followed by the name of the user's program, i.e. the name of the program that Amber is to debug. This is not the name of any source code file, but is a name used by the programmer (i.e user of Amber) to identify the pro-

gram. The scanner will use this name to create the program information file.

-w : This parameter tells the scanner to print the parse tree generated for each file.

This is not the parse tree encoded as facts, but is the real parse tree. This parameter is only used to help debug the scanner.

6.3.1.2. Scanner Output

The scanner produces a set of output files. For each source file scanned a fact file is produced that describes the code contained in the file. Another file called the program information file contains information that lists the source filenames for a program, the name of each function defined in the program, and tells where each function is defined.

Program Information File

The program name passed to the scanner, via the "-p" parameter, is used to name the program information file. The program information file is given the postfix ".fcr". For example, if the program name is "tracer" then the program information file name is "tracer.fcr". For each source file containing program text the file name is stored as a fact in the program information file. The name of each function defined in the program is stored as a fact along with the file name where the function is defined. The format of the program information file is described in a later section.

Source Code Fact File

For each source file that is scanned the scanner produces a fact file that describes the structure of any source code contained in the file and a list of "include" files used in the source file. A fact file has the postfix ".fct". For example, if the name of a source file is "prterr.c" then the fact file name will be "prterr.fct".

6.3.1.3. Scanning Procedure

For each source file the scanner performs the following steps :

- 1) If the source file has the postfix ".y" or ".l" the scanner will ask the programmer if the file is a Yacc or Lex input file respectively. If it is an input file then the scanner will pass the file to the correct utility for processing. This will result in an output file named y.tab.c or lex.yy.c. The output file will be treated as the source file for the remaining steps.
- 2) The source file will be passed to the C preprocessor (part of the C compiler). This step removes any defined constants, macros, "ifdef" statements, any statements that are used by the preprocessor only. The resulting source code will be saved in a temporary file based on the name of the source file but with the postfix ".cpz".
- 3) The "cpz" file is opened.
- 4) The parser is called. The parser will read the input file and create a parse tree that represents the structure of each function defined in the file. A list of include files referenced in the file will be made also.
- 5) The "cpz" file is closed and dropped. The scanner makes only one pass through the file. Note that if the scanner did create a lex.yy.c or y.tab.c file it does not drop that file.
- 6) If the "-w" parameter was set when the scanner was called then the parse tree in printed at this time. This step is only used when debugging the scanner.
- 7) The fact generator is called. The generator will create the fact file for the source program. If the source program is an input file to Yacc or Lex then the name of the fact file is still based on the source file name. The parse tree and include file list are converted into facts that describe the source file. The program information

file is also updated at this time.

After a file has been processed the parser environment is cleaned. Allocated memory is freed and global pointers are reset. The scanner is then ready to process the next file.

6.3.1.4. Scanner Components

The scanner performs two major functions, scan a file and generate facts. The major components for scanning a file are the C language parser and lexical analyzer. The main component for generating facts is the fact generator. These components are presented below.

Lexical Analyzer

The lexical analyzer reads the source file and passes tokens to the parser that describe what was read. The lexical analyzer will accept all C keywords, operators, comments, numeric, and character constants. It does not accept a source file that contains preprocessor commands. A source file is preprocessed before being scanned. Because of this information about defined constants and where they are used is lost. On the other hand, the source file may not be syntactically correct before it is preprocessed.

The lexical analyzer is produced by the UNIX utility Lex. Lex produces a function called `yylex()` that is used by the parser.

C Language Parser

The parser will accept any valid C construct, except a "typedef" declaration. There is a lexical problem related to correctly recognizing such a statement. The parser grammar is based on a LALR grammar for the C language obtained from [HARBI84]. The parser assumes that any file passed to it can be successfully parsed by the C compiler `cc`. The parser does little error checking itself.

The grammar is written in a format accepted by the UNIX utility Yacc. Yacc generates a parser function named `yyparse()`. This parser calls the lexical analyzer to obtain source file information.

Fact Generator

The fact generator is a set of functions (`fact_walker()`) that transverse the parse tree and encode the tree information into facts readable by YAPS. Care is taken that the facts generated describe the structure of the source code and show the order in which statements and expressions will be executed.

One fact is created for each function defined in the file. The structure of the function body is expressed using a list structure. More information about how source code is represented is given in the next section.

6.3.2. Representing Program Knowledge

The scanner produces two groups of facts: program information facts and source code structure facts. Each group of facts is stored in a separate file. Only the program information facts are global to an entire program. A source code fact file is generated for each source code and include file used to create the program.

6.3.2.1. Program Information File

This file contains facts (i.e. information) about each source code and include file used to create the program and the functions defined in each file. Only one program information file is created for a program. The information in this file specifies the names of source code files used to create the program and the name of each function written as part of the program. System functions used by the program are not listed in this file. The facts stored in a program information file are:

FUNC :

Specifies the name of a function defined in a source file. This fact tells Amber the name of the source file where the code for a function is given. The format of this fact is :

(FUNC function_name source_file)

The "function_name" is the name of the function. No data declaration information for the function is given. The "source_file" is the name of the source file where the function is defined.

SRC :

Denotes the name of a source code or include file used to create a program. This fact tells Amber the name of a source file used in a program. The format of this fact is :

(fact SRC source_file)

The "source_file" is the name of a source code or include file.

As an example consider the C program shown in figure 6.2. The name of this program is "countdown" and the source file name is "countdown.c". This file contains the definition of a function called "main."

```

#include <stdio.h>

main()
{
    int i ;

    i = 10 ;
    while ( i > 0 )
    {
        i-- ;
        printf( "Count down at : %d\n", i );
    }
    printf( "BOOM!\n" );
}

```

Figure 6.2 : Example C Program

The program information file created by the scanner would be named "countdown.fcr". A program information file for the program in figure 6.2 is shown in figure 6.3. Notice the file shows the name of the source code file and the name of the only function defined in that file.

```

( fact SRC countdown.c )
( fact FUNC main countdown.c )

```

Figure 6.3 : Program Information File

6.3.2.2. Representing C Code

Before discussing the information in a source code fact file it is necessary to describe how C source code is represented. In chapter 4 it was explained that an abstract syntax representation for C source code would be an a suitable representation style. The implementation of this representation is presented in this section.

Much of the structure in C programs results from the need to express the flow of exe-

cution control in a function. In denotational semantics control, is modeled using "continuations." Continuations can be modeled as stacks where a "control" stack keeps a list of all the statements that need to be evaluated. As a program is executed the stack is repeatedly accessed and the top statement in the stack is popped off. An empty stack means that evaluation is complete. As an example, consider the short code example below:

```
if ( val == TRUE )
    set-of-statements1
else
    set-of-statements2
next-statement
```

Initially, the control contains the IF statement at the top of the stack and next-statement is immediately under it. The set-of-statements that form the THEN and ELSE parts of the IF statement are part of the IF statement. At execution time, the IF statement is popped off the stack and the expression "val == TRUE" is evaluated. If this expression is true, then set-of-statements1 is pushed on the control stack. Then the first statement in set-of-statements1 is popped and executed. After all the statements in set-of-statements1 has been executed, next-statement is popped and executed.

To use a similar method of representing the structure of C code it was decided to represent each statement as a list. A block of statements then becomes a list of lists and the code for a function is represented as a list containing a list of lists. Figures 6.4 and 6.5 show the list format used to represent each C statement and expression type.

(B (body)) ; Block of Code. Body is a list of
; statements ((S1) (S2) (S3)...)

(S C (expr) (stat)) ; case expr : statement

(S D (stat)) ; default : statement

(S E (expr)) ; Expression statement.

(S L ident (stat)) ; Labeled statement: ident : stat

(S I (expr) (stat)) ; IF (expr) statement

(S IE (expr) (stat1) (stat2)) ; IF (expr) stat1 ELSE stat2

(S DW (stat) (expr)) ; DO statement WHILE expr

(S W (expr) (stat)) ; WHILE (expr) statement

(S F (expr1) (expr2) (expr3) (stat))
; FOR (expr1 ; expr2 ; expr3)
; statement

(S S (expr) (stat)) ; SWITCH (expr) statement

(S G ident) ; GOTO ident

(S B) ; BREAK

(S N) ; Null statement

(S CN) ; CONTINUE

(S R (expr)) ; RETURN expr

Figure 6.4 : List Representation of C Statements

(E (expr1) op (expr2))	: op is any operator that takes : two arguments
(E op (expr))	: op is any prefix unary operator
(E (expr) op)	: op is any postfix unary operator
(E CT (cast) (expr))	: Cast expression
(E SZ (expr))	: Sizeof expression
(E IF (e1) (e2) (e3))	: e1 ? e1 : e3 IF expression
(E I ident)	: Identifier
(E N number (type) (size))	: number where type = I (integer), : F (float), H (hex), or O (octal) : and size is D or L. D = default : L = LONG
(E S string)	: String constant
(E C char)	: Character constant
(E P (expr))	: Expression enclosed in ()
(E A (expr1) (expr2))	: expr1[expr2]
(E F (expr1) (expr2))	: expr1(expr2) function call : where expr1 is the function name : and expr2 is the parameter list
(E SP (expr))	: Comma expression. Where : expr = (() () ...)

Figure 6.5 : List Representation of C Expressions

In this scheme of representing C code using lists a "B" as the first element in a list specifies a block of statements. A "S" specifies a statement list and an "E" specifies an expression list. Lists are nested to show the structure and execution order of the elements. For example, for the expression statement:

speed = dist / velocity ;

The variables "speed," "dist," and "velocity" are represented as

```
( E I speed )
( E I dist )
( E I velocity )
```

The division operation is expressed as:

```
(E (E I dist) DIV (E I velocity) )
```

Where "DIV" specifies the division operator. Several of the operators in C are overloaded so a unique name is used to specify the operator, rather than use the operator symbol itself. The entire statement would be represented as:

```
(S E (E (E I speed)
  ASSIGN
  (E (E I dist) DIV (E I velocity))
) )
```

To evaluate this expression the division expression must be evaluated first and then the assignment can be done.

Figure 6.6 shows the representation for the function block from function "main" shown in figure 6.2.

```
(B ( (S E (E (E I i) ASSIGN (E N 10 I D )))
  (S W (E (E I i) GREATER THAN (E N 0 I D ))
    (B ( (S E (E (E I i) POST_SUB ))
      (S E (E F (E I printf)
        (E SP ((E S ! "Count down at : %d0)
          (E I i))))))
    )
  (S E (E F (E I printf)(E S ! "BOOM!0))))
)
```

Figure 6.6 : Representation of a Statement Block

The body of the block contains three lists. Each list corresponds to a statement in the function. To evaluate this block, the three statements would be pushed on a stack.

The first statement, an assignment statement, would be popped and executed. Next the While statement would be popped. The body of the While statement is part of the statement. If the test expression for the loop evaluated to true, the While statement would be pushed back on the stack followed by the statements in the body of the loop. The reason for pushing the While statement back on the stack is that after the statements in the body of the loop have executed, the loop test expression must be evaluated again. If it is still true then the body of the loop must be evaluated again. After the loop is finished executing, the third statement, an expression statement, is executed. This statement contains a function call. If the function was a function written as part of the program instead of a system library function such as "printf," then the list containing the body of that function would be pushed on the stack and the execution would continue with the first statement in the function. The list of statements that form the body of a function is referred to as a "continuation." In the implementation of Amber the continuation for a function refers to the list of statements for that function.

6.3.2.3. Source Code Fact File

Now, that the representation of C source code structure has been covered, the source code fact files produced by the scanner can be presented. Such a file is created for each source code or include file used to create the program. The contents of this file are obtained from the parse tree created by the scanner. The facts contained in this type of file are explained below.

FBOD :

This fact contains the name of a function and a continuation list. The list is an encoding of the body of the function. The format of this fact is:

(FBOD func_name body)

Body is the continuation list and func_name is the name of the function.

INC :

Denotes the name of an include file referenced in a source file. Because an include file may contain source code it may be useful to know which files are included (via the "include" preprocessor statement) in a source file. This fact gives the name of an include file included in a source file. The format of the fact is :

(INC inc_file_name at_line source_name)

The "inc_file_name" is the name of the include file. This name can contain path information. (Currently, the scanner expects all program files to be contained in the same directory, but system include files can have path names attached.) The "at_line" is the line number in the source file where the include statement occurs. The "source_name" is the name of the source file that uses the include file. Figure 6.7 shows the source code fact file that would be produced for the C program given in figure 6.2.

```
( fact INC /usr/include/stdio.h 1 countdown.c )
( fact FBOD main ( B ( ... ) ) )
```

Figure 6.7 : Source Code Fact File

The body of the function main is shown in figure 6.6. Figure 6.7 shows that the source file uses one include file and contains the code for one function.

6.4. Symptom Analyzer

The Symptom Analyzer consists of five rules which are given in the file "symptoms.1". The analyzer is invoked by the goal "run symp_anal" and the first rule to fire is always

start_analyzer. This rule is responsible for classifying the major symptom of the software failure that is appearing. To do this, the programmer is presented with a menu, shown in figure 6.8 and asked to select a category.

Which of the following best describes the problem your program is demonstrating?

- a) The program does not terminate.
- b) The program terminates with a system error message.
- c) The program generates incorrect results.
- d) None of the above seem to fit the problem.
- e) Quit the debugger.

Select a letter :

Figure 6.8 : Symptom Menu

When the programmer identifies a symptom from the menu a symptom fact is created that saves that piece of information. A symptom fact has the form:

(SYM type)

where "type" specifies the type of symptom. The **start_analyzer** rule will produce one symptom fact that describes the category selected. The symptoms fact created will be one of these three:

(SYM No_term) :

The program does not terminate.

(SYM Sys_kill) :

The program is terminated by the system.

(SYM Bad_output) :

The program output is not correct.

Once, the symptom fact is created **start_analyzer** is done. The next rule to fire will depend on which symptom was specified by the programmer. If the programmer

specified that the program has a core dump then the rule `sys_abort` will fire. The lhs of this rule is shown in figure 6.9. The two patterns in the lhs specify that Amber must be running the Symptom Analyzer and that the program is terminated by the system. If the programmer specified that the program does not terminate then the rule `no_end` will fire and if the programmer specified that the program produces incorrect output then the rule `bad_out` will fire.

```
(defp sys_abort
  (goal (run symp_anal))
  (SYM Sys_kill)
  •
  •
  •
)
```

Figure 6.9 : Condition Pattern for Rule "sys_abort"

These rules each try to collect more information about the symptoms of the failure and deduce a set of suspected errors. The complete list of symptoms that the Symptom Analyzer can generate is shown in figure 6.10.

`Bad_output` :

The program output is not correct. The symptoms `Bad_result` and `Garbled` are used to specify how the output is incorrect.

`Bad_result` :

The program output is incorrect. The incorrect values are reasonable in range, but wrong.

`Garbled` :

The program output is incorrect. The incorrect values are out of range or clearly ridiculous.

`No_term` :

The program does not terminate.

`No_output` :

The program does terminate but it produces no output.

`Sys_kill` :

The program is terminated by the system.

Too_little :

The program generates too little output. It generates less output than the programmer expected.

Too_much :

The program generates too much output. It generates more output than the programmer expected.

Wrong_sec :

The program seems to be executing a section of program that the programmer does not think it should be in.

Figure 6.10 : List of Symptoms

Once the symptoms are collected the analyzer can generate the set of suspected errors. For a given set of symptoms there can be several suspected errors where different errors are given weights according to their likelihood. A suspected error fact is denoted as a "hypothesis" and has the form:

(HYP type weight)

The "type" element specifies the type of error suspected. The "weight" element specifies how likely the system thinks the suspected error is. A weight of zero is a middle ground and positive or negative weights are used to specify more or less likely. For example, if a program does not terminate then one suspected error is an infinite loop. This error is always given a weight of 2. Whereas deadlock, which is the other suspected error, is given a weight between -2 and 1 depending on how likely Amber decides that error is. The complete list of suspected errors that Amber work with is shown in figure 6.11. This list includes suspected errors for all groups of symptoms: non-termination, system abort, and bad output.

Bad_format :

The print statement used to print the output may not be correct. It could contain problems such as: data type specifier in format does not match a variable's data type, missing variable, missing variable specifier in format, etc.

Bad_path :

The program may have a bad decision point. For example, an If statement where

the test logic is incorrect or a While loop where the test expression always prevents the loop from being entered.

Bad_value :

The program may contain an incorrect calculation or assignment. For example, an equation that is mistyped and contains the wrong operator or variable, or is missing a variable, or a calculation that assigns its value to the wrong variable.

Buf_ovrflw :

The program generated a core dump and a likely cause is because a buffer was overflowed.

Deadlock :

The program is entering into deadlock.

Flt_div0 :

The program generated a core dump and the system error message specified a floating exception. One possible error is division by zero using floating point arithmetic.

Flt_ovrflw :

The program generated a core dump and the system error message specified a floating exception. One possible error is floating point overflow.

Flt_undflw :

The program generated a core dump and the system error message specified a floating exception. One possible error is floating point underflow.

Inf_loop :

The program is entering into an infinite loop.

Int_div0 :

The program generated a core dump and the system error message specified a floating exception. One possible error is integer division by zero.

Int_ovrflw :

The program generated a core dump and the system error message specified a floating exception. One possible error is integer arithmetic overflow.

Unk_dump :

The program has a core dump and the debugger does not know what could cause that type of core dump.

Unknown :

Amber does not know how to handle the problem. This hypothesis will be used as the debugger is developed to indicate a section that still needs work.

User_quit :

The program generated a core dump, but most likely because the user did a control backslash.

Figure 6.11 : List of Suspected Errors

The weights assigned to suspected error are subjective. Typically, a weight of zero is used, just to specify that the error could occur in this situation. A positive or negative weight is only used if the set of symptoms suggests that an error is not as likely as

other errors or is more likely. That is why an infinite loop is given a weight of 2, because this is almost always the cause of a program not terminating.

In addition to symptom and suspected error facts, the Symptom Analyzer creates one more type of fact. This is the CORE_DUMP fact and is only used if a program is aborted by the system. The CORE_DUMP fact is used to remember which message was displayed by the system when the program was aborted. This fact has the form:

(CORE_DUMP message)

Where "message" is not the message issued by the system, but a keyword the specifies which message was given. The list of core dump messages is shown in figure 6.12.

```
SIGQUIT - quit
SIGILL  - illegal instruction
SIGTRAP - trace trap
SIGIOT  - IOT instruction
SIGEMT  - EMT instruction
SIGFPE  - floating point exception
SIGBUS  - bus error
SIGSEGV - segmentation violation
SIGSYS  - bad argument to system call
```

Figure 6.12 : Core Dump Messages and Keywords

The rule `end_analyzer` will fire when the Symptom Analyzer is finished. This rule removes the goal "run symp_anal" and creates the goal "select error." This new goal will cause control of the debugging session to shift to the Select Error component.

7. Select Error

The Select Error component examines the set of suspected error facts and selects the error with the greatest weight. If two or more errors have the same weight then the error that was added most recently to the YAPS database will be selected. The Select Error component consists of one rule named `select_error` and is given in the file

"control.1." Figure 6.13 shows a portion of this rule. The lhs of the rule selects the suspected error fact (HYP) with the greatest weight. The rhs of the rule contains a list of all possible suspected errors. For the selected error a goal is created that specifies how the debugging session is to proceed.

```

(defp select_error
  (goal (select error))
  (HYP -suspect -weight) ; Select greatest weight
  (~ (HYP -other -mass) with (> -mass -weight))
  ->(remove 1)
  (cond ((= -suspect 'Bad_path)
         ; Suspected cause is a bad logic decision in the program.
         (goal explain unknown)
        )
        ((= -suspect 'Buf_ovrflw)
         ; Suspected cause is a buffer overflow.
         (goal trace coredump)
        )
        ((= -suspect 'Deadlock)
         ; Suspected cause is a deadlock.
         (goal trace deadlock)
        )
        ((= -suspect 'Inf_loop)
         ; Suspected cause is an infinite loop.
         (goal trace loop)
        )
        ((= -suspect 'User_quit)
         ; Suspected cause is a core dump caused by the user interrupting
         ; the program.
         (goal explain coredump)
         •
         •
        )
        ((= -suspect 'Unknown)
         ; The debugger is not prepared to handle the symptoms.
         (goal explain unknown)
        )
  )
) ; end of rule select_error

```

Figure 6.13 : select_error Rule

The goal created by `select_error` specifies how Amber will proceed with the debugging session. Figure 6.13 shows a suspected error for each type of goal that can be created.

These goals are explained below.

explain coredump :

If the suspected error is *User_quit* which can occur with a core dump, then Amber does not need to trace the execution of the program. This is a type of core dump caused by a user generated interrupt to a program. This solution is explained by the *user_quit* rule in the file "coredump.1."

explain unknown :

This goal is generated when Amber realizes it does not know how to proceed. The rule *stumped* in the file "control.1" will fire and inform the programmer that Amber is unable to proceed with the debugging session.

trace coredump :

This goal directs Amber to trace through the program execution looking for a statement where a core dump occurs.

trace deadlock :

This goal directs Amber to trace through the program execution looking for a statement that contains a function that is called, does not return, and for which Amber does not have the source code. (Remember the deadlock detection technique given in chapter 3.)

trace loop :

This goal directs Amber to trace through the program execution looking for a loop that is entered and does not exit.

Once, *select_error* has created the new goal the Select Error component is done. If the new goal is a "trace" goal then the Locate Error component will begin. If the new goal is an "explain" goal then the appropriate explanation will be given to the programmer and then the *more_work* rule, from the file "control.1," will fire. This rule prompts

the programmer to either exit from Amber or go on and debug another error.

7.1. Locate Error

The Locate Error component consists of 93 rules, in the file "trace.l," and is the largest single rule group in Amber. These rules embody the knowledge needed to understand C language constructs and to trace the execution of a program. The rules for this component also incorporate the Execution Oracle component. Currently, execution information must be obtained from the programmer. This knowledge is gained by prompting the programmer, so the oracle is really a set of prompts that appear in the rhs of rules where execution knowledge is needed.

To trace an error the Locate Error component steps through the statements and expressions in a function. The execution trace can move from one function to another as functions are called. How a function is handled by this component is explained below.

- 1) The execution trace determines that a function is entered. A `get_cont` goal is created that specifies the trace must transfer to this function.
- 2) If there is no FBOD fact defined for this function then the rule `need_func_cont` will fire. This rule will use the FUNC fact for this function to locate the source code fact file that contains the continuation for this function and will load that file.
- 3) When the FBOD fact exists in the YAPS database then the rule `get_func_cont` will fire. This rule makes a fact for the first statement in the function body and puts the continuation for the rest of the function in that fact. Once, this is done the rule for handling that statement type will fire and the trace has entered this function.

The Locate Error component has rules for C expressions, C statements, and rules for reasoning about the execution flow in a program. To explain how these different rule groups operate a rule from each is presented. To begin this discussion a rule for reasoning about expressions is shown in figure 6.14.

```

; This rule handles an expression of the form
;   ( E (expr1) MULT (expr2) )
; Where MULT is the "*" operator used to do multiplication.

(defp exp_mult_oper
  (goal (trace -type) )
  ( E -lexp MULT -rexp )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          ; Do nothing for this type of expression.
          ; Check for function calls.
          ( make_fact -rexp )
          ( make_fact -lexp ) ; Check this first.
        )
  )
) ; end of rule exp_mult_oper

```

Figure 6.14 : Expression Handling Rule

The goal "trace -type" is a pattern that matches any "trace" goal. The Locate Error component uses several different methods of tracing an execution and in addition uses several other "trace" goals internally. A separate rule could be written for each type of trace since the set of actions on the rhs can be different for each one. However, YAPS is notorious for running out of memory when too many rules are used. So, as an implementation issue, the set of actions required for each type of trace have been collected into one rule for each lhs side. The variable -type then specifies the style of tracing to use.

The rule shown in figure 6.14 is designed to work with multiplication expressions such

as "a * b". This expression is represented as:

$$(E (E I a) MULT (E I b))$$

For the trace types "coredump," "loop," and "deadlock" we are only interested in knowing if either the expression to the right or left of the multiplication operator contains a function call. To accomplish this the left and right expressions need to be examined by the appropriate rules. These two expression lists are passed to the function **make_fact** which is given in the file "trace.l." This function takes a list and generates a fact that is correctly formatted. For the expression shown above a fact for the right side expression is generated first and then for the left side expression. Because of this the left side expression will be examined first and then the right hand side expression. This order of checking corresponds to the evaluation order performed when the expression is executed. In this way, the semantics of the multiplication operator is given in the knowledge base. This style of handling expressions is similar for other types of expressions.

C statements are more complicated to reason about. Figure 6.15 shows the rule for handling an IF-THEN-ELSE statement. Notice that the second pattern in the lhs contains an extra variable *-cont*. This variable holds the continuation, which is a list containing the remaining statements in the function being traced. The rhs of the rule is a large Lisp cond statement that contains the set of actions required for each trace style.

```

: This rule handles a statement of the form
: ( S IE (expr) (stmt1) (stmt2) )
: IF (expr) stmt1 ELSE stmt2

(defp stmt_bal_if
  (goal (trace -type)
    (S IE -test -then -else -cont)
    -> (remove 2)
      (cond ((or (equal -type 'loop) (equal -type 'deadlock))
        ; Check for function calls in the test first.
        (prt_stmt (list 'S 'IE -test))
        (fact B_IF -test -then -else -cont)
        (make_fact -test))
        ((or (equal -type 'default) (equal -type 'case)
          (equal -type 'label))
        ; Tracing a default,label, or case statement
        (make_fact -cont)); Follow continuation.
        ((equal -type 'coredump)
        ; Does the coredump occur for this statement?
        (prt_stmt (list 'S 'IE -test))
        (prog (ans)
          repeat
            (msg N T
              "Does the test expression execute successfully (y or n)? "
              (setq ans (call_help (read)
                ("HELP MESSAGE OMITTED" )))
              (cond ((null ans)
                ; Repeat the prompt
                (go repeat))
                ((equal ans 'y)
                ; This statement is OK so continue on.
                (fact B_IF -test -then -else -cont))
                ((equal ans 'n)
                ; The core dump occurs on this line
                ; or in a function call on this line.
                (fact DUMPED -test)
                (make_fact -test))
                (t ; Invalid input
                (msg N T "**** Answer using 'y' or 'n'" N)))
              )))
        ((equal -type 'break)
        ; Following a Break statement jump.
        ; Drop the continuation
        ())
        ((equal -type 'continue)
        ; Trying to find the top of a loop.
        ; Drop the continuation.
        ()))

```

```

    ( t ; Bad trace type abort
      ( msg N T
        " *** Internal error : bad trace type : " -type N )
        ( halt ))
    )
) ; end of rule stmt_bal_if

```

Figure 6.15 : Statement Handling Rule

When the rule `stmt_bal_if` shown in figure 6.15 fires, the first action removes the IF statement fact from the YAPS database. That fact has been used successfully to fire this rule and is no longer needed. The information in that fact will be used to create new facts before the rule is finished.

The next action depends on the type of tracing that Locate Error is using. The trace type is specified by the variable `-type` in the first pattern in the lhs of the rule. If the trace type is "loop" or "deadlock," meaning that an infinite loop or deadlock situation is being traced then three actions are performed. First, the first line of the IF statement is displayed to the programmer. The first line is the of the form:

```
if (test_expr)
```

This allows the programmer to see which statement in the code the trace step has reached. The actual code from the source code files is not saved by Amber, so the function `prt_stmt` is used to convert the list representation of the statement back into C format for display. Secondly, a "B_IF" fact is created. This fact specifies to Locate Error that the value of the test expression in the IF statement must be determined so that the correct path can be traced. This fact will be used later by the rule shown in figure 6.17. Finally, the last action is to create a fact for the test expression. Because the type of trace is "loop" or "deadlock" the tracer must determine if there is a function call in the test expression, and if so, does it return. This is done by checking each sub-list in the expression list, via other rules, for a function call. If no function call that does not return is found then the rule `bal_if_branch` will fire because of the "B_IF"

fact.

The next clause in the rhs of the rule is for the trace types "default," "case," and "label." These are internal trace modes used by Locate Error that temporarily override the current trace type. These trace types are explained below.

default :

This trace type specifies that the tracer is attempting to find the Default labeled statement inside a Switch statement.

case :

This trace type specifies that the tracer is attempting to find a Case labeled statement inside a Switch statement.

label :

This trace type specifies that the tracer is attempting to locate a labeled statement.

This trace type is used when a Goto statement has been reached and the tracer is attempting to find the statement to which the execution jumps.

For each of these trace types, the tracer is attempting to locate a specific statement in the body of the current function. For "default" or "case" the statement sought will not be inside the body of the IF statement so the IF statement is skipped and the search will continue with the statement following the IF. For the trace type "label" there is a problem. The tracer assumes that the labeled statement will not be in the body of the IF statement, but actually it could be. The problem is that to search the body of the THEN and ELSE portions of the statement both will need to be placed in the continuation and searched sequentially. If the labeled statement is found in the THEN portion then there is no way to specify in the continuation that the ELSE portion should be skipped. This a problem introduced by the messy semantics of the Goto statement and for now the tracer just lives with it, but it is not handled well at this time.

The next set of actions is for the trace type "coredump". The first action is to display the first line of the IF statement so the programmer will know where the trace is at. Then the Execution Oracle is invoked and a prompt is sent to the programmer asking if the test expression executes successfully, i.e. did the core dump occur here. If the expression is executed successfully then a "B_IF" fact is created. Because the test expression executed successfully it is not necessary to examine the expression, but to continue the trace the tracer needs to know which part of the IF statement is executed. If the test expression did not execute successfully then a "DUMPED" fact is created. This fact specifies a statement or expression where the core dump occurs. This is the core dump location. Also, the test expression is made into a fact. If the test expression contains a function call that does not return then this is not the location of the core dump and the function that was called must be traced. Here the "DUMPED" fact created earlier will be deleted.

The last two sets of action for the rule are for trace types "break" and "continue." These are internal trace types used when the tracer encounters a Break or Continue statement. A Break statement transfers execution out of a loop or Switch statement and a Continue statement transfers execution to the top of a loop that contains the Continue statement. To determine where a Break or Continue statement will cause the execution to jump, two fact types are used: "BREAK" and "CONTINUE."

The "BREAK" fact is created when a Switch, While, Do-While, or For statement is encountered and is created for use when a Break statement is reached. When one of the above statements is reached and the trace enters the body of the statement then a "BREAK" fact is created to show where a break will jump to. For a Switch statement the fact created just before the "BREAK" fact is a fact containing the continuation for the rest of the function. If a "BREAK" fact is reached while tracing a loop then it is

likely that the programmer has entered wrong information about whether the loop exits. A "BREAK" fact has the form:

(BREAK type)

where "type" indicates the type of statement the Break will break out of.

The "CONTINUE" fact is created when a While, Do-While, or For statement is encountered. The fact is created for use when a Continue statement is reached. When one of the above statements is reached and the trace enters the body of the statement then a "CONTINUE" fact is created to show where a continue will jump to.

The above discussion has presented in detail how the `stmt_bal_if` rule operates. Other rules for handling statements operate similarly. After the rule in figure 6.15 fires, it is usually necessary to know the value of the test expression in the IF statement so that the trace can continue either in the THEN or ELSE section of the IF statement. This piece of information must be obtained from the Execution Oracle. The rule `bal_if_branch` is used to collect this piece of information and act on it. The rule `bal_if_branch` is shown in figure 6.16. Notice that this rule uses the "B_IF" fact and the first action of the rule is to remove this fact once it has been used.

```

; This rule determines if the test expression for a balanced IF
; statement evaluates to zero or not. Based on this value the
; trace continues with the body of the "then" portion or the "else"
; portion of the IF statement.

```

```

(defp bal_if_branch
  (goal (trace -type) )
  ( B_IF -test -then -else -cont )
  ->( remove 2 )
  (cond ( (and (equal -type 'loop) (equal In_loop 'true) )
        ; We suspect an infinite loop and are inside a loop.
        ( make_fact -cont )
        ( make_fact -else )
        ( make_fact -then )
        (msg N T "The debugger will check the body of the THEN section of the")
        (msg N T "IF statement and then the ELSE section. The search will")
        (msg N T "then continue with the first statement after the IF." N )
        )
    ( t ; For other trace type.
      ( prog ( ans )
        again
        (msg N T
          "Is the value of the test expression zero (false) or non-zero (true)?" )
          (msg N T "Pick false or true (f or t)? " )
          ( setq ans ( call_help (read)
            ("HELP MESSAGE OMITTED" ) ) )
          ( cond ( (null ans )
                  ; Repeat the prompt
                  ( go again ) )
                ( (equal ans 'f)
                  ; Yes, the "else" portion is followed.
                  (msg N T "Continue tracing with the first statement inside the")
                  (msg N T "'else' portion of the IF statement." N )
                  ( make_fact -cont )
                  ( make_fact -else ))
                ( equal ans 't)
                  ; The "then" portion is executed.
                  (msg N T "Continue tracing with the first statement inside the")
                  (msg N T "'then' portion of the IF statement." N )
                  ( make_fact -cont ) ; continue on
                  ( make_fact -then)) ; Trace the IF body first
                ( t ; Invalid input
                  ( msg N N T "*** Invalid input, try again!" N )
                  ( go again ) )
                )
          ))))
    ) ; end of rule bal_if_branch

```

Figure 6.16 : Execution Flow Handling Rule

There are two sets of actions on the rhs of this rule. If the tracer is searching for an infinite loop and if it is currently tracing the body of a loop that does not exit then the tracer does not need to know the value of the test expression in the IF statement. It was explained in chapter 4 that when a possible infinite loop is traced, branching is ignored and all statements are searched sequentially for function calls that do not return and inner loops that do not exit. To conduct this search a fact is created using the continuation for the IF statement. This fact contains the statement following the IF statement and a continuation for the remaining statements in the loop. A fact is created for the ELSE section of the statement. This is a fact for the first statement in the ELSE section and a continuation containing the remaining statements in the ELSE. Finally, a fact is created for the THEN section of the IF statement. This is a fact for the first statement in the THEN section and a continuation containing the remaining statements in the THEN section. The statements in the THEN section will be examined first, followed by the statements in the ELSE section and finally the statements following the IF will be examined. This ordering makes full use of the technique used by YAPS to select the next rule to fire.

If the tracer is not tracing the inside a loop that does not exit then the second set of actions in the rhs of the rule is used. The first action determines if the test expression in the IF statement evaluates to zero or non-zero. (C does not use a boolean value for conditional expressions.) This information is obtained from the Execution Oracle, which is a prompt issued to the programmer. If the programmer specifies that the test expression is non-zero, then the THEN section of the IF statement will be executed. The continuation for the remainder of the function is first made into a fact. This fact contains the statement following the IF statement. Then the THEN section is made into a fact. This fact contains the first statement in the THEN section and a continuation

containing the remaining statements in that section. If the test expression is zero then the ELSE section of the IF statement is executed and a fact is created for that section of code.

This process of stepping through statements and expressions in the program source code continues until a section of code is found whose behavior matches that predicted by the trace type. The Locate Error component can sometimes detect when the programmer incorrectly supplied information that has led the trace astray. Currently, to recover from this Amber informs the programmer that the execution trace is lost and offers to start the trace over.

The Locate Error component uses what are called "aha" rules that detect when an error has been located in the source code. The rule `aha_loop` detects when an infinite loop has been detected. This rule removes the goal to trace the program execution and creates a goal to explain the error to the programmer. This goal "explain loop" causes control to transfer to the Explain Error component. This rule also creates a "clean trace" goal. This goal will enable rules to fire that remove facts related to performing a trace. These rules are given in the file "cleanup.l."

The rule `aha_coredump` detects when the location of a core dump occurrence has been found. This rule removes the goal to trace the execution and creates a goal to show the dump location to the programmer. This is the "show dump" goal. This rule also creates a "clean trace" goal.

7.2. Explain Error

The rules for the Explain Error component are divided into several files: "forever.l," and "coredump.l." This component is partially implemented at this time. When the source of the error is found the statement is shown to the programmer. For an infinite

loop the programmer is told that an infinite loop was found. The rule `show_loop` in the file "forever.l" shows the loop statement. When the statement where a core dump occurs is found the rule `show_dump` is used to show the statement or expression to the user.

7.3. Locate/Determine Fault

This component is not implemented.

Chapter 7

Conclusions

The concept of a program debugging tool, that can use knowledge about how to debug programs and assists a programmer with the laborious task of debugging, has been presented in this work. The motivation for this work is to relieve the programmer of a portion of the debugging effort. Twenty years ago the debugging tools available to a programmer were the print statement and "snapshots" or dumps of the program taken as it executed. In the early eighties, "break-and-see" debuggers became available that allowed a programmer to examine a program while it executed. These "tools" helped in the debugging process, but their purpose was solely to provide the programmer with information. The programmer was the debugging expert who needed to sort through the information and determine what was going wrong, where it was going wrong, why it was going wrong, and how to fix it. The debugging expertise that is developed over time by programmers can be incorporated into a knowledge based tool. Such a tool assists the programmer with the task of interpreting debugging information.

The process of debugging a program is a diagnostic process. The usefulness of knowledge based tools for diagnostic activities has been shown in several domains: such as medicine and equipment repair. However, knowledge based tools for diagnoses usually are designed to reason about one particular system. For example, a medical system for determining what virus is afflicting a human, will not work on chickens. A

diagnostic system for locating faulty components in a particular model of disk drives will not work well if asked to reason about another model. However, a knowledge based diagnostic system for debugging programs must be able to reason about many different programs. It is not practical to develop such a tool to debug just one program. Therefore, the knowledge in the KBPD must be general enough to support the debugging process and flexible enough to work on different programs.

One area of knowledge that can be generalized for a KBPD is the knowledge about the programming language used to develop programs. The constructs in a programming language have both a syntactic structure and a semantic meaning. Providing a KBPD with the semantics for a programming language gives the KBPD the ability to "understand" the effects of the constructs in a program and to understand when a construct is used incorrectly. Further, the semantics allow the KBPD to reason about the execution behavior of a program.

The Amber debugger was given the semantic rules for constructs in the C programming language. The semantic definition used is fairly complete, except for declarations. These rules allow Amber to understand how each construct should execute and permits Amber to follow the execution flow for any C construct used in any C program. The Locate Error component of Amber makes full use of this knowledge when examining the execution behavior of a program. This knowledge allows Amber to understand what should happen in the execution for any type of statement. This knowledge also gives Amber the ability to understand where execution errors can occur.

Another advantage to using knowledge about the programming language is that it allows the KBPD to reason about common programming mistakes that occur with the language. The Locate/Determine Fault component of a KBPD makes use of this knowledge to predict what faults could cause an observed error. This knowledge also

enables this component to search for suspected faults. The C language permits a programmer to make many different little mistakes, often, with serious consequences to the program. Programmers often forget that arrays are indexed starting at zero, not one. Copying the value of a long integer into a short when the value is too large to be represented in a short, is another common mistake. Mistakes like these can be difficult for a programmer to find. When examining the source for an error a programmer easily forgets about the "little" restrictions and consequences.

Amber knows about common C programming mistakes (faults) and "remembers" to consider these while a programmer tends to forget. This functionality in a KBPD can save a programmer many hours of frustration and effort. A KBPD can determine when a particular fault is likely to be present and knows how to locate the fault.

Providing a KBPD with an understanding of the debugging process is important. Debugging a program is not a simple task. It can be a complex process consisting of several steps, where each step has its own concerns, goals, information needed, and techniques. The debugging procedure is "coded" into the rules and components of Amber. A debugging session will always follow the same sequence of steps, though the specific actions in each component will differ for different errors.

Amber consists of components, where each component supports a phase of the debugging process. Each component is a collection of rules and/or functions that do the necessary actions. The debugging process used by Amber flows from component to component. Each component detects when it should fire. This is an advantage of the rule based approach. Transfer of control between components does not need to be explicitly coded into a function. The rules in the different components can watch the on-going debugging session and simply fire at the appropriate time.

A disadvantage to the design presented here is that Amber can not assist a programmer when the program logic is faulty. The strength of Amber lies in its knowledge about the C language and common C mistakes. This approach does not provide Amber with the ability to understand the intended task or purpose of a C program. When a program incorrectly performs its task because of an error in the program logic the programmer remains responsible for collecting and analyzing the program information. It is possible for Amber to be extended to assist a programmer with tracing a faulty execution path or to follow a faulty variable value to an incorrect calculation. KBPDs that compare specifications to program implementations have been developed and a similar technique could be added to Amber. A combination of Amber's current capabilities with the ability to understand the task of a program would produce a more powerful and useful debugging tool.

A drawback with the current implementation of Amber is that the programmer is responsible for providing Amber with all execution information. This places an unreasonable demand on the programmer. For Amber to become a truly usable tool it must be able to collect program execution information itself. This approach has its own advantages and drawbacks that must be considered.

The emphasis of this project has been to explore the design and development of a knowledge based program debugger that uses knowledge of the C programming language and common programming mistakes to assist a programmer with debugging a C program. This knowledge allows such a debugging tool to assist a programmer with some of the tedious tasks of debugging and relieves a programmer of some the responsibility of analyzing information generated during the debugging process.

Bibliography

- [ALLEN83] Allen, Elizabeth M., "YAPS : Yet Another Production System," *Department of Computer Science, University of Maryland: TR-1146*, December 1983.
- [COLLO87] Collofello, James S., Cousins, Larry., "Towards Automatic Software Fault Location Through Decision-to-Decision Path Analysis," *AFIPS Conference Proceedings: National Computer Conference*, Vol. 56, June 15-18, 1987, Chicago, Illinois, p539-543.
- [DERSH87] Dershowitz, Nachum., Lee, Yuh-jeng., "Deductive Debugging," *Proceedings 1987 Symposium on Logic Programming*, August 32 - Sept. 4, 1987, San Francisco, Calif., p298-306.
- [DONZE84] Donzeau-Gouge, Veronique., Huet, Gerard., Kahn, Gilles., Lang, Bernard., "Programming Environments Based on Structured Editors: The MENTOR Experience," *Interactive Programming Environments*, edited by Barstow, David R., published by McGraw-Hill, New York, 1984.
- [DYER87] Dyer, Michael., "A Formal Approach to Software Error Removal," *Journal of Systems and Software*, Vol. 7, No. 2, (June 1987), p109-114.
- [FOUET84] Fouet, Jean-Marc., "An Expert System For the Manipulation of Programs," *Fourth Jerusalem Conference on Information Technology*, May 21-25, 1984, p460-467, IEEE Computer Society Press.
- [GOULD75] Gould, John D., "Some Psychological Evidence on How People Debug Computer Programs," *International Journal of Man-Machine Studies*, Vol. 7, No. 2, (March 1975), p151-182.
- [GUPTA84] Gupta, N.K., Seviora, R.E., "An expert system approach to real time system debugging," *First Conference on Artificial Intelligence Applications*, Dec. 6-7, 1984, p336-343, IEEE Computer Society Press.
- [HARAN83] Harandi, Mehdi T., "Knowledge-Based Program Debugging : A Heuristic Model," *Softfair : A Conference on Software Development Tools, Techniques, and Alternatives, Proceedings*. July 25-28, 1983, Arlington, VA., p282-288, IEEE Computer Society Press.
- [HARBI84] Harabison, Samuel P., Steele Jr., Guy L., *A C Reference Manual*, published by Prentice-Hall, 1984.
- [HARTL84] Hartley, Roger T., "CRIB: Computer Fault-find Through Knowledge Engineering," *IEEE Computer*, Vol. 17, No. 3 (March 1984), p76-83.
- [HUNTB87] Huntback, Mathew M., "Algorithmic Parlog Debugging," *Proceedings 1987 Symposium on Logic Programming*, August 31 - Sept. 4, 1987, San Francisco, Calif., p288-297.
- [IEEE83] *IEEE Standard Glossary of Software Engineering Terminology*, published by the IEEE, New York, IEEE Std 729-1983, February 18, 1983.

- [JOHNS86] Johnson, W. Lewis, *Intention-Based Diagnosis of Novice Programming Errors*, published by Pitman Publishing Limited, 1986.
- [KERNI78] Kernighan, Brian W., Ritchie, Dennis M., *The C Programming Language*, published by Prentice-Hall, 1978.
- [KOREI86] Korei, Bogdan., "A Program Error Localization Expert System," *Proceedings of SPIE : The International Society for Optical Engineering, Volume 635, Applications of Artificial Intelligence III*, April 1-3, 1986, Orlando, Florida, p111-118.
- [LAFFE86] Laffey, Thomas J., Perkins, Walton A., Nguyen, Tin A., "Reasoning about Fault Diagnosis with LES," *IEEE Expert*, Vol. 1, No. 1 (Sept. 1986), p13-20.
- [LANGE85] Lange, Rense., Harandi, Mehdi, T., "Human Engineering Aspects Of a Program Debugging Expert System," *Proceedings of COMPSAC*, Oct. 9-11, 1985, p394-398.
- [LEUNG87] Leung, Hareton K. N., Reghbbati, Hassan K., "Comments on Program Slicing," *IEEE Transactions on Software Engineering*, Vol. 13, No. 12, (Dec. 1987), p1370-1371.
- [LUKEY80] Lukey, F. J., "Understanding and Debugging Programs," *International Journal of Man-Machine Studies*, Vol. 12, No. 2, (February 1980), p189-202.
- [MYERS79] Myers, Glenford J., *The Art of Software Testing*, published by John Wiley & Sons, 1979.
- [POPLK75] Poplke, H. E., et al., "DIALOG : A Model of Diagnostic Logic for Internal Medicine," *Proceedings of 4 IJCAI* (Sept 1975), p848-855.
- [SCHMI86] Schmidt, David A., *Denotational Semantics*, published by Allyn and Bacon, Inc., 1986.
- [SEDLM83] Sedlmeyer, Robert L., Thompson, William B., Johnson, Paul E., "Knowledge-Based Fault Localization in Debugging," *Journal of Systems and Software*, Vol. 3, No. 4, (Dec 1983), p301-307.
- [SEVIO87] Seviara, Rudolph E., "Knowledge-Based Program Debugging Systems," *IEEE Software*, Vol. 4, No. 3, (May 1987), p20-32.
- [SHAPI82] Shapiro, Ehud Y., *Algorithmic Program Debugging*, (ACM distinguished dissertations), Thesis (Ph.D.), Yale University, 1982, published by The MIT Press.
- [SHORT76] Shortliff, E. H., *Computer Based Medical Consultations: MYCIN*, Elsevier, N.Y., 1976
- [SWAN85] Swan, Randall B., "Common C Bugs," *Programmer's Journal*, Vol. 3, No. 6, (Nov-Dec 1985), p28-30.
- [VESSE85a] Vessey, Iris., "Expertise in debugging computer programs : A process analysis," *International Journal of Man-Machine Studies*, Vol. 23, No. 5, (Nov 1985), p459-494.
- [VESSE85b] Vessey, Iris., "Expertise in Debugging Computer Programs : Situation-Based versus Model-Based Problem Solving" *Proceedings of the Sixth International Conference on Information Systems*, Dec. 16-18, 1985, Indianapolis, Indiana, p288-303.

- [WEISE82] Weiser, Mark., "Programmers Use Slices When Debugging," *Communications of the ACM*, Vol. 25, No. 7, (July 1982), p446-452.
- [WEISE84] Weiser, Mark., "Program Slicing," *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, (July 1984), p352-357.
- [WIEDE86] Wiedenbeck, Susan., "Organization of Programming Knowledge of Novices and Experts," *Journal of the American Society for Information Science*, Vol. 37, No. 5, (Sept. 1986), p294-299.

Appendix A Amber User's Guide

Amber is an experimental system that implements the ideas and design presented in this work. Amber is a knowledge based program debugging tool that assists a programmer by helping to locate errors in a C program. This appendix explains how to use Amber.

Using Amber

There are several restrictions that you should be aware of before you start using Amber.

- 1) Amber must be used in the directory containing the source code for the program. If the source code for the program is distributed between several directories then full path names should be used to specify the location of files.
- 2) Any source code or include file passed to Amber must be compilable. The C compiler "cc" must be able to successfully operate on the file. Amber does not produce useful messages for syntax errors in a source file and any file passed to it should be syntactically correct. Amber will detect when a file is not syntactically correct, but will produce no useful error message about the violation.
- 3) Amber should be passed all the source files used to create a program. Amber can handle C source code files and include files. In addition the scanner can handle files used by Yacc and Lex. Yacc input files should end with the postfix ".y". Lex input files should end with the postfix ".l". System supplied include files (such as stdio.h) should not be passed to the scanner.
- 4) Amber will run all C files through the C preprocessor to remove preprocessor statements. Any Yacc file will be run through Yacc and then the C preprocessor. Any Lex file will be run through Lex and then the C preprocessor.
- 5) Source code processed by Amber can make use of any C language construct except "typedef." Amber will report a syntax error if it scans a "typedef" statement.

Invoking Amber

To run Amber you must use YAPS. To start YAPS type in the command:

```
mllisp yaps
```

YAPS prints out several messages as it loads. When YAPS is done loading it displays the prompt "0_" and this means YAPS is ready for use. To start Amber type in the command:

```
(load 'amber.l)
```

After Amber has loaded it will automatically begin execution.

Passing Source Code Files to Amber

The first thing Amber does is ask for a name for the program. This is not intended to be the name of any source code file, just a name that refers to the program. The next thing Amber does is ask how you want to enter the list of source code file names. If the program source is contained in only one or two files than you can just enter the file names when Amber asks for them. However, if the program source is distributed between twenty files then you will probably not want to enter twenty file names. Amber allows you to create a file that contains the name of each source code and include file used by the program. You may then tell Amber the name of this file and Amber will read the source file names from it. Amber examines each source file. Amber scans the files and converts them into a format it can understand. Once this step is done, Amber will be ready to discuss the problem in your program.

Help Facility

Amber provides a help facility that may be invoked whenever Amber issues a prompt. The help facility may be used to explain a prompt issued by Amber, to escape to the UNIX shell, to halt the debugging session, or to exit from Amber and YAPS.

There are two methods that can be used to invoke the help facility. When Amber issues a prompt to the user, you may enter a "?" followed by a help request, or just enter a "?" and no request. In the former case, the facility will do the requested action and then return you to the same prompt. In the latter case, an explanation of the prompt will be displayed. If the user types "? help" at a prompt then the help facility subsystem is entered. When the help facility subsystem is active it will issue prompts of the form "?>". Help facility commands are shown below:

"?" : Give a help message for the current prompt.

exit :

This command is only valid when the help facility is enabled and you want to return to the debugging session. This action exits the help facility and returns you to the same prompt in the debugging session where the facility was invoked.

help :

When this command is entered at an Amber prompt the help facility is entered. You must use the "exit" command to return to the debugger. This command can not be used in the help facility.

quit :

Terminates the Amber debugger and YAPS and returns you to the UNIX shell. This action causes Amber to stop immediately.

shell :

Creates a UNIX shell. You will then be able to enter any valid UNIX command. To exit the shell and return to Amber enter a CNTL-d.

stop :

Causes Amber to stop running after the current rule is done executing and the user is returned to the mlisp environment. The Amber session can be resumed by typing "run." (The mlisp prompt is a number followed by an underscore.)

As noted above, any help facility command must be preceded by a question mark and a space, if enter at a debugging session prompt. Inside the help facility it is not necessary to use the question mark before a command.

Providing Execution Information

Amber will help you locate an error in your program by stepping through the execution of the program. You must be able to provide Amber with information about what happens at each step in the program execution. Amber needs information such as:

- If a loop is entered does it ever exit?
- If a function is called does the function ever return?
- For an IF statement what is the value of the test expression?
- What is the value of a test expression for a Switch statement? Which case statement matched this value?

To answer these questions you must obtain information about what happens during the execution of the program. You can obtain this information by putting print statements into your code, or using dbx to step through the program. Be sure to answer each question issued by Amber carefully and correctly. Most requests for execution information can be answered with a yes or no answer.

As Amber steps through the execution of your program it will display the statement that it is currently tracing. The code displayed by Amber may not match a source listing for the same code. This is because Amber has run the code through the C preprocessor. This removes all comments, #ifdefs, #defines, etc, from the code used by Amber. Any defined constants used in the source are replaced by their values.

Appendix B

Using Amber - Examples

This appendix shows three example debugging sessions. The first example shows Amber locating an infinite loop. The second example shows Amber locating a core dump and the last example shows a deadlock situation.

Example : Infinite Loop

This example uses the C program shown below. There are two files: "avg.c" and "sum.c." The source code for "avg.c" is :

```

/*****
**
**   Program : avg.c
**
**   Purpose : This program reads in a list of numbers from a data
**             file and computes the sum and average of the numbers.
**
*****/

```

```

#include <stdio.h>

extern long do_sum();

main()
{
    float avg ;      /* Average value of numbers */
    int count ;     /* Number of items of data */
    char name[30];  /* Name of data file */
    long sum ;      /* Sum of the numbers */
    char temp[80];  /* Temporary variable */

    printf( " What is the name of your data file : " );
    gets( name );

    printf( " How many numbers are in your file : " );
    count = atoi( gets( temp ) );
    if ( count < 1 )
    {
        printf( "That is too few numbers. Bye!\n" );
        exit();
    }

    printf( "at avg 10\n" );
    sum = do_sum( name, count );
    printf( "at avg 20\n" );
    avg = sum / (float)count ;

    printf( " The sum of the numbers is : %ld\n", sum );
    printf( " The average of the numbers is : %f\n", avg );

} /* end of main */

```

The source code for "sum.c" is :

```
/******
```

Function : do_sum

Purpose : Reads in a list of numbers from a file and computes the sum of the values.

Arguments : name - Name of the data file
 cnt - Number of data items to read.

```
*****/
```

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

```
long do_sum( name, cnt )  
char *name;  
int cnt ;  
{
```

```
    FILE *fp ; /* Pointer to data file */  
    int i ; /* Used to tell if fscanf worked */  
    long sum ; /* Holds the sum */  
    int val ; /* Holds the value read from the file */
```

```
    printf( "in sum 10 ) :
```

```
        /* Open the data file */  
        fp = fopen( name, "r" ) ;  
        if ( fp == NULL )  
        {  
            printf( " unable to open the file %s.0, name ) ;  
            exit() ;  
        }
```

```
        /* Read the data and compute the sum */  
        sum = 0L ;  
        i = fscanf( fp, "%d", &val ) ;  
        printf( "in sum 20 ) ;  
        while ( i = 1 )  
        {  
            sum += val ;  
            i = fscanf( fp, "%d", &val ) ;  
        }  
        printf( "in sum 30 ) :
```

```
        if ( i == 0 )  
        {  
            printf( " Error while reading the data file!0 ) ;  
            exit() ;
```

```
    }  
    return ( sum ) ;  
} /* end of do_sum */
```

These two source code files also contain test print statement similar to the type a programmer would put into the code to locate an infinite loop. The Amber debugging session for this problem is shown below. Notice that when Amber prompted for the names of the source files the user used the User Interface to escape to the shell and looked up the names of the files.

Welcome to the Amber program debugger.

Please enter the name of your program : average

You may enter either :

- a) The name of a file containing a list of program source code and include file names.
- b) Each source code and include file name separately.

Select : b

Enter each file name below. Enter a period '.' after you have entered all the file names.

Enter the name of a source code file : ? shell

** Press CNTL-d to resume the debugger.

%

avg.c

sum.c

%

** Welcome back.

Enter the name of a source code file : avg.c

Enter the name of a source code file : sum.c

Enter the name of a source code file : .

Preprocessing the file : avg.c

Scanning file : avg.c

Preprocessing the file : sum.c

Scanning file : sum.c

[load average.fcr]

Which of the following best describes the problem your program is demonstrating?

- a) The program does not terminate.
- b) The program terminates with a system error message.
- c) The program generates incorrect results.
- d) None of the above seem to fit the problem.
- e) Quit the debugger.

Select a letter : a

It happens sometimes that the system is running slowly or you do not realize just how much time your program needs to run.

Are you sure that you waited long enough for the program to finish (y or n)? y

Does the program do process-to-process communication or communicate with another program via a network?

Please answer y or n : n

Done running the symptom analyzer!

If seems likely that your program is entering an infinite loop. To find the loop it is necessary to trace the execution of your program.

Because this debugger is unable to simulate the execution of the program it must (unfortunately) prompt you for the information necessary to trace the program's execution.

The trace begins with the main routine.

[load avg.fct]

Entering the code for function : main

```
printf( " What is the name of your data file : " );
```

When the function 'printf' is called does it return (y or n)? y
gets(name);

When the function 'gets' is called does it return (y or n)? y
printf(" How many numbers are in your file : ");

When the function 'printf' is called does it return (y or n)? y
count = atoi(gets(temp));

When the function 'gets' is called does it return (y or n)? y
When the function 'atoi' is called does it return (y or n)? y
if (count < 1)

Is the value of the test expression false (zero) or true (non-zero)?
Pick false or true (f or t)? f

The body of the IF statement is not executed. The statement following the body of the IF statement will be the next statement executed.

```
printf( "at avg 10" );
```

When the function 'printf' is called does it return (y or n)? y
sum = do_sum(name , count);

When the function 'do_sum' is called does it return (y or n)? n
[load sum.fct]

Entering the code for function : do_sum
printf("in sum 10) ;

When the function 'printf' is called does it return (y or n)? y
fp = fopen(name , "r");

When the function 'fopen' is called does it return (y or n)? y
if (fp == 0)

Is the value of the test expression false (zero) or true (non-zero)?
Pick false or true (f or t)? f

The body of the IF statement is not executed. The
statement following the body of the IF statement
will be the next statement executed.

sum = 0L ;
i = fscanf(fp , "%d" , &val);

When the function 'fscanf' is called does it return (y or n)? y
printf("in sum 20) ;

When the function 'printf' is called does it return (y or n)? y
while (i = 1)

Does this While loop eventually exit (y or n)? n

Because the debugger suspects an infinite loop it is necessary
to determine if the loop contains any inner loops that are
entered and never exit or if there are any functions that are
called and never return.

The trace will search the body of the loop, ignoring branches in
the execution flow and just ask about function calls and inner
loops that are found. Please read along!

sum += val ;
i = fscanf(fp , "%d" , &val);

When the function 'fscanf' is called does it return (y or n)? y

The program contains an infinite loop in the function 'do_sum'.
The source code for the loop is :
while (i = 1)
{


```
sum += val ;  
i = fscanf( fp , "%d" , &val ) ;  
}
```

Unfortunately, the debugger can not tell you why the loop does not exit. You must figure that out.

Do you want to quit or do you have another problem?
Please enter 'q' to quit or 'c' to continue : q

Example : Core Dump

This example uses the two source files given in example 1. For this example assume that the program experiences a core dump.

Please enter the name of your program : average

You may enter either :

- a) The name of a file containing a list of program source code and include file names.
- b) Each source code and include file name separately.

Select : b

Enter each file name below. Enter a period '.' after you have entered all the file names.

Enter the name of a source code file : avg.c

Enter the name of a source code file : sum.c

Enter the name of a source code file : .

Preprocessing the file : avg.c
Scanning file : avg.c

Preprocessing the file : sum.c
Scanning file : sum.c
[load average.fcr]

Which of the following best describes the problem your program is demonstrating?

- a) The program does not terminate.
- b) The program terminates with a system error message.
- c) The program generates incorrect results.
- d) None of the above seem to fit the problem.
- e) Quit the debugger.

Select a letter : b

Which message is printed when the program is aborted:

- a) quit (core dumped)
- b) illegal instruction (core dumped)
- c) trace trap (core dumped)
- d) IOT instruction (core dumped)
- e) EMT instruction (core dumped)
- f) floating point exception (core dumped)
- g) bus error (core dumped)
- h) segmentation violation (core dumped)
- i) bad argument to system call (core dumped)

z) None of the above.

Select a letter : g
Done running the symptom analyzer!

It is necessary to trace the execution of your program to find the statement that is executing when the core dump occurs.

Because this debugger is unable to simulate the execution of the program it must (unfortunately) prompt you for the information necessary to trace the program's execution.

The trace begins with the main routine.
[load avg.fct]

Entering the code for function : main
printf(" What is the name of your data file : ");

Does this statement execute successfully (y or n)? y
gets(name);

Does this statement execute successfully (y or n)? y
printf(" How many numbers are in your file : ");

Does this statement execute successfully (y or n)? y
count = atoi(gets(temp));

Does this statement execute successfully (y or n)? y
if (count < 1)

Does the test expression execute successfully (y or n)? y

Is the value of the test expression false (zero) or true (non-zero)?
Pick false or true (f or t)? f

The body of the IF statement is not executed. The statement following the body of the IF statement will be the next statement executed.
printf(" at avg 10);

Does this statement execute successfully (y or n)? y
sum = do_sum(name , count);

Does this statement execute successfully (y or n)? n
When the function 'do_sum' is called does it return (y or n)? y

The core dump occurs in the function 'main'.

The core dump occurs in the statement:
sum = do_sum(name , count);

Unfortunately, the debugger can not tell you why the core dump occurred. You must figure that out.

Do you want to quit or do you have another problem?
Please enter 'q' to quit or 'c' to continue : q

Example : Deadlock

This example uses a program that accesses a network to communicate with another computer. The source code for this example is shown below.

```
#include <stdio.h>

main()
{

    CHANNEL *chan ; /* Pointer to a network channel */
    char buf[512]; /* Buffer for receiving data */

    netinit(); /* Initialize the network routines */

    chan = netopen( "hostB" ); /* Connect to host B */
    if ( chan == NULL )
    {
        printf( " Unable to open channel0 );
        exit();
    }

    netread( buf );

    do_rest_of_work( chan, buf ); /* Rest of program */
}
```

For this example the call to "netread" does not return. Assume that the functions "netinit," "netopen," and "netread" are supplied by the vendor of the network system in use. The debugging session is given below.

Welcome to the Amber program debugger.

Please enter the name of your program : netrun

You may enter either :

- a) The name of a file containing a list of program source code and include file names.
- b) Each source code and include file name separately.

Select : b

Enter each file name below. Enter a period '.' after you have entered all the file names.

Enter the name of a source code file : netrun.c

Enter the name of a source code file : .

Preprocessing the file : netrun.c

Scanning file : netrun.c

[load netrun.fcr]

Which of the following best describes the problem your program is demonstrating?

- a) The program does not terminate.
- b) The program terminates with a system error message.
- c) The program generates incorrect results.
- d) None of the above seem to fit the problem.
- e) Quit the debugger.

Select a letter : a

It happens sometimes that the system is running slowly or you do not realize just how much time your program needs to run.

Are you sure that you waited long enough for the program to finish (y or n)? y

Does the program do process-to-process communication or communicate with another program via a network? y

Done running the symptom analyzer!

If seems likely that your program is entering an infinite loop or reaching a deadlock state. To find the loop or place where deadlock is entered it is necessary to trace the execution of your program.

Because this debugger is unable to simulate the execution of the program it must (unfortunately) prompt you for the information necessary to trace the program's execution.

The trace begins with the main routine.
[load netrun.fct]

Entering the code for function : main
netinit() ;

When the function 'netinit' is called does it return (y or n)? y
chan = netopen("hostB") ;

When the function 'netopen' is called does it return (y or n)? y
if (chan == 0)

Is the value of the test expression false (zero) or true (non-zero)?
Pick false or true (f or t)? f

The body of the IF statement is not executed. The statement following the body of the IF statement will be the next statement executed.
netread(buf) ;

When the function 'netread' is called does it return (y or n)? n

Does the function 'netread' do some type of read operation (y or n)? y

The debugger does not have the source code for the function 'netread'. If this function was written as part of the program and you forgot to give the name of the file containing the source then you must start the debugging session from the beginning.

This function could be where the program is entering into a deadlock state, because this function is trying to read information. You should check that there is data available for the function to read. If there is no data to read then the function may fail to return. If the read operation is attempting to read from another process or across a network then be sure that the other program is sending information at the time your program is attempting to read.

Do you want to quit or do you have another problem?
Please enter 'q' to quit or 'c' to continue : q

Bye! Good luck! Have a nice day!

Appendix C

YAPS Rules and Lisp Functions

This appendix contains listings of the source files for Amber. These are the YAPS rules and Lisp functions. The source code for the scanner is given in appendix D.

The source files in this appendix are:

- amber.l
- cleanup.l
- control.l
- coredump.l
- forever.l
- help.l
- misc.l
- prt_code.l
- run_scan.l
- start.l
- symptoms.l
- trace.l


```

.....
;;
;; File : amber.l
;;
;; Created : 02/04/88          by : Eric Byrne
;;
;; Purpose : This file contains the YAPS commands to load the
;;           knowledge based program debugger, Amber.
;;
;;           To run Amber type : mlist yaps <cr>
;;                               (load 'amber.l) <cr>
;;
.....

```

```

(yaps-untrace)      ; Turn off the yaps trace feature.
(rm *)              ; Remove any facts already in the database.

```

```

(load '/usrb/byrne/grad/kbpd/cleanup.l) ; Rules for cleaning up the fact base.
(load '/usrb/byrne/grad/kbpd/control.l) ; Rules for controlling the debugger.
(load '/usrb/byrne/grad/kbpd/coredump.l) ; Rules for reasoning about core dumps.
(load '/usrb/byrne/grad/kbpd/forever.l) ; Rules for reasoning about deadlock
                                         ; and infinite loops.
(load '/usrb/byrne/grad/kbpd/help.l)    ; The help facility.
(load '/usrb/byrne/grad/kbpd/misc.l)    ; Miscellaneous functions.
(load '/usrb/byrne/grad/kbpd/prt_code.l) ; Functions to display program code.
(load '/usrb/byrne/grad/kbpd/run_scan.l) ; Rules for handling the scanner.
(load '/usrb/byrne/grad/kbpd/symptoms.l) ; The symptom analyzer system.
(load '/usrb/byrne/grad/kbpd/start.l)   ; Functions used to start the system.
(load '/usrb/byrne/grad/kbpd/trace.l)   ; Rules for tracing C programs.

```

```

(banner)           ; When starting up Amber display the banner.
(start)            ; Start running Amber.

```

```

.....
;;
;; File : cleanup.l
;;
;; Created : 03/25/88        by : Eric Byrne
;;
;; Purpose : This file contains the YAPS commands to clean up the
;;           knowledge base of unneeded facts.
;;           knowledge based program debugger, Amber.
;;
.....

```

```

.....
Clean up facts left by the tracer.
.....

```

```
(defp clean_trace_exp1
  (goal (clean trace))
  (E -x -y) ; Any expression fact with two arguments
  ->(remove 2)
) ; end of rule clean_trace_exp1
```

```
(defp clean_trace_exp2
  (goal (clean trace))
  (E -x -y -z) ; Any expression fact with three arguments
  ->(remove 2)
) ; end of rule clean_trace_exp2
```

```
(defp clean_trace_exp3
  (goal (clean trace))
  (E -x -y -z -a) ; Any expression fact with four arguments
  ->(remove 2)
) ; end of rule clean_trace_exp3
```

```
(defp clean_trace_block
  (goal (clean trace))
  (B -body) ; Any block fact
  ->(remove 2)
) ; end of rule clean_trace_block
```

```
(defp clean_trace_stmt1
  (goal (clean trace))
  (S -x -y -cont) ; Any statement fact with two arguments plus cont
  ->(remove 2)
) ; end of rule clean_trace_stmt1
```

```
(defp clean_trace_stmt2
  (goal (clean trace))
  (S -x -y -z -cont) ; Any statement fact with three arguments plus cont
  ->(remove 2)
) ; end of rule clean_trace_stmt2
```

```
(defp clean_trace_stmt3
  (goal (clean trace))
  (S -x -y -z -a -cont) ; Any statement fact with four arguments plus cont
  ->(remove 2)
) ; end of rule clean_trace_stmt3
```

```
(defp clean_trace_stmt4
```

```
( goal (clean trace) )
( S -x -y -z -a -b -cont ) ; Any expression fact with five arguments
->( remove 2 ) ; plus cont
) ; end of rule clean_trace_stmt4
```

```
(defp clean_trace_stmt5
  ( goal (clean trace) )
  ( S -x -cont ) ; Any expression fact with one argument plus cont
->( remove 2 )
) ; end of rule clean_trace_stmt5
```

```
(defp clean_trace_if_exp
  ( goal (clean trace) )
  ( IF_EXP_PATH -a -b -c )
->( remove 2 )
) ; end of rule clean_trace_if_exp
```

```
(defp clean_trace_func
  ( goal (clean trace) )
  ( FUNC_CALL -a )
->( remove 2 )
) ; end of rule clean_trace_func
```

```
(defp clean_trace_if
  ( goal (clean trace) )
  ( U_IF -a -b -c )
->( remove 2 )
) ; end of rule clean_trace_if
```

```
(defp clean_trace_if_else
  ( goal (clean trace) )
  ( B_IF -a -b -c -d )
->( remove 2 )
) ; end of rule clean_trace_if_else
```

```
(defp clean_trace_break
  ( goal (clean trace) )
  ( BREAK -type )
->( remove 2 )
) ; end of rule clean_trace_break
```

```
(defp clean_trace_continue
  ( goal (clean trace) )
```

```
( CONTINUE -type )
-->( remove 2 )
) ; end of rule clean_trace_continue
```

```
(defp clean_trace_return
  ( goal (clean trace) )
  ( RETURN )
-->( remove 2 )
) ; end of rule clean_trace_return
```

```
(defp clean_adw
  ( goal (clean trace) )
  ( ADW -test -body )
-->( remove 2 )
) ; end of rule clean_adw
```

```
(defp clean_aw
  ( goal (clean trace) )
  ( AW -test -body )
-->( remove 2 )
) ; end of rule clean_aw
```

```
(defp clean_af
  ( goal (clean trace) )
  ( AF -test -incr -body )
-->( remove 2 )
) ; end of rule clean_af
```

```
(defp clean_test
  ( goal (clean trace) )
  ( TEST -a -b )
-->( remove 2 )
) ; end of rule clean_test
```

```
(defp clean_trace
  ( goal (clean trace) )
-->( remove 1 )
) ; end of rule clean_trace
```

```
.....
```

```
..... Clean up LOOP facts. Only the most recent LOOP fact is valid.
```

```

.....

(defun clean_loop_fact
  (goal (clean loop -a -b))
  (LOOP -x -y)
  test (not (equal -a -x))
        (not (equal -b -y)))
  ->( remove 1 )
) ; end of rule clean_loop_fact

(defun clean_loop
  (goal (clean loop -a -b))
  ( (LOOP -x -y) with ( and (equal -a -x) (equal -b -y) ) )
  ->( remove 1 )
) ; end of rule clean_loop

( installp 'clean_trace_exp1 'clean_trace_exp2 'clean_trace_exp3
  'clean_trace_block 'clean_trace_stmt1 'clean_trace_stmt2
  'clean_trace_stmt3 'clean_trace_stmt4 'clean_trace_stmt5
  'clean_trace_if_exp 'clean_trace_func 'clean_trace_if
  'clean_trace_if_else 'clean_trace_break 'clean_trace_return
  'clean_trace_continue
  'clean_trace 'clean_loop_fact 'clean_loop
  'clean_adw 'clean_aw 'clean_af
  'clean_test
)

```

```

.....
::
:: File : control.l
::
:: Created : 03/07/88 by : Eric Byrne
::
:: Purpose : This file contains YAPS rules used to control the flow
:: of processing in the debugger.
::
::
.....

```

: This rule is fired when the debugger is finished and wants to know if
: the user has another problem to handle.

```

(defun more_work
  (goal ( run again ))
  ->( remove 1 )
  ( prog (ans)
    prompt_1
    ( msg N N )
  )
)

```

```

(msg N T "Do you want to quit or do you have another problem?")
(msg N T "Please enter 'q' to quit or 'c' to continue : ")
(setq ans ( call_help (lread)
  ("If you are done using the debugger then enter 'q'. If you"
   "have another problem in your program that you want to work"
   "on then enter 'c.'."))
)
(cond ((null ans)
      ; Repeat the prompt
      (go prompt_1)
    )
      ((equal ans 'q)
      ; User wants to quit, so terminate the mlisp session
      (msg N)
      (msg N T "Bye! Good luck! Have a nice day!" N N)
      (exit)
    )
      ((equal ans 'c)
      ; User has another bug to work on. Start up the scanner.
      ; We do that in case the user has changed the program since
      ; we last scanned it.
      (rm '*) ; Clean up the fact base.
      (goal run scanner)
    )
      (t ; User entered a bad value, prompt again
      (msg N)
      (msg N T "**** Bad value! Enter a 'q' or a 'c'!" )
      (go prompt_1)
    )
  )
)
) ; End of rule more_work

```

: This rule is fired when the debugger needs to take the most likely hypothesis and attempt to locate the error in the source code.

```

(defp select_error
  (goal (select error))
  (HYP -suspect -weight) ; Select hypothesis with greatest weight
  (~ (HYP -other -mass) with (>-mass -weight))
  ->(remove 1)
  (cond ((= -suspect 'Bad_format)
        ; Suspected cause is a bad print format statement.
        (goal explain unknown)
      )
        ((= -suspect 'Bad_path)
        ; Suspected cause is a bad logic decision in the program.
      )
  )
)

```

```

( goal explain unknown )
)
( = -suspect 'Bad_value )
; Suspected cause is an incorrect calculation in the program.
( goal explain unknown )
)
( = -suspect 'Buf_ovrflw )
; Suspected cause is a buffer overflow.
( goal trace coredump )
)
( = -suspect 'Deadlock )
; Suspected cause is a deadlock.
( goal trace deadlock )
)
( = -suspect 'Flt_div0 )
; Suspected cause is a floating point division by zero.
( goal trace coredump )
)
( = -suspect 'Flt_ovrflw )
; Suspected cause is a floating point overflow.
( goal trace coredump )
)
( = -suspect 'Flt_undflw )
; Suspected cause is a floating point underflow.
( goal trace coredump )
)
( = -suspect 'Inf_loop )
; Suspected cause is an infinite loop.
( goal trace loop )
)
( = -suspect 'Int_div0 )
; Suspected cause is an integer division by zero.
( goal trace coredump )
)
( = -suspect 'Int_ovrflw )
; Suspected cause is an integer overflow.
( goal trace coredump )
)
( = -suspect 'Unk_dump )
; The debugger does not know what could cause the core dump. but
; at least trace the program to where the core dump occurs.
( goal trace coredump )
)
( = -suspect 'Unknown )
; The debugger is not prepared to handle the symptoms.
( goal explain unknown )
)
( = -suspect 'User_quit )
; Suspected cause is a core dump caused by the user interrupting
; the program.

```

```

        ( goal explain coredump )
      )
    ) ; end of rule select_error

```

; This rule is fired after the symptom analyzer has finished and the
; debugger does not know how to proceed with the symptoms.

```

(defp stumped
  ( goal ( explain unknown ) )
  ->( remove 1 )
  ( msg N N )
  ( msg N T "Given the symptoms this debugger does not know how to deal" )
  ( msg N T "with the problem. Sorry, but you're on your own." )
  ( goal run again )
) ; end of rule select_error

```

; This rule is fired when we have run out of program statements to check
; and the source of the error has not been found yet.

```

(defp oh_no_out_of_code
  ( goal ( trace -type ) )
  ->( remove 1 )
  ( msg N )
  ( msg N T "There are no more program statements left to check in this" )
  ( msg N T "block of code or function. Yet, the source of the error has" )
  ( msg N T "not been found. Most likely a mistake was made in following" )
  ( msg N T "the execution of the program. Unfortunately, at this point" )
  ( msg N T "the only thing we can do is start over. Sorry!." N N )
  ( goal run again )
) ; end of rule oh_no_out_of_code

```

```

( installp 'more_work 'select_error 'stumped 'oh_no_out_of_code )

```

```

.....
::
:: File : coredump.l
::
:: Created : 03/09/88 by : Eric Byrne
::
:: Purpose : This file contains the YAPS rules used to reason about
:: core dumps.
::
::
.....

```



```
; IF the program terminates with a core dump
; THEN start tracing the execution flow from the main routine. main().
```

```
(defp start_coredump_trace
  (goal (trace coredump) )
  ->(msg N N)
  (msg N T "It is necessary to trace the execution of your program to find")
  (msg N T "the statement that is executing when the core dump occurs." N )
  (msg N T "Because this debugger is unable to simulate the execution of" )
  (msg N T "the program it must (unfortunately) prompt you for the")
  (msg N T "information necessary to trace the program's execution." N )
  (msg N T "The trace begins with the main routine." )
  (msg N )

  (goal get_cont main) ; Get the continuation for main.

) ; end of rule start_coredump_trace
```

```
: If we know the cause of a core dump and
: the hypothesis is that the user caused the core dump
: THEN explain it.
```

```
(defp user_quit
  (goal (explain coredump) )
  (HYP User_quit -weight)
  ->(remove 1)
  (msg N N)
  (msg N T "The 'quit' core dump message is generated when the program is" )
  (msg N T "terminated by an interrupt. The interrupt is usually caused by")
  (msg N T "the user (you) typing CNTL . That control sequence kills the")
  (msg N T "program and causes a core dump. If you need to terminate the" )
  (msg N T "program and don't want a core dump then use CNTL z." )
  (msg N )
  (msg N T "If you did not use a control sequence or special key ( such as")
  (msg N T "'break' or 'del' ) to terminate the program then this debugger")
  (msg N T "can't explain the cause." N )

  (goal run again )
) ; end of rule user_quit
```

```
: This rule is fired when we have found the line where the core dump occurs.
```

```
(defp aha_coredump
  (goal (trace coredump) )
  (DUMPED -type -pos -stmt )
  ->(remove 1) ; Stop the trace and
```

```
( goal show dump ) ; Show the user where the dump occurs.  
( goal clean trace ) ; clean up trace facts.  
); end of rule aha_goredump
```

; This rule is fired when we have found that a core dump is occurring in
; the test expression used in a Do-While loop.

```
(defp aha_dw_dump  
  ( goal (trace coredump) )  
  ( DUMP_EXP DW )  
  ( ADW -test -body )  
  ->( remove 2 3 )  
  ( fact DUMPED DW T ^ (list -test -body) )  
); end of rule aha_dw_dump
```

; This rule is fired when we have found that a core dump is occurring in
; the test expression used in a While loop.

```
(defp aha_w_dump  
  ( goal (trace coredump) )  
  ( DUMP_EXP W )  
  ( AW -test -body )  
  ->( remove 2 3 )  
  ( fact DUMPED W T ^ (list -test -body) )  
); end of rule aha_w_dump
```

; This rule is fired when we have found that a core dump is occurring in
; the initialize expression used in a For loop.

```
(defp aha_fi_dump  
  ( goal (trace coredump) )  
  ( DUMP_EXP FI )  
  ( AF -init -test -incr -body )  
  ->( remove 2 3 )  
  ( fact DUMPED F IT ^ (list -init -test -incr -body) )  
); end of rule aha_fi_dump
```

; This rule is fired when we have found that a core dump is occurring in
; the test expression used in a For loop.

```
(defp aha_ft_dump  
  ( goal (trace coredump) )  
  ( DUMP_EXP FT )  
  ( AF -init -test -incr -body )  
  ->( remove 2 3 )
```

```

    ( fact DUMPED F T ^ (list -init -test -incr -body) )
  ) ; end of rule aha_ft_dump

```

: This rule is fired when we have found that a core dump is occurring in
: the increment expression used in a For loop.

```

(defp aha_fn_dump
  ( goal (trace coredump) )
  ( DUMP_EXP FN )
  ( AF -init -test -incr -body )
  ->( remove 2 3 )
  ( fact DUMPED F IN ^ (list -init -test -incr -body) )
) ; end of rule aha_fn_dump

```

: This rule is fired if a coredump occurs in a function for which we do
: not have source code.

```

(defp aha_no_func
  ( goal (trace coredump) )
  ( NO_FUNC -name -sys -read )
  ->( remove 1 2 ) ; Stop the trace
  ( cond ( (equal -sys 'yes)
            ; The core dumps occurs for a system function.
            (msg N T "The core dump is occurring in the system function " -name ".")
            (msg N T "Because the debugger does not have source code for this" )
            (msg N T "function it is not possible to continue the trace. System")
            (msg N T "functions normally work correctly. You should carefully" )
            (msg N T "check that the values of the parameters passed to the" )
            (msg N T "function are valid." )
            )
        ( (equal -sys 'no)
            ; The core dumps occurs for a non-system function.
            (msg N T "The debugger does not have the source code for the function")
            (msg N T "-name ". If this function was written as part of the" )
            (msg N T "program and you forgot to give the name of the file containing")
            (msg N T "the source then you must start the debugging session from the")
            (msg N T "beginning." N )
            (msg N T "If this is a system supplied function or a function that is" )
            (msg N T "part of a vendor supplied function library then you should" )
            (msg N T "carefully check that the values of the parameters passed to" )
            (msg N T "function are valid and that the function is called at a valid" )
            (msg N T "place in the program." )
            )
        )
  )
  ( goal run again ) ; Determine if user has another problem.
  ( goal clean trace ) ; clean up trace facts.
) ; end of rule aha_no_func

```

; This rule is fired when we have found the line where the core dump occurs
; and need to show the line to the user.

```
(defp show_dump
  (goal (show dump))
  (DUMPED -type -pos -stmt)
  ->(remove 1)
  (msg N N)
  (msg N T "The core dump occurs in the function '" cur_func "'." )
  (cond ((or (equal -type 'E) (equal -type 'R))
    ; Dump in expression or return statement.
    (msg N T "The core dump occurs in the statement:" N )
    (prt_stmt (list 'S -type -stmt))
  )
  ((equal -type 'W)
    ; Dump in test expression of While statement
    (msg N T "The core dump occurs in the test expression of this" )
    (msg N T "While statement:" N )
    (prt_body (list 'S 'W (car -stmt) (cadr -stmt)))
  )
  ((equal -type 'DW)
    ; Dump in test expression of Do-While statement
    (msg N T "The core dump occurs in the test expression of this" )
    (msg N T "Do-While statement:" N )
    (prt_body (list 'S 'DW (cadr -stmt) (car -stmt)))
  )
  ((equal -type 'I)
    ; Dump in test expression of If statement
    (msg N T "The core dump occurs in the test expression of this If" )
    (msg N T "statement:" N )
    (prt_body (list 'S 'I (car -stmt) (cadr -stmt)))
  )
  ((equal -type 'IE)
    ; Dump in the test expression of an IF-Then-Else statement
    (msg N T "The core dump occurs in the test expression of this" )
    (msg N T "If-Then-Else statement:" N )
    (prt_body (list 'S 'IE (car -stmt) (cadr -stmt) (caddr -stmt)))
  )
  ((equal -type 'S)
    ; Dump in the test expression of a Switch statement
    (msg N T "The core dump occurs in the test expression of this" )
    (msg N T "Switch statement:" N )
    (prt_body (list 'S 'S (car -stmt) (cadr -stmt)))
  )
  ((equal -type 'F)
    ; Dump in a For loop expression
    (cond ((equal -pos 'IT)
      ; Dumps in initialize expression.
      (msg N T "The core dump occurs in the initialize expression ")
      (msg N T "of this For Loop statement:" N )
    )
  )
  )
)
```

```

    )
    ( (equal -pos 'T)
      ;Dumps in test expression.
      (msg N T "The core dump occurs in the test expression ")
      ( msg N T "of this For Loop statement:" N )
    )
    ( (equal -pos 'IN)
      ;Dumps in increment expression.
      (msg N T "The core dump occurs in the increment expression ")
      ( msg N T "of this For Loop statement:" N )
    )
  )
  ( prt_body (list 'S 'F (car -stmt) (cadr -stmt) (caddr -stmt)
                  (caddr -stmt) ) )
)
)

(msg N T "Unfortunately, the debugger can not tell you why the core dump" )
(msg N T "occurred. You must figure that out." )

) ; end of rule show_dump

```

```

( installp 'start_coredump_trace 'user_quit 'aha_coredump
  'aha_dw_dump 'aha_w_dump 'aha_ft_dump
  'aha_fi_dump 'aha_fn_dump 'show_dump
  'aha_po_func
)

```

```

.....
::
:: File : forever.l
::
:: Created : 03/10/88 by : Eric Byrne
::
:: Purpose : This file contains the YAPS rules used to reason
:: about deadlocks and infinite loops.
::
::
.....

```

```

; IF the program does not halt AND
; infinite loop is suspected.
; THEN start tracing the execution flow from the main routine, main().

```

```

(defp start_loop_trace
  ( goal (trace loop) )
  -> ( msg N N )
)

```

```

(msg N T "If seems likely that your program is entering an infinite loop.")
(msg N T "To find the loop it is necessary to trace the execution of your")
(msg N T "program." N)
(msg N T "Because this debugger is unable to simulate the execution of")
(msg N T "the program it must (unfortunately) prompt you for the")
(msg N T "information necessary to trace the program's execution." N)
(msg N T "The trace begins with the main routine." )
(msg N)

(goal get_cont main) ; Get the continuation for main.

) ; end of rule start_loop_trace

; IF the program does not halt AND
; infinite loop and/or deadlock is suspected.
; THEN start tracing the execution flow from the main routine. main().

(defp start_no_halt_trace
  (goal (trace loop))
  (HYP Deadlock -w2)
test( > -w2 -1)
->(msg N N)
(msg N T "If seems likely that your program is entering an infinite loop")
(msg N T "or reaching a deadlock state. To find the loop or place where")
(msg N T "deadlock is entered it is necessary to trace the execution of")
(msg N T "your program." N)
(msg N T "Because this debugger is unable to simulate the execution of")
(msg N T "the program it must (unfortunately) prompt you for the")
(msg N T "information necessary to trace the program's execution." N)
(msg N T "The trace begins with the main routine." )
(msg N)

(goal get_cont main) ; Get the continuation for main.

) ; end of rule start_no_halt_trace

; IF the program does not halt AND
; deadlock is suspected.
; THEN start tracing the execution flow from the main routine. main().

(defp start_deadlock_trace
  (goal (trace deadlock))
->(msg N N)
(msg N T "If seems likely that your program is entering a deadlock state.")
(msg N T "To find where the deadlock is entered it is necessary to trace")
(msg N T "the execution of your program." N)
(msg N T "Because this debugger is unable to simulate the execution of")

```

```

(msg N T "the program it must (unfortunately) prompt you for the" )
(msg N T "information necessary to trace the program's execution." N )
(msg N T "The trace begins with the main routine." )
(msg N )

(goal get_cont main ) : Get the continuation for main.

) : end of rule start_no_halt_trace

```

```

; This rule is used when the program trace finds a loop that does not exit
(defp aha_loop
  (goal (trace loop))
  (LOOP -a -b )
  ->( remove 1 )      ; Stop tracing and
      (goal explain loop) ; Show the user the loop.
      (goal clean loop -a -b) ; Get rid of any older LOOP facts
      (goal clean trace) ; clean up after the trace.
) ; end of rule aha_loop

```

```

; This rule is fired when we have found the infinite loop. This rule
; shows the loop to the user. At this point the debugger does not know
; why the loop does not exit.

```

```

(defp show_loop
  (goal (explain loop))
  (LOOP -type -info) ; info is a list whose structure depends on type
  ->( remove 1 )
      (msg N N )
      (msg N T "The program contains an infinite loop in the function "
               cur_func ".")
      (msg N T "The source code for the loop is : " N )
      (cond ( (equal -type 'DW)
              ; Show a Do-While loop
              (prt_body (list 'S 'DW (car -info) (cadr -info)))
            )
            ( (equal -type 'W)
              ; Show a While loop
              (prt_body (list 'S 'W (car -info) (cdr -info)))
            )
            ( (equal -type 'F)
              ; Show a For loop
              (prt_body (list 'S 'F (car -info) (cadr -info) (caddr -info)
                              (caddrdr -info)))
            )
          )
)
(msg N T "Unfortunately, the debugger can not tell you why the loop does")

```

```

(msg N T "not exit. You must figure that out." )

( goal run again ) ; Determine if user has another problem.
) ; end of rule show_loop

; This rule is fired if we find a function that does not return and for which
; we have no source code and deadlock is not considered a possibility.

(defp aha_no_source
  ( goal (trace -type) )
  ( NO_FUNC -name -sys -read )
  ( HYP Deadlock -weight )
test(and ( or ( equal -type 'loop) (equal -type 'deadlock) )
  ( <-weight -1 )
  ( not (equal -read 'unk) )
)
->( remove 1 2 )
(cond ( (equal -sys 'yes)
  ; System function does not return
(msg N T "The system function "" -name "" does not return. Because" )
(msg N T "the debugger does not have source code for this function" )
(msg N T "it is not possible to continue the trace. System functions" )
(msg N T "normally work correctly. You should carefully check that" )
(msg N T "that the values of the parameters passed to the function" )
(msg N T "are valid." )
)
  ( (equal -sys 'no)
  ; The core dumps occurs for a non-system function.
(msg N T "The debugger does not have the source code for the function" )
(msg N T "" -name ". If this function was written as part of the" )
(msg N T "program and you forgot to give the name of the file containing" )
(msg N T "the source then you must start the debugging session from the" )
(msg N T "beginning." N )
(msg N T "If this is a system supplied function or a function that is" )
(msg N T "part of a vendor supplied function library then you should" )
(msg N T "carefully check that the values of the parameters passed to" )
(msg N T "function are valid and that the function is called at a valid" )
(msg N T "place in the program." )
)
)
(cond ( (equal -read 'yes)
  ; This function does a read.
  ( msg N )
  ( msg N T "This function is trying to do a read. You should check" )
  ( msg N T "that there is something available for the function to" )
  ( msg N T "read. If there is no data to read then the function" )
  ( msg N T "may fail to return." )
)
)
)

```



```

(goal run again) : Determine if user has another problem.
(goal clean trace) ; clean up trace facts.
) : end of aha_no_source

```

```

; This rule is fired if we find a function that does not return and for
; which we have no source code and deadlock is a possibility.

```

```

(defp aha_deadlock
  (goal (trace -type))
  (NO_FUNC -name -sys -read)
  (HYP Deadlock -weight))
test(and (or (equal -type 'loop) (equal -type 'deadlock))
      (> -weight -1)
      (not (equal -read 'unk)))
)
- > (remove 1 2)
(cond ((equal -sys 'yes)
      ; This is a system function.
      (msg N T "The system function "" -name "" does not return." )
      (msg N T "Because the debugger does not have source code for this" )
      (msg N T "function it is not possible to continue the trace." N)
      )
      (equal -sys 'no)
      ; The core dumps occurs for a non-system function.
      (msg N T "The debugger does not have the source code for the function" )
      (msg N T "" -name ". If this function was written as part of the" )
      (msg N T "program and you forgot to give the name of the file containing" )
      (msg N T "the source then you must start the debugging session from the" )
      (msg N T "beginning." N)
      )
      )
)
(cond ((equal -read 'yes)
      ; This function does a read.
      (msg N T "This function could be where the program is entering into a" )
      (msg N T "deadlock state, because this function is trying to read" )
      (msg N T "information. You should check that there is data available" )
      (msg N T "for the function to read. If there is no data to read then" )
      (msg N T "the function may fail to return. If the read operation is" )
      (msg N T "attempting to read from another process or across a network" )
      (msg N T "then be sure that the other program is sending information" )
      (msg N T "at the time your program is attempting to read." )
      )
      )
)
)
(goal run again) : Determine if user has another problem.
(goal clean trace) ; clean up after the trace.
) : end of aha_deadlock

```

: This rule fired if we find a function that does not return and for which
: we have no source code and deadlock is a possibility.

```
(defp aha_deadlock_read
  (goal (trace -type))
  (NO_FUNC -name -sys unk)
  test (or (equal -type 'loop) (equal -type 'deadlock))
  -->( remove 2 )
  (prog (ans)
    again
    (msg N T "Does the function " -name " do some type of read")
    (msg N T "operation (y or n)? ")
    (setq ans ( call_help (lread)
      ("If the function reads a value from some place then type 'y.'"
       "If the function computes a value or performs some other type"
       "of operation then type 'n.'" ) )
    )
    (cond ( (null ans)
      ; Repeat the prompt
      (go again)
      )
      (equal ans 'y)
      ; Yes, it does a read operation
      (fact NO_FUNC -name -sys yes)
      )
      (equal ans 'n)
      ; No, it does not do a read operation
      (fact NO_FUNC -name -sys no)
      )
      (t ; Invalid input
        (msg N N T "**** Invalid input, try again!" N)
        (go again)
        )
      )
    )
  )
) ; end of aha_deadlock_read
```

```
(installp 'start_loop_trace 'start_no_halt_trace 'start_deadlock_trace
  'aha_loop 'show_loop 'aha_deadlock
  'aha_no_source 'aha_deadlock_read
)
```

```
.....
::
:: File : help.l
::
:: Created : 02/29/88 by : Eric Byrne
::
:: Purpose : This file contains the source code for help facility
```

```
::      used in the kbpd.
```

```
::      Functions :
```

```
::      call_help
```

```
::      help_fac
```

```
::      show_help
```

```
::
```

```
::
```

```
::.....
```

```
::.....
```

```
::
```

```
::      Function : call_help
```

```
::
```

```
::      Purpose : Every time the KBPD issues a prompt to the user it is  
::                possible for the user to invoked the help facility.
```

```
::                The logic for processing the user's input to determine  
::                if the help facility was invoked is the same for most  
::                prompts, so we after a prompt is issued and the user's  
::                response is read then this function is called.
```

```
::
```

```
::      Arguments : line - The user's response.
```

```
::                explain - An explanation for the prompt to be passed  
::                to the help facility in case the user wants  
::                a KBPD prompt explained.
```

```
::
```

```
::      Returns : nil - If the KBPD should issue the prompt again.
```

```
::                atom - An atomic value which is the value of line  
::                converted from a list into an atom.
```

```
::
```

```
(defun call_help ( line explain )
```

```
  ( prog ()
```

```
    ( cond ( (equal '? (nthword 1 line) )
```

```
      ; User is invoking the help facility
```

```
      ( help_fac line explain )
```

```
      ( return nil ) ; Back from the help facility, repeat prompt.
```

```
    )
```

```
    ( (equal '? (getchar (nthword 1 line) 1) ) )
```

```
      ; User did not put a space after the question mark
```

```
      ( msg N T "*** ? must be followed by a space." N )
```

```
      ( return nil )
```

```
    )
```

```
    ( (null line)
```

```
      ; User did not enter anything, repeat the prompt
```

```
      ( return nil )
```

```
    )
```

```
  ) ; Pname is a list assign it an atomic value.
```

```

        ( return (nthword 1 line) )
    )
)
) ; end of call_help

```

```

.....

```

```

:
: Function : help_fac
:
: Purpose : This is the top level function in the help facility. It
:           is passed the line of text entered by the user and will
:           attempt to handle the action requested by the user. If
:           no action is entered then the help facility remains 'on'
:           until the user requests to exit.
:
: Arguments : line - The list of text entered by the user in the
:                 function that called the help facility.
:           explain - If the user is merely asking for an explanation
:                 for the prompt issued by the calling function
:                 then this is the explanation it be given.
:
: Returns : nil
:
:

```

```

(defun help_fac ( line explain )

```

```

  ( prog ( action ; Help action requested by the user.
          enabled ; true, if the help facility is now directing the session.
                ; false, if the help facility has been called solely to
                ; do one action.
        )

```

```

    ( setq enabled 'false )
    ( setq action ( nthword 2 line ) )

```

```

  help_loop

```

```

  (cond ( (and (null action) (equal enabled 'false))
        : User wants an explanation for a KBPD prompt. Print the
        : explanation
        ( msg N )
        ( show_help explain )
      )

```

```

    ( (and (equal action '?') (equal enabled 'true' ) )

```

```

: User wants a description of the help facility commands.
(msg N)
(show_help '( "The help facility commands are:"
              "? - Displays this incredibly useful screen."
              "exit - Exits the help facility and returns to
              the debugger."
              "quit - Terminates the Amber debugger."
              "shell - Creates a UNIX shell."
              "stop - Halts the debugger after the current"
              rule is done firing."
            )
)
)

(( (or (and (equal action 'exit) (equal enabled 'true))
      (and (equal action 'e) (equal enabled 'true))
    )
)
: The user wants to exit the help facility and get back to the
: debugger.
(setq enabled 'false)
(return)
)

(( (or (and (equal action '?) (equal enabled 'false))
      (and (equal action 'help) (equal enabled 'false))
      (and (equal action 'h) (equal enabled 'false))
    )
)
: Users wants to enter help mode, stay until user asks to
: leave
(msg N T "Help Facility entered." N)
(setq enabled 'true)
)

(( (equal action 'quit)
  (equal action 'q)
)
)
: The user wants to quit the debugger so halt everything
(exit) ; Stop YAPS and exits the lisp process.
)

(( (or (equal action 'shell)
      (equal action 'sh)
    )
)
: The user wants to use UNIX for awhile create a shell.
(msg N T "*** Press CNTL-d to resume the debugger." N)
(shell)
(msg N T "*** Welcome back." N)
)
)

```

```

    ( (or (equal action 'stop)
          (equal action 'st)
        )
      ; Halt the debugger as soon as the current rule is done.
      (halt)
    )

    ( (null line)
      ; User did not enter anything, repeat the prompt
      ()
    )

    ( t ; An invalid help action was requested. The user really
      ; needs help.
      (msg N T "Invalid help action!" N)
    )
  )

  ( cond ( (equal enabled 'true)
    ; The help facility has been enabled, issue a prompt for a
    ; help action.
    ( msg T "?> " )
    (setq line ( lread ))
    (setq action ( nthword 1 line ))
    (go help_loop)
  )
)
) ; end of help_fac

```

```

*****
:
: Function : show_help
:
: Purpose : This function is used to display a help message
:           for some prompt.
:
: Arguments : message - A list containing the message to be printed.
:
: Returns : nil
:
:

```

```

(defun show_help (message)
  (cond ( (not (null message))
    ( msg T (car message) N )
    ( show_help (cdr message) )
  )
)

```

```

)
) ;end of show_help
.....
;;
;; File : misc.l
;;
;; Created : 02/24/88          by : Eric Byrne
;;
;; Purpose : This file contains miscellaneous lisp functions that
;;           perform various tasks.
;;
;; Functions :
;;   lread
;;   nthword
;;   sys_func
;;
.....

```

```

.....
;
; Function : lread
;
; Purpose : This function reads in an entire line of text from a
;           terminal. The line is stored as a list with each
;           character being an atom.
;
; Arguments : None
;
; Returns : A list that holds the entire line read from a terminal.
;

```

```

(defun lread ()
  (prog (ans)
    loop
      (cond ((nequal 10 (tyipeek))
             ; While the next character is not a newline read it in
             (setq ans (cons (readc) ans))
             (go loop)
            )
            (t ; The next character is a newline. Read it to remove
              ; it from the input stream and then return the line
              ; read
              (readc)
              (return (reverse ans))
            )
          )
    )
)

```

```
) ; end of lread
```

```
.....
```

```
;  
;  
;  
;  
;  
;  
;  
;  
;  
;  
;  
;  
;
```

```
Function : nthword
```

```
Purpose : Returns the nth word in a list created by a call to lread.
```

```
Arguments : word - A number indicating which word to find.  
line - The list returned by a call to lread.
```

```
Returns : result - An atom whose name is the word found or nil.
```

```
(defun nthword ( word line )  
  ( prog ( buf result )  
    (cond ( (lessp word 1 )  
            ; The word parameter is bad  
            ( msg N " *** Error bad word parameter passed to nthword" N )  
            ( return nil )  
          )  
    )  
    (setq buf line) ; Copy the input line.  
    (setq result nil) ; Initialize the answer to nil.  
    past_space  
    (cond ( (null buf)  
            ; The buffer is empty  
            ( return nil )  
          )  
          ( (equal 'l (car buf))  
            ; First atom in the line is a space. Remove it  
            (setq buf (cdr buf))  
            ( go past_space )  
          )  
    )  
    ; When we reach here we are at the beginning of a word  
    (setq word (diff word 1))  
    (cond ( (equal 0 word)  
            ; If zero then we are at the beginning of the word we  
            ; are looking for.  
            ( prog ()  
              get_it  
              (setq result (cons (car buf) result))  
              (setq buf (cdr buf))  
              (cond ( ( null buf )
```



```

        ; At end of list
        (return (setq result (implode (reverse result ))))
    )
    ( (nequal '11 (car buf) )
      ; Still more characters in the word
      ( go get_it )
    )
    ( t ; Not at end of list and car is a space
      ; so we have reached end of word.
      (return (setq result (implode (reverse result ))))
    )
  )
)
)
)
)
(t ; Not at the start of the word we want so move pass
  ; it.
  (prog ()
    past_word
    ( cond ( (null buf)
              ; The buffer is empty
              ( return nil )
            )
          ( (nequal '11 (car buf) )
              ; We are at a letter, move past it
              (setq buf (cdr buf) )
              ( go past_word )
            )
        )
    )
  )
  ; Go to the start of the next word
  ( go past_space )
)
)
(return result )
)
) ; end of nthword

```

```

.....
:
: Function : sys_func
:
: Purpose : This function checks to see if a function name found
:           during a trace is the name of a C system function.
:
: Arguments : name - The function name.
:
: Returns : A list of the form ( system read ) where "system" is

```

```

:           yes or no depending if the name is a system function.
:           "read" is yes or no or unk depending on if the function
:           is a C function and whether it does a read operation.
:

```

```

(defun sys_func ( name )
  (prog ( funcs choice ans )

```

```

    ; Here is the list of known C system functions.
    (setq funcs '(( (atof no ) (atoi no ) (atol no )
                    (calloc no ) (clearerr no ) (fclose no )
                    (fdopen yes) (feof no ) (ferror no )
                    (fflush no ) (fgetc yes) (fgets yes)
                    (fileno no ) (fopen no ) (fprintf no )
                    (fputc no ) ( fputs no ) (fread yes)
                    (free no ) (freopen no ) (fscanf yes)
                    (fseek no ) (ftell no ) (fwrite no )
                    (getc yes) (getchar yes) (getenv no )
                    (getopt no ) (gets yes) (malloc no )
                    (mktemp no ) (pclose no ) (popen no )
                    (printf no ) (putc no ) (putchar no )
                    (puts no ) (realloc no ) (strncpy no )
                    (strpbrk no ) (strchr no ) (strspn no )
                    (strtod no ) (strtok no ) (strtol no )
                    (system no ) (tempnam no ) (tmpfile no )
                    (ungetc no )
                  ))

```

```

    ; Now search the list to see if the function name is in the list.

```

```

loop
  (setq choice (car funcs))
  (cond ((null choice)
         ; Out of choices
         (setq ans '( no unk ))
        )
        ((equal (car choice) name )
         ; Got a match
         (setq ans (list 'yes (cadr choice)))
        )
        (t ; Try again
         (setq funcs (cdr funcs))
         (go loop)
        )
  )
  (return ans)
)
) ; end of function sys_func

```

```

.....
::
:: File : prt_code.l

```

```

;;
;; Created : 03/15/88          by : Eric Byrne
;;
;; Purpose : This file contains the Lisp functions used to print
;;           a statement or expression so that it resembles C code.
;;

```

```

; Pad is a global variable used to specify the number of blanks to pad the
; beginning of a line with when a statement is printed out.
(setq Pad " ")

```

```

;
; Function : prt_stmt
;
; Purpose  : This function takes a list containing information about
;           a program statement, reconstructs the statement and
;           prints it.
;
; Arguments : stmt - The statement list
;
; Returns  : nothing

```

```

(defun prt_stmt (stmt)
  (prog (type ; Statement type)
    (setq type (car stmt))
    (cond ((equal type 'B)
           ; Don't know what to do with a block just now.
           (msg N "Block start??" N)
          )
          ((equal type 'S)
           ; Yes, it is a statement
           (setq type (cadr stmt))
           (cond ((equal type 'C)
                  ; A statement with a case label
                  (msg Pad "case ")
                  (prt_expr (caddr stmt))
                  (msg " : ")
                  (prt_stmt (caddr stmt))
                )
              ((equal type 'D)
                 ; A statement with a default label
                 (msg Pad "default : ")
                 (prt_stmt (caddr stmt))
              )
            )
          )
  )

```

```

(equal type 'E)
: An expression statement
(msg Pad )
(prt_expr (caddr stmt))
(msg " : " N)
)
(equal type 'L)
: A statement with a normal label
(msg Pad )
(msg (caddr stmt))
(msg " : ")
: Don't print the statement part
)
(equal type 'I)
: An If statement (no else)
(msg Pad "if ( ")
(prt_expr (caddr stmt))
(msg ") " N)
)
(equal type 'IE)
: An If-then-else statement
(msg Pad "if ( ")
(prt_expr (caddr stmt))
(msg ") " N)
)
(equal type 'DW)
: A Do-While statement
(msg Pad "do " N)
)
(equal type 'W)
: A While statement
(msg Pad "while ( ")
(prt_expr (caddr stmt))
(msg ") " N)
)
(equal type 'F)
: A For statement
(msg Pad "for ( ")
(prt_expr (caddr stmt))
(msg " : ")
(prt_expr (caddr stmt))
(msg " : ")
(prt_expr (caddrddr stmt))
(msg ") " N)
)
(equal type 'S)
: A Switch statement
(msg Pad "switch ( ")
(prt_expr (caddr stmt))
(msg ") " N)

```

```

)
((equal type 'G)
 : A Goto statement
 ( msg Pad "goto " )
 ( msg (caddr stmt) )
 ( msg " : " N )
)
((equal type 'B)
 : A Break statement
 ( msg Pad "break ;" N )
)
((equal type 'N)
 : A Null statement
 ( msg Pad " : " N )
)
((equal type 'CN)
 : A Continue statement
 ( msg Pad "continue ;" N )
)
((equal type 'R )
 : A Return statement
 ( msg Pad "return " )
 ( prt_expr (caddr stmt) )
 ( msg " : " N )
)
( t : Invalid statement type
 (msg N )
 (msg N T
  "**** Internal error, prt_stmt, bad statement type "
  type N )
)
)
)
((equal type 'E)
 : It's an expression list, not a statement list
 (prt_expr stmt)
)
( t : Invalid type
 (msg N )
 (msg N T "**** Internal error, prt_stmt, bad type " type N )
)
)
)
) : end of prt_stmt

```

```

.....
:
: Function : prt_expr

```

```

;
; Purpose : This function takes a list containing information about
;          a program expression, reconstructs the expression and
;          prints it.
;
; Arguments : expr - The expression list
;
; Returns  : nothing
;
(defun prt_expr (expr)
  (prog (type ; The type of expression
        )
    (setq type (cadr expr) )
    (cond ( (listp type)
           ; The second element is a list. So it is either a postfix
           ; operator or an operator that takes two arguments.
           (setq type (caddr expr) )
           (cond ( (equal type 'DOT)
                  ; Where DOT is the "." operator used for accessing
                  ; structure elements.
                  ( prt_expr (cadr expr) )
                  ( msg "." )
                  ( prt_expr (caddr expr) )
                  )
              ( (equal type 'MEM_PTR)
                  ; Where MEM_PTR is the operator "->" used to access
                  ; structure elements.
                  ( prt_expr (cadr expr) )
                  ( msg "->" )
                  ( prt_expr (caddr expr) )
                  )
              ( (equal type 'POST_ADD)
                  ; Where POST_ADD is the "++" operator used in a
                  ; postfix style.
                  ( prt_expr (cadr expr) )
                  ( msg "++" )
                  )
              ( (equal type 'POST_SUB)
                  ; Where POST_SUB is the "--" operator used in a
                  ; postfix style.
                  ( prt_expr (cadr expr) )
                  ( msg "--" )
                  )
              ( (equal type 'MULT)
                  ; Where MULT is the "*" operator used to do
                  ; multiplication.
                  ( prt_expr (cadr expr) )
                  ( msg "*" )
                  ( prt_expr (caddr expr) )
                  )
            )
        )
  )

```

```

)
(equal type 'DIV)
: Where DIV is the "/" operator used to do division.
( prt_expr (cadr expr) )
( msg "/" )
( prt_expr (caddr expr) )
)
(equal type 'REMAIN)
: Where REMAIN is the "%" operator used to do integer
: modulo division.
( prt_expr (cadr expr) )
( msg "%" )
( prt_expr (caddr expr) )
)
(equal type 'PLUS)
: Where PLUS is the "+" operator used to do addition.
( prt_expr (cadr expr) )
( msg "+" )
( prt_expr (caddr expr) )
)
(equal type 'MINUS)
: Where MINUS is the "-" operator used for subtraction.
( prt_expr (cadr expr) )
( msg "-" )
( prt_expr (caddr expr) )
)
(equal type 'LEFT_SHIFT)
: Where LEFT_SHIFT is the "<<" operator used for bitwise
: left shift.
( prt_expr (cadr expr) )
( msg "<<" )
( prt_expr (caddr expr) )
)
(equal type 'RIGHT_SHIFT)
: Where RIGHT_SHIFT is the ">>" operator used for bitwise
: right shift.
( prt_expr (cadr expr) )
( msg ">>" )
( prt_expr (caddr expr) )
)
(equal type 'LESSTHAN)
: Where LESSTHAN is the "<" operator used for numeric
: comparison.
( prt_expr (cadr expr) )
( msg "<" )
( prt_expr (caddr expr) )
)
(equal type 'LTE)
: Where LTE is the "<=" operator used for numeric
: comparison.

```

```

( prt_expr (cadr expr) )
( msg " <=" )
( prt_expr (caddr expr) )
)
(equal type 'GTE)
: Where GTE is the ">=" operator used for numeric
: comparison.
( prt_expr (cadr expr) )
( msg ">=" )
( prt_expr (caddr expr) )
)
(equal type 'GREATERTHAN)
: Where GREATERTHAN is the ">" operator used for numeric
: comparison.
( prt_expr (cadr expr) )
( msg ">" )
( prt_expr (caddr expr) )
)
(equal type 'EQUAL)
: Where EQUAL is the "==" operator used for comparison.
( prt_expr (cadr expr) )
( msg "==" )
( prt_expr (caddr expr) )
)
(equal type 'NOT_EQUAL)
: Where NOT_EQUAL is the "!=" operator used for
: comparison.
( prt_expr (cadr expr) )
( msg "!=" )
( prt_expr (caddr expr) )
)
(equal type 'BIT_AND)
: Where BIT_AND is the "&" operator used for bitwise AND
: operation.
( prt_expr (cadr expr) )
( msg "&" )
( prt_expr (caddr expr) )
)
(equal type 'BIT_EXCL_OR)
: Where BIT_EXCL_OR is the "^" operator used for bitwise
: exclusive OR.
( prt_expr (cadr expr) )
( msg "^" )
( prt_expr (caddr expr) )
)
(equal type 'BIT_OR)
: Where BIT_OR is the "|" operator used for bitwise OR
: operations.
( prt_expr (cadr expr) )
( msg "|" )

```



```

    ( prt_expr (caddr expr) )
  )
  ( (equal type 'L_AND)
    ; Where L_AND is the "&&" operator used for logical AND
    ; operations.
    ( prt_expr (cadr expr) )
    ( msg " && " )
    ( prt_expr (caddr expr) )
  )
  ( (equal type 'L_OR)
    ; Where L_OR is the "||" operator used for logical OR
    ; operations.
    ( prt_expr (cadr expr) )
    ( msg " || " )
    ( prt_expr (caddr expr) )
  )
  ( (equal type 'ASSIGN)
    ; Where ASSIGN is the "=" operator used for assignments.
    ( prt_expr (cadr expr) )
    ( msg " = " )
    ( prt_expr (caddr expr) )
  )
  ( (equal type 'ADD_ASGN)
    ; Where ADD_ASGN is the "+=" operator used for
    ; assignments.
    ( prt_expr (cadr expr) )
    ( msg " += " )
    ( prt_expr (caddr expr) )
  )
  ( (equal type 'MINUS_ASGN)
    ; Where MINUS_ASGN is the "-=" operator used for
    ; assignments.
    ( prt_expr (cadr expr) )
    ( msg " -= " )
    ( prt_expr (caddr expr) )
  )
  ( (equal type 'MULT_ASGN)
    ; Where MULT_ASGN is the "*=" operator used for
    ; assignments.
    ( prt_expr (cadr expr) )
    ( msg " *= " )
    ( prt_expr (caddr expr) )
  )
  ( (equal type 'DIV_ASGN)
    ; Where DIV_ASGN is the "/=" operator used for
    ; assignments.
    ( prt_expr (cadr expr) )
    ( msg " /= " )
    ( prt_expr (caddr expr) )
  )
  )

```

```

( (equal type 'REMAIN_ASGN)
; Where REMAIN_ASGN is the "%=" operator used for
; assignments.
( prt_expr (cadr expr) )
( msg "%=" )
( prt_expr (caddr expr) )
)
( (equal type 'RS_ASGN)
; Where RS_ASGN is the ">=" operator used for
; assignments.
( prt_expr (cadr expr) )
( msg ">=" )
( prt_expr (caddr expr) )
)
( (equal type 'LS_ASGN)
; Where LS_ASGN is the "<=" operator used for
; assignments.
( prt_expr (cadr expr) )
( msg "<=" )
( prt_expr (caddr expr) )
)
( (equal type 'BIT_AND_ASGN)
; Where BIT_AND_ASGN is the "&=" operator used for
; assignments.
( prt_expr (cadr expr) )
( msg "&=" )
( prt_expr (caddr expr) )
)
( (equal type 'BIT_E_OR_ASGN)
; Where BIT_E_OR_ASGN is the "^=" operator used for
; assignments.
( prt_expr (cadr expr) )
( msg "^=" )
( prt_expr (caddr expr) )
)
( (equal type 'BIT_OR_ASGN)
; Where BIT_OR_ASGN is the "+=" operator used for
; assignments.
( prt_expr (cadr expr) )
( msg "+=" )
( prt_expr (caddr expr) )
)
)
)
( (equal type 'STAR_VAL)
; Where STAR_VAL is the "*" operator used to access the
; value of a pointer.
( msg "*" )
( prt_expr (caddr expr) )
)
)

```

```

(equal type 'ADDRESS)
: Where ADDRESS is the "&" operator used to get the
; address of an object.
(msg "&")
(prt_expr (caddr expr))
)
(equal type 'PRE_ADD)
: Where PRE_ADD is the "++" operator used in prefix format.
(msg "++")
(prt_expr (caddr expr))
)
(equal type 'PRE_SUB)
: Where PRE_SUB is the "--" operator used in prefix format.
(msg "--")
(prt_expr (caddr expr))
)
(equal type 'NEGATE)
: Where NEGATE is the "-" operator used to take the negative
; of a number.
(msg "-")
(prt_expr (caddr expr))
)
(equal type 'NOT)
: Where NOT is the "!" operator used to perform a boolean
; NOT operation.
(msg "!")
(prt_expr (caddr expr))
)
(equal type 'ONES_COMP)
: Where ONES_COMP is the "~" operator used to do a bitwise
; ones complement.
(msg "~")
(prt_expr (caddr expr))
)
(equal type 'CT)
: A Cast expression
(msg "(")
(prt_expr (caddr expr))
(msg ")")
(prt_expr (caddrr expr))
)
(equal type 'SZ)
: A Sizeof expression
(msg "sizeof ")
(prt_expr (caddr expr))
)
(equal type 'IF)
: An If expression
(prt_expr (caddr expr))
(msg "? ")

```

```

( prt_expr (caddr expr) )
( msg " : " )
( prt_expr (caddr expr) )
)
( (equal type 'I)
  : An identifier
  ( msg (caddr expr) )
)
( (equal type 'N)
  : A number
  (setq type (caddr expr))
  (cond ( (equal type 'I)
          : Integer number
          (msg (caddr expr) )
        )
        ( (equal type 'F)
          : Floating point number
          (msg (caddr expr) )
        )
        ( (equal type 'H)
          : Hexidecimal number
          (msg "X" (caddr expr) )
        )
        ( (equal type 'O)
          : Octal number
          (msg "O" (caddr expr) )
        )
      )
  )
( (cond ( (equal (caddr expr) 'L )
          : Has a long marker with it.
          ( msg "L" )
        )
      )
)
)
( (equal type 'S)
  : A string
  ( msg (caddr expr) )
)
( (equal type 'C)
  : A single character
  ( msg (caddr expr) )
)
( (equal type 'P)
  : An expression is parentheses
  ( msg "(" )
  ( prt_expr (caddr expr) )
  ( msg ")" )
)
( (equal type 'A)
  : A subscript expression

```

```

    ( prt_expr (caddr expr) )
    ( msg "[ " )
    ( prt_expr (caddrr expr) )
    ( msg " ]" )
  )
  ( (equal type 'F)
    ; A function call
    ( prt_expr (caddr expr) )
    ( msg "(" )
    ( prt_expr (caddrr expr) )
    ( msg ")" )
  )
  ( (equal type 'SP)
    ; A list of expression separated by commas
    ( setq type (caddr expr) )
    ( prog ()
      repeat
        ( prt_expr (car type) )
        ( setq type (cdr type) )
        ( cond ( (not (null type))
                  ( msg ", " )
                  (go repeat )
                )
        )
      )
    )
  )
  ( t ; Invalid type
    (msg N )
    (msg N T "**** Internal error : bad expression type : " type N)
  )
)
)
) ; end of prt_expr

```

```

.....

```

```

;
; Function : prt_body
;
; Purpose : Prints a section of code.
;
; Arguments : cont - the continuation containing the code to print.
;
; Returns : Nothing

```

```

(defun prt_body ( cont )
  ( prog ( type ; Statement type
          stmt ; Statement or continuation

```

```

padding ; indentation
)
(setq type (car cont))
(cond ((listp type)
      ; If a list then car it again
      (setq type (car type))
      (setq stmt (car cont))
    )
      (t ; cont is only one list, not part of a sublist
        (setq stmt cont)
        (setq cont nil)
      )
    )
)
(cond ((equal type 'B)
      ; Print brackets
      (msg Pad "[" N)
      (setq padding Pad)
      (setq Pad (concat Pad " "))
      (prt_body (cadr stmt))
      (setq Pad padding)
      (msg Pad "]" N)
    )
      ((equal type 'S)
        ; Yes, it is a statement
        (setq type (cadr stmt))
        (cond ((equal type 'C)
              ; A statement with a case label
              (msg N Pad "case ")
              (prt_expr (caddr stmt))
              (msg " : " N)
              (prt_body (caddr stmt))
            )
              ((equal type 'D)
                ; A statement with a default label
                (msg Pad "default : ")
                (prt_body (caddr stmt))
              )
            )
          ((equal type 'E)
            ; An expression statement
            (msg Pad)
            (prt_expr (caddr stmt))
            (msg " ; " N)
          )
          ((equal type 'L)
            ; A statement with a normal label
            (msg Pad)
            (msg (caddr stmt))
            (msg " ; ")
            (prt_body (car stmt))
          )
        )
    )
)

```

```

(equal type 'I)
; An If statement (no else)
(msg Pad "if ( " )
(prt_expr (caddr stmt) )
(msg " )" N )
(prt_body (caddrddr stmt) )
)
(equal type 'IE)
; An If-then-else statement
(msg Pad "if ( " )
(prt_expr (caddr stmt) )
(msg " )" N )
(prt_body (caddrddr stmt) )
(msg Pad "else" N )
(prt_body (caddrdddr stmt) )
)
(equal type 'DW)
; A Do-While statement
(msg Pad "do " N )
(prt_body (caddr stmt) )
(msg Pad "while ( " )
(prt_expr ( caddrddr stmt) )
(msg Pad " )" N )
)
(equal type 'W)
; A While statement
(msg Pad "while ( " )
(prt_expr (caddr stmt) )
(msg " )" N )
(prt_body (caddrddr stmt) )
)
(equal type 'F)
; A For statement
(msg Pad "for ( " )
(prt_expr (caddr stmt) )
(msg ";" )
(prt_expr (caddrddr stmt) )
(msg ";" )
(prt_expr (caddrdddr stmt) )
(msg " )" N )
(prt_body (caddrdddr stmt) )
)
(equal type 'S)
; A Switch statement
(msg Pad "switch ( " )
(prt_expr (caddr stmt) )
(msg " )" N )
(prt_body (caddrddr stmt) )
)
(equal type 'G)

```

```

      ; A Goto statement
      ( msg Pad "goto " )
      ( msg ( caddr stmt ) )
      ( msg " ;" N )
    )
    ( equal type 'B )
      ; A Break statement
      ( msg Pad "break ;" N )
    )
    ( equal type 'N )
      ; A Null statement
      ( msg Pad " ;" N )
    )
    ( equal type 'CN )
      ; A Continue statement
      ( msg Pad "continue ;" N )
    )
    ( equal type 'R )
      ; A Return statement
      ( msg Pad "return " )
      ( prt_expr ( caddr stmt ) )
      ( msg " ;" N )
    )
    ( t ; Invalid statement type
      ( msg N )
      ( msg N T
        "*** Internal error, prt_stmt, bad statement type "
        type N )
    )
  )
  ( cond ( ( not ( null ( cdr cont ) ) )
    ; More statements so continue following on
    ( prt_body ( cdr cont ) ) ; Continue with the code.
  )
  )
  )
  ( equal type 'E )
  ; It's an expression list, not a statement list
  ( prt_expr stmt )
  ( prt_body ( cdr cont ) )
  )
  ( t ; Invalid type
    ( msg N )
    ( msg N T "*** Internal error, prt_body, bad type " type N )
  )
  )
)
) ; end of prt_body
.....
;;

```



```

:: File : run_scanner.l
::
:: Created : 02/04/88           by : Eric Byrne
::
:: Purpose : This file contains the rules and functions for
::           handling the C scanner.
::
::

```

```

: This rule is used to prompt the user for the names of his source
: code files. We run the scanner on the named files and then load
: the program information file.

```

```

.....
(defp scan_source
  (goal (run scanner) )
  ->(remove 1)
  (prog (ans           ; Holds answers from the user.
        do_it         ; Holds a command for the UNIX shell to execute.
        s_file        ; Holds a file name.
        pname         ; Holds the program name.
        )
    (setq files '()) ; Initialize to zero
    askname
    (msg N T "Please enter the name of your program : " )
    (setq pname (call_help (lread)
      ("The name of the program is not necessarily the name of a
       source file, but rather a name that you use when referring
       to the whole program. The debugger will use this name to
       create a file that contains information about the entire
       program." )
      )
    (cond ( (null pname)
           ; Repeat the prompt
           (go askname)
         )
      )
    )
  re_try
  (msg N T "You may enter either :")
  (msg N T " a) The name of a file containing a list of program source")
  (msg N T " code and include file names." )
  (msg N T " b) Each source code and include file name separately." )
  (msg N N T "Select : " )
  (setq ans (call_help (lread)
    ("Is there a file that contains a list of the source code"
     "file names used to create the program? If so, then select"
     "choice 'a'. If not then you must enter the name of each"
     "source code file; select choice 'b'.")
    )
  )

```

```

(cond ( (null ans)
      ; Repeat the prompt
      ( go re_try )
    )
  ( (equal ans 'a) ; Read a file containing the names.
    ( prog ()
      prompt_1
      ( msg N T "Enter the file name : " )
      ( setq s_file ( call_help (lread)
        ("Enter the name of the file that contains the list of all"
         "source code and include files that are used to create your"
         "program." ) )
      )
      ( cond ( ( null s_file )
              ; Repeat the prompt
              ( go prompt_1 )
            )
        )
      ( setq do_it ( list 'exec '/usrb/byrne/grad/scanner/scanner
        '-p pname '-f s_file ) )
    )
  )
  ( (equal ans 'b) ; Read in each file name.
    ( msg N T
      "Enter each file name below. Enter a period '.' after you have" )
    ( msg N T "entered all the file names." N )
    ( prog ()
      repeat
      ( msg N T "Enter the name of a source code file : " )
      ( setq s_file (call_help (lread)
        ("Enter the name of a file that contains source code or definitions"
         "used when compiling your program. If you are done entering file"
         "names then enter a period." ) )
      )
      ( cond ( (null s_file)
              ; Repeat the prompt
              ( go repeat )
            )
        ( (equal s_file '.')
          ; User wants to stop
          ( setq do_it ( append ( list 'exec
            '/usrb/byrne/grad/scanner/scanner '-p pname '-l )
            files ) )
          )
        ( t ; The user entered a file name
          ( cond ( ( null files )
                  ; The file list is empty
                  ( setq files ( list s_file ) )
                )
            )
        )
    )
  )

```



```

again
  (cond ((equal . (getchar s_file nth))
        : Found the . in the file name
        (return (concat pre_name ".fct" )
                 )
        )
        ((equal nil (getchar s_file nth))
         : Pass the end of the string the file name has no . in it
         (return (concat s_file ".fct" )
                 )
        )
        (t : Otherwise just keep at searching the name for .
         (cond ((equal pre_name nil)
                : First letter in the name
                (setq pre_name (getchar s_file nth) )
                )
              (t : Just tack the letter onto the end of the list
               (setq pre_name (concat pre_name
                                       (getchar s_file nth) )
                       )
              )
            )
        )
    (setq nth (+ nth 1))
    (go again)
  )
)
)
)
)

```

```

.....
::
:: File : start.l
::
:: Created : 02/04/88           by : Eric Byrne
::
:: Purpose : This file contains the Lisp functions used to
::           initialize and start Amber running.
::
::
::
.....

```

```

.....
:
: Function : start
:
: Purpose : This function does any initialization that Amber needs and
:           then starts the system running.
:
:
:
(defun start ()
  (rm '*); Remove all facts currently defined.
  (setq cur_func nil) ; Initialize this global variable.
)

```

```
( setq Pad " " ) ; Initialize tbis global variable.
( setq In_loop 'false ) ; Initialize tbis global variable.
( goal run_scanner ) ; Run the C program scanner.
)
```

```
.....
:
: Function : banner
:
: Purpose : Just displays a little banner.
:
```

```
( defun banner ()
  ( msg N N N N N N N N N N N N N N N N )
  ( msg N T "Welcome to the Amber program debugger." )
  ( msg N N N )
)
```

```
( defun amber ()
  ( banner )
  ( start )
)
```

```
.....
::
:: File : symptoms.l
::
:: Created : 02/05/88 by : Eric Byrne
::
:: Purpose : Tbis file contains the symptom analyzer rule set.
::
.....
```

: Tbis is the top level symptom menu. From tbis menu we can place the major symptom into some category.

```
(defp start_analyzer
  ( goal ( run symp_anal ) )
  ( (SYM-any) ) ; No symptoms facts yet
  -> ( msg N N )
      ( prog ( ans )
          again
            ( msg N )
            ( msg N T "Whicb of the following best describes the problem your" )
            ( msg N T "program is demonstrating?" )
            ( msg N )
            ( msg N T " a) Tbe program does not terminate." )
          )
  )
```

```

(msg N T " b) The program terminates with a system error message." )
(msg N T " c) The program generates incorrect results." )
(msg N T " d) None of the above seem to fit the problem." )
(msg N T " e) Quit the debugger." )
(msg N )
(msg N T "Select a letter : " )
(setq ans ( call_help (lread)
("Enter a letter a - c that best matches the behavior of your program."
 "If the program does not stop executing then enter 'a'. If the program"
 "ends its execution with a system error message, such as core dump,"
 "enter 'b'. If the output generated by the program is not correct"
 "then enter 'c'. If none of these choices seems to describe the"
 "problem, enter 'd'. If you just want to quit then select 'e'.") )
)
(cond ((null ans)
      ; Repeat the prompt
      (go again)
    )
      ((equal ans 'a)
      ; Possible infinite loop or dead lock.
      (fact SYM No_term) ; Call the next menu.
    )
      ((equal ans 'b)
      ; A system error message is usually a core dump.
      (fact SYM Sys_kill) ; Call the next menu.
    )
      ((equal ans 'c)
      ; Not supported yet
      (fact SYM Bad_output) ; Call the next menu.
    )
      ((equal ans 'd)
      ; Not supported yet
      (msg N T "Come on be decisive. Try again." )
      (msg N T "If the program does terminate and is not aborted by the")
      (msg N T "system then pick 'c'. It may be that your program has" )
      (msg N T "more than one error. If it appears that several things" )
      (msg N T "are going wrong then concentrate on the symptoms of the" )
      (msg N T "error that appear first." )
      (go again)
    )
      ((equal ans 'e)
      ; Just stop
      (exit)
    )
      (t ; User did not select a correct answer
      (msg N T "*** Please pick a valid letter." )
      (go again)
    )
  )
)
)

```

```
) ; end start_analyzer rule
```

```
; When no other rule in the symptom analyzer will fire then we are done  
; running the symptom analyzer. This rule detects this situation and  
; will make the kbpd move on to the next step which is examining the  
; program against a hypothesis.
```

```
(defp end_analyzer  
  (goal (run symp_anal))  
  -> (remove 1) ; done running the analyzer  
      (msg N " Done running the symptom analyzer!" N)  
      (goal select error)  
)
```

```
; If the user says the program does not terminate then this rule fires.
```

```
(defp no_end  
  (goal (run symp_anal))  
  (SYM No_term)  
  -> (msg N)  
      (msg N T "It happens sometimes that the system is running slowly or you"  
              (msg N T "do not realize just how much time your program needs to run.")  
              (prog (ans)  
                    loop  
                    (msg N)  
                    (msg N T "Are you sure that you waited long enough for the program"  
                              (msg N T "to finish (y or n)? "  
                    (setq ans (call_help (read)  
                                         ("Have you run the program before? How long does it take to"  
                                          "execute normally? Is the system slow? If the system is"  
                                          "running slowly or if you not sure how long the program will"  
                                          "take to run for the given data and task then it is possible"  
                                          "that you simply did not wait long enough for the program to"  
                                          "finish.")))  
                    )  
              (cond ((null ans)  
                    : Repeat the prompt  
                    (go loop)  
                    )  
                    ((equal ans 'y)  
                    : Then suspect infinite loop or deadlock.  
                    (fact HYP Inf_loop 2)  
                    (msg N N)  
                    (prog (reply)  
                          repeat  
                          (msg N T
```

```

    "Does the program do process-to-process communication")
(msg N T
  "or communicate with another program via a network?" )
(msg N N T "Please answer y or n : " )
(setq reply ( call_help (read)
  ("Does your program spawn another process and communicate"
   "with it? Does your program read or send messages to"
   "other programs, devices, or systems? If so, then answer"
   "yes. A terminal does not count as a device." )))
)
(cond ( ( null reply )
  ; Repeat the prompt
  ( go repeat )
  )
  ( ( equal reply 'y )
  ; Then deadlock is more likely.
  ( fact HYP Deadlock 1 )
  )
  ( ( equal reply 'n )
  ; Then deadlock is not very likely.
  ( fact HYP Deadlock -2 )
  )
  ( t : User gave an invalid answer
  ( msg N N T "*** Pick y for yes or n for no." )
  ( go repeat )
  )
  )
)
)
)
)
)
(( equal ans 'n )
; Maybe the user did not wait long enough.
(msg N T "Try running your program again. Wait twice as long")
(msg N T
  "as you think you should. Also check the system load.")
(exit )
)
(t : User did not select y or n
  ( msg N N T "*** Pick y for yes or n for no." )
  ( go loop )
)
)
)
)
) : end of no_end rule

```

; If the user says the system aborts the program then this rule is fired.
; For information about system error message see the system function


```
: sigvec(2).
```

```
(defp sys_abort
  (goal (run symp_anal))
  (SYM Sys_kill)
  -> (msg N)
  (prog (ans)
    prompt_1
    (msg N T "Which message is printed when the program is aborted:")
    (msg N)
    (msg N T " a) quit (core dumped)" )
    (msg N T " b) illegal instruction (core dumped)" )
    (msg N T " c) trace trap (core dumped)" )
    (msg N T " d) IOT instruction (core dumped)" )
    (msg N T " e) EMT instruction (core dumped)" )
    (msg N T " f) floating point exception (core dumped)" )
    (msg N T " g) bus error (core dumped)" )
    (msg N T " h) segmentation violation (core dumped)" )
    (msg N T " i) bad argument to system call (core dumped)" )
    (msg N)
    (msg N T " z) None of the above." )
    (msg N N T "Select a letter: ")

    (setq ans (call_help (read)
      (" When the system aborts a program execution it prints a short message"
        " that explains why the program was terminated. Select the lettered"
        " explanation that matches the message given. If you did not notice"
        " the message then re-run your program. You may use the help facility"
        " shell command to re-run your program." )))

    (cond ((null ans)
      : Repeat the prompt
      (go prompt_1)
    )
      ((equal ans 'a)
      : Quit, core dump
      (fact CORE_DUMP SIGQUIT)
      (fact HYP User_quit 2)
    )
      ((equal ans 'b)
      : illegal instruction, core dump
      (fact CORE_DUMP SIGILL)
      (fact HYP Buf_ovrflw 2) ; Buffer overflow into memory
    )
      ((equal ans 'c)
      : trace trap, core dump
      (fact CORE_DUMP SIGTRAP)
      (fact HYP Unk_dump 0)
    )
      ((equal ans 'd)
```

```

: IOT instruction, core dump
( fact CORE_DUMP SIGIOT )
( fact HYP Unk_dump 0 )
)
(equal ans 'e )
: EMT instruction, core dump
( fact CORE_DUMP SIGEMT )
( fact HYP Unk_dump 0 )
)
(equal ans 'f )
: floating point exception, core dump
( fact CORE_DUMP SIGFPE )
( fact HYP Int_ovrflw 0 ) ; integer overflow
( fact HYP Int_div0 0 ) ; integer division by zero
( fact HYP Flt_ovrflw 0 ) ; floating overflow
( fact HYP Flt_div0 0 ) ; floating division by zero
( fact HYP Flt_undflw 0 ) ; floating underflow
( fact HYP Sub_range 0 ) ; Subscript range
)
(equal ans 'g )
: bus error, core dump
( fact CORE_DUMP SIGBUS )
( fact HYP Int_div0 2 ) ; integer division by zero
)
(equal ans 'h )
: segmentation violation, core dump
( fact CORE_DUMP SIGSEGV )
( fact HYP Unk_dump 0 )
)
(equal ans 'i )
: bad argument to system call
( fact CORE_DUMP SIGSYS )
( fact HYP Unk_dump 0 )
)
(equal ans 'z )
: We don't know what the message means.
(msg N T "This debugger does not know how to handle any other")
(msg N T "type of core dump." )
(remove 1 )
(goal run again )
)
( t ; User picked a bad letter.
(msg N N T "*** Invalid choice. Pick a letter a - i, or z!" N)
(go prompt_1 )
)
)
) : end of sys_abort rule

```

; If the user said the program is generating incorrect output then this
; rule is fired. We need to determine in what way is the output incorrect.

```
(defp bad_out
  (goal (run symp_anal))
  (SYM Bad_output)
  -> (msg N)
  (prog (ans)
    repeat
      (msg N T "Which choice below best describes how the output is" )
      (msg N T "incorrect?" )
      (msg N)
      (msg N T " a) No output is created by the program." )
      (msg N T " b) The output is garbled. The values printed are" )
      (msg N T "   ridiculous or meaningless garbage." )
      (msg N T " c) The output is incorrect but reasonable." )
      (msg N T " d) Too much output is generated." )
      (msg N T " e) Some output is generated, but not as much as expected." )
      (msg N T " f) The wrong print statement is being used." )
      (msg N T " g) None of the above describes how the output is wrong." )
      (msg N)
      (msg N T "Select a letter : " )
      (setq ans (call_help (lread)
        ("Enter a letter a - f that best describes the problem with the"
          "program output. Garbled output is output where the correct print"
          "statement seems to be used but where the value of a program variable"
          "is expected there is a nonsense value. For example if expecting a"
          "a value between 1 and 100 and the value 23422342341 is print, that"
          "is garbage. A value 230 is reasonable, but wrong. If none of the"
          "choices describe how the program output is incorrect then select"
          "the letter 'g.'"))
      )
      (cond ( (null ans)
        ; Repeat the prompt
        (go repeat)
        )
        ( (equal ans 'a)
          ; No output is generated
          (fact SYM No_output)
          (fact HYP Bad_path 1) ; Suspect bad logic decision
          (fact HYP Bad_value 0) ; Suspect bad value calculation
          )
        ( (equal ans 'b)
          ; Output values are garbled
          (fact SYM Garbled)
          (fact HYP Bad_format 2) ; Print format does not match vars
          (fact HYP Bad_value 1) ; Suspect bad value calculation
          (fact HYP Bad_path 0) ; Suspect bad logic decision
          )
        ( (equal ans 'c)

```

```

; Output is incorrect
( fact SYM Bad_result )
( fact HYP Bad_format 0 ) ; Print format does not match vars
( fact HYP Bad_path 1 ) ; Suspect bad logic decision
( fact HYP Bad_value 1 ) ; Suspect bad value calculation
)
(equal ans 'd)
; Too much output
( fact SYM Too_much )
( fact HYP Bad_value 1 ) ; Suspect bad value calculation
( fact HYP Bad_path 2 ) ; Suspect bad logic decision
)
(equal ans 'e)
; Too little output
( fact SYM Too_little )
( fact HYP Bad_value 1 ) ; Suspect bad value calculation
( fact HYP Bad_path 2 ) ; Suspect bad logic decision
)
(equal ans 'f)
; Wrong print statement reached
( fact SYM Wrong_sec )
( fact HYP Bad_value 1 ) ; Suspect bad value calculation
( fact HYP Bad_path 2 ) ; Suspect bad logic decision
)
(equal ans 'g)
; No good choices
(msg N )
(msg N T "Try to pick one of the choices that seems at least")
(msg N T "similar to the fashion in which the output it wrong.")
(msg N T "If you can't pick one then the debugger can't help")
(msg N T "you. You can quit the debugger by typing '? quit'.")
(msg N N )
(go repeat )
)
(t ; Invalid input
( msg N N T "*** Invalid input, try again!" N )
( go repeat )
)
)
)
) ; end of rule bad_out

(installp 'start_analyzer 'end_analyzer 'no_end 'sys_abort
'bad_out
)

```

```

.....
::
:: File : trace.l
::
:: Created : 03/10/88          by : Eric Byrne
::
:: Purpose : This file contains the YAPS rules used to trace a
::           program.
::
.....

```

```

.....
..... Rules for handling expressions
.....

```

```

: This rule handles an expression of the form
: ( E (expr1) DOT (expr2) )
: Where DOT is the "." operator used for accessing structure elements.

```

```

(defp exp_dot_oper
  ( goal (trace -type) )
  ( E -lexp DOT -rexp )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          ; Do nothing for this type of expression.
          ; Check for function calls.
          ( make_fact -rexp )
          ( make_fact -lexp ) ; Check this first.
        )
  )
) ; end of rule exp_dot_oper

```

```

: This rule handles an expression of the form
: ( E (expr1) MEM_PTR (expr2) )
: Where MEM_PTR is the operator "->" used to access structure elements.

```

```

(defp exp_mem_ptr_oper
  ( goal (trace -type) )
  ( E -lexp MEM_PTR -rexp )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          ; Do nothing for this type of expression.
          ; Check for function calls.
        )
  )
)

```

```

        ( make_fact -rexp )
        ( make_fact -lexp ) ; Check this first.
    )
) ; end of rule exp_mem_ptr_oper

; This rule handles an expression of the form
; ( E (expr) POST_ADD )
; Where POST_ADD is the "+" operator used in a postfix style.

(defp exp_post_add_oper
  ( goal (trace -type) )
  ( E -exp POST_ADD )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          ; Do nothing for this type of expression.
          ; Check for function calls.
          ( make_fact -exp )
        )
  )
) ; end of rule exp_post_add_oper

; This rule handles an expression of the form
; ( E (expr) POST_SUB )
; Where POST_SUB is the "-" operator used in a postfix style.

(defp exp_post_sub_oper
  ( goal (trace -type) )
  ( E -exp POST_SUB )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          ; Do nothing for this type of expression.
          ; Check for function calls.
          ( make_fact -exp )
        )
  )
) ; end of rule exp_post_sub_oper

; This rule handles an expression of the form
; ( E STAR_VAL (expr) )
; Where STAR_VAL is the "*" operator used to access the value of a pointer.

(defp exp_ptr_val_oper

```

```

    ( goal (trace -type) )
    ( E STAR_VAL -exp )
  ->( remove 2 )
    ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
                (equal -type 'coredump)
              )
            ; Do nothing for this type of expression.
            ; Check for function calls.
            ( make_fact -exp )
          )
    )
) ; end of rule exp_ptr_val_oper

```

: This rule handles an expression of the form
 : (E ADDRESS (expr))
 : Where ADDRESS is the "&" operator used to get the address of an object.

```

(defp exp_address_oper
  ( goal (trace -type) )
  ( E ADDRESS -exp )
  ->( remove 2 )
    ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
                (equal -type 'coredump)
              )
            ; Do nothing for this type of expression.
            ; Check for function calls.
            ( make_fact -exp )
          )
    )
) ; end of rule exp_address_oper

```

: This rule handles an expression of the form
 : (E PRE_ADD (expr))
 : Where PRE_ADD is the "++" operator used in prefix format.

```

(defp exp_pre_add_oper
  ( goal (trace -type) )
  ( E PRE_ADD -exp )
  ->( remove 2 )
    ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
                (equal -type 'coredump)
              )
            ; Do nothing for this type of expression.
            ; Check for function calls.
            ( make_fact -exp )
          )
    )
) ; end of rule exp_pre_add_oper

```

```
: This rule handles an expression of the form
: ( E PRE_SUB (expr) )
```

```
(defp exp_pre_sub_oper
  (goal (trace -type) )
  ( E PRE_SUB -exp )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          : Do nothing for this type of expression.
          : Check for function calls.
          ( make_fact -exp )
        )
  )
) ; end of rule exp_pre_sub_oper
```

```
: This rule handles an expression of the form
: ( E NEGATE (expr) )
: Where NEGATE is the "-" operator used to take the negative of a number.
```

```
(defp exp_negate_oper
  (goal (trace -type) )
  ( E NEGATE -exp )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          : Do nothing for this type of expression.
          : Check for function calls.
          ( make_fact -exp )
        )
  )
) ; end of rule exp_negate_oper
```

```
: This rule handles an expression of the form
: ( E NOT (expr) )
: Where NOT is the "!" operator used to perform a boolean NOT operation.
```

```
(defp exp_not_oper
  (goal (trace -type) )
  ( E NOT -exp )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          : Do nothing for this type of expression.
          : Check for function calls.
        )
  )
)
```



```

        ( make_fact -exp )
      )
    ) ; end of rule exp_not_oper

; This rule handles an expression of the form
; ( E ONES_COMP (expr) )
; Where ONES_COMP is the "~" operator used to do a bitwise ones complement.

(defp exp_ones_comp_oper
  ( goal (trace -type) )
  ( E ONES_COMP -exp )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock))
            (equal -type 'coredump)
          )
        ; Do nothing for this type of expression.
        ; Check for function calls.
        ( make_fact -exp )
      )
  )
) ; end of rule exp_ones_comp_oper

; This rule handles an expression of the form
; ( E (expr1) MULT (expr2) )
; Where MULT is the "*" operator used to do multiplication.

(defp exp_mult_oper
  ( goal (trace -type) )
  ( E -lexp MULT -rexp )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock))
            (equal -type 'coredump)
          )
        ; Do nothing for this type of expression.
        ; Check for function calls.
        ( make_fact -rexp )
        ( make_fact -lexp ) ; Check this first.
      )
  )
) ; end of rule exp_mult_oper

; This rule handles an expression of the form
; ( E (expr1) DIV (expr2) )
; Where DIV is the "/" operator used to do division.

(defp exp_div_oper

```

```

(goal (trace -type) )
(E -lexp DIV -rexp )
->( remove 2 )
(cond ( (or (equal -type 'loop) (equal -type 'deadlock)
            (equal -type 'coredump)
          )
      ; Do nothing for this type of expression.
      ; Check for function calls.
      (make_fact -rexp)
      (make_fact -lexp) ; Check this first.
    )
  )
) ; end of rule exp_div_oper

; This rule handles an expression of the form
; (E (expr1) REMAIN (expr2) )
; Where REMAIN is the "%" operator used to do integer modulo division.

```

```

(defp exp_modulo_oper
  (goal (trace -type) )
  (E -lexp REMAIN -rexp )
  ->( remove 2 )
  (cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
      ; Do nothing for this type of expression.
      ; Check for function calls.
      (make_fact -rexp)
      (make_fact -lexp) ; Check this first.
    )
  )
) ; end of rule exp_modulo_oper

```

```

; This rule handles an expression of the form
; (E (expr1) PLUS (expr2) )
; Where PLUS is the "+" operator used to do addition.

```

```

(defp exp_plus_oper
  (goal (trace -type) )
  (E -lexp PLUS -rexp )
  ->( remove 2 )
  (cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
      ; Do nothing for this type of expression.
      ; Check for function calls.
      (make_fact -rexp)
      (make_fact -lexp) ; Check this first.
    )
  )
)

```

```

)
)
) ; end of rule exp_plus_oper

```

```

; This rule handles an expression of the form
; ( E (expr1) MINUS (expr2) )
; Where MINUS is the "-" operator used for subtraction.

```

```

(defp exp_minus_oper
  (goal (trace -type))
  (E -lexp MINUS -rexp)
  ->(remove 2)
  (cond ((or (equal -type 'loop) (equal -type 'deadlock)
             (equal -type 'coredump)
            )
         ; Do nothing for this type of expression.
         ; Check for function calls.
         (make_fact -rexp)
         (make_fact -lexp) ; Check this first.
        )
  )
) ; end of rule exp_minus_oper

```

```

; This rule handles an expression of the form
; ( E (expr1) LEFT_SHIFT (expr2) )
; Where LEFT_SHIFT is the "<<" operator used for bitwise left shift.

```

```

(defp exp_left_shift_oper
  (goal (trace -type))
  (E -lexp LEFT_SHIFT -rexp)
  ->(remove 2)
  (cond ((or (equal -type 'loop) (equal -type 'deadlock)
             (equal -type 'coredump)
            )
         ; Do nothing for this type of expression.
         ; Check for function calls.
         (make_fact -rexp)
         (make_fact -lexp) ; Check this first.
        )
  )
) ; end of rule exp_left_shift_oper

```

```

; This rule handles an expression of the form
; ( E (expr1) RIGHT_SHIFT (expr2) )
; Where RIGHT_SHIFT is the ">>" operator used for bitwise right shift.

```

```

(defp exp_right_shift_oper

```

```

    ( goal (trace -type) )
    ( E -lexp RIGHT_SHIFT -rexp )
->( remove 2 )
    ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
                (equal -type 'coredump)
              )
            ; Do nothing for this type of expression.
            ; Check for function calls.
            ( make_fact -rexp )
            ( make_fact -lexp ) ; Check this first.
          )
    )
) ; end of rule exp_right_shift_oper

```

```

; This rule handles an expression of the form
;   ( E (expr1) LESSTHAN (expr2) )
; Where LESSTHAN is the "<" operator used for numeric comparison.

```

```

(defp exp_lessthan_oper
  ( goal (trace -type) )
  ( E -lexp LESSTHAN -rexp )
->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
                (equal -type 'coredump)
              )
            ; Do nothing for this type of expression.
            ; Check for function calls.
            ( make_fact -rexp )
            ( make_fact -lexp ) ; Check this first.
          )
  )
) ; end of rule exp_lessthan_oper

```

```

; This rule handles an expression of the form
;   ( E (expr1) LTE (expr2) )
; Where LTE is the "<=" operator used for numeric comparison.

```

```

(defp exp_lte_oper
  ( goal (trace -type) )
  ( E -lexp LTE -rexp )
->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
                (equal -type 'coredump)
              )
            ; Do nothing for this type of expression.
            ; Check for function calls.
            ( make_fact -rexp )
            ( make_fact -lexp ) ; Check this first.
          )
  )
)

```

```

)
)
) ; end of rule exp_lte_oper

```

```

; This rule handles an expression of the form
; ( E (expr1) GTE (expr2) )
; Where GTE is the ">" operator used for numeric comparison.

```

```

(defp exp_gte_oper
  (goal (trace -type))
  (E -lexp GTE -rexp)
  ->(remove 2)
  (cond ((or (equal -type 'loop) (equal -type 'deadlock)
             (equal -type 'coredump)
            )
         ; Do nothing for this type of expression.
         ; Check for function calls.
         (make_fact -rexp)
         (make_fact -lexp) ; Check this first.
        )
  )
) ; end of rule exp_gte_oper

```

```

; This rule handles an expression of the form
; ( E (expr1) GREATERTHAN (expr2) )
; Where GREATERTHAN is the ">" operator used for numeric comparison.

```

```

(defp exp_greaterthan_oper
  (goal (trace -type))
  (E -lexp GREATERTHAN -rexp)
  ->(remove 2)
  (cond ((or (equal -type 'loop) (equal -type 'deadlock)
             (equal -type 'coredump)
            )
         ; Do nothing for this type of expression.
         ; Check for function calls.
         (make_fact -rexp)
         (make_fact -lexp) ; Check this first.
        )
  )
) ; end of rule exp_greaterthan_oper

```

```

; This rule handles an expression of the form
; ( E (expr1) EQUAL (expr2) )
; Where EQUAL is the "=" operator used for comparison.

```

```

(defp exp_equal_oper

```

```

    ( goal (trace -type) )
    ( E -lexp EQUAL -rexp )
  ->( remove 2 )
    ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
                (equal -type 'coredump)
              )
            ; Do nothing for this type of expression.
            ; Check for function calls.
            ( make_fact -rexp )
            ( make_fact -lexp ) ; Check this first.
          )
    )
) ; end of rule exp_equal_oper

; This rule handles an expression of the form
; ( E (expr1) NOT_EQUAL (expr2) )
; Where NOT_EQUAL is the "!=" operator used for comparison.

```

```

(defp exp_not_equal_oper
  ( goal (trace -type) )
  ( E -lexp NOT_EQUAL -rexp )
  ->( remove 2 )
    ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
                (equal -type 'coredump)
              )
            ; Do nothing for this type of expression.
            ; Check for function calls.
            ( make_fact -rexp )
            ( make_fact -lexp ) ; Check this first.
          )
    )
) ; end of rule exp_not_equal_oper

```

```

; This rule handles an expression of the form
; ( E (expr1) BIT_AND (expr2) )
; Where BIT_AND is the "&" operator used for bitwise AND operation.

```

```

(defp exp_bit_and_oper
  ( goal (trace -type) )
  ( E -lexp BIT_AND -rexp )
  ->( remove 2 )
    ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
                (equal -type 'coredump)
              )
            ; Do nothing for this type of expression.
            ; Check for function calls.
            ( make_fact -rexp )
            ( make_fact -lexp ) ; Check this first.
          )
    )
)

```

```

    )
  )
) ; end of rule exp_bit_and_oper

; This rule handles an expression of the form
;   ( E (expr1) BIT_EXCL_OR (expr2) )
; Where BIT_EXCL_OR is the "^" operator used for bitwise exclusive OR.

(defp exp_bit_excl_or_oper
  (goal (trace -type) )
  (E -lexp BIT_EXCL_OR -rexp )
  ->( remove 2 )
  (cond ((or (equal -type 'loop) (equal -type 'deadlock))
         (equal -type 'coredump)
        )
        ; Do nothing for this type of expression.
        ; Check for function calls.
        (make_fact -rexp )
        (make_fact -lexp ) ; Check this first.
       )
  )
) ; end of rule exp_bit_excl_or_oper

; This rule handles an expression of the form
;   ( E (expr1) BIT_OR (expr2) )
; Where BIT_OR is the "|" operator used for bitwise OR operations.

(defp exp_bit_or_oper
  (goal (trace -type) )
  (E -lexp BIT_OR -rexp )
  ->( remove 2 )
  (cond ((or (equal -type 'loop) (equal -type 'deadlock))
         (equal -type 'coredump)
        )
        ; Do nothing for this type of expression.
        ; Check for function calls.
        (make_fact -rexp )
        (make_fact -lexp ) ; Check this first.
       )
  )
) ; end of rule exp_bit_or_oper

; This rule handles an expression of the form
;   ( E (expr1) L_AND (expr2) )
; Where L_AND is the "&&" operator used for logical AND operations.

(defp exp_l_and_oper

```

```

(goal (trace -type) )
(E -lexp L_AND -rexp )
--> remove 2 )
(cond ( (or (equal -type 'loop) (equal -type 'deadlock)
            (equal -type 'coredump)
          )
      ; Do nothing for this type of expression.
      ; Check for function calls.
      (make_fact -rexp )
      (make_fact -lexp ) ; Check this first.
    )
  )
) ; end of rule exp_1_and_oper

```

```

; This rule handles an expression of the form
; (E (expr1) L_OR (expr2) )
; Where L_OR is the "||" operator used for logical OR operations.

```

```

(defp exp_1_or_oper
  (goal (trace -type) )
  (E -lexp L_OR -rexp )
  --> remove 2 )
  (cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
        ; Do nothing for this type of expression.
        ; Check for function calls.
        (make_fact -rexp )
        (make_fact -lexp ) ; Check this first.
      )
    )
  )
) ; end of rule exp_1_or_oper

```

```

; This rule handles an expression of the form
; (E (expr1) ASSIGN (expr2) )
; Where ASSIGN is the "=" operator used for assignments.

```

```

(defp exp_assign_oper
  (goal (trace -type) )
  (E -lexp ASSIGN -rexp )
  --> remove 2 )
  (cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
        ; Do nothing for this type of expression.
        ; Check for function calls.
        (make_fact -lexp )
        (make_fact -rexp ) ; Check this first.
      )
    )
  )
) ; end of rule exp_assign_oper

```



```

)
)
) ; end of rule exp_assign_oper

```

```

; This rule handles an expression of the form
;   ( E (expr1) ADD_ASGN (expr2) )
; Where ADD_ASGN is the "+" operator used for assignments.

```

```

(defp exp_add_asgn_oper
  ( goal (trace -type) )
  ( E -lexp ADD_ASGN -rexp )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          ; Do nothing for this type of expression.
          ; Check for function calls.
          ( make_fact -lexp )
          ( make_fact -rexp ) ; Check this first.
        )
  )
) ; end of rule exp_add_asgn_oper

```

```

; This rule handles an expression of the form
;   ( E (expr1) MINUS_ASGN (expr2) )
; Where MINUS_ASGN is the "-" operator used for assignments.

```

```

(defp exp_minus_asgn_oper
  ( goal (trace -type) )
  ( E -lexp MINUS_ASGN -rexp )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          ; Do nothing for this type of expression.
          ; Check for function calls.
          ( make_fact -lexp )
          ( make_fact -rexp ) ; Check this first.
        )
  )
) ; end of rule exp_minus_asgn_oper

```

```

; This rule handles an expression of the form
;   ( E (expr1) MULT_ASGN (expr2) )
; Where MULT_ASGN is the "*" operator used for assignments.

```

```

(defp exp_mult_asgn_oper

```

```

    ( goal (trace -type) )
    ( E -lexp MULT_ASGN -rexp )
->( remove 2 )
    ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
                (equal -type 'coredump)
              )
            ; Do nothing for this type of expression.
            ; Check for function calls.
            ( make_fact -lexp )
            ( make_fact -rexp ) ; Check this first.
          )
    )
) ; end of rule exp_mult_asgn_oper

```

```

; This rule handles an expression of the form
;   ( E (expr1) DIV_ASGN (expr2) )
; Where DIV_ASGN is the "/=" operator used for assignments.

```

```

(defp exp_div_asgn_oper
  ( goal (trace -type) )
  ( E -lexp DIV_ASGN -rexp )
->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
                (equal -type 'coredump)
              )
            ; Do nothing for this type of expression.
            ; Check for function calls.
            ( make_fact -lexp )
            ( make_fact -rexp ) ; Check this first.
          )
  )
) ; end of rule exp_div_asgn_oper

```

```

; This rule handles an expression of the form
;   ( E (expr1) REMAIN_ASGN (expr2) )
; Where REMAIN_ASGN is the "%=" operator used for assignments.

```

```

(defp exp_remain_asgn_oper
  ( goal (trace -type) )
  ( E -lexp REMAIN_ASGN -rexp )
->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
                (equal -type 'coredump)
              )
            ; Do nothing for this type of expression.
            ; Check for function calls.
            ( make_fact -lexp )
            ( make_fact -rexp ) ; Check this first.
          )
  )
)

```

```

)
)
) ; end of rule exp_remain_asgn_oper

; This rule handles an expression of the form
; ( E (expr1) RS_ASGN (expr2) )
; Where RS_ASGN is the ">>=" operator used for assignments.

(defp exp_rs_asgn_oper
  (goal (trace -type))
  (E -lexp RS_ASGN -rexp )
  ->( remove 2 )
  (cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          : Do nothing for this type of expression.
          : Check for function calls.
          (make_fact -lexp )
          (make_fact -rexp ) ; Check this first.
        )
  )
) ; end of rule exp_rs_asgn_oper

; This rule handles an expression of the form
; ( E (expr1) LS_ASGN (expr2) )
; Where LS_ASGN is the "<<=" operator used for assignments.

(defp exp_ls_asgn_oper
  (goal (trace -type))
  (E -lexp LS_ASGN -rexp )
  ->( remove 2 )
  (cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          : Do nothing for this type of expression.
          : Check for function calls.
          (make_fact -lexp )
          (make_fact -rexp ) ; Check this first.
        )
  )
) ; end of rule exp_ls_asgn_oper

; This rule handles an expression of the form
; ( E (expr1) BIT_AND_ASGN (expr2) )
; Where BIT_AND_ASGN is the "&=" operator used for assignments.

(defp exp_bit_and_asgn_oper

```

```

(goal (trace -type)
  (E -lexp BIT_AND_ASGN -rexp)
->(remove 2)
  (cond ((or (equal -type 'loop) (equal -type 'deadlock)
             (equal -type 'coredump)
            )
         ; Do nothing for this type of expression.
         ; Check for function calls.
         (make_fact -lexp)
         (make_fact -rexp) ; Check this first.
        )
  )
) ; end of rule exp_bit_and_asgn_oper

; This rule handles an expression of the form
; (E (expr1) BIT_E_OR_ASGN (expr2) )
; Where BIT_E_OR_ASGN is the "=" operator used for assignments.

(defp exp_bit_e_or_asgn_oper
  (goal (trace -type)
    (E -lexp BIT_E_OR_ASGN -rexp)
  ->(remove 2)
    (cond ((or (equal -type 'loop) (equal -type 'deadlock)
               (equal -type 'coredump)
              )
          ; Do nothing for this type of expression.
          ; Check for function calls.
          (make_fact -lexp)
          (make_fact -rexp) ; Check this first.
         )
    )
  )
) ; end of rule exp_bit_e_or_asgn_oper

; This rule handles an expression of the form
; (E (expr1) BIT_OR_ASGN (expr2) )
; Where BIT_OR_ASGN is the "+=" operator used for assignments.

(defp exp_bit_or_asgn_oper
  (goal (trace -type)
    (E -lexp BIT_OR_ASGN -rexp)
  ->(remove 2)
    (cond ((or (equal -type 'loop) (equal -type 'deadlock)
               (equal -type 'coredump)
              )
          ; Do nothing for this type of expression.
          ; Check for function calls.
          (make_fact -lexp)
          (make_fact -rexp) ; Check this first.
         )
    )
  )
)

```

```

)
)
); end of rule exp_bit_or_asgn_oper

```

```

; This rule handles an expression of the form
; ( E CT (cast) (expr) )

```

```

(defp exp_cast
  (goal (trace -type) )
  ( E CT -cast -expr )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          ; Do nothing for this type of expression.
          ; Check for function calls.
          ( make_fact -expr )
        )
  )
); end of rule exp_cast

```

```

; This rule handles an expression of the form
; ( E SZ (expr) )
; expr will be a cast expression or regular expression

```

```

(defp exp_sizeof
  (goal (trace -type) )
  ( E SZ -expr )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          ; Do nothing for this type of expression.
          ; Check for function calls.
          ( make_fact -expr )
        )
  )
); end of rule exp_sizeof

```

```

; This rule handles an expression of the form
; ( E IF (e1) (e2) (e3) )

```

```

(defp exp_if
  (goal (trace -type) )
  ( E IF -test -then -else )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)

```

```

    (equal -type 'coredump)
  )
  ; Trace the test expression and then determine which part of
  ; the conditional expression is evaluated.
  ( fact IF_EXP_PATH -test -then -else )
  ( make_fact -test ) ; Check this first.
)
)
) : end of rule exp_if

```

: This rule is only fired after an if-then-else expression is detected.
 : Ask the user which part of the expression is executed and then trace
 : just that part.

```

(defp if_exp_path
  ( goal (trace -type) )
  ( IF_EXP_PATH -test -then -else )
  ->( remove 2 )
  ; Need to determine which subexpression is executed.
  ( prog (ans)
    repeat
      (msg N T "Is the value of the expression " )
      (prt_expr -test)
      (msg " false (zero) (y or n)? " )
      ( setq ans ( call_help (lread)
        ("This is the test expression for an If-Then expression. To trace"
         "this expression it is necessary to know if the test expression had a"
         "value of false (zero) or not. If the value is zero then enter yes 'y'."
         "If the value is not zero (i.e true ) then enter no 'n'. If the value is"
         "non-zero (true) then the 'then' expression will be executed.") )
      )
      ( cond ( (null ans)
        ; Repeat the prompt
        ( go repeat)
        )
        ( (equal ans 'y)
          ; The value is zero so the 'else' expression is used.
          ( make_fact -else )
          )
        ( (equal ans 'n)
          ; The value is not zero so the 'then' expression is
          ; used.
          ( make_fact -then )
          )
        ( t ; Invalid input
          ( msg N N T "**** Invalid input, try again! Answer y or n" N )
          ( go repeat )
          )
        )
      )
  )
)

```

```

)
) ; end of rule if_exp_path

; This rule handles an expression of the form
;   ( E I ident )

(defp exp_identifier
  (goal (trace -type))
  (E I -ident)
  ->(remove 2)
  (cond ((or (equal -type 'loop) (equal -type 'deadlock)
             (equal -type 'coredump)
            )
         )
        ; Do nothing for this type of expression.
        ()
       )
  )
) ; end of rule exp_identifier

; This rule handles an expression of the form
;   ( E N number type size )
;   type = I, F, H, O size is D or L. D = default

(defp exp_number
  (goal (trace -type))
  (E N -numb -rep -size)
  ->(remove 2)
  (cond ((or (equal -type 'loop) (equal -type 'deadlock)
             (equal -type 'coredump)
            )
         )
        ; Do nothing for this type of expression.
        ()
       )
  )
) ; end of rule exp_number

; This rule handles an expression of the form
;   ( E S string )

(defp exp_string
  (goal (trace -type))
  (E S -string)
  ->(remove 2)
  (cond ((or (equal -type 'loop) (equal -type 'deadlock)
             (equal -type 'coredump)
            )
         )
        ; Do nothing for this type of expression.
        ()
       )
  )
) ; end of rule exp_string

```

```

    )
  )
) ; end of rule exp_string

```

```

; This rule handles an expression of the form
;   ( E C char )
(defp exp_char
  (goal (trace -type))
  (E C -char)
  ->(remove 2)
  (cond ((or (equal -type 'loop) (equal -type 'deadlock)
             (equal -type 'coredump))
         )
        ; Do nothing for this type of expression.
        )
  )
) ; end of rule exp_char

```

```

; This rule handles an expression of the form
;   ( E P (expr) expression in ()
(defp exp_parens
  (goal (trace -type))
  (E P -expr)
  ->(remove 2)
  (cond ((or (equal -type 'loop) (equal -type 'deadlock)
             (equal -type 'coredump))
         )
        ; Do nothing for this type of expression.
        ; Check for function calls.
        (make_fact -expr)
        )
  )
) ; end fo rule exp_parens

```

```

; This rule handles an expression of the form
;   ( E A (expr1) (expr2) expr1[expr2]
(defp exp_subscript
  (goal (trace -type))
  (E A -array -index)
  ->(remove 2)
  (cond ((or (equal -type 'loop) (equal -type 'deadlock)
             (equal -type 'coredump))
         )
        )
  )
)

```



```

        ; Do nothing for this type of expression.
        ; Check for function calls.
        ( make_fact -index )
        ( make_fact -array ) ; Check this first.
    )
)
) ; end of rule exp_subscript

```

```

; This rule handles an expression of the form
; ( E F (expr1) (expr2) ) expr1(expr2) function call

```

```

(defp exp_func_call
  ( goal (trace -type) )
  ( E F -func -list )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          ; Do nothing for this type of expression.
          ; Check for function calls.
          ( fact FUNC_CALL -func )
          ( make_fact -func ) ; Should just be a string.
          ( make_fact -list ) ; Check this first.
        )
  )
) ; end of rule exp_func_call

```

```

; This rule handles a function call. It is fired after the parameter list
; has been traced and now the debugger needs information about the call.

```

```

(defp func_call
  ( goal (trace -type) )
  ( FUNC_CALL -func )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          ; Does the function return when called. If not then trace it
          ; otherwise skip it.
          ( prog (ans name)
                (setq name (caddr -func)) ; Assume -func is form (E I name)
                again
                (msg N T
                 "When the function " name " is called does it return (y or n)? ")
                (setq ans ( call_help (tread)
                                     "If the function returns then it is not necessary to trace its"
                                     "execution. If it does not return then the problem is occurring"
                                     "somewhere during its execution. Answer yes 'y' if the function"

```

```

      "does return and no 'n' if it does not." ))
    )
    (cond ((null ans)
           ; Repeat the prompt
           (go again)
          )
          ((equal ans 'y)
           ; Yes, the function does return, so don't trace it.
           ()
          )
          ((equal ans 'n)
           ; No, the function does not return so trace it.
           (goal get_cont ^ name)
           (goal clean trace) ; remove present trace garbage
           ; first.
          )
          (t ; Invalid input
           (msg N N T
            "*** Invalid input, try again! Answer y or n" N)
           (go again)
          )
    )
  )
)
)
)
(t ; Bad trace type abort
 (msg N T "*** Internal error : bad trace type : " -type N)
 (halt)
)
)
) ; end of rule func_call

```

```

; This rule handles an expression of the form
;   ( E SP (expr) ) comma expression, expr = ( () () ... )

```

```

(defp exp_comma
  (goal (trace -type))
  (E SP -expr)
  ->(remove 2)
  (cond ((or (equal -type 'loop) (equal -type 'deadlock)
            (equal -type 'coredump)
           )
         ; Do nothing for this type of expression.
         ; Check for function calls.
         (cond ((not (null (cdr -expr)))
                ; If not the last expression then set up the next fact
                (fact E SP ^ (cdr -expr)
                 )
               )
          )
         (make_fact (car -expr)) ; Check this first.

```

```

)
)
) ; end of rule exp_comma

```

```

.....
.....
..... Rules for handling Statements
.....
.....

```

```

; This rule handles a statement of the form
;   ( B (body) )
; body is a set of statements ( (S1) (S2) (S3)... )

```

```

(defp stmt_block
  (goal (trace -type) )
  ( B -body )
  ->( remove 2 )
  ( cond ( (equal -type 'break)
            ; Trying to find the statement a break transfers to so don't
            ; open the body. Forget it.
            ()
          )
        ( (equal -type 'continue)
            ; Trying to find the top of a loop. Don't open the body.
            ()
          )
        ( t ; For any other type of trace look in the body.
          ( make_fact -body )
        )
  )
) ; end of rule stmt_block

```

```

; This rule handles a statement of the form
;   ( S C (expr) (stmt) )   case expr : stmt

```

```

(defp stmt_case
  (goal (trace -type) )
  ( S C -expr -stmt -cont )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump) ) (equal -type 'label)
          )
        ; Do nothing for this type of statement. Ignore case label
        ( make_fact -cont ) ; Follow continuation afterwards.
        ( make_fact -stmt ) ; Check the statement.
  )
)

```

```

(equal -type 'case)
; Tracing a case statement
(prog ( ans )
  again
  ( prt_stmt (list 'S 'C -expr -stmt ) )
  (msg N )
  (msg N T "Does the value of this case expression match the value")
  (msg N T "of the Switch test expression (y or n)? " )
  (setq ans ( call_help (read)
("The debugger is searching for the Case statement where the execution"
"continues. This will be the Case statement whose expression value"
"matches the value of the Switch test expression." ) )
  )
  ( cond ( (null ans )
    ; Repeat the prompt
    ( go again )
  )
  ( (equal ans 'y)
    ; Yes, this is the statement
    (msg N T "The trace will now continue with the labeled statement." )
    (msg N )
    ( make_fact -cont ) ; Follow continuation afterwards.
    ( make_fact -stmt ) ; Check the statement.
    ( remove 1 ) ; Found the label.
  )
  ( (equal ans 'n)
    ; No, this is not the statement
    ( make_fact -cont ) ; Follow continuation.
  )
  ( t ; Invalid input
    ( msg N N T "**** Invalid input, try again!" N )
    ( go again )
  )
  )
  )
)
)
)
(equal -type 'default)
; Tracing a default statement
( make_fact -cont ) ; Follow continuation. Ignore case statement.
)
(equal -type 'break )
; Looking for the statement where a break statement will reach.
; Drop the continuation
()
)
(equal -type 'continue)
; Trying to find the top of a loop. Drop the continuation.
()
)
(t : Bad trace type abort

```

```

        ( msg N T " *** Internal error : bad trace type : " -type N )
        ( halt )
    )
)
) ; end of rule stmt_case

; This rule handles a statement of the form
;   ( S D (stmt) )           default : stmt

(defp stmt_default
  ( goal (trace -type) )
  ( S D -stmt -cont )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump) (equal -type 'label)
            )
          ; Do nothing for this type of statement. Ignore default label
          ( make_fact -cont ) ; Follow continuation afterwards.
          ( make_fact -stmt ) ; Check the statement.
        )
    ( (equal -type 'case)
      ; Tracing a case statement
      ( make_fact -cont ) ; Follow continuation. Ignore case statement.
      ( make_fact -stmt ) ; Check the statement.
    )
    ( (equal -type 'default)
      ; Tracing a default statement
      ( make_fact -cont ) ; Follow continuation. Ignore case statement.
      ( make_fact -stmt ) ; Check the statement.
      ( remove 1 )       ; Found the default statement.
      (msg N T " default : " N )
    )
    ( (equal -type 'break )
      ; Looking for the statement where a break statement will reach.
      ; Drop the continuation
      ()
    )
    ( (equal -type 'continue)
      ; Trying to find the top of a loop. Drop the continuation.
      ()
    )
    ( t ; Bad trace type abort
      ( msg N T " *** Internal error : bad trace type : " -type N )
      ( halt )
    )
  )
) ; end of rule stmt_default

```

```

: This rule handles a statement of the form
:   ( S E (expr) )   expression statement

(defp stmt_expression
  (goal (trace -type))
  (S E -expr -cont)
  ->(remove 2)
  (cond ((or (equal -type 'loop) (equal -type 'deadlock))
    ; Do nothing for this type of statement. Check for function calls.
    (prt_stmt (list 'S 'E -expr))
    (make_fact -cont) ; Check the next statement.
    (make_fact -expr) ; Check the expression first.
  )
  ((or (equal -type 'default) (equal -type 'case)
    (equal -type 'label))
  )
  ; Tracing a default, label, or case statement
  (make_fact -cont) ; Follow continuation. Ignore statement.
  )
  ((equal -type 'coredump)
  ; Does the coredump occur for this statement?
  (prt_stmt (list 'S 'E -expr))
  (prog (ans))
  repeat
    (msg N T "Does this statement execute successfully (y or n)? ")
    (setq ans (call_help (lread)))
    ("If the statement executes without problem then answer 'y' for
    "yes. If the core dump happens when this statement is executed
    "then answer 'n' for no. If this statement calls a function and
    "the core dump is possibly occurring in the function then still
    "answer no 'n'. The statement executes successfully if the next
    "statement is eventually reached."))
  )
  (cond ((null ans)
    ; Repeat the prompt
    (go repeat)
  )
  ((equal ans 'y)
    ; This statement is OK so continue on.
    (make_fact -cont) ; Check the next statement.
  )
  ((equal ans 'n)
    ; The core dump occurs on this line.
    (fact DUMPED E N -expr)
    (make_fact -expr) ; Check the expression for function
    ; calls.
  )
  (t ; Invalid input
    (msg N T "**** Answer using 'y' or 'n' N")
  )
  )

```

```

    )
  )
)
(equal -type 'break )
; Looking for the statement where a break statement will reach.
; Drop the continuation
()
)
(equal -type 'continue)
; Trying to find the top of a loop. Drop the continuation.
()
)
(t ; Bad trace type abort
  (msg N T"*** Internal error : bad trace type : " -type N )
  (halt )
)
)
) : end of rule stmt_expression

```

```

; This rule handles a statement of the form
;   ( S L ident (stmt) ) ident : stmt

```

```

(defp stmt_labeled
  (goal (trace -type))
  (S L -ident -stmt -cont )
  ->(remove 2 )
  (cond ((equal -type 'break )
    ; Looking for the statement where a break statement will reach.
    ; Drop the continuation
    ()
  )
    ((equal -type 'label )
    ; We are looking for a label, but we know this isn't it.
    (make_fact (cons -stmt -cont) )
  )
    ((equal -type 'continue)
    ; Trying to find the top of a loop. Drop the continuation.
    ()
  )
    (t ; For anything else
    ; Print the label part. Push the statement on the continuation
    ; and continue the trace treating the statement as a normal
    ; statement.
    (prt_stmt (list 'S 'L -ident))
    (make_fact (cons -stmt -cont) )
  )
  )
) : end of rule stmt_labeled

```

```

: This rule handles a statement of the form
:   ( S I (expr) (stmt) ) IF (expr) stmt
: To handle an IF statement first check the test, then, determine if the test
: was zero or not then trace the IF body or skip it if the test was zero.

```

```

(defp stmt_unbal_if
  (goal (trace -type) )
  ( S I -test -then -cont )
  -> (remove 2)
  (cond ((or (equal -type 'loop) (equal -type 'deadlock) )
        ; Do nothing for this type of statement. Check for function calls
        ; in the test first.
        (prt_stmt (list 'S 'I -test) )
        (fact U_IF -test -then -cont)
        (make_fact -test) ; Check the expression first.
        )
        ((or (equal -type 'default) (equal -type 'case)
              (equal -type 'label)
              )
         ; Tracing a default or case statement
         (make_fact -cont) ; Follow continuation. Ignore statement.
         )
        ((equal -type 'coredump)
         ; Does the coredump occur for this statement?
         (prt_stmt (list 'S 'I -test) )
         (prog (ans)
              repeat
                (msg N T
                 "Does the test expression execute successfully (y or n)? ")
                (setq ans (call_help (read)
                                     ("If the expression executes without a problem then answer 'y' for"
                                      "yes. If the core dump happens when the expression is executed"
                                      "then answer 'n' for no. If the expression calls a function and"
                                      "the core dump is possibly occurring in the function then still"
                                      "answer no 'n'.")))
                )
                (cond ((null ans)
                       ; Repeat the prompt
                       (go repeat)
                       )
                      ((equal ans 'y)
                       ; This statement is OK so continue on. Check for
                       ; function calls first.
                       (fact U_IF -test -then -cont)
                       )
                      ((equal ans 'n)
                       ; The core dump occurs on this line.
                       (fact DUMPED I T ^ (list -test -then) )
                       (make_fact -test) ; Check the expression for function
                       ; calls.
                       )
                    )
        )

```



```

    )
    ( t : Invalid input
      ( msg N T "*** Answer using 'y' or 'n' N )
    )
  )
)
)
)
( (equal -type 'break )
; Looking for the statement where a break statement will reach.
; Drop the continuation
()
)
( (equal -type 'continue)
; Trying to find the top of a loop. Drop the continuation.
()
)
)
( t : Bad trace type abort
  ( msg N T "*** Internal error: bad trace type: " -type N )
  ( halt )
)
)
)
) ; end of rule stmt_unbal_if

```

; This rule determines if the test expression for an unbalanced IF statement evaluates to zero or not. Based on the value the trace continues with the body of the IF statement or the statement following the IF statement.

```

(defp if_branch
  ( goal (trace -type) )
  ( U_IF -test -then -cont )
  ->( remove 2 )
  (cond ( (and (equal -type 'loop) (equal In_loop 'true))
        ; We suspect an infinite loop and are inside a loop.
        ( make_fact -cont )
        ( make_fact -then )
        (msg N T "The debugger will check the body of the IF statement first" )
        (msg N T "and then continue with the first statement after the IF." N )
        )
    ( t : For anything else
      ( prog ( ans )
        again
        (msg N T
          "Is the value of the test expression false (zero) or true (non-zero)?" )
        (msg N T "Pick false or true (f or t)? " )
        ( setq ans ( call_help (read)
          ("The value of the expression evaluates to an integer value. Is the value"
            "zero (false)? If so, then enter 'f'. If the value is not zero (true)"
            "then enter 't.'.") )

```

```

)
  (cond ((null ans)
        : Repeat the prompt
        (go again)
        )
        ((equal ans 'f)
        : Yes, the test is zero so skip the if body.
        (msg N T
         "The body of the IF statement is not executed. The")
        (msg N T "statement following the body of the IF statement")
        (msg N T "will be the next statement executed." )
        (make_fact -cont)
        )
        ((equal ans 't)
        : The IF body is executed.
        (msg N T "Continue tracing with the first statement inside the"
         (msg N T "body of the IF statement." N )
         (make_fact -cont) ; After tracing the body continue on
         (make_fact -then) ; Trace the IF body first
         )
        (t ; Invalid input
         (msg N N T "**** Invalid input, try again" N )
         (go again)
         )
        )
    )
  )
)
)
) : end of rule if_branch

```

```

: This rule handles a statement of the form
: (S IE (expr) (stmt1) (stmt2)) IF (expr) stmt1 ELSE stmt2

(defp stmt_bal_if
  (goal (trace -type) )
  (S IE -test -then -else -cont)
  -> (remove 2)
  (cond ((or (equal -type 'loop) (equal -type 'deadlock) )
        : Do nothing for this type of statement. Check for function calls
        : in the test first.
        (prt_stmt (list 'S 'IE -test) )
        (fact B_IF -test -then -else -cont)
        (make_fact -test) ; Check the expression first.
        )
        ((or (equal -type 'default) (equal -type 'case)
              (equal -type 'label)
              )
        : Tracing a default,label, or case statement
        (make_fact -cont) ; Follow continuation. Ignore statement.
        )

```

```

)
( (equal -type 'coredump)
  ; Does the coredump occur for this statement?
  ( prt_stmt (list 'S 'IE -test) )
  ( prog ( ans )
    repeat
      (msg N T
        "Does the test expression execute successfully (y or n)? " )
      (setq ans ( call_help (Iread)
        ("If the expression executes without a problem then answer 'y' for"
        "yes. If the core dump happens when the expression is executed"
        "then answer 'n' for no. If the expression calls a function and"
        "the core dump is possibly occurring in the function then still"
        "answer no 'n.'" ) )
      )
      (cond ( (null ans)
        ; Repeat the prompt
        ( go repeat )
        )
        ( (equal ans 'y)
          ; This statement is OK so continue on.
          ( fact B_IF -test -then -else -cont )
          )
        ( (equal ans 'n)
          ; The core dump occurs on this line.
          ( fact DUMPED IE T ^ (list -test -then -else) )
          ( make_fact -test ) ; Check the expression for function
          ; calls.
          )
        ( t ; Invalid input
          ( msg N T "*** Answer using 'y' or 'n'" N )
          )
        )
      )
    )
  )
)
( (equal -type 'break)
  ; Looking for the statement where a break statement will reach.
  ; Drop the continuation
  ()
)
( (equal -type 'continue)
  ; Trying to find the top of a loop. Drop the continuation.
  ()
)
( t ; Bad trace type abort
  ( msg N T "*** Internal error : bad trace type : " -type N )
  ( halt )
)
)
) ; end of rule stmt_bal_if

```

; This rule determines if the test expression for a balanced IF statement
 ; evaluates to zero or not. Based on the value the trace continues with the
 ; body of the "then" portion or the "else" portion of the IF statement.

```
(defp bal_if_branch
  (goal (trace -type) )
  ( B_IF -test -then -else -cont )
  ->( remove 2 )
  (cond ( (and (equal -type 'loop) (equal ln_loop 'true) )
    : We suspect an infinite loop and are inside a loop.
    ( make_fact -cont )
    ( make_fact -else )
    ( make_fact -then )
    (msg N T "The debugger will check the body of the THEN section of the")
    (msg N T "IF statement and then the ELSE section. The search will")
    (msg N T "then continue with the statement following the IF." N )
    )
    ( t : For anything else.
      ( prog ( ans )
        again
        (msg N T
          "Is the value of the test expression zero (false) or non-zero (true)?"
          (msg N T "Pick false or true (f or t)? " )
          ( setq ans ( call_help (read)
            ("The value of the expression evaluates to an integer value. Is the value"
            "zero (false)? If so, then enter 'f'. If the value is not zero (i.e true)"
            "then enter 't'.")
          )
          ( cond ( (null ans )
            : Repeat the prompt
            ( go again )
            )
            ( (equal ans 'f)
              : Yes, the test is zero so the "else" portion.
              (msg N T "Continue tracing with the first statement inside the"
                (msg N T "'else' portion of the IF statement." N )
                (make_fact -cont )
                (make_fact -else )
                )
              )
            ( (equal ans 't)
              : The "then" portion of the IF statement is executed.
              (msg N T "Continue tracing with the first statement inside the"
                (msg N T "'then' portion of the IF statement." N )
                (make_fact -cont) : After tracing the body continue on
                (make_fact -then) : Trace the IF body first
                )
              )
            ( t : Invalid input
              ( msg N N T "**** Invalid input, try again!" N )
              ( go again )
              )
            )
          )
        )
      )
    )
  )
```

```

    )
  )
)
) ; end of rule bal_if_branch

```

```

; This rule handles a statement of the form
; ( S DW (stat) (expr) ) DO stat WHILE expr

```

```

(defp stmt_do_while
  (goal (trace -type) )
  ( S DW -body -test -cont )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          ; Determine if the program execution ever gets past this loop.
          (prt_stmt (list 'S 'DW ))
          (prog (ans )
              again
              (msg N T "Does this Do-While loop eventually exit (y or n)? ")
              (setq ans (call_help (lread)
                ("If the loop does not exit then the statement following the loop is never"
                 "executed. The question is does the loop exit every time it is entered?"
                 "If for every time the loop is entered it is also exited then answer 'y'"
                 "for yes. If there is a time when the loop is entered and does not exit"
                 "then enter 'n' for no." ) )
              )
          ( cond ( (null ans )
                  ; Repeat the prompt
                  ( go again )
                )
              ((equal ans 'y)
               ; The loop always exits so skip it.
               (msg N T "Because the loop always exits skip the body of the")
               (msg N T "loop and continue the trace with the statement")
               (msg N T "after the loop." N )
               ( make_fact -cont )
               )
              ((equal ans 'n)
               ; No the loop does not always exit so trace the body
               ; of the loop and the while test expression.
               (cond ( (equal -type 'loop)
                       ; Looking for infinite loop, save info
                       ( fact LOOP DW (-body -test) )
                       ( setq In_loop 'true )
                     )
                 (msg N T "Because the debugger suspects an infinite loop it is necessary")
                 (msg N T "to determine if the loop contains any inner loops that are" )
                 (msg N T "entered and never exit or if there are any functions that are")
               )
            )
        )
    )

```

```

(msg N T "called and never return." N )
(msg N T "The trace will search the body of the loop, ignoring branches in")
(msg N T "the execution flow and ask only about function calls and inner")
(msg N T "loops that are found. Please read along!" N )
    ( make_fact -test ) ; Trace the test if the
      ; body is ok.
    )
  ( (equal -type 'coredump)
    ; Tracing a coredump, if the body is OK then
    ; coredump must be in test.
    ( fact ADW -test -body ) ; Loop again
    ( fact TEST DW -test ) ; Handle carefully
  )
  ( (equal -type 'deadlock)
    ; Looking for deadlock.
    ( fact ADW -test -body ) ; Loop again
    ( make_fact -test ) ; Trace the test if the
      ; body is ok.
  )
)
( fact CONTINUE DW )
( fact BREAK DW )
( make_fact -body ) ; Trace the body.
)
( t ; Invalid input
  ( msg N N T "*** Invalid input, try again!" N )
  ( go again )
)
)
)
)
( (or (equal -type 'default) (equal -type 'case) )
  ; Tracing a default or case statement
  ( make_fact -cont ) ; Follow continuation. Ignore statement.
)
( (equal -type 'break )
  ; Looking for the statement where a break statement will reach.
  ; Drop the continuation
  ()
)
( (equal -type 'continue)
  ; Trying to find the top of a loop. Drop the continuation.
  ()
)
( (equal -type 'label)
  ; Tracing a labeled statement search the body first.
  ( make_fact -cont )
  ( make_fact -body )
)
( t ; Bad trace type abort

```

```

      ( msg N T " *** Internal error : bad trace type : " -type N )
      ( halt )
    )
  )
) ; end of rule stmt_do_while

```

```

; This rule handles a statement of the form
; ( S W (expr) (stat) ) WHILE (expr) stat

```

```

(defp stmt_while
  ( goal (trace -type) )
  ( S W -test -body -cont )
  ->( remove 2 )
  ( cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          : Determine if the program execution ever gets past this loop.
          ( prt_stmt (list 'S 'W -test -body) )
          ( prog ( ans )
              again
              (msg N T "Does this While loop eventually exit (y or n)? ")
              ( setq ans ( call_help (read) )
                ("If the loop does not exit then the statement following the loop is never"
                 "executed. The question is does the loop exit every time it is entered?"
                 "If for every time the loop is entered it is also exited then answer 'y'"
                 "for yes. If there is a time when the loop is entered and does not exit"
                 "then enter 'n' for no." ) )
              )
          ( cond ( (null ans)
                  : Repeat the prompt
                  ( go again )
                )
              ((equal ans 'y)
               : The loop always exits so skip it.
               (msg N T "Because the loop always exits skip the body of the")
               (msg N T "loop and continue the trace with the statement")
               (msg N T "after the loop." N )
               ( make_fact -cont )
             )
              ((equal ans 'n)
               : No the loop does not always exit so trace the body
               : of the loop and the while test expression.
               (cond ((equal -type 'loop)
                      : Looking for infinite loop, save info
                      ( fact LOOP W (-test -body) )
                      ( setq In_loop 'true )
                    )
                 (msg N T "Because the debugger suspects an infinite loop it is necessary")
                 (msg N T "to determine if the loop contains any inner loops that are")
                 (msg N T "entered and never exit or if there are any functions that are")
               )
            )
        )

```

```

(msg N T "called and never return." N )
(msg N T "The trace will search the body of the loop, ignoring branches in")
(msg N T "the execution flow and just ask about function calls and inner")
(msg N T "loops that are found. Please read along!" N )
  )
  ( t ; For any other type
    ( fact AW -test -body )
  )
)
( fact BREAK W )
( fact CONTINUE W )
( make_fact -body ); Trace the body.
(cond ( (equal -type 'coredump)
        ; Tracing a coredump, handle test first then
        ; check the body.
        ( fact TEST W -test ) ; Handle carefully
      )
      ( t ; Tracing loop or deadlock.
        ( make_fact -test ) ; Trace the test if the
          ; body is ok.
      )
    )
)
)
( t ; Invalid input
  ( msg N N T "**** Invalid input, try again" N )
  ( go again )
)
)
)
)
( (or (equal -type 'default) (equal -type 'case) )
  ; Tracing a default or case statement
  ( make_fact -cont ) ; Follow continuation. Ignore statement.
)
( (equal -type 'break )
  ; Looking for the statement where a break statement will reach.
  ; Drop the continuation
  ()
)
( (equal -type 'label)
  ; Tracing a labeled statement search the body first.
  ( make_fact -cont )
  ( make_fact -body )
)
( (equal -type 'continue)
  ; Trying to find the top of a loop. Drop the continuation.
  ()
)
( t ; Bad trace type abort
  ( msg N T " **** Internal error : bad trace type : " -type N )
)

```



```

        ( halt )
      )
    ) ; end of rule stmt_while

```

```

: This rule handles a statement of the form
:   ( S F (expr1) (expr2) (expr3) (stat) )
:   FOR (expr1 : expr2 : expr3 ) stat

```

```

(defp stmt_for
  (goal (trace -type) )
  (S F -init -test -incr -body -cont )
  -X(remove 2)
  (cond ( (or (equal -type 'loop) (equal -type 'deadlock)
              (equal -type 'coredump)
            )
          ; Determine if the program execution ever gets past this loop.
          (prt_stmt (list 'S 'F -init -test -incr -body) )
          (prog (ans)
                again
                (msg N T "Does this For loop eventually exit (y or n)? ")
                (setf ans (call_help (read)
                                     ("If the loop does not exit then the statement following the loop is not"
                                      "executed. The question is does the loop exit every time it is entered?"
                                      "If for every time the loop is entered it is also exited then answer 'y'"
                                      "for yes. If there is a time when the loop is entered and does not exit"
                                      "then enter 'n' for no." )
                                     )
                )
          (cond ( (null ans)
                  ; Repeat the prompt
                  (go again)
                )
                ((equal ans 'y)
                 ; The loop always exits so skip it.
                 (msg N T "Because the loop always exits skip the body of the")
                 (msg N T "loop and continue the trace with the statement")
                 (msg N T "after the loop." N)
                 (make_fact -cont)
                )
                ((equal ans 'n)
                 ; No the loop does not always exit so trace the body
                 ; of the loop and the For expressions.
                 (cond ( (equal -type 'loop)
                         ; Looking for infinite loop, save info
                         (fact LOOP F (-init -test -incr -body) )
                         (setf In_loop 'true)
                       )
                     (msg N T "Because the debugger suspects an infinite loop it is necessary")
                     (msg N T "to determine if the loop contains any inner loops that are")
                     (msg N T "entered and never exit or if there are any functions that are")
                   )
                )
          )

```

```

(msg N T "called and never return." N )
(msg N T "The trace will search the body of the loop, ignoring branches in")
(msg N T "the execution flow and just ask about function calls and inner")
(msg N T "loops that are found. Please read along!" N )
)
  ( t ; For any other type
    ( fact AF -init -test -incr -body )
  )
)

( fact BREAK F )
(cond ( (equal -type 'coredump)
      ; Tracing a coredump. check the For loop in
      ; pieces.
      ( fact TEST FT -test ) ; Trace test.
      ( fact TEST FN -incr ) ; Trace increment.
      ( fact CONTINUE F )
      ( make_fact -body ) ; Trace the body.
      ( fact TEST FI -init ) ; Trace initialize.
    )
  ( t ; Tracing loop or deadlock.
    (make_fact -test) ; Trace test expression.
    (make_fact -incr) ; Trace increment expression.
    ( fact CONTINUE F )
    (make_fact -body) ; Trace the body.
    (make_fact -init) ; Trace initialize expression
  )
)
)
)
( t ; Invalid input
  ( msg N N T "**** Invalid input, try again" N )
  ( go again )
)
)
)
)
)
( (or (equal -type 'default) (equal -type 'case) )
  ; Tracing a default or case statement
  ( make_fact -cont ) ; Follow continuation. Ignore statement.
)
( (equal -type 'break )
  ; Looking for the statement where a break statement will reach.
  ; Drop the continuation
  ()
)
( (equal -type 'label)
  ; Tracing a labeled statement search the body first.
  ( make_fact -cont )
  ( make_fact -body )
)
)

```

```

    ( (equal -type 'continue)
      ; Trying to find the top of a loop. Drop the continuation.
      ()
    )
    ( t ; Bad trace type abort
      ( msg N T " *** Internal error : bad trace type : " -type N )
      ( halt )
    )
  )
) ; end of rule stmt_for

```

: This rule is fired when the tracer needs to trace the test
 : expression for a loop while looking for a coredump source.

```

(defp handle_test
  ( goal (trace coredump) )
  ( TEST -type -expr )
  ->( remove 2 )
  ( prog (ans)
    repeat
      (cond ( (equal -type 'DW)
              ; Handling the test expression for a Do-While loop.
              ; Does the coredump occur for this expression?
              (msg N T "According to you the Do-While loop does not exit, that means" )
              (msg N T "the statement following the loop is never reached. You have" )
              (msg N T "indicated that the coredump did not occur for this iteration of" )
              (msg N T "the loop. If the test expression executes successfully then" )
              (msg N T "the trace goes back to the top of the loop." )
              (msg N )
              ( prt_stmt -expr )
              (msg N )
              (msg N T "Does the test expression execute successfully (y or n)? " )
              (setq ans ( call_help (read)
                ("If the core dump occurs in the test expression there is an"
                 "of the loop where the last statement in the loop executes but"
                 "the first statement in the loop is not reached." ) )
              )
            )
          ( (equal -type 'FI)
            ; Handling the initialize expression for a For loop.
            ; Does the coredump occur for this expression?
            (msg N T "The first time the For loop is entered the initialize" )
            (msg N T "expression is executed:" N )
            ( prt_stmt -expr )
            (msg N )
            (msg N T "Does the initialize expression execute successfully (y or n)? " )
            (setq ans ( call_help (read)

```

```

("If the initialize expression executes successfully then the"
 "statement preceding the For Loop executes successfully and"
 "the first statement in the loop is reached." )
)
)

(equal -type 'FN)
: Handling the increment expression for a For loop.
: Does the core dump occur for this expression?
(msg N T "After the last statement in the loop has executed the")
(msg N T "increment expression is executed." )
(msg N )
( prt_stmt -expr )
(msg N )
(msg N T "Does the increment expression execute successfully (y or n)? ")
(setq ans ( call_help (read)
("If the increment expression executes successfully then the"
 "last statement in the loop executes successfully and the"
 "first statement in the loop is reached. However, the core"
 "dump may occur in the test expression, thus preventing the"
 "first statement in the loop from being reached."
 "One way to determine which expression the core dump"
 "is occurring in is to move the increment expression to the"
 "bottom of the loop so that it is the last statement in the"
 "loop." ) )
)
)

(equal -type 'FT)
: Handling the test expression for a For loop.
: Does the core dump occur for this statement?
(msg N T "The core dump did not occur for this iteration of the loop.")
(msg N T "If the test expression executes successfully then the trace" )
(msg N T "continues with the first statement in the loop." )
(msg N )
( prt_stmt -expr )
(msg N )
(msg N T "Does the test expression execute successfully (y or n)? ")
(setq ans ( call_help (read)
("If the expression executes without a problem then answer 'y' for"
 "yes. If the core dump happens when the expression is executed"
 "then answer 'n' for no. If the expression calls a function and"
 "the core dump is possibly occurring in the function then still"
 "answer no 'n.'" ) )
)
)

(equal -type 'W)
: Handling the test expression for a While loop.
: Does the core dump occur for this expression?

```

```

(msg N T "The While test expression is: " N )
  ( prt_stmt -expr )
  (msg N )
(msg N T "Does the While test expression execute successfully (y or n)? " )
  (setq ans ( call_help (read)
    ("After the last statement in the loop is executed is the first"
     "statement in loop reached? Or after the statement preceding"
     "the loop executes is the first statement in the loop reached?" ) )
  )
)
)
; Now handle the user's reply
(cond ( (null ans)
  ; Repeat the prompt
  ( go repeat )
)
  ( (equal ans 'y )
  ; Yes, the expression executes fine.
  ; function calls first.
  (cond ( (or (equal -type 'DW) (equal -type 'FT)
    (equal -type 'FI) (equal -type 'FN )
    (equal -type 'W)
  )
  )
  ; No problem check the rest of the loop.
  )
)
)
)
  ( (equal ans 'n )
  ; The core dump occurs on this line.
  ( fact DUMP_EXP -type )
  ( make_fact -expr ) ; Check the expression for function
  ; calls.
)
  ( t ; Invalid input
  ( msg N T "*** Answer using 'y' or 'n'" N )
)
)
)
) ; end of rule handle_test

; This rule handles a statement of the form
; ( S S (expr) (stat) ) SWITCH (expr) stat
; This rule just checks the test and then tries to find which case statement
; is executed.

(defp stmt_switch
  ( goal (trace -type) )
  ( S S -test -body -cont )
  ->( remove 2 )

```

```

( cond ( (or (equal -type 'loop) (equal -type 'deadlock) )
  : Do nothing for this type of statement. Check for function calls
  ; in the test first.
  ( prt_stmt (list 'S 'S -test) )
  (cond( (equal In_loop 'true)
    : Switch is in a possible infinite loop.
    ( make_fact -cont )
    ( make_fact -body )
    (msg N T "The debugger will check the entire body of the SWITCH" )
    (msg N T "ignoring CASE and DEFAULT labels. BREAKs will also be" )
    (msg N T "ignored. After searching the body the search will continue")
    (msg N T "with the first statement after the SWITCH." N )
  )
  ( t ; otherwise
    ( fact SWITCH -test -body -cont )
  )
)
( make_fact -test ) ; Check the expression first.
)
( (equal -type 'coredump)
  ; Does the coredump occur for this statement?
  ( prt_stmt (list 'S 'S -test) )
  ( prog ( ans )
  repeat
    (msg N T
      "Does the Switch expression execute successfully (y or n)? " )
    (setq ans ( call_help (read)
      ("If the expression executes without a problem then answer 'y' for
      "yes. If the core dump happens when the expression is executed"
      "then answer 'n' for no. If the expression calls a function and"
      "the core dump is possibly occurring in the function then still"
      "answer no 'n'." ) )
    )
    (cond ( (null ans)
      : Repeat the prompt
      ( go repeat )
    )
    ( (equal ans 'y )
      : This expression is OK so continue on.
      ( fact SWITCH -test -body -cont )
    )
    ( (equal ans 'n )
      : The core dump occurs on this line.
      ( fact DUMPED S T ^ (list -test -body) )
      ( make_fact -test ) ; Check the expression for function
      ; calls.
    )
  )
  ( t ; Invalid input
    ( msg N T "*** Answer using 'y' or 'n'" N )
  )
)

```

```

    )
  )
  ( (or (equal -type 'default) (equal -type 'case) )
    ; Tracing a default or case statement
    ( make_fact -cont ) ; Follow continuation. Ignore statement.
  )
  ( (equal -type 'break )
    ; Looking for the statement where a break statement will reach.
    ; Drop the continuation
    ()
  )
  ( (equal -type 'label)
    ; Tracing a labeled statement search the body first.
    ( make_fact -cont )
    ( make_fact -body )
  )
  ( (equal -type 'continue)
    ; Trying to find the top of a loop. Drop the continuation.
    ()
  )
  ( t ; Bad trace type abort
    ( msg N T " *** Internal error : bad trace type : " -type N )
    ( halt )
  )
)
) ; end of rule stmt_switch

```

; This rule determines the value of a switch expression for a Switch statement.
 ; Based on the value, the trace searches for the case label that matches. The
 ; trace will then continue with the labeled statement. If no appropriate
 ; case label is found then the trace continues with the default labeled
 ; statement. If no default statement is found then the trace continues with
 ; the statement following the Switch statement.

```

(defp switch_branch
  ( goal (trace -type) )
  ( SWITCH -test -body -cont )
  ->( remove 2 )
  ( prog ( ans )
    again
    (msg N T "The Switch expression is: ( " )
    (prt_expr -test )
    (msg " " ) )
    (msg N T "Determine the value of the Switch expression. Does that value")
    (msg N T "match a Case expression value in the switch (y or n)? " )
    ( setq ans ( call_help (lread)
      ("The expression value is compared to constants in all 'case' statements."

```

```

"and the execution will jump to the statement whose 'case' constant"
"matches the expression value. Determine the value of the expression"
"and see if it matches the value of one of the case constants." ) )
)
(cond ( (null ans )
      ; Repeat the prompt
      ( go again )
    )
  ( (equal ans 'y)
    ; Yes, the value matches a case label.
    ; Search the switch body and look for the case label.
    ( goal trace case )
    ( make_fact -cont ) ; Handle after body.
    ( fact BREAK S )
    ( make_fact -body ) ; Hope it is a hlock list (B ... )
  )
  ( (equal ans 'n)
    ; No, the value does not match a case label.
    ( prog ( reply )
      repeat
      (msg N T "Is there a default statement in the Switch body (y or n)? " )
      ( setq reply ( call_help (lread)
        ("You are right the debugger could check, hut things will go faster"
        "if you just tell it whether there is a statement with a 'default'"
        "label on it." ) )
      )
      ( cond ( (null reply )
            ; Repeat the prompt
            ( go repeat )
          )
        ( (equal reply 'y)
          ; Yes, there is a default statement
          (msg N T "The trace will continue at the default statement." N )
          ( make_fact -cont ) ; Handle after body.
          ( fact BREAK S )
          ( goal trace default )
          ( make_fact -body )
        )
        ( (equal reply 'n)
          ; No, there is not a default statement
          (msg N T "The trace will continue with the statement following the" )
          (msg N T "Switch statement." N )
          ( make_fact -cont )
        )
        ( t ; Invalid input
          ( msg N N T "**** Answer using 'y' or 'n'" N )
          ( go repeat )
        )
      )
    )
  )
)

```



```

    )
  )
  ( t ; Invalid input
    ( msg N T "*** Answer using 'y' or 'n' N )
    ( go again )
  )
)
)
) ; end of rule switch_branch

```

```

: This rule handles a statement of the form
:   ( S G ident ) GOTO ident
: Note: the tracer currently can not detect the case where a goto inside
:       a loop is reached and the goto jumps out of the loop. There of
:       course should not be legal given the way we trace things.

```

```

(defp stmt_goto
  (goal (trace -type) )
  ( S G -ident -cont )
  ->( remove 2 )
  ( cond ( (equal -type 'loop)
            ; Find out what happens.
            ( prt_stmt (list 'S 'G -ident) )
            (msg N T "HEY, what is this? A goto statement! You're programming in C")
            (msg N T "here not Fortran, not Basic." N )
            (prog (ans)
                  again
                  (msg N T "Is the statement jumped to farther down in the function")
                  (msg N T "or is it closer to the top of the function (t or d)? ")
                  ( setq ans ( call_help (lread)
                                     ("If the labeled statement jumped to comes after the Goto"
                                      "statement in the function then enter 'd'. If the labeled"
                                      "statement is found before the Goto statement then enter 't:'.")
                                     )
                  )
                  ( cond ( ( null ans )
                           ; Repeat the prompt
                           ( go again )
                         )
                    ( (equal 't ans)
                      ; Possible loop here.
                      (prog (ans)
                            again
                            (msg N T "Because this is a backwards jump the debugger needs to know")
                            (msg N T "if the execution ever moves past this point in the function?")
                            (msg N T "Is any statement that follows this place in the function" )
                            (msg N T "ever executed (y or n)? ")
                            (setq ans ( call_help (lread)

```

```

("If the execution never moves that this Goto statement the then"
 "infinite loop may be caused by this backwards branch. If no"
 "statement in the function that comes after this statement is"
 "ever reached then type 'n', else type 'y.'" )
)
  ( cond ( (null ans )
           ; Repeat the prompt
           ( go again )
         )
        ( (equal ans 'y)
           ; Yes, we do reach another statement.
           ; Continue the trace then.
           ( fact LABEL -ident ) ; Remember the label
           ( goal trace label )
           ( goal get_cont cur_func ) ; Start at top.
         )
        ( (equal ans 'n)
           ; No, we don't reach another statement.
           ; Debugger can't handle back trace.
           (msg N T "There is a possibility that this Goto statement is the source")
           (msg N T "of the infinite loop. Check that there are no other loops or")
           (msg N T "function calls between the labeled statement and this loop")
           (msg N T "that do not return. If you find one then investigate it using")
           (msg N T "debugger, but unfortunately, the debugger can not trace a")
           (msg N T "backwards jump while tracing an infinite loop. Sorry!" N)
           ( goal run again )
         )
        ( t ; Invalid input
          (msg N N T "*** Invalid input, try again!" N )
          ( go again )
        )
      )
    )
    )
  )
)
  )
)
  ( (equal 'd ans )
    ; Jump to the labeled statement.
    ( make_fact -cont ) ; Search for the label
    ( fact LABEL -ident ) ; Remember the label
    ( goal trace label )
  )
  ( t ; Invalid input
    ( msg N N T "*** Invalid input, try again!" N )
    ( go again )
  )
)
)
)
  )
)
  ( (or (equal -type 'deadlock) (equal -type 'coredump)
    )
    ( prt_stmt (list 'S 'G -ident) )
  )
)

```

```

(msg N T "HEY, what is this? A goto statement! You're programming in C")
(msg N T "here not Fortran, not Basic." N)
  (prog ( ans )
    again
    (msg N T "Is the statement jumped to farther down in the function")
    (msg N T "or is it closer to the top of the function (t or d)? ")
      (setq ans ( call_help (read)
        ("If the labeled statement jumped to comes after the Goto"
         "statement in the function then enter 'd'. If the labeled"
         "statement is found before the Goto statement then enter 't'.")
        )
        (cond ( (null ans )
          ; Repeat the prompt
          ( go again )
          )
          ( (equal 't ans )
            ; Jump to the label at the top.
            ( fact LABEL -ident ) ; Remember the label
            ( goal trace label )
            ( goal get_cont cur_func ) ; Start at top.
            )
          ( (equal 'd ans )
            ; Jump to the label
            ( make_fact -cont ) ; Search for the label
            ( fact LABEL -ident ) ; Remember the label
            ( goal trace label )
            )
          ( t ; Bad answer
            (msg N N T "**** Invalid input, try again!" N )
            ( go again )
          )
        )
      )
    )
  )
)
(or (equal -type 'case) (equal -type 'default) )
; Tracing a case or default statement
()
)
(equal -type 'break)
; Tracing a break statement. Ignore the continuation.
()
)
(equal -type 'continue)
; Trying to find the top of a loop. Drop the continuation.
()
)
(equal -type 'label)
; Tracing a labeled statement search the body first.
( make_fact -cont )
)

```

```

      ( t ; Bad trace type abort
        ( msg N T " *** Internal error : bad trace type : " -type N )
        ( halt )
      )
    )
  ) ; end of rule stmt_goto

: This rule handles a statement of the form
:   ( S B )      BREAK

(defp stmt_break
  ( goal (trace -type) )
  ( S B -cont )
  ->( remove 2 )
  ( cond ( (or (equal -type 'case) ( equal -type 'default)
              )
            ; Searching for a case or default label. Continue on.
            ( make_fact -cont )
            )
        ( (or (equal -type 'coredump) (equal -type 'deadlock)
              (equal -type 'loop)
              )
          ; Drop the cont. Somewhere out there is a BREAK fact. Find it!
          ( prt_stmt (list 'S 'B) )
          ( cond ( (equal In_loop 'false)
                  ; If not in a suspected infinite loop then break.
                  ( goal trace break )
                )
              )
          )
        ( (equal -type 'break)
          ; Already hunting for a break jump. Ignore this one.
          ( )
          )
        ( (equal -type 'label)
          ; Tracing a labeled statement search the body first.
          ( make_fact -cont )
          )
        ( (equal -type 'continue)
          ; Trying to find the top of a loop. Drop the continuation.
          ( )
          )
        ( t ; Invalid input
          ( msg N " *** Internal error : Bad trace type : " -type N )
          ( halt )
        )
      )
  ) ; end of rule stmt_break

```

```

: This rule handles a statement of the form
:   ( S N )      null statement
: There is nothing to be done for a NULL statement.

```

```

(defp stmt_null
  (goal (trace -type) )
  ( S N -cont )
  ->(remove 2)
  (cond ((equal -type 'break)
        ; Attempting to find where a break jumps to. Ignore continuation.
        ()
        )
        ((equal -type 'continue)
        ; Trying to find the top of a loop. Drop the continuation.
        ()
        )
        (t ; For any other type of trace.
        (prt_stmt (list 'S 'N -cont) )
        (make_fact -cont)
        )
        )
  )
) ; end of stmt_null

```

```

: This rule handles a statement of the form
:   ( S CN )     CONTINUE

```

```

(defp stmt_continue
  (goal (trace -type) )
  ( S CN -cont )
  ->(remove 2)
  (cond ((or (equal -type 'loop) (equal -type 'deadlock)
            (equal -type 'coredump)
            )
        ; Show that we have reached a continue statement and then see
        ; where it goes.
        (prt_stmt (list 'S 'CN) )
        (cond ((equal In_loop 'false)
              ; If not in a suspected infinite loop then jump
              (goal trace continue)
              )
              )
        )
        ((equal -type 'case)
        ; Tracing a case statement. Ignore it and continue
        (make_fact -cont)
        )
        ((equal -type 'default)
        ; Tracing a default statement
        (make_fact -cont)
        )
        )
  )
)

```

```

)
(equal -type 'break )
; Tracing a break, so just ignore this continuation
()
)
(equal -type 'label)
; Tracing a labeled statement search the body first.
(make_fact -cont )
)
(equal -type 'continue)
; Trying to find the top of a loop. Drop the continuation.
()
)
(t ; Bad trace type abort
(msg N T " *** Internal error : bad trace type : " -type N )
(halt )
)
)
) ; end of rule stmt_continue

```

```

; This rule handles a statement of the form
; ( S R (expr) )
; return expr, if expr is empty then no value to return.

```

```

(defp stmt_return
  (goal (trace -type) )
  (S R -expr -cont )
  ->(remove 2 )
  (cond ( (or (equal -type 'loop) (equal -type 'deadlock) )
    ; Do nothing for this type of statement. Check for function calls.
    (prt_stmt (list 'S 'R -expr) )
    (cond ( (and (equal -type 'loop) (equal In_loop 'true) )
      ; Suspect an infinite loop so ignore return.
      (make_fact -cont )
    )
      (t ; Otherwise return
        (fact RETURN )
      )
    )
  )
  (make_fact -expr ) ; Check the expression first.
)
(equal -type 'coredump)
; Does the coredump occur for this statement?
(prt_stmt (list 'S 'R -expr) )
(prog ( ans )
repeat
(msg N T "If this statement executes successfully then the" )
(msg N T "execution returns to the function that called the" )
(msg N T "current function." N )

```

```

(msg N T "Does this statement execute successfully (y or n)? ")
(setq ans ( call_help (read)
("If the statement executes without a problem then answer 'y' for"
"yes. If the core dump happens when this statement is executed"
"then answer 'n' for no. If this statement calls a function and"
"the core dump is possibly occurring in the function then still"
"answer no 'n'." ))
)
(cond ( (null ans)
; Repeat the prompt
(go repeat)
)
(equal ans 'y)
; This statement is OK so continue on.
(fact RETURN)
)
(equal ans 'n)
; The core dump occurs on this line.
(fact DUMPED R N -expr)
(make_fact -expr) ; Check the expression for function
; calls.
)
(t ; Invalid input
(msg N T "**** Answer using 'y' or 'n'" N)
)
)
)
)
(equal -type 'case)
; Tracing a case statement. Ignore this statement.
()
)
(equal -type 'default)
; Tracing a default statement. Ignore this statement.
()
)
(equal -type 'break)
; Attempting to determine where a break jumps to. Ignore this.
()
)
(equal -type 'label)
; Tracing a labeled statement search the body first.
(make_fact -cont)
)
(equal -type 'continue)
; Trying to find the top of a loop. Drop the continuation.
()
)
(t ; Bad trace type abort
(msg N T " **** Internal error : bad trace type : " -type N)
)
)
)
)

```

```

        ( halt )
      )
    )
  ) ; end of rule stmt_return

```

```

.....
..... Handling a trace
.....
.....

```

```

: This rule is fired when we are tracing a program and need to start
: on the continuation (function body) for a function and the continuation
: for the function is loaded.

```

```

(defp get_func_cont
  ( goal (trace -type) )
  ( goal (get_cont -func) )
  ( ( DUMPED -s -t -what ) with ( equal -type 'coredump) )
  ( ~ ( DUMP_EXP -t ) with ( equal -type 'coredump) )
  ( ~ ( LOOP -a -b ) with ( equal -type 'loop ) )
  ( FUNC -func -file )
  ( FBOD -func -cont )
  ->( remove 2 )
  ( make_fact -cont )
  ( setq cur_func -func ) ; Remember what function we are examining
  ( setq In_loop 'false ) ; Not inside a loop looking for infinite loop.
  ( cond ( (not (equal -type 'label))
            ( msg N T "Entering the code for function : " -func N )
          )
  )
) ; end of rule get_func_cont

```

```

: IF we want to trace a function and we suspect a line that causes a core
: dump and we are tracing a core dump
: THEN delete the fact that we found a suspect statement.
: What this means if we found a line that core dumps and it contains a
: call to a function that did not return then the core dump does not occur
: on the line but in the function.

```

```

(defp not_dump_site
  ( goal (trace coredump) )
  ( goal (get_cont -func) )
  ( DUMPED -s -t -what )
  ->( remove 3 )
) ; end of rule not_dump_site

```



```

; IF we want to trace a function and we suspect a line that causes a core
; dump and we are tracing a core dump
; THEN delete the fact that we found a suspect expression.
; What this means if we found an expression that core dumps and it contains a
; call to a function that did not return then the core dump does not occur
; in the expression but in the function.

```

```

(defp not_dump_expr
  ( goal (trace coredump) )
  ( goal (get_cont -func) )
  ( DUMP_EXP -s )
  ->( remove 3 )
) ; end of rule not_dump_expr

```

```

; IF we want to trace a function and we suspect an infinite loop and the
; loop contains a function that does not return
; THEN delete the fact that we found a suspect loop.
; What this means if we found a loop and it contains a
; call to a function that did not return then the loop is not an infinite
; loop.

```

```

(defp not_loop_site
  ( goal (trace loop) )
  ( goal (get_cont -func) )
  ( LOOP -type -body )
  ->( remove 3 )
) ; end of rule not_loop_site

```

```

; This rule is fired when we are tracing a program and need to start
; on the continuation (function body) for a function and the continuation
; for the function is not loaded yet.

```

```

(defp need_func_cont
  ( goal (trace -type) )
  ( goal (get_cont -func) )
  ( FUNC -func -file )
  ( ~ (FBOD -func -cont) ) ; Function body continuation is not loaded
  ->( load (file_name -file) )
) ; end of rule need_func_cont

```

```

; This rule is fired when we are tracing a program and need to start on
; the continuation (function body) for a function and we don't know where
; to find the source code information.

```

```

(defp missing_func_def
  ( goal (trace -type) )
  ( goal (get_cont -func) )

```

```

( ^ (FUNC -func -file) )
->( remove 2 )
( prog (temp)
  (setq temp (sys_func -func)) ; If this a system function
    ( fact NO_FUNC -func ^ (car temp) ^ (cadr temp) )
  )
) ; end of missing_func_def

```

; If a Break statement is detect then the tracer will look for a BREAK fact.
; This rule is fired when a BREAK fact is found.

```

(defp found_break
  ( goal (trace break) )
  ( BREAK -type )
->( remove 2 )
  ( remove 1 )
  (cond ( (equal -type 'S)
    : Break in a Switch statement
    : Breaking out of a Switch is OK. Do nothing.
  )
  ( ( or (equal -type 'W) (equal -type 'DW) (equal -type 'F) )
    ; Break in a While, Do-While, or For statement
    (msg N)
    (msg N T "The debugger is currently tracing a " )
    (cond ( (equal -type 'W)
      (msg "While" )
    )
      ( (equal -type 'DW)
        (msg "Do-While" )
      )
      ( (equal -type 'F)
        (msg "For" )
      )
    )
  )
  (msg " loop. You have" )
  (msg N T "stated that the loop does not exit, yet the trace has reached")
  (msg N T "a Break statement. The execution of this statement will cause")
  (msg N T "the execution to jump out of the loop." N )
  (msg N T "Unfortunately, it seems the trace has not correctly followed" )
  (msg N T "the execution of the program. Please, determine for yourself" )
  (msg N T "whether the loop really does exit and try the trace again." N)
  ( goal run again ) : Abort the trace.
  )
  ( t ; Don't know )
  (msg N " **** Bad BREAK type : " -type N )
  )
)

```

```
) ; end of rule found_break
```

; If we encounter a BREAK fact while not searching for one then drop it.

```
(defp hit_break
  (goal (trace -type)
        (BREAK -what)
        test (not (equal -type 'break))
        -->(remove 2)
  ) ; end of rule hit_break
```

; If a continue statement is detect then the tracer will look for a continue fact. This rule is fired when a continue fact is found.

```
(defp found_continue
  (goal (trace continue)
        (CONTINUE -type)
        -->(remove 2)
        (remove 1)
        (msg N T "The continue statement causes the execution to jump to")
        (cond ((equal -type 'W)
              ; In a While loop.
              (msg N T "the top of the loop. Trace continues with the test" )
              (msg N T "expression." N )
              )
              ((equal -type 'DW)
              ; In a Do-While loop.
              (msg N T "the bottom of the loop. Trace continues with the test" )
              (msg N T "expression." N )
              )
              ((equal -type 'F)
              ; In a For loop.
              (msg N T "the top of the loop. Trace continues with the increment" )
              (msg N T "and test expressions." N )
              )
              (t ; Invalid type
              (msg N N T "**** Invalid loop type for continue : " -type N )
              )
        )
  ) ; end of rule found_continue
```

; If we encounter a CONTINUE fact while not searching for one then drop it.

```
(defp hit_continue
  (goal (trace -type))
```

```

( CONTINUE -what )
test ( not (equal -type 'continue) )
-× remove 2 )
) ; end of rule hit_continue

```

: If a Return statement is reached and no problem with its expression is found then we have a problem and this rule is fired.

```

(defp hit_return
  ( goal (trace -type) )
  ( RETURN )
  -× remove 2 )
  (cond ( (or (equal -type 'loop) (equal -type 'deadlock)
             (equal -type 'coredump)
           )
        ; Reached a Return when we should not have done so.
        (msg N T "The debugger is currently tracing function " cur_func ".")
        (msg N T "You have indicated that the function does not exit, yet the")
        (msg N T "trace has reached a Return statement that seems to execute")
        (msg N T "successfully. The execution of this statement will cause")
        (msg N T "the execution to jump out of the function." N )
        (msg N T "Unfortunately, it seems the trace has not correctly followed" )
        (msg N T "the execution of the program. Please, determine for yourself" )
        (msg N T "whether the function really does exit and try the trace again." N)
        ( goal run again ) ; Abort the trace.
      )
    ( (or (equal -type 'case) (equal -type 'default)
          (equal -type 'break)
        )
      ; Error, we should be here with this type of trace.
      ( msg N T " *** Internal error : bad trace for return : "
        -type N )
      ( halt )
    )
    ( t ; For any other type
      ( msg N T " *** Internal error : bad trace type : " -type N )
      ( halt )
    )
  )
) ; end of rule hit_return

```

: This rule is fired when the tracer is attempting to find a labeled statement and one has been found.

```

(defp hit_label
  ( goal (trace label) )
  ( S L -ident -stmt -cont )
)

```

```

( LABEL -label )
test( equal -label -ident )
-×( remove 3 )
( remove 1 ) ; The rule stmt_labeled will fire now.
) ; end of rule hit_label

```

: This rule is fired when the tracer needs to do another iteration of a
: Do-While Loop.

```

(defp loop_do_while
  ( goal (trace -type) )
  ( ADW -test -body )
-×( remove 2 )
  (msg N T "Continue trace with the first statement in the loop." N)
  (cond ( (equal -type 'coredump)
          ; Loop the Do-While loop to the top.
          ( ADW -test -body )
          ( fact TEST DW -test )
          ( fact CONTINUE DW )
          ( fact BREAK DW )
          ( make_fact -body )
        )
        ( (equal -type 'deadlock)
          ; Looking for deadlock.
          ( fact ADW -test -body ) ; Loop again
          ( make_fact -test ) ; Trace the test if the
                                ; body is ok.
          ( fact CONTINUE DW )
          ( fact BREAK DW )
          ( make_fact -body )
        )
        ( t ; Should not be here for any other type
          (msg N N T "**** Internal error : bad type type : " -type N )
        )
      )
  )
) ; end of rule loop_while

```

: This rule is fired when the tracer needs to do another iteration of a
: While Loop.

```

(defp loop_while
  ( goal (trace -type) )
  ( AW -test -body )
-×( remove 2 )
  (msg N T "Continue trace with the While loop test expression." N)
  (cond ( (equal -type 'coredump)
          ; Loop the Do-While loop to the top.

```

```

    ( fact AW -test -body )
    ( fact BREAK W )
    ( fact CONTINUE W )
    ( make_fact -body ) ; Trace the body.
    ( fact TEST W -test ) ; Handle carefully
  )
  ( (equal -type 'deadlock)
    : Looking for deadlock.
    ( fact AW -test -body )
    ( fact BREAK W )
    ( fact CONTINUE W )
    ( make_fact -body ) ; Trace the body.
    ( make_fact -test ) ; Trace the test if the body is ok.
  )
  ( t ; Should not be here for any other type
    (msg N N T "**** Internal error : bad type type : " -type N )
  )
)
) ; end of rule loop_while

; This rule is fired when the tracer needs to do another iteration of a
; For Loop.

(defp loop_for
  ( goal (trace -type) )
  ( AF-init -test -incr -body )
  ->( remove 2 )
  (msg N T "Continue trace with the first statement in the loop." N)
  (cond ( (equal -type 'coredump)
    : Loop the Do-While loop to the top.
    ( fact AF -init -test -incr -body )
    ( fact BREAK F )
    ( fact TEST FT -test ) ; Trace test.
    ( fact TEST FN -incr ) ; Trace increment.
    ( fact CONTINUE F )
    ( make_fact -body ) ; Trace the body.
  )
  ( (equal -type 'deadlock)
    : Looking for deadlock.
    ( fact AF -init -test -incr -body )
    ( fact BREAK F )
    (make_fact -test) ; Trace test expression.
    (make_fact -incr) ; Trace increment expression.
    ( fact CONTINUE F )
    (make_fact -body) ; Trace the body.
  )
  ( t ; Should not be here for any other type
    (msg N N T "**** Internal error : bad type type : " -type N )
  )
)
)

```

```
) ; end of rule loop_for
```

```
.....  
:  
: Function : make_fact  
:  
: Purpose : This function takes a continuation and generates a fact.  
:  
: Arguments : cont - The current continuation (may be an expression  
: list)  
:  
: Returns : nothing  
:
```

```
(defun make_fact ( cont )  
  ( cond ( not (null cont) )  
    ; The continuation is not empty so create the fact  
    ( prog ( result ; The fact list  
            temp ; first list in the cont  
            type ; B for block, S for statement, E for expression  
            )  
      (setq temp (car cont) )  
      (setq result ( list 'fact ) )  
      ( cond ( (or (listp temp) (equal temp 'E) (equal temp 'S)  
                )  
        ; We have a list therefore it is a continuation  
        (cond ( (or (equal temp 'E) (equal temp 'S)  
                    )  
              ; Expression list so use all of cont  
              (setq temp cont)  
              (setq cont nil ) ; No additional stmts.  
            )  
          )  
        (prog ()  
          (setq type (car temp) )  
          repeat  
            (setq result ( cons (car temp) result ) )  
            (setq temp (cdr temp) )  
            (cond ( ( not (null temp) )  
                  ; Keep taking it apart  
                  ( go repeat )  
                )  
              )  
            )  
          )  
        ( (equal temp 'B )  
          ; We have a block header
```

```

        (setq type 'B)
        (setq result (cons temp result))
    )
    (t ; Temp is not a list, that means it was a block
     (msg N)
     (msg N T
      "**** Internal error, make_func bad value " temp N)
    )
)

; Build the fact now
(cond ((equal type 'E)
      ; The cont passed was not a real continuation but
      ; an expression list.
      (eval (reverse result))
    )
      ((equal type 'S)
      ; The cont passed is a real continuation
      (eval (reverse (cons (cdr cont) result)))
    )
      ((equal type 'B)
      ; The cont passed is for a block.
      (eval (reverse (cons (car (cdr cont)) result)))
    )
      (t ; Internal error, we don't know the type
       (msg N)
       (msg N T "**** Internal error : unknown type " type
        N)
      )
    )
)
)
)
)
) : end of make_fact

(installp 'exp_dot_oper      'exp_mem_ptr_oper  'exp_post_add_oper
'exp_post_sub_oper  'exp_ptr_val_oper  'exp_address_oper
'exp_pre_add_oper   'exp_pre_sub_oper   'exp_negate_oper
'exp_not_oper       'exp_ones_comp_oper 'exp_mult_oper
'exp_div_oper       'exp_modulo_oper    'exp_plus_oper
'exp_minus_oper     'exp_left_shift_oper 'exp_right_shift_oper
'exp_less_than_oper 'exp_lte_oper       'exp_gte_oper
'exp_greater_than_oper 'exp_equal_oper   'exp_not_equal_oper
'exp_bit_and_oper    'exp_bit_excl_or_oper 'exp_bit_or_oper
'exp_l_and_oper      'exp_l_or_oper      'exp_assign_oper
'exp_add_asgn_oper   'exp_minus_asgn_oper 'exp_mult_asgn_oper
'exp_div_asgn_oper   'exp_remain_asgn_oper 'exp_rs_asgn_oper
'exp_ls_asgn_oper    'exp_bit_and_asgn_oper 'exp_bit_e_or_asgn_oper
'exp_bit_or_asgn_oper

```


'exp_cast	'exp_sizeof	'exp_if
'exp_identifier	'exp_number	'exp_string
'exp_char	'exp_parens	'exp_subscript
'exp_func_call	'exp_comma	
'stmt_block	'stmt_case	'stmt_default
'stmt_expression	'stmt_labeled	'stmt_unbal_if
'stmt_bal_if	'stmt_do_while	'stmt_while
'stmt_for	'stmt_switch	'stmt_goto
'stmt_break	'stmt_null	'stmt_continue
'stmt_return		
'get_func_cont	'need_func_cont	'not_dump_site
'if_exp_path	'func_call	'if_branch
'bal_if_branch	'not_loop_site	'handle_test
'switch_branch	'found_break	'hit_break
'hit_return	'hit_label	'hit_continue
'loop_do_while	'loop_while	'loop_for
'found_continue	'not_dump_expr	'missing_func_def

)

Appendix D Scanner Source Code

This appendix contains the source code for the scanner. The source code files are:

```
flags.def  
tokens.def  
tree_info.def  
  
clean_up.c  
code_scan.c  
error.c  
facts.c  
lex_tabs.c  
node_names.c  
tree_han.c  
  
c_gram.y  
c_lex.l
```

The main function is contained in the file "code_scan.c". The file "c_gram.y" is a YACC input file and the file "c_lex.l" is a LEX input file.

```
/******  
*  
* File : flags.def  
*  
* Created : 04/15/87 by : Eric Byrne  
*  
* Purpose : This include file contains the definitions of various  
* constants used by the scanner tool.  
*  
******/
```

```
/* Used to indicate if an integer constant is a long or an int */
```

```
#define IS_LONG 1  
#define IS_INT 2
```

```
/* Define true and false values. */  
#define TRUE 1  
#define FALSE 0
```

```
/* Returned by some routines to indicate if any errors were detected.  
** These values are matched to the values yyparse() expects yylex() to  
** return. */
```

```
#define OK 0  
#define FAILURE 1
```

```
/* Define NULL and a NULL pointer. */
```

```
#define NULL 0  
#define NULL_CHAR (char *)NULL
```

```
/******  
*  
* File : tokens.def  
*  
* Created : 04/12/87 by : Eric Byrne  
*  
* Purpose : This include file defines tokens for the C programming  
* language. These tokens are the terminal symbols  
* recognized and returned by the lexical analyzer.  
*  
******/
```

```
/* Group 1 : These tokens represent values or classes of values. For
```

```

**      example <identifier> is a token that represents any valid
**      name.
*/

#define t_IDENTIFIER  101  /* Name of an identifier      */
#define t_TYPEDEF_NAME 102  /* Name of a user defined data type */
#define t_DEC_INT     103  /* A value of type integer      */
#define t_HEX_INT     104  /* A value in hex                */
#define t_OCT_INT     105  /* A value in octal              */
#define t_FLOAT       106  /* A value of type float         */
#define t_STRING      107  /* A character string            */
#define t_CHAR        108  /* A single character value     */

/* Group 2 : These are keywords for the C language. */

/* Data Types */
#define k_CHAR        201
#define k_DOUBLE      202
#define k_ENUM        203
#define k_FLOAT       204
#define k_INT         205
#define k_LONG        206
#define k_SHORT       207
#define k_STRUCT      208
#define k_UNION       209
#define k_UNSIGNED    210
#define k_VOID        211

/* Misc */
#define k_SIZEOF      215
#define k_TYPEDEF     216

/* Storage Classes */
#define k_AUTO        220
#define k_EXTERN      221
#define k_REGISTER    222
#define k_STATIC      223

/* Statements */
#define k_BREAK       225
#define k_CASE        226
#define k_CONTINUE    227
#define k_DEFAULT     228
#define k_DO          229
#define k_ELSE        230
#define k_FOR         231
#define k_GOTO        232
#define k_IF          233
#define k_RETURN      234

```

```
#define k_SWITCH      235
#define k_WHILE       236
```

```
/* Group 3 : These are the operators and expressions for C */
```

```
/* Arithmetic */
#define t_ADD_ONE     301 /* ++ */
#define t_DIVIDE      302 /* / */
#define t_MINUS       303 /* - */
#define t_REMAINDER   304 /* % */
#define t_PLUS        305 /* + */
#define t_STAR        306 /* * */
#define t_SUB_ONE     307 /* - */

/* Assignment */
#define t_ADD_ASSIGN  310 /* += */
#define t_ASSIGN      311 /* = */
#define t_B_AND_ASSIGN 312 /* &= */
#define t_B_E_OR_ASSIGN 313 /* ^= */
#define t_B_OR_ASSIGN 314 /* |= */
#define t_DIV_ASSIGN  315 /* /= */
#define t_L_ASSIGN    316 /* <= */
#define t_MINUS_ASSIGN 317 /* -= */
#define t_MULT_ASSIGN 318 /* *= */
#define t_REMAIN_ASSIGN 319 /* %= */
#define t_RS_ASSIGN   320 /* >= */

/* Relational */
#define t_EQUAL       325 /* == */
#define t_GREATERTHAN 326 /* > */
#define t_GTE        327 /* >= */
#define t_LESSTHAN   328 /* < */
#define t_LTE        329 /* <= */
#define t_NOT_EQUAL  330 /* != */

/* Boolean */
#define t_L_AND       335 /* && */
#define t_L_OR        336 /* || */
#define t_NOT         337 /* ! */

/* Bitwise */
#define t_AMPERSAND   340 /* & */
#define t_B_EXCLUSIVE_OR 341 /* ^ */
#define t_B_OR        342 /* | */
#define t_LEFT_SHIFT  343 /* << */
#define t_ONES_COMP   344 /* ~ */
#define t_RIGHT_SHIFT 345 /* >> */

/* Address */
```

```

/* Due to overloading of symbols & and * are the addressing
** operators, but they also occur in other operator classes. */

/* Array */
#define t_LEFT_SQ_BRAC 350 /* [ */
#define t_RIGHT_SQ_BRAC 351 /* ] */

/* Structure/Union */
#define t_DOT 355 /* . */
#define t_MEM_POINTER 356 /* -> */

/* Miscellaneous */
#define t_COLON 360 /* : */
#define t_COMMA 361 /* , */
#define t_COND_THEN 362 /* ? */
#define t_LEFT_CUR_BRAC 363 /* { */
#define t_LEFT_PAREN 364 /* ( */
#define t_RIGHT_CUR_BRAC 365 /* } */
#define t_RIGHT_PAREN 366 /* ) */
#define t_SEMICOLON 367 /* ; */

/*****
*
* File : tree_info.def
*
* Created : 04/26/87 by : Eric Byrne
*
* Purpose : This include file contains definitions and
* declarations related to the parse tree.
*
*****/

/* The parse tree structure. */
struct pnode {
    struct pnode *right ; /* Pointer to right subtree */
    struct pnode *left ; /* Pointer to left subtree */
    int type ; /* The node type */
    char *value ; /* Pointer to any associated
** node value.
*/
};

#define NULL_PNODE (struct pnode *)0

/* For holding information about numbers. */
struct num_node {

```

```

char *num ; /* Points to the character string holding the
            ** number. Memory is allocated in c_lex.1 */
int size ; /* Indicates if the number was entered using
            ** the long mark. i.e. 123L */
};

/* For holding information about include files used in a file */
struct inc_files {
char *name ; /* Points to the name of an include file */
int line ; /* The line number in the source file where the
            ** #include statement is located. */
struct inc_files *next ; /* Pointer to the next node */
};

/* Node type definitions : Used in structure pNode for field "type" */

/* Terminal nodes */
#define n_BREAK 1 /* No value in tree->val */
#define n_CHAR 3 /* tree->val holds char value */
#define n_CONTINUE 5 /* No value in tree->val */
#define n_DEC_INT 7 /* tree->val points to num_node */
#define n_DEFAULT 9 /* No value in tree->val */
#define n_FLOAT 11 /* tree->val points to num_node */
#define n_HEX_INT 13 /* tree->val points to num_node */
#define n_IDENT 15 /* tree->val points to string */
#define n_NULL 17 /* No value in tree->val */
#define n_OCT_INT 19 /* tree->val points to num_node */
#define n_STRING 21 /* tree->val points to string */
#define n_TYPEDEF 22 /* not used yet. */

/* Non-terminal nodes */
#define n_ASSIGN 30 /* R = exp L = val,
                    ** Val = assignment type */
#define n_BLOCK 33 /* R = statement list, L=NULL compound
                    ** statement block */
#define n_CASE 36 /* R = case exp . L = NULL */
#define n_CAST_EXP 39 /* R = exp L = type */
#define n_DO_STMT 42 /* R = stmt, L = cond exp */
#define n_ELSE 45 /* R = then exp, L = else exp */
#define n_ELSE_EXP 48 /* R = then exp L = else exp ? */
#define n_E_STMT 51 /* R = list exp, L = NULL */
#define n_EXP_SEPAR 54 /* R = list or first exp, L = exp */
#define n_FOR 57 /* R = init exp, L = FOR_EXIT */
#define n_FOR_EXIT 60 /* R = exit exp, L = FOR_INCR */
#define n_FOR_INCR 63 /* R = increment exp, L = stmt */
#define n_FUNC_CALL 66 /* R = func_name L = params */
#define n_FUNC_DEF 69 /* R = func_name, L = body */
#define n_GOTO 72 /* R = label node, L = NULL */

```

```

#define n_IF_BAL          75 /* R = cond, L = else node */
#define n_IF_EXP         78 /* R = cond, L = ELSE_EXP node ? : */
#define n_IF_UNBAL       81 /* R = cond, L = then exp, no else */
#define n_LABEL          84 /* R = label node, L = stmt */
#define n_OPERATOR       87 /* R = Rexp L = Lexp, Val has oper */
#define n_PAREN_EXP      91 /* R = exp in ( ) */
#define n_RETURN         94 /* R = exp, L = NULL */
#define n_SIZEOF_EXP     97 /* R = exp to take size of, L = NULL */
#define n_STATEMENT     100 /* R = single statmt, L = statmt
** list, statements following R */
#define n_SUBSCRIPT     103 /* R = array L = subscript_exp */
#define n_SWITCH        106 /* R = exp, L = stmts */
#define n_TOP_DECL      109 /* R is single decl, L is decl list */
#define n_WHILE_STMT    112 /* R = cond exp, L = stmt */

```

```

/*****
*
*   File : clean_up.c
*
*   Created : 10/17/87                by : Eric Byrne
*
*   Purpose : This file contains routines that are used to clean
*             global pointers after a file has been scanned and
*             the facts generated. All allocated memory is also
*             freed up after the facts are generated for a file.
*
*   Routines :
*             clean_up()
*             clean_inc()
*             clean_tree()
*
*****/

```

```

#include "flags.def"
#include "tree_info.def"

```

```

/***** EXTERNAL VARIABLES *****/

```

```

extern struct inc_files *G_includes ; /* Defined in c_lex.l */
extern struct pnode *G_root ; /* Defined in tree_han.c */

```

```

/*****

```

```

Routine : clean_up()

```


Purpose : This routine directs the activity of cleaning up after
a file has been scanned and facts generated.

Arguments : None

*****/

```
clean_up()
{
    clean_inc();
    clean_tree( G_root );

    G_root = NULL_PNODE ;
} /* end of clean_up() */
```

Routine : clean_inc()

Purpose : This routine frees up all memory allocated to build
the linked list of include file names.

Arguments : None

*****/

```
static clean_inc()
{
    register struct inc_files *ptr ; /* Used to move down the linked list */
    register struct inc_files *nptr ; /* Used to point to next node */

    ptr = G_includes ;

    while ( ptr != (struct inc_files *)NULL )
    {
        nptr = ptr->next ;
        free( ptr->name ) ; /* Free the include name buffer */
        free( ptr ) ; /* Free the entire node */
        ptr = nptr ;
    } ;

    G_includes = (struct inc_files *)NULL ; /* Set the list to empty */
} /* end of clean_inc() */

*****
```

Routine : `clean_tree()`

Purpose : This routine is used to travel the parse tree and free all the memory allocated for each node. When this routine is done the entire parse tree will have been deallocated.

Arguments : `node` - The current or top tree node to work with.

*****/

```
static clean_tree( node )
struct pnode *node ;
{

    struct num_node *tnum; /* Used to point to a number node */

    if ( node == NULL_PNODE )
        return ; /* Nothing else to do about it! */

    switch( node->type )
    {

        case n_ASSIGN :
        case n_BLOCK :
        case n_BREAK :
        case n_CASE :
        case n_CAST_EXP :
        case n_CHAR :
        case n_CONTINUE :
        case n_DEFAULT :
        case n_DO_STMT :
        case n_E_STMT :
        case n_ELSE :
        case n_ELSE_EXP :
        case n_EXP_SEPAR :
        case n_FOR :
        case n_FOR_EXIT :
        case n_FOR_INCR :
        case n_FUNC_CALL :
        case n_FUNC_DEF :
        case n_GOTO :
        case n_IDENT :
        case n_IF_BAL :
        case n_IF_EXP :
        case n_IF_UNBAL :
        case n_LABEL :
        case n_OPERATOR :
        case n_NULL :
        case n_PAREN_EXP :
```

```

case n_RETURN      :
case n_SIZEOF_EXP  :
case n_STATEMENT   :
case n_STRING      :
case n_SUBSCRIPT   :
case n_SWITCH      :
case n_TOP_DECL    :
case n_WHILE_STMT  :

        if ( node->right != NULL_PNODE )
            clean_tree( node->right );

        if ( node->left != NULL_PNODE )
            clean_tree( node->left );

        if ( node->value != NULL_CHAR )
            free( node->value );

        free( node );
        node = NULL_PNODE ;
        return ;
        break ;

case n_DEC_INT     :
case n_FLOAT       :
case n_HEX_INT     :
case n_OCT_INT     :
        /* These are terminal nodes */
        tnum = (struct num_node *)node->value ;
        free( tnum->num ) ;
        free( tnum ) ;

        free( node->value ) ;

        free( node ) ;
        node = NULL_PNODE ;
        return ;
        break ;

default           :
        /* Found a node type that we don't have a case
        ** statement for yet. */

        err_prt(" *** Error : don't know how to clean node type = %s0,
        node_name( node->type ) ) ;

        free( node ) ;
        node = NULL_PNODE ;

} ; /* end of switch */

```

```
} /* end of clean_tree() */
```

```
/******  
*  
* File : code_scan.c  
*  
* Created : 04/14/87          by : Eric Byrne  
*  
* Purpose : This file contains the main routine for the scanner.  
*           The scanner is a tool used to parse a C source file  
*           and generate a set of facts about the C source file  
*           that the yaps KBPD tool can use.  
*  
* Routines : main()  
*            make_name()  
*            read_file()  
*            scan_file()  
*  
*****/
```

```
#include <stdio.h>  
#include "flags.def"  
#include "tree_info.def"
```

```
/****** EXTERNAL VARIABLES *****/
```

```
extern int      errno      ; /* The C system error variable */  
extern struct pnode *G_root ; /* Defined in tree_han.info */  
extern struct inc_files *G_includes ; /* Defined in c_lex.l */  
extern int      yylineno  ; /* Defined in lex.yy.c */
```

```
/****** GLOBAL VARIABLES *****/
```

```
FILE *G_fp      ; /* Pointer to the source file to be scanned */  
char  G_fname[255] ; /* Holds the name of the file being scanned */  
FILE *G_pif     ; /* Pointer to program information file */
```

```
/****** STATIC VARIABLES *****/
```

```
static int S_display ; /* If true then display the parse tree */
```

```

main( argc, argv )
int  argc ;          /* Argument count          */
char *argv[] ;      /* Argument list          */
{

    char prog_name[80] ; /* Holds name of program being scanned */
    char prog_info[90] ; /* Name of program info file.          */

    /* Initialize these flags.          */
    S_display = FALSE ; /* Don't show the parse tree by default */
    G_root = NULL_PNODE ;
    G_includes = (struct inc_files *)NULL ;

    argc-- ; /* Move past the scanner name          */
    argv++ ;
    if ( (*argv)[1] == 'p' )
    { /* The first parameter is the program name          */
        --argc ; /* Move past the -p          */
        argv++ ;
        strcpy ( prog_name, *argv ) ;

        /* Create the program information file.          */
        make_name( prog_name, prog_info, ".fcr" ) ;
        G_pif = fopen( prog_info, "w" ) ;
        if ( G_pif == (FILE *)NULL )
        {
            err_prt(
                " *** Error : Unable to create program information file.0);
            exit(1);
        }
    }
    else /* The program name was not passed first          */
    {
        err_prt(
            " *** Error : The -p parameter MUST be first on the command line.0);
        exit(1); /* Just quit          */
    }

    while ( --argc > 0 )
    {

        argv++ ;

        switch( (int)(*argv)[0] )
        {
            case '-': /* Command line argument          */
                switch( (int)(*argv)[1] )
                {
                    case 'f': /* Source files listed in a file          */
                        argv++ ;
                }
            }
        }
    }
}

```

```

    argc--;
    if ( read_file( *argv ) == FAILURE )
        exit(1); /* Quit due to err */
    break ;

case 'l': /* Source files listed on line */
    while ( --argc > 0 )
    {
        argv++;
        if ( scan_file( *argv, TRUE ) ==
            FAILURE )
            exit(1); /* Quit due to err*/
    };
    break ;

case 'w': /* Display the parse tree */
    S_display = TRUE;
    break ;

default : /* Illegal argument */
    err_prt(
    " *** Error: Illegal command line argument ->%c<-0,
      argv[1] );
    };
    break ;

default : /* C source code file name */
    err_prt(
    " *** Error: Illegal command line argument ->%s<-0,
      *argv );
    exit(1); /* Just quit */

}; /* end of switch */

}; /* end of while */

/* Close the program information file. */
fclose( G_pif );

} /* end of main() */

/*****

```

Routine : make_name

Purpose : One trick that the scanner does several times is to take the name of a file and change the postfix on it (i.e facts.c to facts.fct). This routine performs this simple transformation.

Arguments : from - A buffer containing the old file name.
 into - A buffer which we will fill with the new name.
 postfix - The new postfix to use. This string should
 start with a '.'.

```

*****/

make_name( from, into, postfix )
char *from ;
char *into ;
char *postfix ;
{

    char *ptr ; /* Pointer into the original file name */

    extern char *strchr() ; /* System function */

    strcpy( into, from ) ;

    ptr = strchr( into, '.' ) ; /* Find the last . in the name */
    if ( ptr == NULL_CHAR )
        /* There was no . in the name so concatenate the postfix */
        strcat( into, postfix ) ;
    else
    { /* There was a . in the name */

        ptr[0] = ' ' ; /* Puts a NULL at the last . */
        strcat( into, postfix ) ;
    }
} /* end of make_name() */

```

Function : read_file

Purpose : If the "-f" parameter is passed to the program then we need to read the file containing the list of source code files. This function handles the reading of the list file and calls scan_file() to handle the scanning of the source files.

Arguments : name - The name of the list file.

Returns : FAILURE - If an error is detected.
 OK - If no errors are detected.

*****/

```
read_file( name )
```

```

char *name ;
{
    register int len ; /* Length of source file name + newline */
    register FILE *lst ; /* Pointer to the list file */
    char sfile[255] ; /* The name of a source file. */

    errno = 0 ;

    if ( name == NULL_CHAR )
    { /* Seem to be missing the list file name */
        err_prt(
            " *** Error : The -f parameter must be followed by a file name.0);
            return( FAILURE );
        }

    /* Open up the list file */
    lst = fopen( name, "r" );
    if ( lst == (FILE *)NULL )
    {
        err_prt(" *** Error : Unable to open file %s : system error = %d0,
            name, errno );
        return( FAILURE );
    }

    /* Read the name of each source file from the list file and scan it */
    while ( !feof( lst ) )
    {
        if ( fgets( sfile, 255, lst ) != NULL_CHAR )
        {
            len = strlen( sfile ); /* Get rid of the newline after the */
            sfile[len-1] = NULL ; /* file name in the buffer. */
            if ( scan_file( sfile, TRUE ) == FAILURE )
                return( FAILURE );
        }
        else
        {
            if ( !feof( lst ) )
            { /* Read error detected. */
                err_prt(
                    " *** Error : error reading from file %s : system error = %d0,
                    name, errno );
                fclose( lst );
                return( FAILURE );
            }
        }
    }

    /* Wrap up and return. Everything is OK. */
    fclose( lst );
    return( OK );
}

```



```

} /* end of read_file() */

/*****

Function : scan_file

Purpose : This function directs the scanning of a single file.

Arguments : name - The name of the file to scan.
            yacc_lex - This is a flag. If TRUE then file names
                        ending in ".y" or ".l" are checked to see
                        if they are input files for yacc or lex.

Returns : OK - If the file is scanned successfully.
          FAILURE - If an error is detected while scanning the file.

*****/

scan_file( name, yacc_lex )
char *name ;
int yacc_lex ;
{

    int err ; /* Flag used to mark the occurrence of an error */
    int num ; /* Scratch area */
    char buf[255] ; /* A working area */
    char oname[255] ; /* Name of the preprocessor output file */
    register char *ptr ; /* Scratch pointer */
    char wname[255] ; /* Another working area for names */

    extern char *rindex() ; /* System function */

    errno = 0 ;

    strcpy( wname, name ) ;

    /* Step 1 : Are we worried about yacc or lex input source files? */
    ptr = rindex( name, '.' ) ;
    if ( ( yacc_lex == TRUE ) && ( ptr != NULL_CHAR ) )
    {
        /* Check the last two characters in the file name for .y or .l */
        if ( strcmp( ptr, ".y" ) == 0 )
        { /* We have a possible yacc input file. */
            do
            {
                printf( " Is the file %s used as input to yacc (y or n)? ",
                        wname ) ;
                gets( buf ) ;
            } while ( buf[0] != 'y' && buf[0] != 'Y' &&
                    buf[0] != 'n' && buf[0] != 'N' ) ;
        }
    }
}

```

```

if ( ( buf[0] == 'y' ) || ( buf[0] == 'Y' ) )
{ /* Yes, it is yacc input, so yacc it and scan the result */
  sprintf( buf, "yacc %s0, wname );
  if ( system ( buf ) != 0 )
  {
    err_prt(" *** Error : unable to yacc the file '%s'.0.
            wname );
    return( FAILURE );
  }
  /* Yacc puts "# line filename" in the y.tab.c file and
  ** that screws up the include file skipping done in the
  ** lexical analyzer. So delete those from y.tab.c */
  sprintf( buf, "sed '/^# line/d' y.tab.c > khkfgra2.q " );
  if ( system ( buf ) != 0 )
  {
    err_prt(
      " *** Error : deletings '# line' from y.tab.c.0);
    return( FAILURE );
  }
  sprintf( buf, "mv khkfgra2.q y.tab.c" );
  if ( system ( buf ) != 0 )
  {
    err_prt(
      " *** Error : deletings '# line' from y.tab.c.0);
    return( FAILURE );
  }
  /* From now on we work with y.tab.c */
  strcpy( wname, "y.tab.c" );
}
}

if ( strcmp( ptr, ".l" ) == 0 )
{ /* We have a possible lex input file. */
  do
  {
    printf(" Is the file %s used as input to lex (y or n)? ",
           wname );
    gets( buf );
  } while ( buf[0] != 'y' && buf[0] != 'Y' &&
           buf[0] != 'n' && buf[0] != 'N' );

  if ( ( buf[0] == 'y' ) || ( buf[0] == 'Y' ) )
  { /* Yes, it is lex input, so lex it and scan the result */
    sprintf( buf, "lex %s0, wname );
    if ( system( buf ) != 0 )
    {
      err_prt(" *** Error : unable to lex the file '%s'.0.
              wname );
      return( FAILURE );
    }
  }
}

```

```

        }
        else
            strcpy( wname, "lex.yy.c" );
    }
}

/* Step 2 : Run the file through the preprocessor          */
make_name( wname, oname, ".cpz" );

/* The -E says just run the preprocessor and -C says leave comments */
sprintf( buf, "cc -E -C %s >%s0, wname, oname );
printf( " Preprocessing the file : %s0, wname );
if ( system( buf ) != 0 )
{
    err_prt( " *** Error : unable to preprocess '%s'.0, wname );
    return( FAILURE );
}

/* Step 3 : open the file to be scanned.                */
G_fp = fopen( oname, "r" );
if ( G_fp == (FILE *)NULL )
{
    err_prt( " *** Error : Unable to open file '%s' : system error = %d0,
            name, errno );
    return( FAILURE );
}

/* Step 4 : Parse through the file.                    */
err = OK ;
strcpy( G_fname, wname );
yylineno = 1 ;
printf( " Scanning file : %s0, wname );
if ( yyparse() != OK )
{
    err_prt( " *** Error : Unable to successfully parse the file.0 );
    err = FAILURE ;
}

/* Step 5 : Close and drop the preprocess file before moving on. */
fclose( G_fp );
sprintf( buf, "rm %s0, oname );
if ( system( buf ) != 0 )
{
    err_prt( " *** Error : unable to drop the temp file '%s'.0, oname );
    return( FAILURE );
}

if ( err == FAILURE )

```

```

        return( FAILURE ); /* Parse failed, but we waited to close the file*/

/* Step 6 : Do we display the parse tree.          */
if ( S_display == TRUE )
{
    num = 1 ;
    printf( " ROOT : %8X0, G_root ) ;
    tree_walker( G_root, &num ) ;
}

/* Step 7 : Generate the facts and store them      */
if ( fact_gen( name ) == FAILURE )
    return( FAILURE ) ;

/* Step 8 : Need to free up the memory allocated for the tree */
clean_up() ;

} /* end of scan_file() */

```

```

/*****
*
*   File : error.c
*
*   Created : 04/14/87           by : Eric Byrne
*
*   Purpose : This file contains routines for handling error messages.
*             This way errors are only printed from one location in
*             the program.
*
*   Routines :
*             err_prt()
*             yyerror()
*
*****/

```

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

```

/*****

```

```

Routine : err_prt()

```

```

Purpose : This routine is passed all the parameters that are
normally passed to printf. This routine just passes
these parameters on to printf.

```

```

Arguments : format - The format parameter for printf.

```

```

arg1 - First parameter for the format.
arg2 - Second " "
arg3 - Third " "
arg4 - Fourth " "
arg5 - Fifth " "

```

```

*****/

```

```

err_prt( format, arg1, arg2, arg3, arg4, arg5 )

```

```

char *format ;
char *arg1 ;
char *arg2 ;
char *arg3 ;
char *arg4 ;
char *arg5 ;
{

```

```

    printf( format, arg1, arg2, arg3, arg4, arg5 ) ;

```

```

} /* end of err_prt() */

```

```

/*****

```

```

Routine : yyerror()

```

```

Purpose : The yacc produced parser requires that the user supply
          an error reporting function called yyerror(). This is
          it.

```

```

Arguments : mess - The yacc produced error message.

```

```

*****/

```

```

yyerror( mess )

```

```

char *mess ;
{

```

```

    extern int yylineno ;    /* Defined in the lex produced file. */

```

```

    err_prt( " *** Error at line : %d, %s0, yylineno, mess ) ;

```

```

} /* end of yyerror() */

```

```

/*****

```

```

*

```

```

* File : facts.c
*
* Created : 4/30/87 by : Eric Byrne
*
* Purpose : This file contains the routines that are used to
* generate the facts for the knowledge base about the
* program source code.
*
* Routines :
* fact_gen()
* fact_inc()
* fact_walker()
*
*****/

```

```

#include "flags.def"
#include "tree_info.def"

#include <stdio.h>

```

```

/***** EXTERNAL VARIABLES *****/

```

```

extern int errno ; /* A system supplied variable */
extern struct inc_files *G_includes ; /* Defined in c_lex.l */
extern FILE *G_pif ; /* Defined in code_scan.c */
extern struct pnode *G_root ; /* Defined in tree_han.c */

```

```

/***** EXTERNAL FUNCTIONS *****/

```

```

extern char *malloc() ; /* System defined function. */
extern char *node_name() ; /* Defined in node_names.c */

```

```

/***** STATIC VARIABLES *****/

```

```

static char S_file[90] ; /* Holds the name of the C source file. */

```

```

/*****

```

```

Function : fact_gen()

```

```

Purpose : This function is the head of the fact generating action.
From here the recursive routine fact_walker() is called
to transverse the parse tree and generate facts for the
knowledge base.

```

Arguments : source - The name of the C source file that was just
 parsed.

Returns : FAILURE - If a serious error is detected.
 OK - If it all works fine.

*****/

```
fact_gen( source )
char *source ;
{

    int i      : /* Scratch */
    FILE *out  : /* Pointer to the facts output file */
    char name[90] : /* The name of the fact file. */
    char temp[100] : /* Scratch */

    /* First, build the name of the parse tree fact file. The
    ** postfix .fct denotes a fact file with the same "first" name
    ** as the source file. */

    strcpy( S_file, source ); /* Save the file name */
    make_name( source, name, ".fct" );

    /* Now open the new facts file. */
    errno = 0 ;
    out = fopen( name, "w+" );
    if ( out == (FILE *)NULL )
    { /* Unable to open the file */
        err_prt(
            " *** Error : unable to create the file : %s : system error = %d\n",
            name, errno );
        return( FAILURE ); /* Punt */
    }

    /* Generate a fact that indicates the source file is part of the
    ** program */

    fprintf( G_pif, "( fact SRC %s )0, S_file );

    /* Print the list of include files used in the source file. */
    fact_inc( out );

    fact_walker( G_root, out, FALSE );

    /* Finally, close the fact file. */
    fclose( out );

    return( OK );
}
```

```

} /* end of fact_gen() */

/*****

Routine : fact_inc

Purpose : If a source file uses include files then the list of
include files referenced in the source file is converted
into a set of facts in this routine.

The fact for an include file reference is
( fact INC inc_file_name at_line source_file_name )

Arguments : fp - The file pointer to the facts output file.

*****/

static fact_inc( fp )
FILE *fp;
{
    register struct inc_files *ptr; /* Used to move down the linked list */

    ptr = G_includes;

    while ( ptr != (struct inc_files *)NULL )
    {
        fprintf( fp, "( fact INC %s %d %s )0, ptr->name, ptr->line,
                S_file );
        ptr = ptr->next;
    };
} /* end of fact_inc() */

/*****

Routine : fact_walker()

Purpose : This routine is used to travel the parse tree and
generate facts for the knowledge base for the source
file.

Arguments : node - The current or top tree node to work with.
fp - The file pointer to the facts output file.
is_e_list - TRUE if we are tracing a comma expression list.

*****/

static fact_walker( node, fp, is_e_list )
struct pnode *node ;

```



```

FILE      *fp      ;
int       is_e_list ;
{

char      val ; /* Scratch variable */
struct num_node *tnum ; /* Used to point to a number node */

if ( node == NULL_PNODE )
    return ; /* Nothing else to do about it! */

/* The action we take to generate a fact depends on the type of
** tree node we were passed. */

switch( node->type )
{
case n_ASSIGN :
    /* An assignment expression. The right subtree is the
    ** expression that is evaluated and the left subtree
    ** is the variable ( or address calculation ) where the
    ** value is to be stored. */
    /* ( E (expr1) op (expr2) ) */

    fprintf( fp, "(E " ) ;
    /* Travel the left subtree : the left exp who gets the val */
    fact_walker( node->left, fp, FALSE ) ;
    fprintf( fp, "%s", node->value ) ;

    /* Travel the right subtree first : the evaluated exp */
    fact_walker( node->right, fp, FALSE ) ;
    fprintf( fp, ")0 ) ) ;
    break ;

case n_BLOCK :
    /* The right subnode points to the declaration and
    ** statement list for the block. The left subtree is
    ** empty. */
    /* ( B (body) ) */

    fprintf( fp, "(B ( " ) ;
    fact_walker( node->right, fp, FALSE ) ;
    fprintf( fp, ")0 ) ) ;
    break ;

case n_BREAK :
    /* A break statement. This will be a terminal tree node. */
    fprintf( fp, "(S B)0 ) ) ;
    break ;

case n_CAST_EXP :

```

```

/* A cast expression. The left pointer is the cast
** expression itself. The right pointer is the subtree
** for the expression being cast. */
/* ( E CT (cast) (expr) ) */

fprintf( fp, "(E CT ( " );
fact_walker( node->left, fp, FALSE ); /* Cast expression */
fprintf( fp, " ) );
fact_walker( node->right, fp, FALSE ); /* Expr to be cast */
fprintf( fp, " ) );
break ;

case n_CHAR :
/* This is a character constant. They appear only in
** expressions and are terminal nodes in the parse tree.
** A character constant can be 'A' or '0. */
/* ( E C char ) */

fprintf( fp, "(E C %s)0, node->value );
break ;

case n_CONTINUE :
/* A continue statement. This will be a terminal tree node. */
fprintf( fp, "(S CN)0 );
break ;

case n_DEC_INT :
case n_FLOAT :
case n_HEX_INT :
case n_OCT_INT :
/* These are numeric constants. They appear only in
** expressions and are terminal nodes in the parse tree. */
/* ( E N number type size ) */

switch( node->type )
{
case n_DEC_INT : val = 'I' ; break ;
case n_FLOAT : val = 'F' ; break ;
case n_HEX_INT : val = 'H' ; break ;
case n_OCT_INT : val = 'O' ; break ;
}
tnum = (struct num_node *)node->value ;
fprintf( fp, "(E N %s %c %s)", tnum->num, val,
(tnum->size == IS_LONG) ? "L" : "D" );
break ;

case n_DO_STMT :
/* A DO - WHILE statement node. The right subtree is the
** body of the loop and the left subtree is the condition
** expression in the while clause. */

```

```

/* ( S DW (body) (expr) ) */

fprintf( fp, "S DW " );
/* Travel the statement subtree */
fact_walker( node->right, fp, FALSE );
/* Travel the expression subtree */
fact_walker( node->left, fp, FALSE );
fprintf( fp, " ) 0 );
break ;

case n_ELSE :
/* An ELSE node. This is part of a balanced IF statement.
** The right subtree is the body of the "then" section of
** the IF statement. The left subtree is the body of the
** "else" section. */

fact_walker( node->right, fp, FALSE ); /* Do the then body */
fact_walker( node->left, fp, FALSE ); /* Do the else body */
break ;

case n_ELSE_EXP :
/* This node holds the "then" and "else" expressions for a
** conditional expression. See n_IF_EXP. The right pointer
** is the "then" subtree and the left pointer is the "else"
** subtree. */

fact_walker( node->right, fp, FALSE );
fact_walker( node->left, fp, FALSE );
fprintf( fp, " ) 0 );
break ;

case n_EXP_SEPAR :
/* This node indicates a set of expressions separated by
** commas. The right subtree is a list of EXP_SEPAR nodes
** ending in a terminal node ( such as an operation node).
** The left subtree is an expression. */
/* ( E SP (expr1) (expr2) ) */

/* Travel to the end of the list at the right and then
** start generating facts. */

if ( node->right->type != n_EXP_SEPAR )
{ /* That is the first expression in the list. */
  fprintf( fp, "E SP ( " );
}
fact_walker( node->right, fp, TRUE );

/* Now travel the expression associated on the left */
fact_walker( node->left, fp, TRUE );

```

```

if ( is_e_list == FALSE )
    fprintf( fp, " ) ) ; /* Is the top of the list */

break ;

case n_E_STMT :
    /* An expression statement. The right subtree points to
    ** an expression tree ( i.e assignment statement, sole
    ** function call, i++ statement, etc ) and the left subtree
    ** is NULL. */
    /* ( S E (expr) ) */

    fprintf( fp, "( S E " ) ;
    /* Travel the expression subtree */
    fact_walker( node->right, fp, FALSE ) ;
    fprintf( fp, " ) ) ;
    break ;

case n_FOR :
    /* A For statement. This is a root node for a For subtree.
    ** The right pointer is the init expression. The left
    ** pointer is the remainder of the for expression and
    ** statement body. See n_FOR_EXIT, n_FOR_INCR. */
    /* ( S F (E1) (E2) (E3) (S) ) */

    fprintf( fp, "( S F " ) ;
    fact_walker( node->right, fp, FALSE ) ; /* The init expr */
    fact_walker( node->left, fp, FALSE ) ; /* Rest of statement */
    break ;

case n_FOR_EXIT :
    /* This holds most of the structure of a For statement.
    ** The right pointer is the exit expression for the For
    ** and the left pointer is a n_FOR_INCR node. */

    fact_walker( node->right, fp, FALSE ) ;
    fact_walker( node->left, fp, FALSE ) ;
    break ;

case n_FOR_INCR :
    /* This holds the increment expression for a For statement
    ** in the right subtree. The left pointer is the body of
    ** the For statement. */

    fact_walker( node->right, fp, FALSE ) ;
    fact_walker( node->left, fp, FALSE ) ;
    fprintf( fp, " ) ) ;
    break ;

case n_FUNC_CALL :

```

```

/* A function call node. The right subnode is the function
** name and the left subtree is the parameter list. */
/* ( E F (name) (params) ) */

fprintf( fp, "(E F " );
fact_walker( node->right, fp, FALSE ); /* Func name */
fact_walker( node->left, fp, FALSE ); /* Parameters */
fprintf( fp, ")0 );
break ;

case n_FUNC_DEF :
/* This is a function declaration node. Generate a function
** defined fact. The function name is on the right subtree
** and the left subtree is the function body. */
/* ( fact FBOD name body ) */

/* Record the function defined fact in the program
** information file. */
fprintf( G_pif, "( fact FUNC %s %s )0, node->right->value,
S_file );

/* Now handle the fact file information */
fprintf( fp, "( fact FBOD %s ", node->right->value );
fact_walker( node->left, fp, FALSE );
fprintf( fp, ")0 );
break ;

case n_GOTO :
/* A goto statement. The right pointer should hold
** the label name to jump to. */
/* ( S G label ) */

fprintf( fp, "(S G " );
if ( node->right->type == n_IDENT )
{ /* Just grab the identifier */
fprintf( fp, "%s )0, node->right->value );
}
else
{ /* How strange! It should have been an identifier */
err_prt(
" *** Error : goto statement has bad label node, node = %s0,
node_name( node->right->type ) );
}
break ;

case n_IDENT :
/* An identifier node. */
/* ( E I ident ) */

fprintf( fp, "(E I %s)", node->value );

```

```

break ;

case n_IF_BAL :
/* A balanced IF statement. ( That is an IF statement with
** an ELSE clause. ) The right subtree node is the condition
** expression. The left subtree node is an ELSE node. */
/* ( S IE (expr) (S1) (S2) ) */

fprintf( fp, "(S IE " );
/* Travel the expression body first. */
fact_walker( node->right, fp, FALSE );

/* Travel the ELSE node next. This covers the "then" and
** "else" bodies. */

fact_walker( node->left, fp, FALSE );
fprintf( fp, ")0 );
break ;

case n_IF_EXP :
/* A conditional expression, i.e. "exp1 ? exp2 : exp3".
** The right pointer is the exp1 subtree. The left pointer
** points to a node n_ELSE_EXP that has the rest of the
** subtree. */
/* ( E IF (E1) (E2) (E3) ) */

fprintf( fp, "(E IF " );
fact_walker( node->right, fp, FALSE );
fact_walker( node->left, fp, FALSE );
break ;

case n_IF_UNBAL :
/* An unbalanced IF statement. ( That is an IF statement with
** no ELSE clause. ) The right subtree node is the condition
** expression. The left subtree node is body of the IF. */
/* ( S I (expr) (S) ) */

fprintf( fp, "(S I " );

/* Travel the expression body first. */
fact_walker( node->right, fp, FALSE );

/* Travel the body of the IF statement now. */
fact_walker( node->left, fp, FALSE );
fprintf( fp, ")0 );
break ;

case n_LABEL :
/* A labelled statement. May be a simple label, or a case
** or default label. Right pointer points to the label

```

```

** subtree and the left pointer points to the statement. */
/* ( S L ident (S) ) simple label
** ( S C (expr) (S) ) case label
** ( S D (S) ) default label */

fprintf( fp, "(S " );

switch( node->right->type )
{
case n_IDENT : /* A simple label */
    fprintf( fp, "L %s ", node->right->value);
    break ;
case n_CASE : /* A case label */
    fprintf( fp, "C " );
    fact_walker( node->right->right, fp, FALSE);
    break ;
case n_DEFAULT: /* A default label */
    fprintf( fp, "D " );
    break ;
}

/* Handle the labelled statement */
fact_walker( node->left, fp, FALSE ) ;
fprintf( fp, ")0 ) ;
break ;

case n_NULL :
/* A null statement. We include just so we know it was in
** the source code. This will be a terminal tree node. */
fprintf( fp, "(S N)0 ) ;
break ;

case n_OPERATOR :
/* An operator node. These only appear in expressions.
** The right pointer is the subtree for the expression that
** will be evaluated first. The left pointer is the
** is the expression that will be evaluated second. Either
** pointer may be NULL depending on the arity of the operator
** and whether it is a prefix or postfix operator. */
/* ( E (expr1) op (expr2) )
** ( E (expr) op )
** ( E op (expr) ) */

fprintf( fp, "(E " );
if ( node->right == NULL_PNODE || node->left == NULL_PNODE )
{ /* It is an unary operator. */
    if ( node->right != NULL_PNODE )
    { /* It is a prefix operator. */
        fprintf( fp, "%s ", node->value ) ;
        fact_walker( node->right, fp, FALSE ) ;
    }
}

```

```

    }
    else /* It is a postfix operator.          */
    {
        fact_walker( node->left, fp, FALSE );
        fprintf( fp, "%s ", node->value );
    }
}
else /* It is an operator of arity two.      */
{
    fact_walker( node->right, fp, FALSE );
    fprintf( fp, "%s ", node->value );
    fact_walker( node->left, fp, FALSE );
}
fprintf( fp, "%0 ) );
break ;

case n_PAREN_EXP :
/* This node indicates an expression enclosed by '(' and ')'.
** The enclosed expression is on the right subtree.          */
/* ( E P (expr) )                                           */

fprintf( fp, "(E P " );
fact_walker( node->right, fp, FALSE );
fprintf( fp, ")0 ) );
break ;

case n_RETURN :
/* A Return statement. The right subtree points to
** an expression tree ( i.e assignment statement, sole
** function call, i++ statement, etc ) and the left subtree
** is NULL.                                                 */
/* ( S R (expr) )                                           */

fprintf( fp, "(S R " );
/* Travel the expression subtree                               */
fact_walker( node->right, fp, FALSE );
fprintf( fp, ")0 ) );
break ;

case n_SIZEOF_EXP :
/* A sizeof expression for the C "sizeof" operator. The
** right pointer is the expression or type whose size is
** needed.                                                 */
/* ( E SZ (expr) )                                           */

fprintf( fp, "(E SZ " );
fact_walker( node->right, fp, FALSE );
fprintf( fp, ")" );
break ;

```



```

case n_STATEMENT :
/* The statements for a block of code are stored in the
** tree with the first statement at the bottom of the tree
** and the last statement in the block at the top of the
** tree. So travel to the bottom and then work our way up
** generating statement facts. */

fact_walker( node->left, fp, FALSE );
fact_walker( node->right, fp, FALSE );
break ;

case n_STRING :
/* This is a string constant. They appear only in
** expressions and are terminal nodes in the parse tree. */
/* ( E S string ) */

fprintf( fp, "(E S %s)\0", node->value );
break ;

case n_SUBSCRIPT :
/* An array subscript expression. The right subtree is
** the expression that is to be subscripted. This
** can be an array name or an address calculation. The
** left subtree is the actual subscript expression. */
/* ( E A (expr1) (expr2) ) */

fprintf( fp, "(E A " );

/* Travel the array address expression first. */
fact_walker( node->right, fp, FALSE );

/* Travel the subscript expression tree now. */
fact_walker( node->left, fp, FALSE );
fprintf( fp, ")\0 );
break ;

case n_SWITCH :
/* A switch statement. The right pointer is the expression.
** The left pointer is the switch statement body. */
/* ( S S (E) (S) ) */

fprintf( fp, "(S S " );

/* Handle the switch expression. */
fact_walker( node->right, fp, FALSE );

/* Handle the statement body. */
fact_walker( node->left, fp, FALSE );
fprintf( fp, ")\0 );
break ;

```

```

case n_TOP_DECL :
    /* Any function definitions are along this tree. The first
    ** function from the file is at the bottom of the subtree on
    ** the left. Travel to the bottom and generate facts in
    ** reverse order i.e bottom to top */

    fact_walker( node->left, fp, FALSE );
    /* Now travel the function definition for this node. */
    fact_walker( node->right, fp, FALSE );
    break ;

case n_WHILE_STMT :
    /* A While statement. The right subtree points to the
    ** condition expression. The left subtree points to the
    ** body of the WHILE. */
    /* ( S W (expr) (body) ) */

    fprintf( fp, "S W " );

    /* Travel the expression subtree */
    fact_walker( node->right, fp, FALSE );

    /* Travel the body subtree */
    fact_walker( node->left, fp, FALSE );
    fprintf( fp, ")\n" );
    break ;

default :
    /* Found a node type that we don't have a case
    ** statement for yet. */

    err_prt(
        " *** Error : don't know how to make a fact for node type = %s0.
        node_name( node->type ) );

} ; /* end of switch */

} /* end of fact_walker() */

/*****
*
* File : lex_tabs.c
*
* Created : 04/13/87 by : Eric Byrne
*
* Purpose : This file contains the definition of tables that
* relate keywords to their token values and operators
* to their token values. Also contained in this file
*
*****/

```

```

*         are the routines which manipulate these tables.
*
* Routines :
*     get_key_tok()
*     get_oper_tok()
*     get_tok_name()
*
*****//

#include "flags.def"
#include "tokens.def"

/* Define a structure to relate keywords to their token values. */
struct tok_vals {
    char *name ; /* A keyword name or operator string */
    int token ; /* The token value for the name */
};

/* Define the keyword token value table. */
static struct tok_vals keyword[] =
{
    /* Data types */
    "char", k_CHAR,
    "double", k_DOUBLE,
    "enum", k_ENUM,
    "float", k_FLOAT,
    "int", k_INT,
    "long", k_LONG,
    "short", k_SHORT,
    "struct", k_STRUCT,
    "union", k_UNION,
    "unsigned", k_UNSIGNED,
    "void", k_VOID,
    /* Misc */
    "sizeof", k_SIZEOF,
    "typedef", k_TYPEDEF,
    /* Storage Classes */
    "auto", k_AUTO,
    "extern", k_EXTERN,
    "register", k_REGISTER,
    "static", k_STATIC,
    /* Statements */
    "break", k_BREAK,
    "case", k_CASE,
    "continue", k_CONTINUE,
    "default", k_DEFAULT,
    "do", k_DO,

```

```

"else",    k_ELSE,
"for",     k_FOR,
"goto",    k_GOTO,
"if",      k_IF,
"return",  k_RETURN,
"switch",  k_SWITCH,
"while",   k_WHILE
};

```

```

/* Define the operator token value table. */
static struct tok_vals operator[] =
{
    /* Arithmetic */
    "+", t_ADD_ONE,
    "/", t_DIVIDE,
    "-", t_MINUS,
    "%", t_REMAINDER,
    "+", t_PLUS,
    "*", t_STAR,
    "-", t_SUB_ONE,

    /* Assignment */
    "+=", t_ADD_ASSIGN,
    "=", t_ASSIGN,
    "&=", t_B_AND_ASSIGN,
    "_B_E_OR_ASSIGN",
    "_B_OR_ASSIGN",
    "/=", t_DIV_ASSIGN,
    "<<=", t_LS_ASSIGN,
    "-=", t_MINUS_ASSIGN,
    "*=", t_MULT_ASSIGN,
    "%=", t_REMAIN_ASSIGN,
    ">>=", t_RS_ASSIGN,

    /* Relational */
    "=", t_EQUAL,
    ">", t_GREATERTHAN,
    ">=", t_GTE,
    "<", t_LESSTHAN,
    "<=", t_LTE,
    "!=", t_NOT_EQUAL,

    /* Boolean */
    "&&", t_L_AND,
    "||", t_L_OR,
    "!", t_NOT,

    /* Bitwise */
    "&", t_AMPERSAND,
    "_B_EXCLUSIVE_OR",
    "_B_OR",
    "<<=", t_LEFT_SHIFT,
    "_B_ONES_COMP,

```

```

">>", t_RIGHT_SHIFT,
/* Array */
"[", t_LEFT_SQ_BRAC,
"]", t_RIGHT_SQ_BRAC,
/* Structure/Union */
".", t_DOT,
">", t_MEM_POINTER,
/* Miscellaneous */
":", t_COLON,
",", t_COMMA,
";", t_COND_THEN,
"{", t_LEFT_CUR_BRAC,
"(", t_LEFT_PAREN,
"}", t_RIGHT_CUR_BRAC,
")", t_RIGHT_PAREN,
";", t_SEMICOLON
};

/*****

Function : get_key_tok()

Purpose : This function is passed a C keyword and returns the
token value for that keyword.

Arguments : key - The keyword whose token value is wanted.

Returns : The token value for the keyword.
0 - If the keyword is not found in the tables.

*****/

get_key_tok( key )
register char *key ;
{

register int i ; /* Used to move through the table. */
register int max ; /* The number of entries in the keyword table */

max = sizeof( keyword ) / sizeof ( struct tok_vals ) ;

for ( i=0 ; i<max ; i++ )
{ /* Search the keyword table */
if ( strcmp( key, keyword[i].name ) == 0 )
{ /* Found the keyword in the table. */
return( keyword[i].token ) ;
}
}

}

/* If we reach here then the keyword was not found in the table. */

```

```

    return( 0 );
} /* end of get_key_tok() */

/*****

Function : get_oper_tok()

Purpose  : This function is passed a character string containing a
           C operator. This function will return the token value
           for that operator.

Arguments: oper - The operator.

Returns  : The operator token value.
           0 - If the operator is not located in the tables.

*****/

get_oper_tok( oper )
register char *oper ;
{
    register int i ; /* Used to move through the table.      */
    register int max ; /* The number of entries in the operator table */

    max = sizeof( operator ) / sizeof ( struct tok_vals ) ;

    for ( i=0 ; i<max ; i++ )
    { /* Search the operator table                               */
        if ( strcmp( oper, operator[i].name ) == 0 )
        { /* Found the operator in the table.                  */
            return( operator[i].token ) ;
        }
    }

    /* If we reach here then the operator was not found in the table. */
    return( 0 ) ;
} /* end of get_oper_tok() */

/*****

Function : get_tok_name()

Purpose  : This function is passed a token value and it will search
           the token tables and return the name of the C keyword or
           operator associated with that string.

```

Arguments : tok - The token value.

Returns : A pointer to a character string containing the token name.
A NULL pointer if the token does not match any entries.

```
*****/  
  
char *get_tok_name( tok )  
register int tok ;  
{  
  
    register int i ; /* Used to move through the tables.      */  
    register int max ; /* The number of entries in a table    */  
  
    /* Search the keyword table first.                          */  
  
    max = sizeof( keyword ) / sizeof ( struct tok_vals ) ;  
  
    for ( i=0; i<max; i++ )  
    { /* Search the keyword table                                */  
        if ( tok == keyword[i].token )  
        { /* Found the keyword token in the table.             */  
            return( keyword[i].name ) ;  
        }  
    }  
  
    /* Did not find it in the keyword table so search the operator table */  
  
    max = sizeof( operator ) / sizeof( struct tok_vals ) ;  
  
    for( i=0 ; i<max ; i++ )  
    { /* Search the operator table.                              */  
        if ( tok == operator[i].token )  
        { /* Found the token in the operator table.             */  
            return( operator[i].name ) ;  
        }  
    }  
  
    /* If we reach here then the keyword was not found in the table. */  
    return( NULL_CHAR ) ;  
  
} /* end of get_tok_name() */  
  
/*****  
*  
*   File : node_names.c  
*  
*****/
```

```

* Created : 4/29/87 by : Eric Byrne
*
* Purpose : This file contains the definition of a structure that
* maps node type values to node names. This names are
* used when displaying the parse tree and when
* generating facts.
*
* Routines : node_name()
*
*****/

```

```

#include "flags.def"
#include "tree_info.def"

```

```

/* This is the structure used to map node types to names. */

```

```

struct name_map {
    char *name ; /* A name for the node */
    int n_type ; /* A node type value. */
} ;

```

```

/* Define the node type to name map here */

```

```

static struct name_map names[] =
{
    /* Terminal nodes */
    "Break", n_BREAK,
    "Char", n_CHAR,
    "Continue", n_CONTINUE,
    "DEC_INT", n_DEC_INT,
    "Default", n_DEFAULT,
    "Float", n_FLOAT,
    "HEX_INT", n_HEX_INT,
    "IDENT", n_IDENT,
    "NULL_STMT", n_NULL,
    "OCT_INT", n_OCT_INT,
    "STRING", n_STRING,
    "Typedef", n_TPEDEF,
    /* Non-terminal nodes */
    "ASSIGN", n_ASSIGN,
    "BLOCK", n_BLOCK,
    "CASE", n_CASE,
    "CAST_EXP", n_CAST_EXP,
    "DO_STMT", n_DO_STMT,
    "ELSE", n_ELSE,
    "ELSE_EXP", n_ELSE_EXP,

```



```

"E_STMT", n_E_STMT,
"EXP_SEPAR", n_EXP_SEPAR,
"FOR_STMT", n_FOR,
"FOR_EXIT", n_FOR_EXIT,
"FOR_INCR", n_FOR_INCR,
"FUNC_CALL", n_FUNC_CALL,
"FUNC_DEF", n_FUNC_DEF,
"GOTO_STMT", n_GOTO,
"IF_BAL", n_IF_BAL,
"IF_EXP", n_IF_EXP,
"IF_UNBAL", n_IF_UNBAL,
"LABEL", n_LABEL,
"PAREN_EXP", n_PAREN_EXP,
"OPERATOR", n_OPERATOR,
"Return", n_RETURN,
"Sizeof", n_SIZEOF_EXP,
"STATEMENT", n_STATEMENT,
"SUBSCRIPT", n_SUBSCRIPT,
"SWITCH_STMT", n_SWITCH,
"TOP_DECL", n_TOP_DECL,
"WHILE_STMT", n_WHILE_STMT
};

```

```
static int S_map_size = sizeof( names ) / sizeof( struct name_map );
```

```
/******
```

```
Function : node_name()
```

```
Purpose : Given a node type this function will return a character
string that contains a name for the type.
```

```
Arguments : type - The node type.
```

```
Returns : A pointer to a character string containing the name or
A NULL pointer if the type is not found in the list.
```

```
*****/
```

```
char *node_name( type )
```

```
register int type;
```

```
{
```

```
register int i ; /* Used to move through the table */
```

```
register int max ; /* The maximum number of entries */
```

```

max = S_map_size ;

for ( i=0 ; i<max ; i++ )
{ /* Search the name map table */
  if ( type == names[i].n_type )
  { /* Found the type in the table */
    return( names[i].name ) ;
  }
}

/* If we reach here then the type was not found in the table. */
err_prt(
  "*** Error : unable to match type %d to a node name. Internal error.0.
  type ) ;
return( NULL_CHAR ) ;
} /* end of node_name() */

```

```

/*****
*
* File : tree_han.c
*
* Created : 04/26/87 by : Eric Byrne
*
* Purpose : This file contains routines for constructing and
*           transversing the parse tree.
*
*           The parse tree is constructed such that it should
*           be transversed from right to left in order to
*           reflect the structure of original code.
*
* Routines :
*           ptree()
*           numeric_node()
*           print_node()
*           string_node()
*           tree_walker()
*
*****/

```

```

#include "flags.def"
#include "tree_info.def"

```

```

/***** GLOBAL VARIABLES *****/

```

```

struct pnode *G_root ;    /* The root of the parse tree.    */

/***** EXTERNAL VARIABLES *****/
extern int errno ;    /* A System variable    */

/***** STATIC VARIABLES *****/
static int S_node_size = (sizeof (struct pnode)); /* A tree node size */
static int S_num_size = (sizeof (struct num_node));

/*****

Function : ptree

Purpose : This function is used to create a new parse tree node.
         The node is created and the struct elements are set.

Arguments : right - A pointer to the right subtree for the new node.
            left  - A pointer to the left subtree for the new node.
            type  - A value that indicates the type of node.
            value - If the node is a leaf then it will have a node
                    value. Some non-terminal nodes also have node
                    values.

Returns : Pointer to a new tree node if no errors are detected.
         A NULL pointer.

*****/

struct pnode *ptree( right, left, type, value )
struct pnode *right ;
struct pnode *left ;
int    type ;
char   *value ;
{

    register struct pnode *new ; /* Pointer to new node    */
    char   *temp ; /* Pointer to temporary area */

    extern char *malloc() ; /* System function    */

    /* First, allocate the new node    */
    errno = 0 ;
    new = (struct pnode *) malloc( S_node_size ) ;
    if ( new == NULL_PNODE )
    { /* Malloc failed, report the error and return    */
        err_prt(

```

```

    *** Error : unable to allocate parse tree node. System error = %d0,
        errno );
    return( NULL_PNODE );
}

/* Now fill the node.                                     */
new->right = right ;
new->left = left ;
new->type = type ;
new->value = value ;

return( new ) ; /* Return successfully                  */
} /* end of ptree() */

/*****

Function : numeric_node()

Purpose : This function builds a terminal parse tree node that
          points to a num_node for its value.

Arguments : number - The number stored as a character string.
            l_or_i - A flag to indicate if the number was entered
                    with a long mark or not. i.e. 123L
            type - The node type of the parse tree node.

Returns : A pointer to the new parse tree node, or
          A NULL pointer if any trouble is detected.

*****/

struct pnode *numeric_node( number, l_or_i, type )
char *number ;
int l_or_i ;
int type ;
{

    struct pnode *new ; /* Pointer to the new parse tree node */
    struct num_node *temp ; /* Pointer to the num_node */

    /* Build a num_node to hold it. */
    temp = (struct num_node *) malloc( S_num_size ) ;
    if ( temp == (struct num_node *)NULL )
    { /* Unable to malloc the number node. */
        err_prt( " *** Error : unable to malloc memory for number ->%s<-0,
                number );
        return( NULL_PNODE );
    }
}

```

```

/* Fill the number node.                                     */
temp->num = number ;
temp->size = l_or_i ;

/* Now create a tree node                                   */
new = ptree( NULL_PNODE, NULL_PNODE, type, temp ) ;
if ( new == NULL_PNODE )
{ /* Unable to build the parse tree node.                 */
  return( NULL_PNODE ) ;
}

return( new ) ; /* Return the new parse tree node.        */
} /* end of numeric_node() */

```

```

/*****

```

```

Routine : print_node()

```

```

Purpose : This routine is used by tree_walker to display a node.

```

```

Arguments : node - The node to be printed.
           numb  - A identifying node number.

```

```

Returns : It does not return any value, but it does increment the
          value of numb by one.

```

```

*****/

```

```

static print_node( node, numb )
struct pnode *node ;
int          *numb ;
{
    struct num_node *temp ;

    printf( "NUM : %d ; ADDR : %8X ; TYP : %s ;", *numb, node,
           node_name( node->type ) ) ;

    if ( node->right == NULL_PNODE )
        printf( " R -> NULL ;" ) ;
    else
        printf( " R -> %8X ;", node->right ) ;

    if ( node->left == NULL_PNODE )
        printf( " L -> NULL ;" ) ;
    else
        printf( " L -> %8X ;", node->left ) ;
}

```

```

switch( node->type )
{
    case n_ASSIGN :
    case n_CHAR :
    case n_IDENT :
    case n_OPERATOR:
    case n_STRING :
        printf( " VAL = ->%s<-0, node->value );
        break ;
    case n_DEC_INT :
    case n_HEX_INT :
    case n_OCT_INT :
        temp = (struct num_node *)node->value ;
        printf(
            " VAL = NUM_NODE_PTR, NUM = %s ; SIZE = %s0,
            temp->num, (temp->size == IS_LONG )
            ? "LONG" : "DEFAULT" );
        break ;

    case n_FLOAT :
        temp = (struct num_node *)node->value ;
        printf( " VAL = NUM_NODE_PTR, NUM = %s0,
            temp->num );

    default :
        printf( " VAL = %s0, (node->value == NULL_CHAR)
            ? "NULL" : "PTR" );
};

```

```
*numb += 1 ;
```

```
} /* end of print_node() */
```

```
/*****
```

Function : string_node()

Purpose : This function builds a terminal parse tree node that points to a character string for its value.

Arguments : string - Pointer to the character string.
 len - The number of bytes needed to store the string.
 type - The node type of the parse tree node.

Returns : A pointer to the new parse tree node, or
 A NULL pointer if any trouble is detected.

```
*****/
```

```
struct pnode *string_node( string, len, type )
char *string ;
```

```

int len ;
int type ;
{

    struct pnode *new ; /* Pointer to the new parse tree node */
    char *temp ; /* Pointer to the allocated memory area */

    /* Allocate memory to store the character string in. */
    temp = malloc( len ) ;
    if ( temp == NULL_CHAR )
    { /* Unable to malloc the memory */
        err_prt( " *** Error : unable to malloc memory for string ->%s<-0.
                string ) ;
        return( NULL_PNODE ) ;
    }

    strcpy( temp, string ) ;

    /* Now create a tree node */
    new = ptree( NULL_PNODE, NULL_PNODE, type, temp ) ;
    if ( new == NULL_PNODE )
    { /* Unable to build the parse tree node. */
        return( NULL_PNODE ) ;
    }

    return( new ) ; /* Return the new parse tree node. */
} /* end of string_node() */

/*****

```

Routine : tree_walker()

Purpose : This routine is used to print a copy of the parse tree.
 The main use of this routine is just to allow examination
 of the tree and will be used mainly for debugging.

Note : This is a recursive routine.

Arguments : root - The root of the tree or subtree whose node and
 subnodes need to be printed.

num - A number to use to number the nodes.

*****/

```

tree_walker( root, num )
struct pnode *root ;
int *num ;
{

```

```

if ( root == NULL_PNODE )
    return ; /* The node is a NULL pointer so just return */

/* Handle the right subtree first. */
tree_walker( root->right, num ) ;

/* Display the root node now */
print_node( root, num ) ;

/* Handle the left subtree now. */
tree_walker( root->left, num ) ;

} /* end of tree_walker() */

```

```

/*****
*
* File : c_gram.y
*
* Created : 04/12/87          by : Eric Byrne
*
* Purpose : This file contains a grammar specification for the C
*           programming language. The content of this file is
*           input for yacc. The resulting parser may be used to
*           parse a C program and generate a model of the program
*           that is in a format usable by the KBPD tool.
*****/

%{

#include "flags.def"
#include "tree_info.def"

/***** External Functions *****/

extern char *get_tok_name() ; /* Declared in lex_tabs.c */
extern struct pnode *numeric_node() ; /* Declared in tree_han.c */
extern struct pnode *ptree() ; /* Declared in tree_han.c */
extern struct pnode *string_node() ; /* Declared in tree_han.c */

/***** External Variables *****/

extern int G_intlen ; /* Declared in c_lex.1 */
extern struct pnode *G_root ; /* Declared in tree_han.c */

```



```

/***** STATIC VARIABLES *****/
static char *temp ; /* Temporary string pointer */

typedef struct pnode *yystype ; /* Affects the yacc $$ and $# vars */
#define YYSTYPE yystype

%}

/***** TOKEN NAMES *****/

/* The Token definitions are added to the grammar file by the
** Makefile just before being passed to yacc. See the .y.o rule
** in the makefile to see exactly what happens.
*/

%start program

%% /***** GRAMMAR RULES *****/

/*****
|
|
DECLARATION RULES |
|
V
*****/

empty :
{ /* Empty */
  $$ = NULL_PNODE ;
}
;

/* This is the top rule for the program it can either accept
** or reject the program. */
program : alt_Q01
{ /* Accept the program */
  G_root = $1 ; /* Save the parse tree */
  YYACCEPT ;
}
| error
{ /* Error : reject the program. */
  G_root = NULL_PNODE ; /* No parse tree. */
  YYABORT ;
}
;

```

```

alt_001 : alt_001 top_lev_decl
        { /* alt_001 for top level declarations */
          if ( $2 == NULL_PNODE )
            { /* Empty or data declaration was parsed ignore that
              ** and just pass the tree on up. */

                $$ = $1 ;
            }
          else
            { /* A function was parsed add that to the tree */
              $$ = ptree( $2, $1, n_TOP_DECL, NULL_CHAR ) ;
              if ( $$ == NULL_PNODE )
                YYERROR ;
            }
        }
| empty
  { /* alt_001 : empty */
    $$ = NULL_PNODE ;
  }
;

/* Top level program declarations consist of data declarations
** and function declarations. Data declarations should be
** added to the symbol table. Function declaration should be
** added to the parse tree for the file. */

top_lev_decl : top_lev_data_decl
              { /* data declarations */
                $$ = NULL_PNODE ; /* Declaration statement are
                ** not added to the parse tree */
              }
| top_lev_func_decl
  { /* function declarations */
    $$ = $1 ; /* Just pass the node on up. */
  }
;

/* Top level data declaration */
top_lev_data_decl : opt_decl_spec t_SEMICOLON
                  { /* Storage class specifier and type specifier */
                    $$ = NULL_PNODE ;
                  }
| opt_decl_spec list_init_dcltr t_SEMICOLON
  { /* Storage class and type specifier with initialization */
    $$ = NULL_PNODE ;
  }
;

alt_002 : empty
| alt_002 param_decl

```

```

    { /* alt_002 : function parameter data type declarations */
      }
    :

    /* Top level function declaration : Involves declaring the
    ** function data type, name, parameter data types and body. */

top_level_func_decl : opt_decl_spec decltr alt_002 compound_stmt
  { /* Function header declaration */
    if ( $2->type != n_IDENT )
      { /* Until we support data declarations this had better
        ** be a node with the function name */
        err_prt(
          " *** Error : Unsupported function declaration syle.0);
          YYERROR ;
        }
        $$ = ptree( $2, $4, n_FUNC_DEF, NULL_CHAR ) ;
        if ( $$ == NULL_PNODE )
          YYERROR ;
      }
    :

loc_data_decl : decl_spec t_SEMICOLON
  {
  }
  | decl_spec list_init_decltr t_SEMICOLON
  {
  }
  :

    /* Function parameter data type declarations */

param_decl : decl_spec t_SEMICOLON
  { /* Function parameter data type declarations */
  }
  | decl_spec list_decltr t_SEMICOLON
  { /* Function parameter non-basic data type declarations */
  }
  :

typename_decl : decl_spec abs_decltr
  {
    $$ = NULL_PNODE ;
  }
  :

alt_003 : empty
  | alt_003 struct_decl
  {
  }

```

```

;
list_struct_decl : alt_003
  {
  }
;

/*
alt_004 : formal_decl
  {
  | alt_004 formal_decl
  }
;
*/

formals_decl : empty
  {
  | alt_008
  } /* I changed this, it was a reference to alt_004 */
;

/*
formal_decl : opt_decl_spec decltr
  {
  }
;
*/

/* Optional storage class and type specifier information like
** for a function */
opt_decl_spec : empty
  {
  | decl_spec
  } /* Optional declaration specifiers */
;

alt_005 : tc_spec
  {
  }
  | alt_005 tc_spec
  {
  }
;

decl_spec : alt_005
  {
  }
;

```

```

tc_spec : std_class
{
  | type_spec
}
;

```

```

std_class : k_AUTO
{
  | k_STATIC
  | k_EXTERN
  | k_REGISTER
  | k_TYPEDEF
}
;

```

```

/*****
*****
***** TYPE DECLARATION RULES I
*****
*****/

```

```

|
|
V

```

```

type_spec : t_TYPEDEF_NAME
{
  /* Be sure to verify that the IDENTIFIER is a typedef
  ** name and not just a variable name or such. */
}
| std_type
{
}
| struct_spec
{
}
| enum_spec
{
}
;

```

```

std_type : k_CHAR
{
}
| k_FLOAT

```

```

    {
    }
| k_DOUBLE
    {
    }
| k_INT
    {
    }
| k_SHORT
    {
    }
| k_LONG
    {
    }
| k_UNSIGNED
    {
    }
| k_VOID
    {
    }
;

struct_spec : struct_ctype struct_name
    {
    }
| union_ctype union_name
    {
    }
| struct_ctype t_LEFT_CUR_BRAC list_struct_decl t_RIGHT_CUR_BRAC
    {
    }
| struct_ctype str_tag_dcltr t_LEFT_CUR_BRAC list_struct_decl
    t_RIGHT_CUR_BRAC
    {
    }
| union_ctype t_LEFT_CUR_BRAC list_struct_decl t_RIGHT_CUR_BRAC
    {
    }
| union_ctype un_tag_dcltr t_LEFT_CUR_BRAC list_struct_decl
    t_RIGHT_CUR_BRAC
    {
    }
;

struct_ctype : k_STRUCT
    {
    }
;

union_ctype : k_UNION

```

```

    {
    }
;

struct_decl : decl_spec list_struct_dcltr t_SEMICOLON
    {
    }
;

struct_dcltr : dcltr
    {
    }
| t_COLON exp
    {
    }
| dcltr t_COLON exp
    {
    }
;

alt_006 : enum_dcltr
    {
    }
| alt_006 t_COMMA enum_dcltr
    {
    }
;

alt_007 : t_LEFT_CUR_BRAC alt_006 t_RIGHT_CUR_BRAC
    {
    }
| t_LEFT_CUR_BRAC alt_006 t_COMMA t_RIGHT_CUR_BRAC
    {
    }
;

enum_spec : enum_ctype enum_name
    {
    }
| enum_ctype alt_007
    {
    }
| enum_ctype en_tag_dcltr alt_007
    {
    }
;

enum_ctype : k_ENUM
    {
    }

```

```

;
enum_dcltr : name_dcltr
  {
  | name_dcltr t_ASSIGN exp
  }
;

name_dcltr : t_IDENTIFIER
  { /* A name declaration has been found */
    $$ = ptree( NULL_PNODE, NULL_PNODE, n_IDENT, (char *)yy1val);
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

str_tag_dcltr : t_IDENTIFIER
  { /* A struct tag declaration name has been found */
    $$ = ptree( NULL_PNODE, NULL_PNODE, n_IDENT, (char *)yy1val);
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

un_tag_dcltr : t_IDENTIFIER
  { /* A union union tag_dcltr has been found */
    $$ = ptree( NULL_PNODE, NULL_PNODE, n_IDENT, (char *)yy1val);
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

en_tag_dcltr : t_IDENTIFIER
  { /* A enum tag dcltr has been found */
    $$ = ptree( NULL_PNODE, NULL_PNODE, n_IDENT, (char *)yy1val);
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

key_dcltr : t_IDENTIFIER
  { /* A key dcltr has been found */
    $$ = ptree( NULL_PNODE, NULL_PNODE, n_IDENT, (char *)yy1val);
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

```



```

p1_dcltr : name_dcltr
  {
  | t_LEFT_PAREN dcltr t_RIGHT_PAREN
  }
;

p2_dcltr : p1_dcltr
  {
  | p2_dcltr t_LEFT_PAREN formals_decl t_RIGHT_PAREN
  }
  {
  | p2_dcltr t_LEFT_SQ_BRAC t_RIGHT_SQ_BRAC
  }
  {
  | p2_dcltr t_LEFT_SQ_BRAC list_exp t_RIGHT_SQ_BRAC
  }
;

alt_008 : key_dcltr
  {
  | alt_008 t_COMMA key_dcltr
  }
;

p3_dcltr : p2_dcltr
  {
  | t_STAR p3_dcltr
  { /* Right now until we support data declarations just
    ** forget the STAR. This will need to be fixed later. */
    $$ = $2 ;
  }
  }
;

dcltr : p3_dcltr
  {
  }
;

init_dcltr : dcltr
  {
  | dcltr t_ASSIGN init_exp
  }

```

```

    }
;

alt_009 : init_exp
{
}
| alt_009 t_COMMA init_exp
{
}
;

init_exp : exp
{
}
| t_LEFT_CUR_BRAC alt_009 t_RIGHT_CUR_BRAC
{
}
| t_LEFT_CUR_BRAC alt_009 t_COMMA t_RIGHT_CUR_BRAC
{
}
;

p1_abs_dcltr : t_LEFT_PAREN p3_abs_dcltr t_RIGHT_PAREN
{
}
;

p2_abs_dcltr : p1_abs_dcltr
{
}
| p2_abs_dcltr t_LEFT_PAREN t_RIGHT_PAREN
{
}
| t_LEFT_PAREN t_RIGHT_PAREN
{
}
| p2_abs_dcltr t_LEFT_SQ_BRAC t_RIGHT_SQ_BRAC
{
}
| p2_abs_dcltr t_LEFT_SQ_BRAC list_exp t_RIGHT_SQ_BRAC
{
}
| e_abs_dcltr t_LEFT_SQ_BRAC t_RIGHT_SQ_BRAC
{
}
| e_abs_dcltr t_LEFT_SQ_BRAC list_exp t_RIGHT_SQ_BRAC
{
}
;

```

```

p3_abs_dcltr : p2_abs_dcltr
  {
    | t_STAR p3_abs_dcltr
    {
      | t_STAR e_abs_dcltr
      {
        }
      }
    }
  }
;

e_abs_dcltr :
abs_dcltr : p3_abs_dcltr
  {
    | e_abs_dcltr
    {
      }
    }
  }
;

alt_010 : init_dcltr
  {
    | alt_010 t_COMMA init_dcltr
    {
      }
    }
  }
;

list_init_dcltr : alt_010
  {
    }
  }
;

alt_011 : struct_dcltr
  {
    | alt_011 t_COMMA struct_dcltr
    {
      }
    }
  }
;

list_struct_dcltr : alt_011
  {
    }
  }
;

alt_012 : dcltr
  {
    }
  }
;

```

```

|alt_012 t_COMMA decltr
{
}
;

list_decltr : alt_012
{
}
;

/*****          |
*****          |
***** STATEMENT RULES          |
*****          V
*****/

alt_013 : empty
{ /* alt_013 : empty */
  $$ = NULL_PNODE ;
}

|alt_013 decl_or_stmt
{ /* alt_013 : alt_013 decl_or_stmt */
  if ( $2 == NULL_PNODE )
  { /* declaration line was parsed */
    $$ = $1 ; /* Pass the statement list on up */
  }
  else
  { /* A statement was parsed. */
    $$ = ptree( $2, $1, n_STATEMENT, NULL_CHAR ) ;
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
}
;

compound_stmt : t_LEFT_CUR_BRAC alt_013 t_RIGHT_CUR_BRAC
{ /* compound_stmt : { alt_013 } */
  $$ = ptree( $2, NULL_PNODE, n_BLOCK, NULL_CHAR ) ;
  if ( $$ == NULL_PNODE )
    YYERROR ;
}
;

decl_or_stmt : loc_data_decl
{ /* decl_or_stmt : loc_data_decl */
  $$ = NULL_PNODE ; /* Declarations are added to the
                    ** environment table, not the parse
                    ** tree. */
}
|stmt

```

```

    { /* decl_or_stmt : stmt */
      $$ = $1 ; /* Just pass the tree node on up */
    }
  ;

basic_stmt : e_stmt
  { /* basic_stmt : e_stmt */
    $$ = $1 ; /* Just pass the tree node on up */
  }
| compound_stmt
  { /* basic_stmt : compound_stmt */
    $$ = $1 ; /* Just pass the tree node on up */
  }
| do_stmt
  { /* basic_stmt : do_stmt */
    $$ = $1 ; /* Just pass the tree node on up */
  }
| break_stmt
  { /* basic_stmt : break_stmt */
    $$ = $1 ; /* Just pass the tree node on up */
  }
| continue_stmt
  { /* basic_stmt : continue_stmt */
    $$ = $1 ; /* Just pass the tree node on up */
  }
| return_stmt
  { /* basic_stmt : return_stmt */
    $$ = $1 ; /* Just pass the tree node on up */
  }
| goto_stmt
  { /* basic_stmt : goto_stmt */
    $$ = $1 ; /* Just pass the tree node on up */
  }
| null_stmt
  { /* basic_stmt : null_stmt */
    $$ = $1 ; /* Just pass the tree node on up */
  }
;

bal_stmt : basic_stmt
  { /* bal_stmt : basic_stmt */
    $$ = $1 ; /* Just pass the tree node on up */
  }
| bal_while_stmt
  { /* bal_stmt : bal_while_stmt */
  }
| bal_for_stmt
  { /* bal_stmt : bal_for_stmt */
    $$ = $1 ; /* Just pass the tree node on up */
  }
;

```

```

|bal_ifelse_stmt
{ /* bal_stmt : bal_ifelse_stmt */
  $$ = $1; /* Just pass the tree node on up */
}
|bal_switch_stmt
{ /* bal_stmt : bal_switch_stmt */
  $$ = $1; /* Just pass the tree node on up */
}
|label bal_stmt
{ /* bal_stmt : label bal_stmt */
  $$ = ptree( $1, $2, n_LABEL, NULL_CHAR );
  if ( $$ == NULL_PNODE )
    YYERROR;
}
;

unbal_stmt : unbal_while_stmt
{ /* unbal_stmt : unbal_while_stmt */
  $$ = $1; /* Just pass the tree node on up */
}
|unbal_for_stmt
{ /* unbal_stmt : unbal_for_stmt */
  $$ = $1; /* Just pass the tree node on up */
}
|unbal_if_stmt
{ /* unbal_stmt : unbal_if_stmt */
  $$ = $1; /* Just pass the tree node on up */
}
|unbal_ifelse_stmt
{ /* unbal_stmt : unbal_ifelse_stmt */
  $$ = $1; /* Just pass the tree node on up */
}
|unbal_switch_stmt
{ /* unbal_stmt : unbal_switch_stmt */
  $$ = $1; /* Just pass the tree node on up */
}
|label unbal_stmt
{ /* unbal_stmt : label unbal_stmt */
  $$ = ptree( $1, $2, n_LABEL, NULL_CHAR );
  if ( $$ == NULL_PNODE )
    YYERROR;
}
;

bal_ifelse_stmt : k_IF t_LEFT_PAREN list_exp t_RIGHT_PAREN bal_stmt
k_ELSE bal_stmt
{ /* bal_ifelse_stmt : IF cond then_exp else else_exp */
  /* Build the "else" node, Right -> then exp, L = else exp */
  $6 = ptree( $5, $7, n_ELSE, NULL_CHAR );
  if ( $6 == NULL_PNODE )

```

```

YYERROR ;

/* Build the "if" node, Right -> cond exp, L = else node */
$$ = ptree( $3, $6, n_IF_BAL, NULL_CHAR );
if ( $$ == NULL_PNODE )
    YYERROR ;
}
;

unbal_ifelse_stmt : k_IF t_LEFT_PAREN list_exp t_RIGHT_PAREN bal_stmt
                  k_ELSE unbal_stmt
    { /* unbal_ifelse_stmt : IF ( cond ) then_exp ELSE else_exp */
      /* Build the "else" node, Right -> then exp, L = else exp */
      $6 = ptree( $5, $7 , n_ELSE, NULL_CHAR );
      if ( $6 == NULL_PNODE )
          YYERROR ;

      /* Build the "if" node, Right -> cond exp, L = else node */
      $$ = ptree( $3, $6, n_IF_BAL, NULL_CHAR );
      if ( $$ == NULL_PNODE )
          YYERROR ;
    }
;

unbal_if_stmt : k_IF t_LEFT_PAREN list_exp t_RIGHT_PAREN stmt
    { /* unbal_if_stmt : IF ( cond ) then_stmt */
      $$ = ptree( $3, $5, n_IF_UNBAL, NULL_CHAR );
      if ( $$ == NULL_PNODE )
          YYERROR ;
    }
;

stmt      : bal_stmt
    { /* stmt : bal_stmt */
      $$ = $1 ; /* Just pass the tree node on up */
    }
| unbal_stmt
    { /* stmt : unbal_stmt */
      $$ = $1 ; /* Just pass the tree node on up */
    }
;

e_stmt    : list_exp t_SEMICOLON
    { /* e_stmt : list_exp ; */
      $$ = ptree( $1, NULL_PNODE, n_E_STMT, NULL_CHAR );
      if ( $$ == NULL_PNODE )
          YYERROR ;
    }
;

```

```

bal_while_stmt : k_WHILE t_LEFT_PAREN list_exp t_RIGHT_PAREN bal_stmt
  { /* bal_while_stmt : WHILE ( cond ) stmt */
    $$ = ptree( $3, $5, n_WHILE_STMT, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      . YYERROR ;
  }
;

unbal_while_stmt : k_WHILE t_LEFT_PAREN list_exp t_RIGHT_PAREN unbal_stmt
  { /* unbal_while_stmt : WHILE ( cond ) stmt */
    $$ = ptree( $3, $5, n_WHILE_STMT, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

do_stmt : k_DO stmt k_WHILE t_LEFT_PAREN list_exp t_RIGHT_PAREN
         t_SEMICOLON
  { /* do_stmt : DO stmt WHILE ( cond ) */
    $$ = ptree( $2, $5, n_DO_STMT, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

/* This is the increment statement of a balanced FOR statement */
alt_014 : t_SEMICOLON t_RIGHT_PAREN bal_stmt
  { /* alt_014 : ; ) bal_stmt */
    /* There are no increment statements for the FOR */
    $$ = ptree( NULL_PNODE, $3, n_FOR_INCR, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
| t_SEMICOLON list_exp t_RIGHT_PAREN bal_stmt
  { /* alt_014 : ; list_exp ) bal_stmt */
    /* There are increment statements for the FOR statement */
    $$ = ptree( $2, $4, n_FOR_INCR, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

/* This is the exit condition section of a balanced FOR statement */
alt_015 : t_SEMICOLON alt_014
  { /* alt_015 : ; alt_014 */
    /* There is no exit conditions specified for the FOR */
    $$ = ptree( NULL_PNODE, $2, n_FOR_EXIT, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

```



```

!t_SEMICOLON list_exp alt_014
  { /* alt_015 : ; list_exp alt_014 */
    /* There are exit conditions specified for the FOR */
    $$ = ptree( $2, $3, n_FOR_EXIT, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

bal_for_stmt : k_FOR t_LEFT_PAREN alt_015
  { /* bal_for_stmt : FOR ( alt_015 */
    /* No initialize section for the FOR statement */
    $$ = ptree( NULL_PNODE, $3, n_FOR, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
!k_FOR t_LEFT_PAREN list_exp alt_015
  { /* bal_for_stmt : FOR ( exp alt_015 */
    /* There is an initialize section for the FOR statment */
    $$ = ptree( $3, $4, n_FOR, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

/* This is the increment statement of an unbalanced FOR statement */
alt_016 : t_SEMICOLON t_RIGHT_PAREN unbal_stmt
  { /* alt_016 : ;) unbal_stmt */
    /* There are no increment statements for the FOR */
    $$ = ptree( NULL_PNODE, $3, n_FOR_INCR, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
!t_SEMICOLON list_exp t_RIGHT_PAREN unbal_stmt
  { /* alt_016 : ; list_exp ) unbal_stmt */
    /* There are increment statements for the FOR statement */
    $$ = ptree( $2, $4, n_FOR_INCR, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

/* This is the exit condition section of an unbalanced FOR statement */
alt_017 : t_SEMICOLON alt_016
  { /* alt_017 : ; alt_016 */
    /* There is no exit conditions specified for the FOR */
    $$ = ptree( NULL_PNODE, $2, n_FOR_EXIT, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }

```

```

| t_SEMICOLON list_exp alt_016
  { /* alt_017 : ; list_exp alt016 */
    /* There are exit conditions specified for the FOR */
    $$ = ptree( $2, $3, n_FOR_EXIT, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

unbal_for_stmt : k_FOR t_LEFT_PAREN alt_017
  { /* unbal_for_stmt : FOR ( alt_017 */
    /* No initialize section for the FOR statement */
    $$ = ptree( NULL_PNODE, $3, n_FOR, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
| k_FOR t_LEFT_PAREN list_exp alt_017
  { /* unbal_for_stmt : FOR ( list_exp alt_017 */
    /* There is an initialize section for the FOR statement */
    $$ = ptree( $3, $4, n_FOR, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

bal_switch_stmt : k_SWITCH t_LEFT_PAREN list_exp t_RIGHT_PAREN bal_stmt
  { /* bal_switch_stmt : SWITCH ( list_exp ) bal_stmt */
    $$ = ptree( $3, $5, n_SWITCH, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

unbal_switch_stmt : k_SWITCH t_LEFT_PAREN list_exp t_RIGHT_PAREN unbal_stmt
  { /* unbal_switch_stmt : SWITCH ( list_exp ) unbal_stmt */
    $$ = ptree( $3, $5, n_SWITCH, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

break_stmt : k_BREAK t_SEMICOLON
  { /* break_stmt : BREAK ; */
    $$ = ptree( NULL_PNODE, NULL_PNODE, n_BREAK, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

continue_stmt : k_CONTINUE t_SEMICOLON

```

```

    { /* continue_stmt : CONTINUE ; */
      $$ = ptree( NULL_PNODE, NULL_PNODE, n_CONTINUE, NULL_CHAR );
      if ( $$ == NULL_PNODE )
        YYERROR ;
    }
;

return_stmt : k_RETURN t_SEMICOLON
  { /* return_stmt : RETURN ; */
    $$ = ptree( NULL_PNODE, NULL_PNODE, n_RETURN, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
| k_RETURN list_exp t_SEMICOLON
  { /* return_stmt : RETURN list_exp ; */
    $$ = ptree( $2, NULL_PNODE, n_RETURN, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

goto_stmt : k_GOTO label_name t_SEMICOLON
  { /* goto_stmt : GOTO label_name ; */
    $$ = ptree( $2, NULL_PNODE, n_GOTO, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

null_stmt : t_SEMICOLON
  { /* null_stmt : ; */
    $$ = ptree( NULL_PNODE, NULL_PNODE, n_NULL, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

label : name_label
  { /* label : name_label */
    $$ = $1 ; /* Just pass the tree node on up. */
  }
| case_label
  { /* label : case_label */
    $$ = $1 ; /* Just pass the tree node on up. */
  }
| default_label
  { /* label : default_label */
    $$ = $1 ; /* Just pass the tree node on up. */
  }
;

```

```

name_label: t_IDENTIFIER t_COLON
    { /* A name_label followed by a colon has been found */
      $$ = ptree( NULL_PNODE, NULL_PNODE, n_IDENT, (char *)yyval);
      if ( $$ == NULL_PNODE )
        YYERROR;
    }
;

case_label: k_CASE exp t_COLON
    { /* case_label : CASE exp */
      $$ = ptree( $2, NULL_PNODE, n_CASE, NULL_CHAR );
      if ( $$ == NULL_PNODE )
        YYERROR;
    }
;

default_label: k_DEFAULT t_COLON
    { /* default_label : DEFAULT */
      $$ = ptree( NULL_PNODE, NULL_PNODE, n_DEFAULT, NULL_CHAR );
      if ( $$ == NULL_PNODE )
        YYERROR;
    }
;

/*****
***** |
***** |
***** EXPRESSION RULES |
***** v
*****/

/* literals are used as terminals in expressions. */
literal: t_DEC_INT
    { /* A decimal integer has been found */
      $$ = numeric_node( (char *)yyval, G_intlen, n_DEC_INT );
      if ( $$ == NULL_PNODE )
        YYERROR;
    }
| t_OCT_INT
    { /* A octal constant has been found. */
      $$ = numeric_node( (char *)yyval, G_intlen, n_OCT_INT );
      if ( $$ == NULL_PNODE )
        YYERROR;
    }
| t_HEX_INT
    { /* A hexadecimal constant has been found. */
      $$ = numeric_node( (char *)yyval, G_intlen, n_HEX_INT );
      if ( $$ == NULL_PNODE )
        YYERROR;
    }
| t_FLOAT

```

```

    { /* A floating point number has been found */
      $$ = numeric_node( (char *)yy1val, 0, n_FLOAT );
      if ( $$ == NULL_PNODE )
        YYERROR ;
    }
| t_CHAR
  { /* A single character constant has been found */
    $$ = ptree( NULL_PNODE, NULL_PNODE, n_CHAR, (char *)yy1val);
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
| t_STRING
  { /* A character string constant has been found */
    $$ = ptree(NULL_PNODE, NULL_PNODE, n_STRING, (char *)yy1val);
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

id_name : t_IDENTIFIER
  { /* An identifier has been read, id_name */
    $$ = ptree( NULL_PNODE, NULL_PNODE, n_IDENT, (char *)yy1val);
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

struct_name : t_IDENTIFIER
  { /* A structure name has been found. */
    $$ = ptree( NULL_PNODE, NULL_PNODE, n_IDENT, (char *)yy1val);
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

union_name : t_IDENTIFIER
  { /* A union name has been found */
    $$ = ptree( NULL_PNODE, NULL_PNODE, n_IDENT, (char *)yy1val);
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

enum_name : t_IDENTIFIER
  { /* An enum name has been found */
    $$ = ptree( NULL_PNODE, NULL_PNODE, n_IDENT, (char *)yy1val);
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

```

```

label_name : t_IDENTIFIER
  { /* A label name has been found. */
    $$ = ptree( NULL_PNODE, NULL_PNODE, n_IDENT, (char *)yyval);
    if ( $$ == NULL_PNODE )
      YYERROR;
  }
;

field_name : t_IDENTIFIER
  { /* A field_name has been found */
    $$ = ptree( NULL_PNODE, NULL_PNODE, n_IDENT, (char *)yyval);
    if ( $$ == NULL_PNODE )
      YYERROR;
  }
;

paren_exp : t_LEFT_PAREN list_exp t_RIGHT_PAREN
  { /* paren_exp rule */
    $$ = ptree( $2, NULL_PNODE, n_PAREN_EXP, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR;
  }
;

primary_p1_exp : id_name
  { /* primary_p1_exp : id_name */
    $$ = $1; /* Just pass the tree node on up */
  }
| literal
  { /* primary_p1_exp : literal */
    $$ = $1; /* Just pass the tree node on up */
  }
| paren_exp
  { /* primary_p1_exp : paren_exp */
    $$ = $1; /* Just pass the tree node on up */
  }
| k_SIZEOF t_LEFT_PAREN typename_decl t_RIGHT_PAREN
  { /* primary_p1_exp : sizeof ( typename_decl ) */
    $$ = ptree( $3, NULL_PNODE, n_SIZEOF_EXP, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR;
  }
;

primary_p2_exp : primary_p1_exp
  { /* primary_p2_exp : primary_p1_exp */
    $$ = $1; /* Just pass the tree node on up */
  }
| primary_p2_exp t_LEFT_SQ_BRAC list_exp t_RIGHT_SQ_BRAC
  { /* primary_p2_exp : array subscript */

```

```

    $$ = ptree( $1, $3, n_SUBSCRIPT, NULL_CHAR );
    if ( $$ == NULL_PNODE )
        YYERROR;
}
| primary_p2_exp t_LEFT_PAREN t_RIGHT_PAREN
  { /* primary_p2_exp : function call no arguments */
    $$ = ptree( $1, NULL_PNODE, n_FUNC_CALL, NULL_CHAR );
    if ( $$ == NULL_PNODE )
        YYERROR;
  }
| primary_p2_exp t_LEFT_PAREN list_exp t_RIGHT_PAREN
  { /* primary_p2_exp : function call with arguments */
    $$ = ptree( $1, $3, n_FUNC_CALL, NULL_CHAR );
    if ( $$ == NULL_PNODE )
        YYERROR;
  }
| primary_p2_exp t_DOT field_name
  { /* primary_p2_exp : structure dot reference */
    $$ = string_node( "DOT", 4, n_OPERATOR );
    if ( $$ == NULL_PNODE )
        YYERROR;
    $$->right = $1;
    $$->left = $3;
  }
| primary_p2_exp t_MEM_POINTER field_name
  { /* primary_p2_exp : structure member pointer reference */
    $$ = string_node( "MEM_PTR", 8, n_OPERATOR );
    if ( $$ == NULL_PNODE )
        YYERROR;
    $$->right = $1;
    $$->left = $3;
  }
;

primary_exp : primary_p2_exp
  { /* primary_exp : primary_p2_exp */
    $$ = $1; /* Just pass the tree node on up */
  }
;

postfix_exp : primary_exp
  { /* postfix_exp : primary_exp */
    $$ = $1; /* Just pass the tree node on up */
  }
| postfix_exp postfix_op
  { /* postfix_exp : postfix_exp postfix_op */
    $2->left = $1; /* Operator node, left exp only */
    $$ = $2;
  }
;

```

```

postfix_op : t_ADD_ONE
  { /* postfix_op : t_ADD_ONE */
    $$ = string_node( "POST_ADD", 9, n_OPERATOR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
| t_SUB_ONE
  { /* postfix_op : t_SUB_ONE */
    $$ = string_node( "POST_SUB", 9, n_OPERATOR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

prefix_exp : postfix_exp
  { /* prefix_exp : postfix_exp */
    $$ = $1 ; /* Just pass the node on up. */
  }
| k_SIZEOF prefix_exp
  { /* prefix_exp : sizeof prefix_exp */
    $$ = ptree( $2, NULL_PNODE, n_SIZEOF_EXP, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
| prefix_op cast_exp
  { /* prefix_exp : prefix_op cast_exp */
    $1->right = $2 ; /* prefix only has right exp */
    $$ = $1 ;
  }
| t_STAR cast_exp
  { /* prefix_exp : * cast_exp */
    /* Build a node to hold the operator */
    $1 = string_node( "STAR_VAL", 9, n_OPERATOR );
    if ( $1 == NULL_PNODE )
      YYERROR ;
    $1->right = $2 ; /* Star only has right exp */
    $$ = $1 ;
  }
| t_AMPERSAND cast_exp
  { /* prefix_exp : & cast_exp */
    /* Build a node to hold the operator */
    $1 = string_node( "ADDRESS", 8, n_OPERATOR );
    if ( $1 == NULL_PNODE )
      YYERROR ;
    $1->right = $2 ; /* & only has right exp to work on */
    $$ = $1 ;
  }
| negate_op cast_exp
  { /* prefix_exp : negate_op cast_exp */
    $1->right = $2 ; /* only has right exp to work on */
  }

```



```

        $$ = $1 :
    }
;

prefix_op : t_ADD_ONE
{ /* prefix_op : t_ADD_ONE */
  $$ = string_node( "PRE_ADD", 8, n_OPERATOR );
  if ( $$ == NULL_PNODE )
    YYERROR ;
}
| t_SUB_ONE
{ /* prefix_op : t_SUB_ONE */
  $$ = string_node( "PRE_SUB", 8, n_OPERATOR );
  if ( $$ == NULL_PNODE )
    YYERROR ;
}
;

negate_op : t_MINUS
{ /* negate_op : t_MINUS */
  $$ = string_node( "NEGATE", 7, n_OPERATOR );
  if ( $$ == NULL_PNODE )
    YYERROR ;
}
| t_NOT
{ /* negate_op : t_NOT */
  $$ = string_node( "NOT", 4, n_OPERATOR );
  if ( $$ == NULL_PNODE )
    YYERROR ;
}
| t_ONES_COMP
{ /* negate_op : t_ONES_COMP */
  $$ = string_node( "ONES_COMP", 10, n_OPERATOR );
  if ( $$ == NULL_PNODE )
    YYERROR ;
}
;

cast_exp : prefix_exp
{ /* cast_exp : prefix_exp */
  $$ = $1 ; /* Just pass the node on up. */
}
| t_LEFT_PAREN typename_decl t_RIGHT_PAREN cast_exp
{ /* cast_exp : ( typename_decl ) cast_exp */
  $$ = ptree( $4, $2, n_CAST_EXP, NULL_CHAR );
  if ( $$ == NULL_PNODE )
    YYERROR ;
}
;

```

```

unary_oper_exp : cast_exp
  { /* unary_oper_exp : cast_exp */
    $$ = $1 ; /* Just pass the node on up. */
  }
;

mult_oper_exp : unary_oper_exp
  { /* mult_oper_exp : unary_oper_exp */
    $$ = $1 ; /* Just pass the node on up. */
  }
| mult_oper_exp mult_op unary_oper_exp
  { /* mult_oper_exp : mult_oper_exp mult_op unary_oper_exp*/
    $2->right = $1 ; /* Left expression goes on right tree */
    $2->left = $3 ; /* Right expression goes on left tree */
    $$ = $2 ;
  }
;

mult_op : t_STAR
  { /* mult_op : t_STAR */
    $$ = string_node("MULT", 5, n_OPERATOR);
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
| t_DIVIDE
  { /* mult_op : t_DIVIDE */
    $$ = string_node("DIV", 4, n_OPERATOR);
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
| t_REMAINDER
  { /* mult_op : t_REMAINDER */
    $$ = string_node("REMAIN", 7, n_OPERATOR);
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

add_oper_exp : mult_oper_exp
  { /* add_oper_exp : mult_oper_exp */
    $$ = $1 ; /* Just pass the node on up. */
  }
| add_oper_exp add_op mult_oper_exp
  { /* add_oper_exp : add_oper_exp add_op mult_oper_exp */
    $2->right = $1 ;
    $2->left = $3 ;
    $$ = $2 ;
  }
;

```

```

add_op : t_PLUS
    { /* add_op : t_PLUS */
      $$ = string_node( "PLUS", 5, n_OPERATOR );
      if ( $$ == NULL_PNODE )
        YYERROR ;
    }
| t_MINUS
    { /* add_op : t_MINUS */
      $$ = string_node( "MINUS", 6, n_OPERATOR );
      if ( $$ == NULL_PNODE )
        YYERROR ;
    }
;

shift_oper_exp : add_oper_exp
    { /* shift_oper_exp : add_oper_exp */
      $$ = $1 ; /* Just pass the node on up. */
    }
| shift_oper_exp shift_op add_oper_exp
    { /* shift_oper_exp : shift_oper_exp shift_op add_oper_exp */
      $2->right = $1 ;
      $2->left = $3 ;
      $$ = $2 ;
    }
;

shift_op : t_LEFT_SHIFT
    { /* shift_op : t_LEFT_SHIFT */
      $$ = string_node( "LEFT_SHIFT", 11, n_OPERATOR );
      if ( $$ == NULL_PNODE )
        YYERROR ;
    }
| t_RIGHT_SHIFT
    { /* shift_op : t_RIGHT_SHIFT */
      $$ = string_node( "RIGHT_SHIFT", 12, n_OPERATOR );
      if ( $$ == NULL_PNODE )
        YYERROR ;
    }
;

rel_oper_exp : shift_oper_exp
    { /* rel_oper_exp : shift_oper_exp */
      $$ = $1 ; /* Just pass the node on up. */
    }
| rel_oper_exp rel_op shift_oper_exp
    { /* rel_oper_exp : rel_oper_exp rel_op shift_oper_exp */
      $2->right = $1 ;
      $2->left = $3 ;
      $$ = $2 ;
    }
;

```

```

;
rel_op : t_LESSTHAN
  { /* rel_op : t_LESSTHAN */
    $$ = string_node( "LESSTHAN", 9, n_OPERATOR );
    if ( $$ == NULL_PNODE )
      YYERROR;
  }
| t_LTE
  { /* rel_op : t_LTE */
    $$ = string_node( "LTE", 4, n_OPERATOR );
    if ( $$ == NULL_PNODE )
      YYERROR;
  }
| t_GTE
  { /* rel_op : t_GTE */
    $$ = string_node( "GTE", 4, n_OPERATOR );
    if ( $$ == NULL_PNODE )
      YYERROR;
  }
| t_GREATERTHAN
  { /* rel_op : t_GREATERTHAN */
    $$ = string_node( "GREATERTHAN", 12, n_OPERATOR );
    if ( $$ == NULL_PNODE )
      YYERROR;
  }
;

equ_oper_exp : rel_oper_exp
  { /* equ_oper_exp : rel_oper_exp */
    $$ = $1 ; /* Just pass the node on up. */
  }
| equ_oper_exp equ_op rel_oper_exp
  { /* equ_oper_exp : equ_oper_exp equ_op rel_oper_exp */
    $2->right = $1 ;
    $2->left = $3 ;
    $$ = $2 ;
  }
;

/* These are == and != */
equ_op : t_EQUAL
  { /* equ_op : t_EQUAL */
    $$ = string_node( "EQUAL", 6, n_OPERATOR );
    if ( $$ == NULL_PNODE )
      YYERROR;
  }
| t_NOT_EQUAL
  { /* equ_op : t_NOT_EQUAL */
    $$ = string_node( "NOT_EQUAL", 10, n_OPERATOR );
  }

```

```

        if ( $$ == NULL_PNODE )
            YYERROR ;
    }
;

bitand_oper_exp : equ_oper_exp
    { /* bitand_oper_exp : equ_oper_exp */
      $$ = $1 ; /* Just pass the node on up. */
    }
| bitand_oper_exp t_AMPERSAND equ_oper_exp
    { /* bitand_oper_exp : bitand_oper_exp & */
      $2 = string_node( "BIT_AND", 8, n_OPERATOR ) ;
      if ( $2 == NULL_PNODE )
          YYERROR ;
      $2->right = $1 ;
      $2->left = $3 ;
      $$ = $2 ;
    }
;

bitxor_oper_exp : bitand_oper_exp
    { /* bitxor_oper_exp : bitand_oper_exp */
      $$ = $1 ; /* Just pass the node on up. */
    }
| bitxor_oper_exp t_B_EXCLUSIVE_OR bitand_oper_exp
    { /* bitxor_oper_exp : bitxor_oper_exp ^ bitand_oper_exp */
      $2 = string_node( "BIT_EXCL_OR", 12, n_OPERATOR ) ;
      if ( $2 == NULL_PNODE )
          YYERROR ;
      $2->right = $1 ;
      $2->left = $3 ;
      $$ = $2 ;
    }
;

bitor_oper_exp : bitxor_oper_exp
    { /* bitor_oper_exp : bitxor_oper_exp */
      $$ = $1 ; /* Just pass the node on up. */
    }
| bitor_oper_exp t_B_OR bitxor_oper_exp
    { /* bitor_oper_exp : bitor_oper_exp | bitxor_oper_exp */
      $2 = string_node( "BIT_OR", 7, n_OPERATOR ) ;
      if ( $2 == NULL_PNODE )
          YYERROR ;
      $2->right = $1 ;
      $2->left = $3 ;
      $$ = $2 ;
    }
;

```

```

and_oper_exp : bitor_oper_exp
  { /* and_oper_exp : bitor_oper_exp */
    $$ = $1 ; /* Just pass the node on up. */
  }
| and_oper_exp t_L_AND bitor_oper_exp
  { /* and_oper_exp : and_oper_exp && bitor_oper_exp */
    $2 = string_node( "L_AND", 6, n_OPERATOR );
    if ( $2 == NULL_PNODE )
      YYERROR ;
    $2->right = $1 ;
    $2->left = $3 ;
    $$ = $2 ;
  }
;

or_oper_exp : and_oper_exp
  { /* or_oper_exp : and_oper_exp */
    $$ = $1 ; /* Just pass the node on up. */
  }
| or_oper_exp t_L_OR and_oper_exp
  { /* or_oper_exp : or_oper_exp || and_oper_exp */
    $2 = string_node( "L_OR", 5, n_OPERATOR );
    if ( $2 == NULL_PNODE )
      YYERROR ;
    $2->right = $1 ;
    $2->left = $3 ;
    $$ = $2 ;
  }
;

cond_exp : or_oper_exp
  { /* cond_exp : or_oper_exp */
    $$ = $1 ; /* Just pass the node on up. */
  }
| or_oper_exp t_COND_THEN list_exp t_COLON cond_exp
  { /* cond_exp : or_oper_exp ? list_exp : cond_exp */
    /* Build the "if" node, Right -> cond, Left -> else */
    $$ = ptree( NULL_PNODE, NULL_PNODE, n_IF_EXP, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;

    /* Build the "else" node */
    $4 = ptree( $3, $5, n_ELSE_EXP, NULL_CHAR );
    if ( $4 == NULL_PNODE )
      YYERROR ;

    $$->right = $1 ; /* This is the conditional expression */
    $$->left = $4 ; /* The ELSE_EXP points to the "then"
      ** and "else" expressions. */
  }
;

```

```

    }
;

assign_exp : cond_exp
{ /* assign_exp : cond_exp */
  $$ = $1 ; /* Just pass the node on up. */
}

| cond_exp asgn_op exp
{ /* assign_exp : cond_exp asgn_op exp */
  $2->right = $3 ;
  $2->left = $1 ;
  $$ = $2 ;
}
;

asgn_op : t_ASSIGN
{ /* asgn_op : t_ASSIGN */
  $$ = string_node( "ASSIGN", 7, n_ASSIGN );
  if ( $$ == NULL_PNODE )
    YYERROR ;
}

| t_ADD_ASSIGN
{ /* asgn_op : t_ADD_ASSIGN */
  $$ = string_node( "ADD_ASSIGN", 9, n_ASSIGN );
  if ( $$ == NULL_PNODE )
    YYERROR ;
}

| t_MINUS_ASSIGN
{ /* asgn_op : t_MINUS_ASSIGN */
  $$ = string_node( "MINUS_ASSIGN", 11, n_ASSIGN );
  if ( $$ == NULL_PNODE )
    YYERROR ;
}

| t_MULT_ASSIGN
{ /* asgn_op : t_MULT_ASSIGN */
  $$ = string_node( "MULT_ASSIGN", 10, n_ASSIGN );
  if ( $$ == NULL_PNODE )
    YYERROR ;
}

| t_DIV_ASSIGN
{ /* asgn_op : t_DIV_ASSIGN */
  $$ = string_node( "DIV_ASSIGN", 9, n_ASSIGN );
  if ( $$ == NULL_PNODE )
    YYERROR ;
}

| t_REMAIN_ASSIGN
{ /* asgn_op : t_REMAIN_ASSIGN */
  $$ = string_node( "REMAIN_ASSIGN", 12, n_ASSIGN );
  if ( $$ == NULL_PNODE )
    YYERROR ;
}

```

```

    }
| t_RS_ASSIGN
  { /* asgn_op : t_RS_ASSIGN */
    $$ = string_node( "RS_ASSIGN", 8, n_ASSIGN );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
| t_LS_ASSIGN
  { /* asgn_op : t_LS_ASSIGN */
    $$ = string_node( "LS_ASSIGN", 8, n_ASSIGN );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
| t_B_AND_ASSIGN
  { /* asgn_op : t_B_AND_ASSIGN */
    $$ = string_node( "BIT_AND_ASSIGN", 13, n_ASSIGN );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
| t_B_E_OR_ASSIGN
  { /* asgn_op : t_B_E_OR_ASSIGN */
    $$ = string_node( "BIT_E_OR_ASSIGN", 14, n_ASSIGN );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
| t_B_OR_ASSIGN
  { /* asgn_op : t_B_OR_ASSIGN */
    $$ = string_node( "BIT_OR_ASSIGN", 12, n_ASSIGN );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

exp : assign_exp
  { /* exp : assign_exp */
    $$ = $1 ; /* Just pass the node on up. */
  }
;

list_exp : exp
  { /* list_exp : exp */
    $$ = $1 ; /* Just pass the node on up. */
  }
| list_exp t_COMMA exp
  { /* list_exp : list_exp . exp */
    $$ = ptree( $1, $3, n_EXP_SEPAR, NULL_CHAR );
    if ( $$ == NULL_PNODE )
      YYERROR ;
  }
;

```


%%

```
/*
 *
 * File : c_lex.l
 *
 * Created : 04/13/87 by : Eric Byrne
 *
 * Purpose : This file contains a lexical analyzer for the C
 * programming language. The content of this file
 * is formatted to be input to the UNIX utility lex.
 *
 * Routines : yylex() - created by lex
 *            input()
 *            save_name()
 *            sep_line()
 *            yywrap()
 */
*****/

/* A few definitions for the lexical analyzer. */

/* D : Digits, OD : Octal Digits, HD : Hex Digits, E : Exponents */
/* EC : Escape Codes, L : Letters, LN : Letters and Numerals */
/* PC : Printable Characters, sans " */
D [0-9]
OD [0-7]
HD [0-9a-fA-F]
E [eE[+]?{D}+]
EC [ntbrfv'
L [_A-Za-z]
LN [_A-Za-za0-9]
PC [A-Za-z0-9@#&--+~[] ;'/?>.<]

%{

#undef input /* Ignore the lex defined input macro. */
#include "flags.def"
#include "tokens.def"
#include "tree_info.def"

/****** External Variables *****/

extern FILE *G_fp ; /* Defined in code_scan.c */
extern char *yyval ; /* Defined in yacc generated file */
extern char G_fname[] ; /* Defined in code_scan.c */
```

```

/***** External Functions *****/
extern char *malloc(); /* System function */

/***** Global Variables *****/

struct inc_files *G_includes; /* Pointer to top of linked list
                               ** containing names of include files
                               ** used in a source file. */

int G_intlen; /* Indicates if an integer constant was
              ** typed as long in the program. */

/***** Local Variables *****/

static int tok; /* Holds a token value. */
static int temp; /* Scratch variable. */
static char buf[256]; /* Temporary buffer */
static int i,j; /* Two more rogue variables. */

%)
%% /***** LEX RULES *****/

^# ]*define[ ]*[L]{LN}* |
^# ]*undef[ ]*[L]{LN}* |
^# ]*include |
^# ]*if |
^# ]*elif |
^# ]*else |
^# ]*endif |
^# ]*ifdef |
^# ]*ifndef { /* These Preprocessor lines are not valid. */
err_prt(
" *** Error : Preprocessor statments are not supported.0);
return( FAILURE );
}
^# ]*line { /* Yacc puts this into y.tab.c. So we need to recognize
** it and ignore it for now. */
}
^# ](D){D}* [
/* The C preprocessor leaves this in its output file.
** This information gives a line number and a file name.
** We need it to be able to handle the preprocessor
** output and to recognize where include files have been
** expanded into the source text. We want to ignore
** include file text that has been expanded into the
** current file. */

```

```

int line : /* The line number mentioned */
int nline : /* Next expected line # to find */
char iname[255] : /* The file name mentioned */

sep_line( yytext, &line, iname ) ;

if ( strcmp( G_fname, iname ) == 0 )
{
    yylineno - ;
    continue ; /* Is for the current file */
}
else
{ /* Save iname in include file name list unless we
** are dealing with y.tab.c. Then worry about a
** special case. */

    nline = yylineno ;
    if ( save_name( iname, yylineno-1 ) == FAILURE )
        return( FAILURE ) ;
}

/* Now read until we find another line like this
** and iname == the current scan file name. */

i = FALSE ;
do
{ /* Read until the end of the include file text
** is reached. */
    j = input() ;
    while ( (char)j != '\0' && j != 0 )
        j = input() ;

    j = input() ;
    while ( (char)j == '\0' )
        j = input() ; /* Read past multiple 0s */

    if ( j == 0 )
    { /* Reached EOF */
        err_prt(
*** Error : reached EOF while looking for end of include file.0 ) ;
        return( FAILURE ) ;
    }

    if ( (char)j == '#' )
    { /* A line starting with # is what we
** are looking for. # d "name" */
        temp = 0 ;
        while ( (char)j != '\0' )
        {
            buf[temp++] = j ;

```

```

        j = input() ;
    } ;
    sep_line( buf, &line, iname ) ;
    if ( ( line == nline ) && ( strcmp( G_fname,
        iname ) == 0 ) )
    { /* Reached the end of include file */
        i = TRUE ;
        yylineno = line ;
    }
} while ( i == FALSE ) ;
continue ;

}
{L}{LN}* { /* Identifier */
/* Check if it is a keyword */
if ( ( tok = get_key_tok( yytext ) ) != 0 )
{ /* Yes, it is a keyword */
return( tok ) ;
}
else
{ /* No, it is not a keyword, just an identifier */
/* Save the identifier string. */
yylval = malloc( strlen( yytext ) + 1 ) ;
if ( yylval == NULL_CHAR )
{
err_prt(
" *** Error : unable to allocate memory for identifier ->%s<-0,
yytext ) ;
return( FAILURE ) ;
}
strcpy( yylval, yytext ) ;
return( t_IDENTIFIER ) ;
}
}
{D}{D}*{11L}? { /* Decimal-constant : What I call an integer. */
/* Store the number */
yylval = malloc( strlen( yytext ) + 1 ) ;
if ( yylval == NULL_CHAR )
{
err_prt(
" *** Error : unable to allocate memory for ->%s<-0,
yytext ) ;
return( FAILURE ) ;
}
strcpy( yylval, yytext ) ;

/* Is the constant typed as long in the program? */
temp = strlen( yytext ) - 1 ;
if ( yytext[temp] == 'l' || yytext[temp] == 'L' )

```

```

    {
        G_intlen = IS_LONG ;
        yyival[temp] = ' ' ; /* Delete the L from the num */
    }
    else
        G_intlen = IS_INT ; /* It is default int */

    return( t_DEC_INT ) ;
}
0(OD)*[IL]? { /* Octal-constant */
    /* Keep the string for now */
    yyival = malloc( strlen( yytext ) + 1 ) ;
    if ( yyival == NULL_CHAR )
    {
        err_prt(
            " *** Error : unable to allocate memory for ->%s<-0,
            yytext ) ;
        return( FAILURE ) ;
    }
    strcpy( yyival, yytext ) ;

    /* Is the constant typed as long in the program? */
    temp = strlen( yytext ) - 1 ;
    if ( yytext[temp] == 'l' || yytext[temp] == 'L' )
    {
        G_intlen = IS_LONG ;
        yyival[temp] = ' ' ; /* Delete the L from the num */
    }
    else
        G_intlen = IS_INT ; /* It is default int */
    return( t_OCT_INT ) ;
}
[0xX]{HD}+[IL]? { /* Hexadecimal-constant */
    /* Keep the string for now */
    yyival = malloc( strlen( yytext ) + 1 ) ;
    if ( yyival == NULL_CHAR )
    {
        err_prt(
            " *** Error : unable to allocate memory for ->%s<-0,
            yytext ) ;
        return( FAILURE ) ;
    }
    strcpy( yyival, yytext ) ;

    /* Is the constant typed as long in the program? */
    temp = strlen( yytext ) - 1 ;
    if ( yytext[temp] == 'l' || yytext[temp] == 'L' )
    {
        G_intlen = IS_LONG ;
        yyival[temp] = ' ' ; /* Delete the L from the num */
    }

```

```

    }
    else
        G_intlen = IS_INT ; /* It is default int */
    return( t_HEX_INT ) ;
}
(D)+{E} |
{D}+"{E}? |
{D}+"{D}+({E})? |
"."{D}+({E})? { /* Floating Point-constant */
/* Keep the string for now */
yyval = malloc( strlen( yytext ) + 1 ) ;
if ( yyval == NULL_CHAR )
{
    err_prt(
        " *** Error : unable to allocate memory for ->%s<-0,
        yytext ) ;
    return( FAILURE ) ;
}
strcpy( yyval, yytext ) ;
return( t_FLOAT ) ;
}
"."(PC)(
/* Character-constant */
/* Double up any backslashes in the constant string */
temp = strlen( yytext ) ;
j = 0 ;
for( i=0 ; i<temp ; i++ )
{
    if ( yytext[i] == " )
        buf[j++] = yytext[i] ; /* Add another one */
    else
    {
        if ( yytext[i] == \ )
        { /* Put a backslash in front of it. The
            ** vertical has a meaning to Lisp. */
            buf[j++] = " ;
        }
    }
    buf[j++] = yytext[i] ;
}
buf[j] = ' ' ;

/* Copy the character */
yyval = malloc( strlen( buf ) + 1 ) ;
if ( yyval == NULL_CHAR )
{
    err_prt(
        " *** Error : unable to allocate memory for ->%s<-0,
        yytext ) ;
    return( FAILURE ) ;
}

```

```

    }
    strcpy( yyval, buf );
    return( t_CHAR );
}

/* String-constant */
/* Double up any backslashes in the constant string */
temp = strlen( yytext );
j = 0;
for( i=0; i<temp; i++ )
{
    if ( yytext[i] == '"' )
        buf[j++] = yytext[i]; /* Add another one */
    else
    {
        if ( yytext[i] == '\\' )
        { /* Put a backslash in front of it. The
            ** vertical has a meaning to Lisp. */
            buf[j++] = '\\';
        }
        buf[j++] = yytext[i];
    }
}
buf[j] = '\0';

/* Copy the string */
yyval = malloc( strlen( buf ) + 1 );
if ( yyval == NULL_CHAR )
{
    err_prt(
        " *** Error : unable to allocate memory for ->%s<-0,
        yytext );
    return( FAILURE );
}
strcpy( yyval, buf );
return( t_STRING );
}
;
.
-
)
]
[
-

```

```

'
/
+
^
v
?
{ /* Single character operators */
  if ( ( tok = get_oper_tok( yytext ) ) == 0 )
  {
    err_prt( " *** Error : Invalid operator
              yytext );
    return( FAILURE );
  }
  else /* It is a valid operator */
  {
    return( tok );
  }
}

/* Multi-character operators */
/* Get rid of any spaces between the characters */
temp = strlen( yytext );
j = 0;
for( i=0; i<temp; i++)
  if ( yytext[i] != ' ' && yytext[i] != '\n' )
    buf[j++] = yytext[i];
buf[j] = '\0';

```



```

/* Get the Token value for the operator          */
if ( ( tok = get_oper_tok( buf ) ) == 0 )
{
    err_prt( " *** Error : Invalid operator
            yytext );
    return( FAILURE );
}
else /* It is a valid operator                  */
{
    return( tok );
}
}

[ 0 { /* Ignore blanks, tabs, newlines, and formfeeds. */
}

"/*" { /* Ignore comments, read until an end-of-comment is found */
temp = FALSE;
do
{
    i = input(); /* Read the next character */
    switch( i )
    {
        case " : /* Read next char and ignore it */
            i = input();
            break ;

        case "*" : /* Possible end-of-comment */
            i = input(); /* Found the end */
            if ( i == '/' ) /* of comment. */
                temp = TRUE ;
            else
                unput( i ); /* Put the char back */
            break ;

        case '/' : /* Possible begin-comment */
            i = input(); /* Found a begin-comment */
            if ( i == '*' ) /* inside a comment */
            {
                err_prt(
" *** Warning : A begin-comment mark was found inside a comment0);
                err_prt(
"         at line %d. Could a previous comment be missing0, yylineno );
                err_prt(
"         an end-of-comment mark?0 ) ;
            }
            else
                unput( i ); /* Put the char back */
            break ;
    }
}
}

```

```

        case 0 : /* EOF reached, bad news      */
                err_prt(
" *** Error : Reached end of file while scanning for end-of-comment.0);
                return( FAILURE );
        )
    } while( temp == FALSE );
    }
    err_prt( " *** Error : Unable to lex this ->%s<-0,
            yytext );
    return( FAILURE ); /* Means error in lex terms. */
}
%% /***** SUPPORTING LEX ROUTINES *****/

```

```

/*****

```

Function : input()

Purpose : This function replaces Lex's input macro which prefers to read from stdin. In this version we read from the source code file. The file has been opened previously to yylex() being called.

Arguments : None

Returns : The character read.
0 - To indicate the end of input. (Lex requires this.)

```

*****/

```

```

static input()
{
    if ( yysptr > yysbuf )
        yytchar = U(*--yysptr) ; /* Get a character that has been
                                ** inputted already and unputted */
    else
    { /* Read from the input file.          */
        yytchar = fgetc( G_fp );
    }

    if ( yytchar == '0 )
        yylineno++ ; /* Increment the line count.      */

    if ( yytchar == EOF )
    {
        yylineno-- ; /* Assumes we hit a newline and then EOF */
        return( 0 ) ; /* Required by lex to indicate EOF */
    }
    else
        return( yytchar ) ; /* Just return the character */
}

```

```
} /* end of input() */
```

```
/******
```

Function : save_name()

Purpose : When an include file is found in a source file we save the name of the include file so that we will know what include files are used in a source file. The include files used are kept in a linked list and later converted into a set of facts about the source file.

Arguments : name - The name of the include file.
line - The line number in the source file where the "#include name" line appears.

Returns : FAILURE - If unable to create the new linked list node.
OK - If no errors are detected.

```
*****/
```

```
static save_name( name, line )
```

```
char *name ;
```

```
int line ;
```

```
{
```

```
    struct inc_files *ptr ; /* Temporary pointer to a node */  
    char *np ; /* Pointer into the name */
```

```
    /* Create the new node first */  
    ptr = (struct inc_files *)malloc( sizeof( struct inc_files ) ) ;  
    if ( ptr == (struct inc_files *)NULL )
```

```
    {  
        err_prt(  
            " *** ERROR : unable to allocate memory for include names linked list.0);  
        return( FAILURE ) ;  
    }
```

```
    /* Check the include file name. The preprocessor puts "./" on the  
    ** front of include file names when the include file is located in  
    ** the same directory as the source file. We want to remove the "./" */
```

```
    if ( (name[0] == '.') && ( name[1] == '/' ) )  
        np = &(name[2]) ;  
    else  
        np = name ;
```

```
    /* Now fill in the node */  
    ptr->name = malloc( strlen(np) + 1 ) ;  
    if ( ptr->name == NULL_CHAR )
```

```

    {
        err_prt(
            "*** ERROR : unable to allocate memory for include name.0);
        return( FAILURE );
    }

strcpy( ptr->name, np );
ptr->line = line ;

/* Now add the node to the linked list. Order is not important so add
** it to the top. */

ptr->next = G_includes ;
G_includes = ptr ;

return( OK ) ;

} /* end of save_name() */

```

/*****

Routine : sep_line()

Purpose : The C preprocessor leaves lines in its output file that indicate where a file begins and ends. This information is used to denote the starting and ending locations of expanded include file text into a source file. These lines have the form :
 # line_no "file_name"
 This routine takes a buffer containing such a line and extracts the line_no and file_name items and returns these.

Arguments : buf - A character buffer containing the line to be separated.
 line - The line number goes here.
 name - The file name goes here.

*****/

```

static sep_line( buf, line, name )
register char *buf ;
int *line ;
char *name ;
{
    register int i ; /* Used to move through buf */
    register int j ; /* Used to move through temp */
    char temp[255] ; /* Temporary holding area */

```

```

/* Extract the line number from the buffer, start after the # */
i = 2 ;
j = 0 ;
while ( buf[i] != ' ' )
    temp[j++] = buf[i++] ;
temp[j] = ' ' ;
*line = atoi( temp ) ;

/* Extract the file name from the buffer */
i = i + 2 ;
j = 0 ;
while ( buf[i] != ' ' )
    temp[j++] = buf[i++] ;
temp[j] = ' ' ;
strcpy( name, temp ) ;
} /* end of sep_line() */

/*****

Function : yywrap()

Purpose  : This is a function for lex. Lex will call it when EOF
          is detected.

Arguments : None

Returns  : 1 - Indicates done, besides lex requires this value.

*****/

static yywrap()
{
    return( 1 ) ;
} /* end of yywrap() */

```

A Knowledge Based Program Diagnostic System

by

Eric J. Byrne

B.S. University of Nebraska - Lincoln, 1983

AN ABSTRACT OF A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1988

Abstract

Debugging consists of taking a program that does not run correctly because of a software failure, locating the fault that is causing the failure, and rewriting the program so that it does run correctly. Locating the source of the failure and understanding it can consume as much as 95% of the debugging effort. Programmers become experts in program debugging over a period of years, yet debugging is perhaps the least liked activity of program development.

Human expertise in debugging can be implemented into a knowledge based program debugger (KBPD) that can assist programmers in the debugging process. The main purpose of a KBPD is to help analyze information and locate an error for the programmer. This relieves the programmer of much of the burden of the debugging process.

This thesis presents a design and implementation of a knowledge based program debugger called Amber. Amber is able to help a programmer debug programs written in the programming language C. Amber makes use of knowledge about the C language semantics and common C mistakes to help a programmer locate an error and determine the fault that is causing the error.