REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	Approved for public release; distribution is unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School	6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000

8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Cognitive Modeling and the Evolution of the Student Model in Intelligent Tutoring Systems

12. PERSONAL AUTHOR(S)   William Charles Hoppe

13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED FROM 08/90 TO: 09/92	14. DATE OF REPORT (Year, Month, Day) 1992, September 24	15. PAGE COUNT 123

16. SUPPLEMENTARY NOTATION   The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17.	COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Intelligent Tutoring Systems, Computer-Aided Instruction, Cognitive modeling, Student Model.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
This thesis deals with the design and development of a student module for an intelligent tutoring system (ITS). Within the context of this thesis a student module has two components: a student model, and a diagnostic component. We present an in-depth review of the history of ITS, on the design, development, and limitations of the system . We review the methods of cognitive modeling used in some historical systems and a variety of methods of program analysis used in previous works. We also discuss some Ada language issues that are both language specific and compiler specific, and related to tutoring in Ada. Our approach was to take an existing intelligent tutoring system, an Ada language tutor written in Ada, and extend the capabilities of the student module without altering the existing student model or diagnostic package. We designed heuristics to analyze the programming constructs involved in the student learning process and were able to generate information which may indicate the deficiencies and possible causes in the student model. The implementation of this idea required the integration of CLIPS and the Ada programming language. Using a small portion of the Ada language we have successfully diagnosed common programming misconceptions and missing concepts with novice Ada programmers.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [X] UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Yuh-jeng Lee	22b. TELEPHONE (Include Area Code) (408) 646-2361	22c. OFFICE SYMBOL CS/LE

DD FORM 1473, 84 MAR	83 APR edition may be used until exhausted All other editions are obsolete	SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED

Cognitive Modeling and the Evolution of the Student Model
in Intelligent Tutoring Systems

by

William Charles Hoppe
Captain, United States Army
B. S. General Engineering, United States Military Academy , 1983

ABSTRACT

This thesis deals with the design and development of a student module for an intelligent tutoring system (ITS). Within the context of this thesis a student module has two components: a student model, and a diagnostic component. We present an in-depth review of the history of ITS, on the design, development, and limitations of the system. We review the methods of cognitive modeling used in some historical systems and a variety of methods of program analysis used in previous works. We also discuss some Ada language issues that are both language specific and compiler specific, and related to tutoring in Ada.

Our approach was to take an existing intelligent tutoring system, an Ada language tutor written in Ada, and extend the capabilities of the student module without altering the existing student model or diagnostic package. We designed heuristics to analyze the programming constructs involved in the student learning process and were able to generate information which may indicate the deficiencies and possible causes in the student model.

The implementation of this idea required the integration of CLIPS and the Ada programming language. Using a small portion of the Ada language we have successfully diagnosed common programming misconceptions and missing concepts with novice Ada programmers.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I. INTRODUCTION

A.   BACKGROUND.

Computers have long been recognized for their enormous potential as an educational tool [WENG 87]. Intelligent tutoring systems are an outgrowth of the explosive and rapid advances in technology applied to the domain of computer-aided instruction. Artificial intelligence in the domain of education manifests itself in systems called intelligent tutors, which are a step beyond traditional computer-assisted instruction [BURN 88].

Intelligent tutors are smart systems designed to assist, coach, or teach, skills or knowledge traditionally left to the human educators. The methods applied in the past, and currently being investigated, are as diverse as the domains of skills and knowledge being tutored. These methods include the *mixed-initiative* dialogue of the tutor SCHOLAR [BARR 82], the *socratic* teaching paradigm of the tutor WHY [BARR 82],the *overlay* student model of the LISP tutor [ANDE 85], and the *reactive learning environment* tutor SOPHIE [WENG 87]. This list is not inclusive, nor is it representative of all the methods.

There are four basic components of an ITS: the student-machine interface, the expert module, the instructional or tutor module, and the student module. We need to make a distinction between *module* and *model*. Throughout the remainder of this work the term *module* will refer to the entire abstract component of the ITS. A module may have several sub-components. The term *model* will refer to a specific method, theory, or data structure. A model is normally a component of a module.

This thesis deals with the student module, more specifically, the modeling of the student's knowledge or lack of knowledge, in a particular application domain. Ideally the student module should contain all the aspects of the student's behavior and knowledge that have repercussions for the student's performance and learning [WENG 87]. The student module can be, and often is, divided into two parts: a student model and a diagnostic component. The student model is that component of the ITS that represents the student's current state of knowledge normally measured in respect to the expert module. The

diagnostic component attempts to uncover cognitive states from the student's observable behavior. The student model can be visualized as the data structure. The diagnosis is then the process by which the student model is manipulated. It is this tightly coupled nature of the two components that gives rise to the design issue referred to as the *student modeling problem* [VANL 88].

Cognitive modeling is in essence the ability of the tutoring system to keep track of the student's current state of knowledge and understanding of the domain subject matter. The goal of the tutoring system is then to use this individualized student model to tailor the system and adapt the instruction to the specific student's needs. It is the manipulation of the student model that gives the ITS it's advantage over other types of computerized learning systems [HOLM 91b].

B. OBJECTIVES

This thesis was embarked upon to answer three questions. How can the student model and other knowledge obtained by the ITS be best captured by the expert system shell CLIPS, in order to facilitate further manipulation of that knowledge? How can the knowledge obtained by the ITS be used to tailor the lessons of the system for the student? Can a rule based system be functionally integrated with Ada? These general questions evolved once the scope of the issues began to immerge. The key elements at issue here however remained the same. They are:

1. Augmentation of an existing student module in order to make it more robust;

2. Integration of the expert system shell CLIPS with Verdix Ada; and

3. Environment enhancement for CLIPS within the Emacs programming environment.

C. SCOPE

ITS Ada [DELO 91] is an existing Ada language tutor, written in Ada. It has a student model and diagnostic component that easily identifies syntactic knowledge issues to the student. These characteristics were designed into the system. Using the expert systems

shell CLIPS, we have attempted to make the student model more robust in its ability to model the cognitive processes of the student. This was to be done with as little alteration of the original tutor as possible. Due in large part to the size of the Ada language, only a specific subset of concepts have been undertaken in order to get a functioning prototype. A functioning prototype was necessary in order to ascertain the problems associated with the process, design issues related to changing the student model, and control issues arising from the interaction of Ada and CLIPS. The specific Ada constructs used in the prototype are the "if" conditional statement, and the three basic looping constructs: the simple loop, the "while" statement, and the "for" statement.

Many significant design decisions have to be addressed, including language compatibility, control flows associated with language compatibility, and the basic architecture of the student module. All of these issues, along with an extensive historical background of the types of student models that have been implemented, encompass the bulk of the work of this thesis.

D.   ORGANIZATION

Chapter II of this thesis provides an overview of intelligent tutoring systems in general with a specific focus on student models in several historical tutoring systems. Chapter III covers the area of program analysis and the issues that must be addressed from a student modeling perspective. Chapter IV provides an introduction of the features and functionality of the expert system shell CLIPS. Chapter V analyzes the complexity and language issues related to the Ada language in respect to the modeling of the students knowledge. Chapter VI contains the details of the design and integration of the student model adopted. Chapter VII discusses the implementation details of the student model, integration details of the two languages, and environment enhancements used in the development of the system. Chapter VIII summarizes the work accomplished and areas for future research. The appendices contain the source code for the interface package for

the source code for the interface package for CLIPS and Ada, the elisp code for the clips-mode within the Emacs programming environment, and the CLIPS code for system.

II. INTELLIGENT TUTORING SYSTEMS

A.   HISTORY OF INTELLIGENT TUTORING SYSTEMS

In order to look at the history of ITS it is essential to look first at the history of Computer Based Instruction (CBI) and Computer-Assisted Instruction (CAI). CBI has it's foundations in the development of machines that build interactive teaching devices. The term CBI can be traced back in the literature as far as 1927 [PARK 87]. Most of the historically significant work in CBI was done in 1950's and 1960's and used programmed instruction as the base model for the design of the systems. This is not to say that CBI ceased to exist after 1969. On the contrary, CBI is still an active area of research. It was the basic goals of CBI, to use machines to teach, that drove researchers to ask more of the instructional models they were designing.

Another significant factor in the development of CBI was the teaching methodology that was most prevalent during this period. Skinnerian behaviorism was the prevailing theoretical position. Skinnerian behaviorism was the belief in taking small, linearly traceable steps in the presentation of knowledge with immediate feedback to the student [PARK 87]. This made most CBI systems very flat, programs of instruction. It would be analogous to putting the pages from a book into a computer. The student reads the pages in order, and is then asked pertinent questions at the end of the reading. The CBI system provides all the data, questions, and feedback. However, there was no flexibility in the system. The co-location of the expert knowledge and pedagogical knowledge in the same module was a common characteristic of early CBI. The most apparent deficiency of the systems was their inability to handle questions the student may have had which were not covered in the review.

This frame-oriented and predefined format of CBI had it's obvious limitations. Attempts to overcome these limitations lead to the development of generative CAI in 1969 [PARK 87]. As the name implies, these systems could generate problems within the subject domain. The subject domains for these systems were restricted to arithmetic and

vocabulary-recall. These first systems generated problems without regard to the mastery level of the student. The next version of these types of systems provided drill and practice sessions with an ability to select problems commensurate with the student's ability. These systems were labeled adaptive CAI [SLEE 82]. These traditional CAI programs were still very statically organized and had only a limited ability to adapt to the student. They were designed to encompass both the domain specific knowledge and the teaching (pedagogical) knowledge of expert teachers [WENG 87].

CAI went through change in the early 1970s. There is no distinctive boundary between CAI and Intelligent CAI (ICAI) or synonymously, ITS. The desire to have the systems reason about the ability of the student and present the appropriate problem set leads to the development of ITS. One of the original goals of ITS was to take the existing adaptive CAI systems and extend their application domain, their power, and the accuracy of their predictions of the student's level of mastery [SLEE 82]. In order to achieve this goal a design change would be necessary. As previously noted, in CAI systems, the student model and pedagogical knowledge were all contained in the same module. ITS separated the domain knowledge, the teaching strategy and the student knowledge into separate modules. This separation allowed independent manipulation of the student knowledge to tailor the session to each student's needs [BARR 82]. The major intuitive difference between CAI and ITS is that in CAI the computer system possessed the inductive power of its human counterpart. In ITS the student interacts with the system. It is the students response that drives the dialogue.

There is another area in the history of ITS that needs to be noted. There are three major disciplinary domains that have an impact on ITS: education, psychology, and computer science. As stated earlier, ITS were designed to be educational tools. For this reason the field of education (and training) has a vested interest in these systems. The development of models of thought falls under the purview of psychology. Hence development of the student model, and the cognitive process itself, has peeked the interest and involvement of many psychologist. Finally, the use of artificial intelligence technology in computer

systems clearly falls in the discipline of computer science. ITS falls in the intersection of these disciplines (see Figure 1)[KEAR 87].
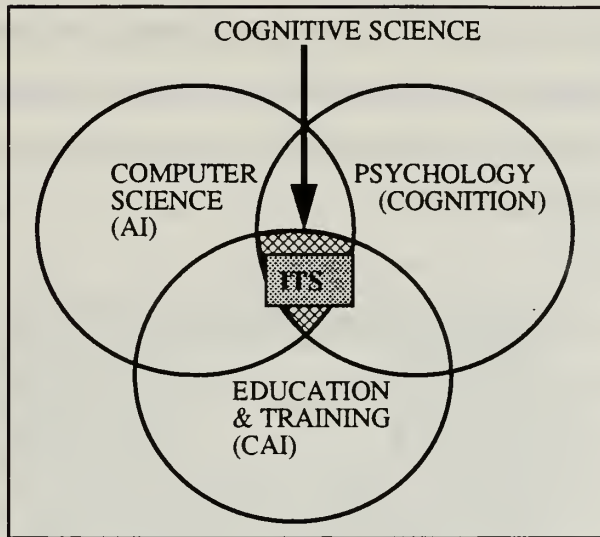
Figure 1: Intelligent Computer Assisted Instruction Domains.

The interdependency of the disciplines means each discipline has an impact on the domain of ITS. An independent discovery in one area may affect research in another.

B.   BASIC DESIGN OF INTELLIGENT TUTORING SYSTEMS

An intelligent tutoring system has what has been called an anatomy [BURN 88], a general architecture which most systems follow. It consists of an expert module, student diagnostic module, instructor module, and a human-computer interface (see Figure 2). The individual components of ITS have at least one of the previously mentioned disciplines in which research has been dedicated.

As stated in Chapter I, the expert module contains the domain knowledge. This is probably the most labor intensive portion of ITS development. It requires the discovery, collection, and codification of the domain knowledge from expert sources. These sources may be human experts or other resources. There are three traditional designs in this knowledge engineering process: black box, traditional expert systems, and cognitive models. The black box model does not require codification of the entire domain's

knowledge. The model revolves around an algorithm or series of algorithms that simulate the appropriate behavior correctly. The expert systems model requires the normal development cycle of any expert system to codify the domain knowledge. Cognitive models also require codification of the explicit domain knowledge. There are three cognitive models depending on the type of knowledge: procedural, declarative, and qualitative. The cognitive models require a simulation of the way the humans use the domain knowledge [ANDE 88].
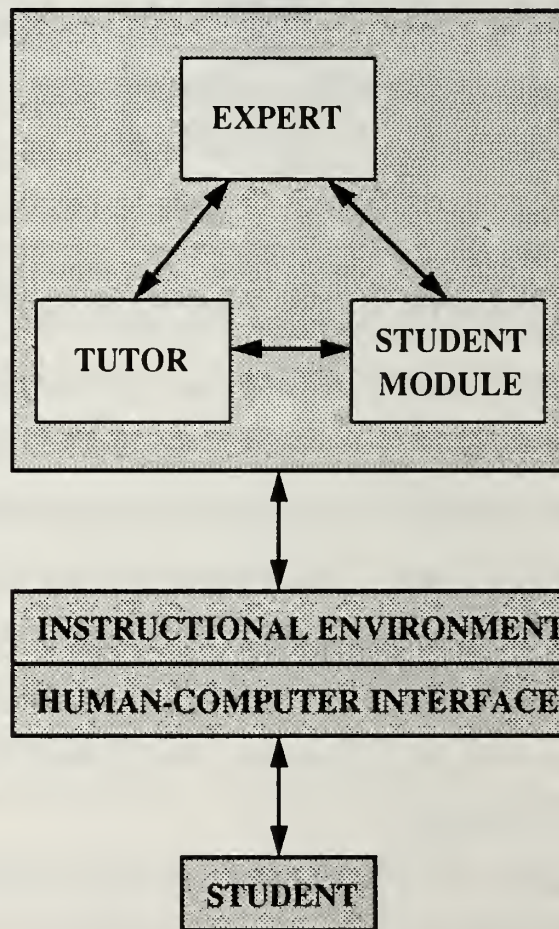
Figure 2: Anatomy of an Intelligent Tutoring System.

The tutor is the instructor module and is used to focus the system on the appropriate knowledge deficiency. This module deals with the teaching technique to be used in the tutor. There are several teaching strategies that can be used. The intended use of the

system, domain application, amount of time delay between the student error and a system response, the method of system response, are all factors in deciding this strategy. The most common strategies are: explanation, guided discovery, coaching, coaxing and critiquing. They range in theory from despotic to diagnostic. Despotic systems provide immediate feedback as soon as the error is detected. Diagnostic systems compare student and expert behaviors then reasons about how to elicit better learning performance [WOOL 91].

The instructional environment and human-computer interface facilitate the effective interaction of the tutorial session. This module can have the greatest impact on the usefulness and utility of the ITS. Since the student is interfaced to the ITS through this module, its design is critical to providing the appropriate learning environment. If it is poorly designed the effectiveness of the interactive session will suffer. If it is well designed it can enhance the learning process. The technique chosen will affect two aspects of the ITS. It will affect how the student interacts with the system and how the student interacts with the domain [MILL 88].

C.   THE STUDENT MODULE

The student module, as defined here, has two components: the student model and a diagnostic component. The two components are very tightly coupled and should be designed together. The student model can be thought of as the data structure that maintains the student's current state of knowledge. The diagnostic component manipulates this data structure in order to infer the student's cognitive state [VANL 88]. The purpose of this module is to make a hypothesis about any misconception or less than optimal performance strategies so the tutoring module can help the student correct the error or strategy [BARR 82].

1.   STUDENT MODELING

The student model contains the system's knowledge of the student. It should be as encompassing as possible and updated dynamically throughout the tutorial session [KEAR 87]. Van Lehn [VANL 88] developed a three dimensional classification of the

student model based partially on the structural properties of the student model and partially on the properties of the input available to the diagnostic component. The classification has three dimensions. Each dimension has three possible values.

Bandwidth relates to the input to the system. It is a rough characterization of the amount and quality of data input. The three values of bandwidth are: *mental states*, *intermediate states*, and *final states*. In an ideal situation we would know the mental states the student went through to get to the solution. Unfortunately, such an ideal situation is likely unattainable. The highest attainable bandwidth then is an *approximation* of the mental states where both physical and mental activity is available. The next highest bandwidth is intermediate states. Here only the physical activity is available. These states are observable in problem domains where the student can manipulate the problem into simpler problems enroute to the final solution. The lowest bandwidth is final states. Final states are just that, the final solution. With final states the systems gets no other information except the student's final solution.

In order to solve the problem some type of interpretation of the knowledge is necessary. Classification of the *knowledge type* dictates the type of interpretation employed. The type of knowledge in the student model can be *procedural-flat*, *procedural-hierarchical*, and *declarative*. Van Lehn did not include qualitative type knowledge in his theory. Declarative knowledge is that pure knowledge that has just been discovered [ANDE 90]. This is analogous to the information gained when reading a book. The new knowledge is declarative. Declarative knowledge interpretation searches over the entire knowledge base each time a decision needs to be made [VANL 88].

Procedural knowledge is attained through the compilation of declarative knowledge. Procedural knowledge can only be acquired through the use of declarative knowledge [ANDE 90]. This is analogous to taking the information gained from reading a book on programming in LISP and writing a LISP program. Procedural knowledge can be thought of as knowledge gained through the application of declarative knowledge. Procedural knowledge interpretation makes decisions based solely on local knowledge

[VANL 88]. To visualize this, let us consider the traversal of a tree. Once at a node the only choices available are the branches out of that node. Knowledge about the rest of the tree structure is not available to the decision process. Procedural-flat knowledge interpretation is done solely on the current state of the problem. Procedural-hierarchical knowledge interpretation allows the problem to be broken down into sub-goals.

The student model is basically the expert knowledge of the system plus the student's misconceptions and missing concepts. Misconceptions are concepts the student has but the expert does not. Missing concepts are concepts the expert has that the student does not. The representation of these differences make up the third dimension of the classification, *student-expert difference*. The student-expert difference dimension can take on one of the following values (models): *overlay*, *bug libraries*, and *bug part libraries*. The overlay model can only represent missing concepts. The student's knowledge is considered a subset of the expert's knowledge. Bug libraries are labor intensive to build but can represent both missing concepts and misconceptions. It is a predefined library of missing concepts and misconceptions. The bug part library constructs the bug from a library of predefined bug parts during diagnosis [VANL 88].

Knowledge Type / Bandwidth	Procedural-flat	Procedural Hierarchical	Declarative
Mental States		**Kimball's calculus tutor **Anderson's LISP tutor **Anderson's Geometry tutor	GUIDON
Intermediate States	WEST WUSOR	**The MACSYMA Advisor **Spade **Image	*SCHOLAR *WHY *GUIDON
Final States	**LMS **Pixie **ACM	*BUGGY *DEBUGGY *IDEBUGGY	*MENO *PROUST

Figure 3: The Space of Student Models.

Given the definitions of the three dimensions of the student model, each with three possible values, there are 3^3, or 27, possible student models within this framework. A table of some of the more well known tutors have been collated by Kurt Van Lehn and are reproduced here for comparison (see Figure 3). The student-expert difference dimension is represented by the asterisks: no asterisks indicates an overlay, single asterisks (*) indicates bug library, and double asterisks (**) indicates bug parts library [VANL 88]. This is not an inclusive list of all tutors and does not reflect any of the current work ongoing.

2. DIAGNOSIS

Diagnosis can have multiple purposes. This purpose normally requires a combination of the following three tasks: inference, interpretation, and classification. These steps are almost sequential. Inferences reconstruct internal processes and states on the basis of observable behavior. Interpretation makes sense of the inferences and observations by placing them in the context of the domain. Finally, classification makes relevant distinctions that will allow different actions to be triggered. Diagnosis in ITS is normally done for pedagogical purposes. In this context there are three levels at which this knowledge may be relevant: the behavior level, epistemic level, and the individual level. The individual level can be sub-divided into eight categories: architectural, learning, stereotypical, motivational, circumstantial, intentional, reflexive, and reciprocal [WENG 87]. We are now fully entrenched in the cognitive science discipline of ITS. This introduction to the terms will be the extent of our cognitive science discussion.

The introduction of terms used in diagnosis was necessary to reiterate the vastness of the development knowledge used for ITS. We felt this was necessary before moving on to any discussion on the practical issues of diagnosis. There are three basic issues that must be addressed when attempting to model the student's behavior: credit allotment, combinatorial explosion, and noisy data. These problems directly affect diagnosis.

The credit allotment problem stems from trying to assign credit or blame to appropriate construct within the diagnostic component (rule, procedure, path, etc.). The

component that identified the inconsistency in the student's knowledge should get the credit. Credit assignment can be hindered by non-resolution of multiple possible correct solutions or an inability to find a construct that matches the inconsistency. Systems that use a generative approach to diagnosis must account for the combinatorial explosion potential inherent in producing the primitive actions, plans, or rules that explain the student's behavior. Non-systemic mistakes are those mistakes that are not associated with the knowledge misconceptions or missing concepts. They are errors associated with fatigue, typographical errors, cognitive overload and the like. These non-systemic errors are called noisy data. Noisy data should not be handled the same way as errors made from a systemic misunderstanding of the domain knowledge [SLEE 82]. These three issues are a function of the structure of the data maintained in the student model, the granularity of the input data, and the diagnostic method employed.

In the assignment of credit problem, bandwidth and knowledge type have the most significant impact. The type of knowledge in the system will dictate the constructs used to represent misconceptions and missing concepts. The lower the bandwidth the harder the inference problem. The more data that is available the more information the system has to make the appropriate credit assignment. A combination of low bandwidth and procedural-flat knowledge generally results in general constructs. The more general the construct the harder it is to decide which construct receives credit.

In the combinatorial explosion problem the student-expert difference has the most impact. In the overlay model the choice is binary. The error is detected with the existing constructs or the error is not detected. This places more emphasis on the diagnostic technique employed. Bug libraries are finite but attempting to resolve compound errors can generate infinite combinations. Bug part libraries, normally significantly smaller than bug libraries, compound the problem of bug generation. The number of possible bugs becomes an exponential factor of the number of bug parts.

Bandwidth is the major factor in accounting for noisy data. The more input the systems has the more noisy possibilities the system must filter. With little analysis it is

clear that the bandwidth, knowledge type and student-expert difference values directly impact not only on the existence, but the severity of the problem.

The diagnostic strategy also has an impact on the severity of these problems. Kurt Van Lehn details nine of the most prevalent strategies: model training, path planning, condition induction, plan recognition, issue tracing, expert systems, decision trees, generate and test, and interactive diagnosis. Replacing the systems from Figure 3 with the diagnostic strategy of the systems generates Figure 4 [VANL 88].

Knowledge Type Bandwidth	Procedural-flat	Procedural Hierarchical	Declarative
Mental States		Model tracing	
Intermediate States	Issue tracing	Plan recognition	Expert system
Final States	Path finding Condition Induction	Decision tree Generate and test Interactive	Generate and test

Figure 4: Diagnostic Strategies.

We will dispense with the individual analysis of each strategy. However, it is obvious that those strategies that have multiple paths or are of a generative nature must deal with the combinatorial explosion problem.

D.   PREVIOUS SYSTEMS AND THEIR STUDENT MODULES

In order to see how all these factors come together in actual systems we will look at the student modules of four systems. We will look at these systems from the perspective of the three dimensions of the student model and the diagnosis technique employed.

1. THE LISP TUTOR

The LISP tutor was designed at Carnegie-Mellon University in 1985. Its purpose was to teach novice users the LISP programming language. The system was menu driven and interactive. The system received all keystrokes as input. It also contained a detailed cognitive model of how students learn to program [ANDE 85]. Since both the physical and mental activity of the student are available, the system has a *mental states bandwidth*.

The development of a LISP program starts some place. From that starting point only the current state of the solution is relevant to the next step. Most programmers do not start a function (function in LISP, procedure in an imperative language) until the completion of the current one. This classifies the knowledge type as *procedural*. But, is it hierarchical or flat? The more pertinent question is whether the goal can be divided into sub-goals? A LISP program can be attacked in this manner. The LISP tutor does attack the goal in this manner. Therefore, the knowledge type of the system is *procedural-hierarchical*.

There were approximately 325 production rules in the original version of the LISP tutor that were dedicated to planning and writing LISP programs. There were approximately 475 buggy versions of those same production rules. These bug parts were designed so they could account for all possible student deviations from the ideal solution path [ANDE 85]. The *student-expert difference* in this systems was modelled by a *bug part library*.

The LISP tutor follows the student input as it is entered. The tutor analysis the student's input and tries to match it to a production rule. If the production rule is one of the correct rules the system remains silent waiting for more input. If the production rule is one of the buggy rules then the system response with advice. The key here is that the system has an ideal model of what the system would like the student to know. It can generate a correct path through the sub-goals based on the specification of the problem. It compares the student stages of program development path with the ideal path. A deviation in the path is an error [ANDE 85]. This is a *model tracing* diagnosis strategy.

2. PROUST

PROUST was designed at Yale University in 1985. It's purpose was to help novice programmers learn how to program in PASCAL [JONS 85]. Figure 5 [JONS 85] shows how the system interacts with the student. The student gets a problem. The student solves the problem. PROUST analyzes the solution and provides feedback.

```
Problem:  Read in numbers, taking their sum, until
the number 99999 is seen.  Report the average.  Do
not include the final 99999 in the average.

    1    PROGRAM Average( input, output );
    2    VAR Sum, Count, New, Avg:  REAL;
    3    BEGIN
    4        Sum := 0
    5        Count := 0;
    6        Read( New );
    7        WHILE New <> 99999 DO
    8        BEGIN
    9            Sum := Sum + New;
    10           Count := Count + 1;
    11           New := New + 1;
    12       END;
    13       Avg := Sum / Count;
    14       Writeln( 'The average is ', avg );
    15   END;
```

PROUST output:

It appears that you were trying to use line 11 to read the next input value. Incrementing NEW will not cause the next b value to be read in. You need to use a READ statement here, such as you use in line 6.

Figure 5: Example of a Buggy Program in PROUST.

The input to the system is the completed PASCAL program. The system has no idea of the cognitive states the student went through to get to the solution. All the physical activity is not available to the systems. The **only** input the system receives is the final program solution. This is an example of *final states bandwidth*.

The *knowledge type* in PROUST is declarative. Recall the definition of declarative knowledge. Declarative knowledge is that pure knowledge that has just been

discovered [ANDE 90]. In PROUST the student solution suddenly appears. There is no way for the system to reconstruct the process by which the student reached that solution. It is raw data to the system.

The *student-expert difference* model in PROUST is a *bug library* of known types of bugs [WENG 87]. The bugs are managed as rules that explain mismatches in paths between student solution and expert solution. Only these error rules are reported to the student. The system has a set of transformation rules to preserve the equivalence of two code segments. If an inconsistency occurs for which the system does not have an error rule or transformation rule, it is unable to reconcile the difference and conducts a partial analysis. PROUST tells the student about those parts of the solution it did not understand [JONS 85].

PROUST uses a generate and test diagnostic strategy. The possible programming plans are generated and tested to see how well they match the student solution. Those possible explanations of the student bugs are evaluated to determine which solution matches, near-misses, or misses [JONS 85]. The plan that best matches the student solution is tried first.

3. SCHOLAR

SCHOLAR was a product of Bolt Bernek and Newman, Incorporated in 1970. It was designed to tutor students on simple facts about South American geography [BARR 82]. It was one of the early, mixed-initiative, tutors. Mixed-initiative tutors were those that both the student and the system could initiate the dialogue. Figure 6 [WENG 87] is an excerpt from a dialogue with SCHOLAR.

SCHOLAR allows the student to initiate the dialogue with the tutor. Allowing the student to control the initiative does not allow us to capture the cognitive states that may have lead the student to his last answer or the current question. Since we do not know what direction the student may take, attempting to account for all possibilities leads to the familiar combinatorial explosion problem. SCHOLAR does not try to model the students

cognitive state by priori assumptions about the possible state space. It takes all the physical input from the keyboard; all physical input is made available to the system. This is defined as *intermediate states bandwidth*.

```
SCHOLAR:  Approx what is the area of Brazil?
 Student:  2,500,000 square miles.
SCHOLAR:  Wrong.  Please indicate if the following statement is
          correct:  the area of Paraguay is approx 47,432 square
          miles.
 Student:  What is the area of Brazil?
SCHOLAR:  The area of Brazil is approx 3,600,000 square miles.
          Now answer the question you didn't answer before.
 Student:  False.
SCHOLAR:  Rephrase your statement. I can't understand it.
 Student:  Incorrect.
SCHOLAR:  Very Good.  The area of Paraguay is approx 157,000
          square miles.  The language in Brazil is French.  Right
          or wrong?
 Student:  Wrong.
SCHOLAR:  You are correct.  The language of Brazil is
          Portuguese.   Is it true....
```

Figure 6: Excerpt from a Dialogue with SCHOLAR.

SCHOLAR's *knowledge type* is declarative for the same reasons that PROUST had a declarative knowledge type. The information provided to the system is new knowledge each time the student and systems interact. The interpretation strategy forces the system to review the entire knowledge base in order to properly construct a solution.

SCHOLAR uses the socratic method of tutoring. It does not provide a direct response to incorrect answers but uses a set of rules to guide the student to discover his own error. This set of rules is basically a *bug library*.

SCHOLAR uses a semantic network structure for knowledge representation. The theory was that the system and student should have similar representations of knowledge. The author, Jaime Carbonell, believed that semantic networks were an accurate model of how humans store and access knowledge. Carbonell believed a complete semantic network would represent the perfect student. Information gained during the dialogue would adjust

(perturb) this model to reflect the student's actual performance. This was the theory, the actual system was not complete in that the semantic network was not complete. The diagnosis was carried out with incomplete sets of knowledge. The *student-expert difference* was an *overlay model* [WENG 87].

4. SARAH

SARAH is not an ITS in the context presented here. It's design and function however mirror that of an ITS. It is a system developed in France to assist in the rehabilitation of aphasic patients. Aphasia is a language impairment resulting from brain damage. The patient does not understand that they have a problem because they do not have the language skills to understand the explanation. The system presents grammatical exercises to the patient so he can discover his own errors [MASS 90].

SARAH uses a clinical diagnosis, and knowledge gained during the dialogue to build the patient (student) model. The clinical diagnosis provides the cognitive state, the mental activity. The patient's input is the physical activity input. These are the two components of a *mental states bandwidth*.

SARAH has a *procedural-flat knowledge type*. The data coming into the system is already classified in the context that the system already knows something about the patient's condition. The system is getting feedback on the application of that knowledge. Decisions made about the interpretation of the error is dependent only on the current state of the problem.

The student-expert difference is not one we have discussed. SARAH uses a model based on machine learning: program synthesis from examples of input and output. The idea is to synthesize a logic program from the student model. Bugs in the logic program would then represent the patients bug. The nature of the disease allows categorization of the types of potential bugs into one of three categories: lost grammatical processes, misused processes, and incorrect processes [MASS 90].

The diagnostic strategy is to perform *program induction* on the logic program synthesized from the patient model. There are two algorithms used in the system: Model Inference System (MIS) and Program Diagnosis System (PDS). MIS induces a Prolog program satisfying all the facts provided to the system. MIS relies on PDS to detect errors in one of two cases. Incorrect Prolog clauses in respect to facts in the fact base. Missing Prolog clauses needed to satisfy facts in the fact base when the program fails [MASS 90].

SARAH is an example of the ITS design in a specific application domain. Several of these non-traditional techniques used in SARAH are possible in part by the nature of the domain of application.

5. *ITS* Ada

a. *Architecture*

ITS Ada [DELO 91] contains the four basic components of what has been traditionally called an intelligent tutoring system as outlined in Chapter I. Figure 7 gives a logical schematic representation of the architecture.

The instructional module is the human-computer interface. Since it's design was aimed at high portability between different hardware platforms, the implementation utilized only text input and output.

The expert module contains the domain knowledge for the tutor. The main component is an Ada parser produced using the AYACC compiler generator from the University of California, Irvine [DELO 91]. The parser was modified to accept program segments as input instead of complete programs. The rest of the files were produced by the compiler generator for the parser's use.

The diagnostic module is a single Ada package which takes input from the student and expert and creates a meaning list. This meaning list is first checked for syntactic correctness. If it is syntactically correct the diagnostic module compares the meaning lists of the two solutions. Structural discrepancies in the student meaning list from the expert meaning list as evaluated from inside out constitutes an error. If there are

elements of the meaning list in the expert solution but not in the student solution the element is identified as a missing concept and an error is raised. If there is a element of the student meaning list that is not in the expert meaning list the element is identified as a misconception or unnecessary and an error is raised. The expert meaning list becomes the template by which the structure of the student meaning list is compared.
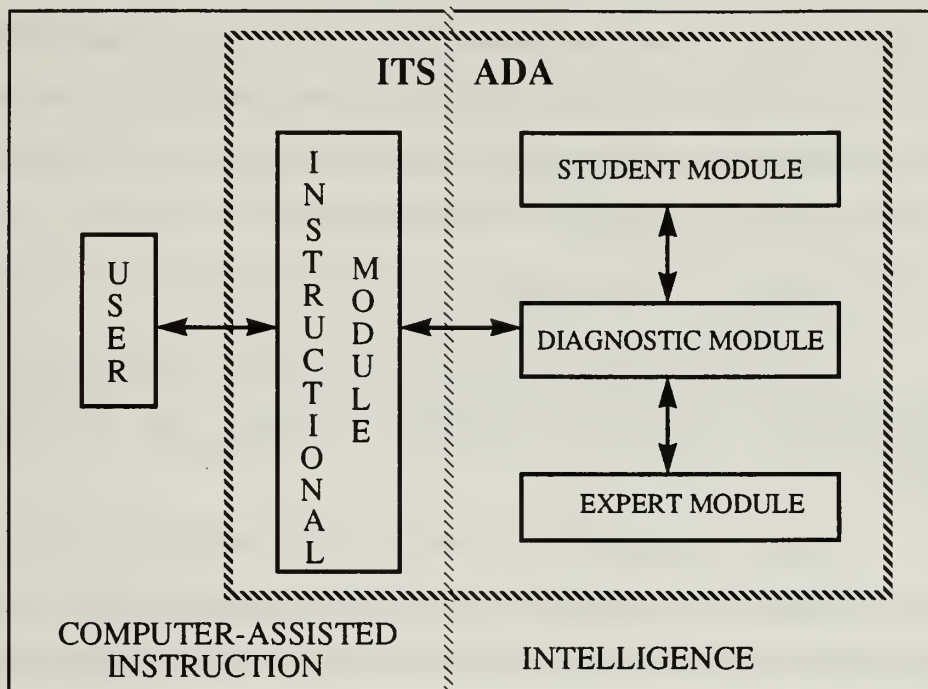
Figure 7: Architecture of *ITS Ada*.

The student module is the student model. It is an Ada data structure contained in an external file. The file is accessed each time a user begins a session and contains the students history. The file basically contains an array of records for each student known to have or currently using the tutor.

b. *The Student Module*

The student module in *ITS Ada* is the student model. As previously stated the student model for *ITS Ada* is an array of records saved in an external file called "user.dat". This file is accessed when the student invokes the tutor. The tutor goes to the

file and searches it to see whether the student has used the tutor previously. If so the data in the student's file is used to start the session where the student level of "mastery" was last demonstrated. This automatic placement can be overridden. If this is the first time the student has used the tutor a new record is created.

The student's level of "mastery" is represented by an Ada enumerated type: *unknown, exposed, demonstrated.* Each of the topic areas, 12 total, is represented in this record. Once a topic area has been introduced the value *exposed* is added to the student model. In each topic area there are five predefined problems to test the student's knowledge of that topic. If the student gets two of the five test problems correct the student's level of mastery is updated to reflect *demonstrated.*

c. Evaluation of "ITS Ada"

In evaluating *ITS Ada*, we will use the three characteristics of tutoring systems presented in Chapter III, bandwidth, knowledge type, and student-expert difference.

Recall that bandwidth is a measure of how much of the student's activity is available to the diagnostic program. *ITS Ada* presents completed student solutions in the form of Ada code segments. This is similar to the approach used in PROUST [LEWI 85] for the Pascal programming language. Since only the answer is available the bandwidth is classified as final states. Given how the student model is updated, this presents a problem. *ITS Ada* will diagnose a problem and present a new problem. Eventually the student, if persistent in the error, will repeatedly see all five problems. Given enough chances the student should eventually get the required two problems of five correct and move on to the next topic area without understanding his errors.

ITS Ada basically uses a template, the expert meaning list, and attempts to match the student's meaning list to this template. This use of templates allows us to classify *ITS Ada*'s knowledge type as declarative. *ITS Ada* uses it's entire knowledge base, represented by the parser, to find an answer to the student's problem. The system then uses

that knowledge to draw a conclusion about the student's level of knowledge. The tutor has no knowledge of how a student reached this current state.

Finally, the student-expert difference. *ITS Ada* uses an overlay representation of this difference [DELO 91]. The student's knowledge is assumed to be a subset of the experts knowledge.

ITS Ada will catch all the syntactic errors presented by the student. This statement assumes the correctness of the parser generated by AYACC. However, any inference made by the system about the cognitive process the student went through would have little evidence to support the conclusion. The system has had no access to any of that knowledge. Cognitively, any error inferred by differences in the meaning lists is attributable to an insufficient mastery of individual skills related to the problem domain. The system does not consider the possibility of misconceptions of correct domain knowledge[WENG 87]. The system does not take into account the possibility that the error was made because of fatigue or other human factors, noisy data, etcetera. [SLEE 82].

d. Limitations

This system will catch all syntactic errors in the language. It attempts to finds inconsistencies in the meaning lists however this is dependant on the expert solution. However, there could be multiple correct implementations of the problem within a given language construct. Basing the logical correctness of the program on a single structural match limits the possible solution set to one solution.

There is no semantic analysis of the student solution. The reason for this is the amount of knowledge available to the system about how the student arrived at the solution. This low bandwidth of information about the student's cognitive process is the limitation upon which we focused the efforts of the enhanced student module.

III. PROGRAM ANALYSIS

When a programmer debugs a program he has a considerable amount of knowledge about the program, it's design, and it's intended use. Specifically, he knows what problem was to be solved. He knows the flow of control of the program. He knows how to trace down an error in the program if it does not behave as intended. If he encounters a bug, he knows how to correct it. Ideally an automated program debugger will have the same knowledge [LEE 89]. Automated program debugging decomposes into two categories: syntactic and semantic. A reasonably good parser will handle the syntactic analysis of the program and identify any syntax errors. Semantic or logic errors in the program are significantly more difficult to detect [JONS 90]. Correcting semantic errors is at least as difficult as locating the errors and requires different skills and strategies. Since syntactic analysis is well defined in parser and compiler design we will focus on some of those techniques of semantic analysis.

We will review four noteworthy approaches to program analysis. These four systems are unique in their approach and provide a broad overview of some of the implementations used in the past.

A. PROUST

PROUST uses plan generation and recognition as the basis of program analysis [JONS 85]. There are three types of plans: programming plans, variable plans, and control plans. These plans can be thought of as templates. PROUST takes these templates and generates a plan from scratch each time a new problem is presented. A limited bug library of bug rules is maintained which contains specific common misconceptions. Instead of a large bug library PROUST has a plan library that associates task goals with plan templates. Heuristics are used to both control the plan generation process, thus avoiding the combinatorial explosion problem, and the plan selection. PROUST also has what are called plan transformation rules. These rules are essentially rewrite rules that account for equivalent variations of correct plans.

PROUST takes in a non-algorithmic description of the problem and generates the design and implementation steps necessary to write the program. The student's program is then compared to the generated plan. Differences in the plan and the program are first compared against the transformation rules to determine if the segments are equivalent. If there is a transformation rule that activates, the difference is not labeled as a bug. If no transformation rules match, the bug rules are compared to find common bugs. In this case either a bug rule activates and generates a bug report or the difference is not reconcilable and is generated as a bug report [MURR 86]. Another effect of unreconcilable differences is that PROUST now can only make a partial evaluation of the program since it can no longer account for future differences in the overall plan of the program. PROUST continues it's analysis but with less confidence in any inference made about plan differences.

B. LAURA

The LAURA [ADAM 80] system attempts to prove equivalence of two programs. LAURA takes in as input two FORTRAN programs. The program model is the *expert* solution to a problem. The other input is the student program. The initial goal of LAURA is to prove the student program is equivalent to the program model. If this fails then LAURA attempts to debug the student program.

LAURA transforms both programs into a graph representation. These graphs represent the calculus process that the program was intended to solve. The nodes in the graphs represent arithmetic operations or equations. The arcs in the graphs represent the control flow. The transformation is done in four steps: standardization, separation of variables, composition, and linear induction. The result is a graph for each program that if solved produces a calculus function. If the two programs are equivalent the solutions of the two graphs are identical.

The first step in comparison tries to bind variables, nodes and arcs in order to get the two graphs to match. Normally an exact match is unlikely since the structure of the two

graphs are rarely the same. The first level tries only to resolve the components of the graphs. The second step in comparison attempts to transform the structure of the two graphs and then does a comparison of the structure. If the result of the previous steps results in a match of the graphs the two programs are equivalent and LAURA halts. If they are not the same LAURA attempts to diagnose errors. The process in diagnosis is the same as before except it is done with more complex conditions [ADAM 80].

C.   ALGORITHMIC PROGRAM DEBUGGING

An example of a system that uses intermediate results from the user is the system written by Shapiro [SHAP 83]. This system applies the techniques of inductive program synthesis to the automatic correction of programs. It is based on interactive dialogue with the author of the Prolog program to be analyzed. The system simulates the execution of the target program on a given input. It generates results for each procedure call. The programmer is queried to determine the correctness of some of the intermediate results produced by the procedure call. The basic algorithm of the program is in Figure 8 [WERT 87].

reading of the program **P** to be corrected
  repeat
    read the next example of input/output
    while **P** behaves in an incorrect manner *do*
        identify a bug in **P** by use of a diagnostic algorithm
        correct the bug by use of a correction algorithm
    write
  until there are no more examples

Figure 8: Basic Algorithm of Shapiro's Algorithmic Debugger.

The system consists of two parts: a diagnostic algorithm and a correction algorithm. The diagnostic algorithm highly interactive and detects three types of errors:

1. Termination with an incorrect result;

2. Termination but fails to return a result; and

3. Non-termination.

The first two errors are detected directly from the input/output pairs and the dialogue with the programmer. The third error type is detected by an induction algorithm on the arguments of the procedure calls. The correction algorithm is an inductive inference algorithm like those used in program synthesis [WERT 87].

D. TALUS

TALUS is an automated debugger for LISP programs [MURR 86] designed to be used in a tutoring environment. TALUS represents knowledge about the correct programming solution as a program, called the reference function. It then uses the Boyer-Moore theorem prover to construct an induction proof that the reference function is equivalent to the student's function [ALLE 91]. The debugging process has four steps: program simplification, algorithm analysis, bug detection, and bug correction.

Program simplification takes the student functions and simplifies them by using transformations that maintain the computational equivalence of the original function. The simplified functions are parsed into frames which are matched to frames in the task representation. This allows identification of the student algorithm and pairing of the student functions with reference functions. Each of the function pairs has the case splitting performed for an inductive proof of equivalence. Verification conditions that establish the base case and induction steps are generated. Violations of the verification conditions identify a bug in the function at the point the induction proof fails. In order to correct the bug, violated verification conditions are repaired by altering the student function. Reference code supplies the corrections which are traced back through the transformations to the original student code.

E. SUMMARY

There is another topic which directly impacts the program analysis method decision and that is the source or type of knowledge required. A caveat to this issue is how much, if any, of that knowledge is supplied by the user. There are several alternative solutions all

27

with impacts on the analysis technique. Program analysis requires at least two pieces of knowledge [JONS 90]. The obvious piece is the code to be analyzed. The not so obvious piece is knowledge about the intended use of the code. This piece can be provided by the user in terms of a high level specification. It can be generated by the program analysis algorithm or it can be inferred from interaction between the analysis program and the user.

The source of knowledge for program analysis in PROUST is the plan library. The plan libraries are a codification of the expert knowledge. PROUST only requires a completed Pascal program from the user. It does not rely on any interaction or additional information. PROUST is algorithmic in its program analysis.

LAURA is also algorithmic in its program analysis. It's source of knowledge is the program model (expert solution). Prior knowledge of the exact problem to be solved and a sophisticated transformation algorithm are the key elements of the program analysis in LAURA. It too does not require any additional information from the

Shapiro's Algorithmic Program Debugging is a combination of algorithmic program analysis and user input. The programmer verifies intermediate solutions before the algorithm continues to the next stage of program analysis. After providing the initial solution program as input the debugger needs intermediate feedback from the user. The source of knowledge is actually the programmer.

Talus is also algorithmic in its program analysis. It's source of knowledge is the Boyer-Moore theorem prover and a reference function (expert solution). Once the student solution is provided the student provides no other input to the system.

ITS Ada in it original configuration was dependent solely on the program for it's program analysis knowledge. There was no other input to the system. The parser generated by the AYACC compiler generator was the sole source of knowledge. This was similar to PROUST except the diagnostic package of PROUST could generate a correct solution using path planning. The diagnostic package of ITS Ada can not generate a correct solution nor can it diagnose errors outside a very narrow structural definition.

Our enhanced student module does not alter the syntactic analysis of the original *ITS* Ada tutor but provides more diagnostic capability. This increased diagnostic capability is possible by increasing the knowledge available to the system. This new knowledge is in the form of a rule base and the intermediate solutions states. The student still provides only the solution to the problem but the system captures more of the student's cognitive process which enhances the inference capability of the system.

IV. CLIPS

A. DESCRIPTION OF THE CLIPS LANGUAGE

CLIPS stands for the "C Language Integrated Production System". It is a product of the National Aeronautics and Space Administration's Software Technology Branch [NASA 91a]. It was designed to provide high portability, low cost, and easy integration with external systems [GIAR 89]. To this end it has been implemented on a varied number of different architectures from personal computers to SPARC architectures.

The "C" in CLIPS has been made somewhat misleading. The original system was implemented in the C programming language. However, since it's original development Ada has appeared on the scene. The Software Technology Branch has tried to keep an Ada version in production to keep pace with the C version. CLIPS has undergone numerous revisions and upgrades. This thesis uses the C version of CLIPS, version 5.0, compiled on Sun workstation with an ansi C compiler.

B. HISTORY OF CLIPS

CLIPS dates back to 1984 and was a product of necessity. Prior to CLIPS, NASA, as well as the rest of the artificial intelligent community, relied almost exclusively on LISP as the base language of all expert systems applications. The low availability of LISP across a variety of platforms hindered portability and forced the rewrite of the entire system each time. State-of-the-art LISP tools and hardware were expensive. And finally, the difficulty experienced with attempts to integrate LISP with other languages made embedded applications difficult.

A prototype version of CLIPS was completed in 1985 and was used primarily as a training tool to research the issues surrounding construction of expert systems tools. It was originally modelled syntactically after the ART expert system tool developed by Inference Corporation. CLIPS version 3.0 was released to organizations outside NASA in 1986 and went through several revisions: version 4.0 released in early 1987, version 4.1 released in

late 1987, version 4.2 released in 1988, version 4.3 released in 1989, version 5.0 released in 1991, and version 5.1 was released in 1992 [NASA 91a].

Originally CLIPS was only a forward chaining rule language methodology based on the Rete Algorithm that had inferencing and representation capabilities similar to that of OPS5 [GIAR 89]. Version 5.0 added both a procedural and object-oriented programming paradigm to the language [NASA 91a]. We will not be dealing with the CLIPS Object Oriented Language (COOL) in this thesis. The OOP paradigm certainly has interesting applications within the scope of this thesis. Throughout the remainder of the description of CLIPS, COOL constructs will be mentioned but not elaborate.

C.   THE CLIPS ENVIRONMENT

There are three basic components to what we term the CLIPS environment; the raw data, the knowledge base, and the inference engine [GIAR 89]. The raw data is found in the form of a fact-list and once asserted is global. The knowledge base consists of all the rules in the system. The inference engine is the control mechanism for the system. The normal execution cycle is shown in Figure 9.
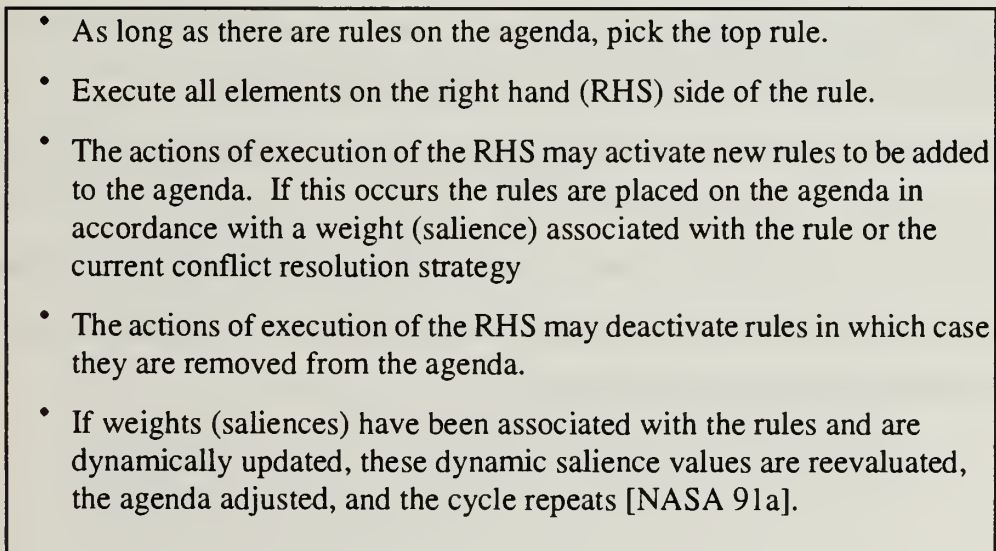
- As long as there are rules on the agenda, pick the top rule.

- Execute all elements on the right hand (RHS) side of the rule.

- The actions of execution of the RHS may activate new rules to be added to the agenda. If this occurs the rules are placed on the agenda in accordance with a weight (salience) associated with the rule or the current conflict resolution strategy

- The actions of execution of the RHS may deactivate rules in which case they are removed from the agenda.

- If weights (saliences) have been associated with the rules and are dynamically updated, these dynamic salience values are reevaluated, the agenda adjusted, and the cycle repeats [NASA 91a].

Figure 9: CLIPS Normal Execution Cycle.

The agenda can be thought of as a stack of rules whose RHS have been satisfied but the rule has not had a chance to activate (fire). A new rule's location on the agenda is determined by the application of the following three rules in the order they appear (see Figure 10).

> - The new rule is positioned above all other rules whose salience is lower than the new rule's salience and below all other rules whose salience is higher.
>
> - If the rules have an equal salience value, the current conflict resolution strategy is used to determine the new rules location on the agenda.
>
> - If a rule (or several rules) is activated and no ordering has been specified to this point, then the rule is "arbitrarily (not randomly)" placed on the agenda in relation to the existing rules [NASA 91a]. The order the rules were written will have an affect.

Figure 10: Algorithm to Add a New Rule to the CLIPS Agenda.

CLIPS has seven conflict resolution strategies: depth, breadth, simplicity, complexity, lex, mea, and random. The default strategy is depth but may be changed even during execution. Briefly, the methodology of each strategy is described in Figure 11.

This is the basic definition of the runtime environment of CLIPS. It's flexibility is apparent and was one of the main reasons for it's use in the thesis.

For those programmers familiar with the programming languages LISP and Prolog, this is a familiar cross between the two. CLIPS code takes on a LISP appearance recognized by it's parenthetic closure. It also resembles LISP in that it has primitive data types, functions and constructs that define the operation of it's rules. It reads like a Prolog statement, from left to right, if p then q.

- Depth: new rules are placed above all rules of equal salience.
- Breadth: new rules are placed below all rules of equal salience.
- Simplicity: among rules with the same salience, new rules are placed above all rules whose LHS has the same number or more comparisons to be performed. This is called specificity.
- Complexity: among rules with the same salience, new rules are placed above all rules whose LHS has the same number or fewer comparisons to be performed.
- LEX: This is the same strategy of as that of the OPS5 strategy of the same name.
- MEA: This is the same strategy of as that of the OPS5 strategy of the same name.
- Random: each rule is given a system generated random value that is used to determine the new rules placement among other rules with equal salience [NASA 91a].

Figure 11: CLIPS Conflict Resolution Strategies.

CLIPS, like Prolog, has built in operators (assert and retract) that allow the manipulation of the fact base. The major difference from Prolog is that the LHS can contain more than one clause. For example, if the rule has the following description:

There is no error present, the token 'for' is present, and the variable to be incremented is on the LHS of an assignment statement, then change the error status to reflect a concept error and print a message.

The CLIPS code for this rule may look like this:

```
(defrule for-loop-concept-error1
  ?error-status <- (no error)
  ?token1 <- (token for)
  ?variable1 <- (variable ?v)
  (statement ?v := $?)
=>
  (printout t "The Ada for loop automatically increments
  its counter. Within the sequence of statements the
  loop parameter is a constant. Hence a for loop
  parameter is not allowed as a variable on the LHS of an
  assignment statement (LRM 5.5.6)." crlf)
  (retract ?error-status)
  (assert (concept error)))
```

All the clauses to the left of the implication symbol, "=>", must be true before the rule will activate and go on the agenda. If it is selected from the agenda to fire, the clauses to the right of the implication are executed. The similarities of CLIPS to LISP and Prolog are apparent. To review the results of this rule in the CLIPS run-time environment see Appendix C.

It is because of its portability, extensibility, capabilities, and low-cost that as of 1991, CLIPS had over 3000 registered users including all NASA sites and branches of the military, numerous federal bureaus, government contractors, 160 universities, and many companies [NASA 91a]. It is for these same attributes that we chose it for this thesis.

V. COMPLEXITY AND LANGUAGE ISSUES

A.   COMPLEXITY ISSUES

In Chapter II the issue of combinatorial explosion arose. In an intuitive sense, most of us have a definition in mind that tells us that this phenomena makes the problem big. The question is how big? Even more fundamental is the question of the problem even being *decidable* (solvable). Trying to answer the decidability question brings up the *halting problem*. The use of formal logic to prove theorem correctness raises the question of *satisfiability*.

1.   DECIDABILITY

What do we mean when we say something is *undecidable* or *unsolvable*? There are two possibilities: (1) it is intuitively undecidable, or (2) it is undecidable by Turing *machines* (TM's) [LOEC 72]. In the case of TM's, the set of recursively enumerable (r.e.) languages are countably infinite and therefore do not contain some languages. It has been proven that an extended TM model computes all and only the r.e. languages [HOPC 79]. Therefore there are languages (problems) that exist that can not be computed by a TM [LOEC 72].

In the intuitive case it depends on whether or not one accepts or rejects Church's thesis. If one accepts it, then one believes that the intuitively computable functions are precisely the Turing-computable functions. If one rejects it, then one has in mind some other model of computation that may accept languages (problems) outside the r.e. set. However, in this new model, effective instructions for accepting the language must be provided [HOPC 69]. Effective in this context means the machine must be able to carry out the instructions.

Take as given the following two statements: (1) that the combinatorial problem is finite and (2) the previous definition of decidability. Given those two statements, the combinatorial problem is *decidable* both intuitively and by TM. However, the resources necessary to conduct the computation may not be available or reasonable.

2. THE HALTING PROBLEM

The halting problem basically asks the question of whether or not a TM stops. Formally, assume that we have some mapping function g which maps the positive integers onto the set of TM's; also g is a totally computable function. Given a positive integer, n, as an input to g, $g(n)$ denotes the nth TM. The function $g(n)$ is also written A_n. If the TM A_n takes as input x, there are two possible outcomes: $A_n(x)$ eventually halts, or $A_n(x)$ runs forever. The halting problem is the problem of deciding, for any number n, whether $A_n(x)$ halts or not [HARR 78]. Look at this a different way. Suppose we have a program that determines whether another program terminates. There is one function inside this program called **halts**. The assumption is that **halts** exists. The proof follows:

Given the program:

$$P(Q) = \text{if } \textbf{halts} \ (Q \ (Q)) \text{ then}$$

loop forever

else false.

$Q(Q)$ halts implies $P(Q)$ does not halt. By the contrapositive, $P(Q)$ halts implies $Q(Q)$ does not halt. $Q(Q)$ does not halt implies $P(Q)$ halts. $P(Q)$ halts if and only if $Q(Q)$ does not halt. Now, run the program on itself. Substitute P for Q. $P(P)$ halts if and only if $P(P)$ does not halt. This is an obvious contradiction therefore the assumption that the predicate **halts** exists must be incorrect [HARR 78]. The application of the program to itself in the proof is the trick. It is also the source of the halting problem.

In 1969 Hoare introduced a logic and axiomatic method for proving that a program is partially correct with respect to a specification [COUS 90]. In 1977 Clarke provided a language definition that included Algol-like and Pascal-like languages. The definition of a Clarke language is:

A programming language allowing procedures (with a finite number of local variable and parameters taking a finite number of values, without sharing via aliases) and the following features:

(i) procedures as parameters of procedure calls (without self-application);

(ii) recursion;

(iii) static scoping;

(iv) use of global variables in procedure bodies;

(v) nested internal procedures as parameters of procedure calls.

Clarke languages have an undecidable halting problem for finite interpretations with $|D| \geq 2$ [COUS 90]. This restriction on the cardinality of the domain ($|D|$) means there must be at least two possible interpretations for the lemma to hold. Algol-like and Pascal-like languages thus have an undecidable halting problem for finite interpretations with $|D| \geq 2$. Ada can also be categorized as a Clarke language. This implies that Ada also has an undecidable halting problem. The intuitive description of this is that the names in the predicates P and Q and the command C are all used in a similar fashion. All the names are considered objects, and at a given instance of time, are all given different names. Therefore, variables in P, Q, and C can be interpreted in exactly the same way by means of states. With Algol- or Pascal-like languages the naming conventions in P, Q, and C are different. For example, objects buried deep in the runtime stack cannot be accessed by their name. However, they can be modified using procedure calls [COUS 90].

3. SATISFIABILITY

Many of the methods for proving program correctness deal with some type of logic. In dealing with conventional, imperative programs, one has to define a program in the form of "computational logic" in order for the theorem prover to process it. The theorem prover then must process the "computational logic" program and the input program. However in logic based programs (such as LISP and Prolog) it is easier to have a theorem prover process the program without the additional overhead.

TALUS is a program debugger that uses the Boyer-Moore logic and theorem prover to detect and correct non-syntactic errors in programs written in a *restricted* subset of LISP [MURR 86]. The Boyer-Moore logic contains primitive constructs, data types, and functions that resemble LISP. It contains constructs from ordinary predicate logic that

37

permit formulation of well-formed formula. The Boyer-Moore theorem prover automatically proves the correctness of these well-formed formula by constructing an inductive proof of the equivalence of the student's function and a reference function [ALLE 91].

Note the emphasis we added to the word restricted in the description of Talus. The predicates of a pure LISP function can easily be represented in conjunctive normal form (CNF). A boolean expression is said to be in CNF if it is of the form $E_1 \wedge E_2 \wedge \ldots \wedge E_k$, where each E_i, called a clause (or conjunct), is of the form $\alpha_{i1} \vee \alpha_{i2} \vee \ldots \vee \alpha_{ir}$, where each αi_j is a literal $(x, \neg x)$ [HOPC 79]. For example, $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge \neg x_3$ is in CNF. The expression is in 3-CNF if there are exactly three literals. The previous example is not in 3-CNF because the first and third literals do not have exactly three literals. If we let the clauses of student program and the reference function represent the literals in an expression in CNF the results are significant. The question becomes one of satisfiability of the boolean expression. An expression is *satisfiable* if it yields the value *true* for some interpretation which assigns values *true* or *false* to the literals in the expression. This would be necessary to prove the two functions equivalent. This is called the satisfiability problem and has been proven to be NP-complete [HOPC 79]. That is, the problem is in the class NP and the problem is hard for NP with respect to log-space reduction. The Boyer-Moore theorem prover does not work on a purely mathematically model. It uses heuristics to choose the substitutions for the inductive proof [MURR 86]. The previous result is interesting but the language we are working with is Ada, which is an imperative language, not a functional language.

B.   THE SCOPE OF THE ADA PROBLEM

Is it possible to prove total correctness of **all** possible Ada programs within the Ada language? No, not in a mathematical sense. To prove total correctness of **all** Ada programs equates to proving the halting problem for Clarke languages is decidable, which it is not. Proving total correctness of a single Ada program is decidable. These restrictions are the

source of what has been termed significant variability [MURR 86]. Each procedure (or function) in the *language* can be represented in an infinite number of correct implementations. These various implementations would have different control flow, programming constructs, identifiers, order of formal parameters, and role of variables. If there are an infinite number of possible implementations in the language then there are potentially an infinite number of bugs [MURR 86].

The problems associated with the halting problem and significant variability mandate that instead of total correctness we provide partial correctness proofs of the language. This translates to limited debugging capabilities with any ITS that deals with program *language* correctness.

VI. DESIGN

A. SYSTEM INTEGRATION

The integration of *ITS Ada* and an extended student module is a question of which runtime environment should be the main control unit. There were two options: CLIPS as the controller calling the Ada tutor packages, or *ITS Ada* as the controlling program invoking the CLIPS interpreter when needed. We chose to implement the later.

With Ada as the controlling program it was not necessary to break up the original program and find the appropriate locations to make calls to CLIPS. It was not as critical to understand the control flow of *ITS Ada* because it was not going to be interrupted as would have been the case had CLIPS been the controlling environment.

The flow of control of the original code would stay intact to the point of editing the student code. Instead of waiting until the student's answer was ready and the student called for the editor to check the solution, each line of code would be written off to a file and the CLIPS environment invoked. The CLIPS code would access this file along with a fact list and check for any semantic or cognitive errors. Errors that are specifically enumerated as rules in the rule base. This increases the bandwidth of information available to the system, and allows semantic error detection as the lines of code are input. Syntactic errors would still be checked by the parser upon completion of the program segment. Those semantic errors that would not have normally been detected will have already been brought to the student's attention. Since the student can edit any line of code until the student calls for the solution to be checked CLIPS would not have to do anything to the student solution. The normal flow of control would resume allowing the parser to check for syntactic errors and the normal evaluation of the solution in accordance with *ITS Ada*'s original design.

B. DIAGNOSIS

Diagnosis consists of rule matching. The fact base is built up of tokens, constructs and historical data about the session. The CLIPS code parses the lines of code as they are input. This is not necessarily a syntactic check though certain syntax errors also indicate potential

knowledge misconceptions and missing concepts. As the fact base is built, the system is looking for a rule match to one of the semantic error rules. If all the necessary facts are present the rule will activate. If no rules fire, control is returned to the editor in *ITS Ada*. This does not mean that there are no misconceptions or missing concepts. It only means that of those potential errors coded into the rule base, none of the rules were activated. This is an inherent limitation to an enumerated bug library.

C. EXTENDED STUDENT MODEL

The extended student model is a file that contains a define facts construct (deffacts). This construct is built by the Ada main program and CLIPS. The wrapper of the construct is built by Ada. The fact list contained in the construct is built by execution of a CLIPS error rule. The fact is written out to a temporary file by CLIPS to be read and copied by the Ada main program. This fact list is updated dynamically by the CLIPS rules. If a rule is activated and that rule contains a fact that is to be asserted onto the fact list and it is critical to the future analysis of the student's cognitive state, then it is also written out to the student file.

D. STUDENT MODULE

As previously stated the student module contains the student model and the diagnostic component of the ITS (See "INTRODUCTION" on page 1.). The intent of the system is to take in the student solution one line of code at a time. As the student enters the carriage return the Ada main program writes that line of code to a temporary file where the CLIPS module has been coded to look for it. The Ada program then initializes and runs the CLIPS analysis code on that line of code. During the analysis of the code, if the necessary preconditions are met in terms of facts on the fact base, the appropriate error rule is activated and fired. This provides significantly more responsive feedback to the student on errors that may or may not have been identified by the parser in the main Ada program. The knowledge that the student has this missing concept or misconception is captured in the extended student model. The error rule writes out a code to a file. In order to get that

information into the appropriate student file it is copied by the Ada main program from the CLIPS temporary file into the appropriate student file.

There are three pieces of information that must be made available for the system to work. The lines of code, the problem number being solved by the student, and an *expert* solution to the problem being solved. The word expert is a misnomer. The solution is labeled an expert solution but is in reality *one* correct solution of any number of alternatively correct solution. The problem number will be used to determine which expert solution file of facts to load. Once loaded the expert solution fact base remains on the fact list and is used in comparisons (rule matching) of the error rules. The student solution is parsed into tokens that are asserted into the fact base. Knowing the problem to be solved and having a correct solution and the student's solution, there is significant information being provided to the rules such that rigorous decomposition of the original code is not necessary to identify a wide variety of missing concepts and misconceptions.

In order to build a realistic bug library a survey was given to a beginning Ada programming class, one week after they had covered all the constructs in question (the if conditional, and the three Ada looping constructs). The survey was written such that the student had a choice as to which construct to use (See "APPENDIX F. ADA SURVEY" on page 107.). The questions had a *best* solution. These best solutions provided the facts for the expert solution files.

VII. IMPLEMENTATION

A.   ENHANCING THE CLIPS ENVIRONMENT

A method to ease the burden of going from editing CLIPS code to running CLIPS code in the runtime environment was sought for two reasons. The first reason was to decrease the number of open interactive windows when in a windowing environment. The second reason was to ease the transition from editing code to the CLIPS runtime environment when working in a non-windowing environment. This normally involves terminating the editing and invoking the CLIPS runtime environment. Any editing changes would thus require leaving the CLIPS interpreter and invoking the editor. GNU Emacs [LEWI 90] allows customizing of the editing environment. In this case the file *clips.el* is an emacs lisp file that customizes emacs to edit CLIPS source code. It contains the code for two major modes: clips mode and inferior clips mode.

GNU Emacs Lisp is a product of Richard Stallman and the Free Software Foundation Inc. It is a full computer programming language. It is designed for use in an editor and hence has a myriad of features for scanning and parsing text as well as features for handling files, buffers, displays, subprocesses, etcetera. It is similar to common lisp with the above noted exceptions and two more that are worth noting here. The first is that elisp is a dynamically bound dialect of lisp. The second is that in elisp there is only one primitive numeric type, *integer*. This is due to the fact that elisp programs are intended for use in editors, not computation. The GNU Emacs Lisp Reference Manual [LEWI 90] outlines a coding convention for writing major modes and is the convention we followed. Most of the declarations and function definitions in the code will match up directly to the paragraphs in the GNU Emacs programming convention.

1.   CLIPS MODE

A major mode in the GNU Emacs venacular customizes Emacs for editing particular kinds of text.   CLIPS mode (See "APPENDIX A. EMACS CLIPS MODE" on page 63.) is an editing environment that expects CLIPS source code.

CLIPS source code is parenthetic in nature, similar to LISP, and has a special symbol to separate left hand side (LHS) from right hand side (RHS): =>, similar to Prolog. These two characteristics and the use of special characters by CLIPS are the foundation of majority of the elisp code for the mode.

Each ASCII character can be mapped into one of several categories. Nine characters needed to be redefined: # & ' , . : | ~ ?. This map is shown in the definition of the syntax table for CLIPS mode (see Figure 12).

```
;define the clips-mode-syntax-table
(if clips-mode-syntax-table
  () ;clips-mode-syntax-table already loaded, do not change
  (let ((table (make-syntax-table)))
  (modify-syntax-entry ?# "_" table) ;symbol constituent
  (modify-syntax-entry ?& "'" table) ;expression prefix operator
  (modify-syntax-entry ?' "_" table) ;symbol constituent
  (modify-syntax-entry ?, "_" table) ;symbol constituent
  (modify-syntax-entry ?. "_" table) ;symbol constituent
  (modify-syntax-entry ?: "'" table) ;expression prefix operator
  (modify-syntax-entry ?| "'" table) ;expression prefix operator
  (modify-syntax-entry ?~ "'" table) ;expression prefix operator
  (modify-syntax-entry ?? "_" table) ;symbol constituent
  (setq clips-mode-syntax-table table) ) )
```

Figure 12: CLIPS Mode Syntax Table Definition.

Symbol constituents are the extra characters that are used in variable and command names along with word constituents. In CLIPS the rule name **get-valid-#s** is legal so the # had to be added to the character set. An expression prefix operator is used for syntactic operators that are part of an expression if they appear next to one but are not part of an adjoining symbol. In CLIPS a variable is denoted by ?*variable-name*, e.g ?**input**.

The majority of the code in this mode deals with proper indentation. The code is lengthy and can be found in Appendix A on page 63. There are two major functions: clips-indent-line and calculate-clips-indent. Clips-indent-line returns an integer that is the correct column position to indent the current line of code. It calls calculate-clips-indent which actually parseses the current function keeping a count of the number of opening and closing parenthesis. It will return the value of the column that corresponds to the last open

parenthesis. In order to prevent having a function for each form found in CLIPS an association list is built up of equivalent syntactic forms. This is a hook which permits invocation whenever needed (see Figure 13).

```
(put 'deffacts 'clips-indent-hook 'defrule)
(put 'defglobal 'clips-indent-hook 'defrule)
(put 'deftemplate 'clips-indent-hook 'defrule)
(put 'deffunction 'clips-indent-hook 'defrule)
```

Figure 13: CLIPS Mode Form Association List.

The CLIPS forms *deffacts*, *defglobal*, *deftemplate*, and *deffunction* are all syntactically equivalent to the *defrule* form. This prevents having to write a new indentation function for each form.

```
(defun clips-mode ()
 "Major mode for editing CLIPS 5.0 code for CLIPS. ';' starts
comments. Normally blank lines separate structures and
blocks.
Use LF for automatic indentation. RTN will reset that lines
indentation to 0.
Commands:
\\{clips-mode-map}
Entry to this mode call the value of clips-mode-hook if that
value is non-nil."
 (interactive)
 (kill-all-local-variables)
 (use-local-map clips-mode-map)
 (set-syntax-table clips-mode-syntax-table)
 (setq major-mode 'clips-mode)
 (setq mode-name "CLIPS")
 (clips-mode-variables)
 (run-hooks 'clips-mode-hook) )
```

Figure 14: Main Function for CLIPS Mode.

The main function is *clips-mode* (see Figure 14). It interacts with several other elisp files, the system and the terminal. The function really does little more than change the current bindings in local variables, buffers, maps, syntax table, and valid hooks. It sets

the emacs environment so the other functions can operate given the proper syntax of CLIPS code.

This package of elisp functions made writing CLIPS code easier but did nothing to ease the transition from editing to the CLIPS interpreter. Another major mode was written to allow the CLIPS interpreter to be activated inside the emacs process.

2. INFERIOR CLIPS MODE

The tables defined for CLIPS mode are identical to those needed for inferior CLIPS mode. This is the reason the code for both modes are found in the same file. Inferior CLIPS mode is an emulation of the CLIPS runtime environment. The emacs buffer that is currently being operated in and designated a CLIPS buffer takes all input from the terminal and sends it to the CLIPS runtime environment. The output from the CLIPS interpreter is echoed in the CLIPS buffer. Since all the data in the buffer is sent, it is necessary to "peel off" the CLIPS interpreter prompt. Inside the main function, inferior-clips-mode, there are two lines of code that take the interpreter prompt out (see Figure 15). The rest of the code for this mode is relatively standard. Substituting the name of the mode and the proper buffer names into a format provided in the programmer's guide provides most of the functionality.

```
(make-variable-buffer-local 'shell-prompt-pattern)
(setq shell-prompt-pattern "^CLIPS> ") ;set clips prompt
```

Figure 15: CLIPS Interpreter Prompt in Emacs.

This mode allows invocation of the CLIPS interpreter from inside an emacs process. This significantly eased the transition from editing code to running code inside the interpreter. There are multiple ways to invoke the interpreter. The two most common are with the meta-x (ESC x) *run-clips* command in the minibuffer (see Figure 16). This causes

a response in the minibuffer which tells the user of the abbreviated key bindings that can be used instead (see Figure 17).
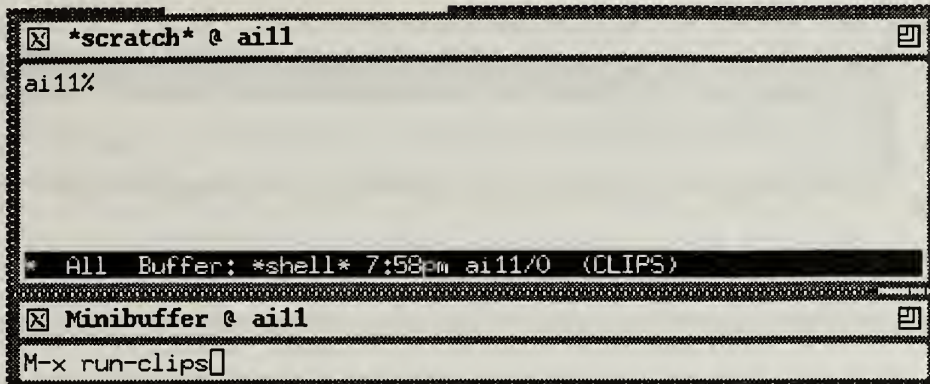
Figure 16: Invoking CLIPS with *ESC-x run-clips*, from Emacs.
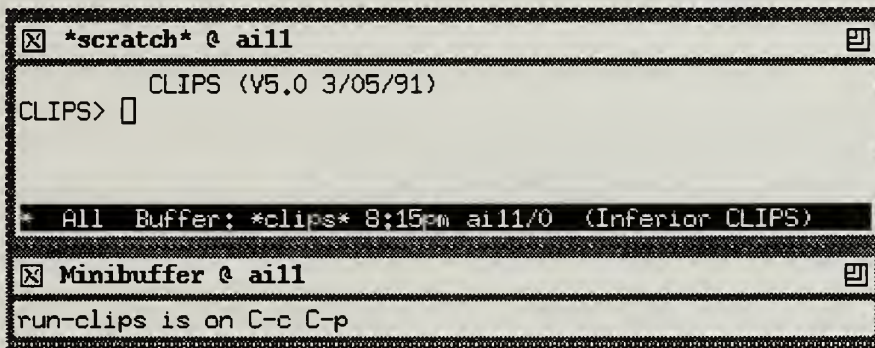
Figure 17: Minibuffer Message After Invoking CLIPS with *ESC-x run-clips*.

The abbreviated key bindings are coded in the function *clips-mode-commands*. The abbreviated key bindings to run the CLIPS interpreter from inside emacs are: Control c, Control p (C-c C-p) (see Figure 18).
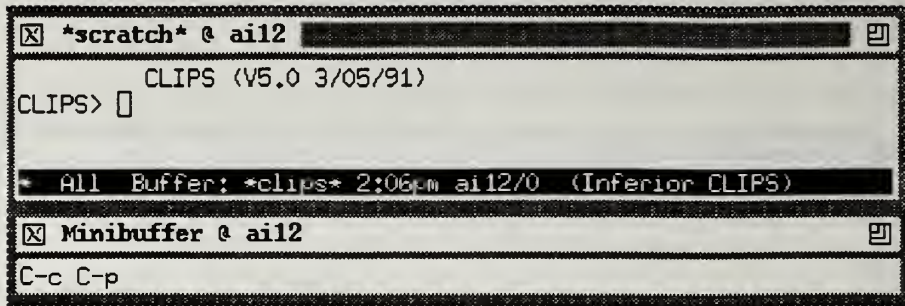
Figure 18: Emacs Windows after Invoking the CLIPS Interpreter.

There are several functions defined that allow the editing of a CLIPS function or region, selection of that area in the buffer, then execution of the CLIPS interpreter with that region as input. These functions are: *clips-consult-region-and-go*, and *clips-reconsult-buffer-and-go*. These functions have abbreviations defined in the function *clips-mode-commands* (see Figure 19).

```
;Key bindings specific for CLIPS mode
(defun clips-mode-commands (map)
 (define-key map "\t" 'clips-indent-line)
;; (define-key map "\C-m" 'newline-and-indent)
 (define-key map "\C-c\C-p" 'run-clips)
 (define-key map "\C-c\C-b" 'clips-reconsult-buffer-and-go)
 (define-key map "\C-c\C-r" 'clips-consult-region-and-go) )
```

Figure 19: CLIPS Mode Command Key Bindings.

Both major modes greatly increased our ability to go from one mode, editing, to the other without forcing major changes in the current runtime environment. It's greatest benefit was in being able to run emacs in a non-windowing environment, edit a file and then run the file without the need for additional windows or terminating the current process.

B. INTEGRATION OF CLIPS AND ADA

In order to pass control from the Ada main program to the CLIPS runtime environment Ada and CLIPS have to communicate. Recall in Chapter IV that the version of CLIPS we

are running is written in the programming language C. This requirement for Ada and CLIPS to communicate translates to a requirement for Ada and C to communicate.

The advanced programmers guide [NASA 91b] has an example of an Ada package that mapped DEC Ada to CLIPS. There is, however, a problem with using the example as it is currently written. The Ada Language Reference Manual [GONZ 91] lists 14 pragmas defined by the language. The different compilers must implement these pragmas but are not limited to the additional pragmas they can add to their compiler. The majority of Verdix Ada and Dec Ada pragmas are thus compiler specific with the exception of the 14 language defined pragmas. Verdix Ada [VERD 90] has a total of 28 pragmas including the language defined implementations. This necessitated rewriting the package for Verdix Ada.

The basic concept is quite simple. From Ada, if you want to call a function or procedure in another programming language, the compiler needs to know a couple of things. It needs to know the target language and the subprogram name as it will appear in the Ada source code. In order for the linker to resolve external references it needs to know the Ada subprogram name as it will appear in the Ada source code and the representation of the object file link it will be referencing. DEC VMS uses different pragmas depending on the type of construct, procedure or function (see Figure 20) [NASA 91b]. Verdix Ada uses the same pragmas regardless of the Ada construct (see Figure 21)[VERD 90].

There is another change in the way the compilers handle parameters. In order to pass a parameter, such as a file name, in the DEC VMS compiler all one needs to do is convert an Ada string to a C string and call the function **cLoadConstructs** with the C string parameter. In Verdix the parameter could only be of type scalar, access, or the predefined type ADDRESS found in the package SYSTEM. This requires rewriting the conversion function (see Figure 22) to return a value of type ADDRESS instead of type STRING.

```
pragma INTERFACE (C, xInitializeCLIPS);
pragma IMPORT_PROCEDURE(INTERNAL => xInitializeCLIPS,
                        EXTERNAL => InitializeCLIPS);

function cLoadConstructs (File_Name : in string) return integer;
pragma INTERFACE(C, cLoadConstructs);
pragma IMPORT_FUNCTION(INTERNAL => cLoadConstructs,
                       EXTERNAL => LoadConstructs,,
                       MECHANISM => REFERENCE);

pragma INTERFACE(C, xRunCLIPS);
pragma IMPORT_FUNCTION(INTERNAL => xRunCLIPS,
                       EXTERNAL => RunCLIPS,
                       MECHANISM => REFERENCE);
```

Figure 20: DEC VMS Ada Pragma for CLIPS.

```
pragma INTERFACE (C, xInitializeCLIPS);
pragma INTERFACE_NAME (xInitializeCLIPS, "_InitializeCLIPS");

function cLoadConstructs (File_Name : in Address) return INTEGER;
pragma INTERFACE (C, cLoadConstructs);
pragma INTERFACE_NAME (cLoadConstructs, "_LoadConstructs");

pragma INTERFACE (C, xRunCLIPS);
pragma INTERFACE_NAME (xRunCLIPS, "_RunCLIPS");
```

Figure 21: Verdix Ada Pragma for CLIPS.

The flow of control works like this. For example, the CLIPS source code to be loaded from inside Ada is contained in a file, *test.clp*. Since **xLoadConstructs** is a function the value returned must be assigned to some variable of the same type. The function call may look like this:

```
File_Name(1..8) := "test.clp";
File_Open_Status := xLoadConstructs(File_Name(1..8));
```

The Ada string *test.clp* must be converted to an attribute of type ADDRESS in order for the pragma mapping from Ada to C to function properly. The return value from the function call is *integer* therefore *File_Open_Status* must be declared to be of type

integer. In the package body of package *clips.a* (See "APPENDIX B. CLIPS/ADA INTERFACE PACKAGE" on page 73.) the function **xLoadConstructs** looks like this:

```
function xLoadConstructs(File_Name : in STRING) return INTEGER is
begin
   return cLoadConstructs(Ada_to_C_String(File_Name));
end xLoadConstructs;
```

The Ada string *test.clp* is an *in* parameter to **xLoadConstructs** which in turn calls **cLoadConstructs** with the return value, of type ADDRESS, from the imbedded function call to **Ada_to_C_String**. The pragma INTERFACE_NAME (see Figure 21) allows the linker to replace all occurrences of **cLoadConstructs** with the object code representation of **_LoadConstructs**. The C function **_LoadConstructs** gets a pointer (type ADDRESS in Ada) that is the address of the file *test.clp* which is the expected in parameter. The function will return an integer value of zero if it was able to open the file or a value of one if an error was encountered. This is one of the more complicated examples because it requires the passing of a parameter.

```
function Ada_to_C_String (Input_String : in STRING) return ADDRESS is
 Out_String : string(1..Input_String'last+1);
begin
 for I in Input_String'range loop
  if (Input_String(I) in ' '..'~' or
    Input_String(I) = ASCII.Cr or
    Input_String(I) = ASCII.Lf ) then
    Out_String(I) := Input_String(I);
  else
    Out_String(I) := ASCII.Nul;
  end if;
 end loop;
 Out_String (Out_String'last) := ASCII.Nul;
 return Out_String'Address;
end Ada_to_C_String;
```

Figure 22: Verdix Ada, Ada_to_C_String Conversion Function.

Many of the functions invoked by Ada do not require the passing of parameters. However, not all of the CLIPS functions have been incorporated into the package clips.a. Many of theses non-incorporated functions do require passing parameters other than strings.

51

C.   ENHANCED STUDENT MODEL

As stated in Chapter VII, the enhanced student model is a file that contains a CLIPS deffacts construct. The file is created when the student logs into the system for the first time. The prototype emulates the input and output (I/O) of *ITS Ada*. When the student first executes *ITS Ada* a general introduction screen appears and prompts the user for their name or user identification (see Figure 23). Once *ITS Ada* has this information it searches a file called *user.dat* for the student model data (See "THE "ITS ADA" INTELLIGENT TUTORING SYSTEM" on page 31.). The emulation provides the same prompt for the same reasons.
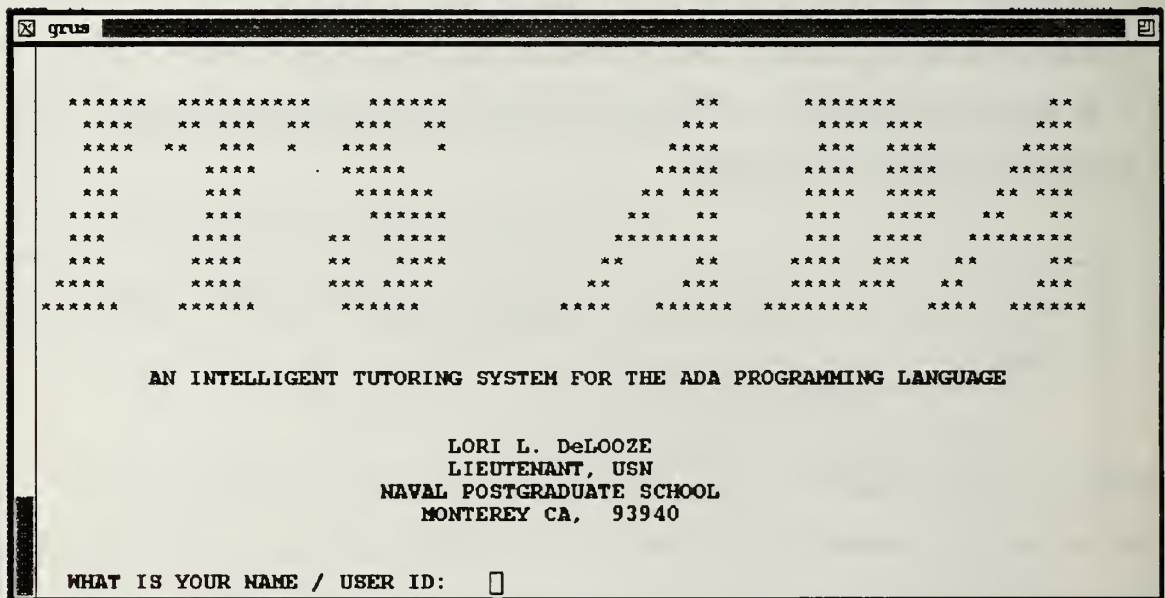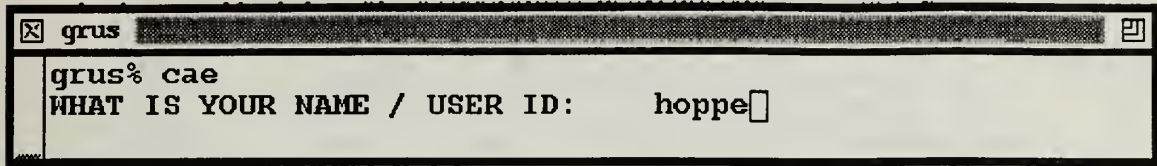
Figure 23: Screen 1 from *ITS Ada*.

The information provided by the user becomes the student model retrieval index for both *ITS Ada* and the enhanced student model. In the enhanced student model the information is needed to first construct the file name and then check to see if the file exists. If the file exists the file is loaded when the CLIPS interpreter is invoked. For example, in Figure 24 the response was **hoppe**. Our system concatenates the suffix, **.mod**, to the input data to build the file name **hoppe.mod**. It then checks to see if the file exists in the current
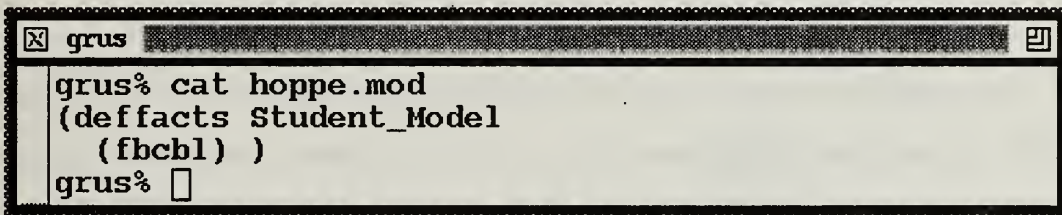
directory. If the file exists, the file is relabeled as the student model file name. If it does not exist it is created and then relabeled. This file name is a parameter to the procedure **Invoke_CLIPS** (See "APPENDIX D. ADA EMULATOR SOURCE CODE" on page 80.).

```
⊠  grus                                                          🔲
grus% cae
WHAT IS YOUR NAME / USER ID:      hoppe▯
```

Figure 24: Invocation and Initial Prompt of *ITS Ada* Emulation.

The file **hoppe.mod** may contain the deffacts construct in Figure 25. The fact **(fbcbl)** is a representation of the error rule that fired during a session. It identifies the rule **faulty-boolean-cond-basic-loop** (See "APPENDIX E. CLIPS RULES" on page 89.).

```
⊠  grus                                                          🔲
grus% cat hoppe.mod
(deffacts Student_Model
   (fbcbl) )
grus% ▯
```

Figure 25: Contents of Enhanced Student Model.

D. THE STUDENT MODULE

The student model was discussed in the previous section. The diagnostic component of the system is part Ada code for preprocessing the student solution and the CLIPS rule base. CLIPS recognizes certain characters as delimiters and therefor necessitates preprocessing the student solution prior to invoking the CLIPS interpreter. Specifically, the ;, =, (, and) are all delimiters in CLIPS. They terminate the current line and are ignored by the inference engine so they can not be assert as part of the fact base. This makes Ada statements such as, *if value = 0*, or *Balance := Compound_Interest(Balance);* , difficult to parse in CLIPS without preprocessing the problem characters into tokens. The Ada statement, *Wait := True;*, becomes *Wait := True semicolon.* The actual student solution is

written to the file **stdnt.dat**. This is the same file the ITS Ada parser will use as input to check for syntactic correctness. The files **temp.dat** and **temp.out** contain the preprocessed solutions used by the CLIPS rule base (see Figure 26).

```
stdnt.dat                                temp.out

                                         for
   for index in 1..100 loop              index
   index := index + 1;                   in
   end loop;                             1..100
                                         loop
                                         index
temp.dat                                 :=
                                         index
   for index in 1..100 loop              +
   index := index + 1 semicolon          1
   end loop semicolon                    semicolon
                                         end
                                         loop
                                         semicolon
```

Figure 26: Solution Files for the Enhanced System.

The preprocessed student solution (**temp.dat**) is then read, a token at a time, by a small CLIPS file that rewrites the tokens out to the file **temp.out**. The student model is then loaded into the fact base along with the rule base (**parser.clp**). Inside the CLIPS parser the input files **temp.dat** and **temp.out** are opened. The output file **model.out** is opened to capture changes to the student model. An Ada procedure writes the problem number out to the file called **current_prb.dat**. This file contains one fact. That one fact is of the form (**current-problem prob#.clp**). Where the '#' is replaced by the integer value of the problem being solved. The first executable command on the RHS of the rule **open-files** is the statement, (**load-facts current_prob.dat**). This asserts the fact, (**current-problem prob#.clp**) on to the fact list and activates the rule **load-expert-solution** (see Figure 28). Each problem has a corresponding expert solution file named **prob#.clp** (see Figure 29). It contains the facts about the expert solution necessary to identify the errors coded in the rule base. Once this file is loaded the system has the three components necessary to analyze the solution. Those components are again, the problem being solved, the student's solution,

and the expert solution. Figure 27 is a data flow diagram of this process. It illustrates the process and the dependency of information within the enhanced student module.
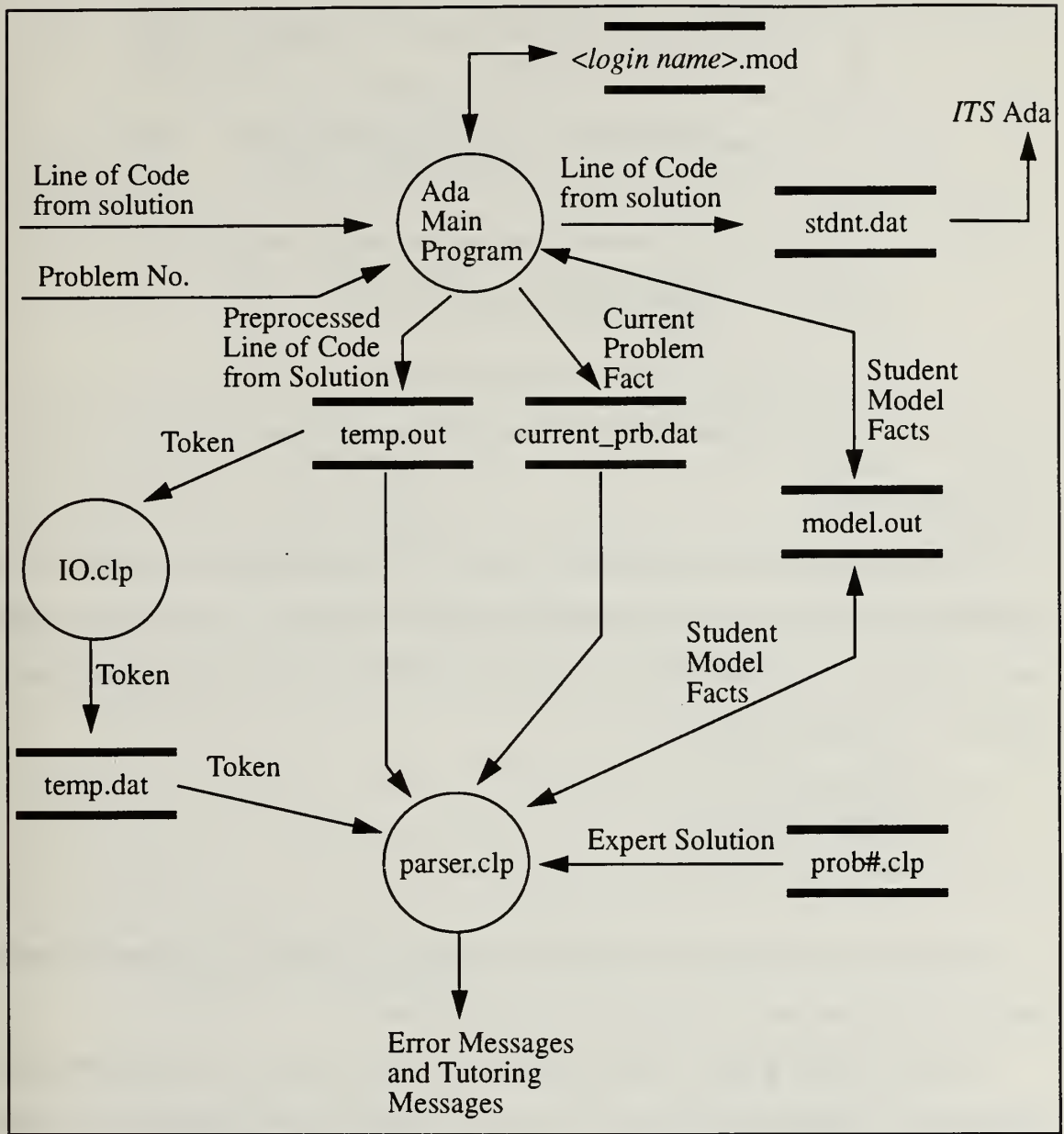
Figure 27: Data Flow Diagram of Enhanced System.

```
(defrule load-expert-solution
  (declare (salience 15))
  (current-problem ?number)
=>
  (load-facts ?number)
  (bind ?num (eval (sub-string 5 5 ?number)))
  (assert (problem ?num)))
```

Figure 28: CLIPS Rule **load-expert-solution**.

```
(expert range 1..100)
(expert stmnt sum := sum + I)
(expert token end loop)
```

Figure 29: Facts Contained in **prob2.clp**.

E.   EXAMPLES

Most of the figures in the preceding section dealt with different aspects of one of the problems in the Ada survey (See "APPENDIX F. ADA SURVEY" on page 107.). Problem two has the following description: Write a loop that sums the numbers from 1 to 100 inclusive. The expert solution is:

        Sum := 0;
        for I in 1..100 loop
          Sum := Sum + I;
        end loop;

The expert solution fact base is shown in Figure 29. The knowledge that the problem uses a for loop construct is captured in the problem number.

The error we are going to induce is not only a syntax error but a common error found in the novice solutions to the survey. The error is manually incrementing the loop counter of a for loop. Since the loop counter is considered a constant it can not be allowed on the LHS of an assignment statement. With the original *ITS Ada* diagnostics the student would not get any feedback about this error until the solution was completed and the student

wanted to have the answer checked. This enhanced system notifies the student as soon as the error is detected (see Figure 30).

```
WHAT IS YOUR NAME / USER ID:     hoppe
Enter problem number:  2
@  1  for index in 1..100 loop

@  2  index := index + 1;
ERROR: index := index + 1 semicolon
The Ada for loop automatically increments its counter.
Within the sequence of statements the loop parameter is a constant. Hence
a for loop parameter is not allowed as a variable on the LHS of an
assignment statement (LRM 5.5.6).
Do you want to see an explanation and example (yes or no)? yes


According to the LRM a loop statement with a "for" iteration
scheme, the loop parameter specification is the declaration of the loop
parameter with the given identifier.  The loop parameter is an object whose
type is the base type of the discrete range (see 3.6.1).  Within the sequence
of statements, the loop parameter is a
constant.  Hence a loop parameter must not be given as an "out" or "in out"
parameter of a procedure or entry call statement, or as an "in out" parameter
of a generic instantiation.

For the execution of a loop statement with a "for" iteration
scheme, the loop parameter specification is first elaborated.  This
elaboration
creates the loop parameter and evaluates the discrete range.

If the discrete range is a null range, the execution of the loop
statement is complete.  Otherwise, the sequence of statements is executed
once
for each value of the discrete range.  Prior to each such iteration, the
corresponding value of the discrete range is assigned to the loop parameter.

The syntax of the for loop is:
     for [loop parameter specification] loop
       [sequence of statements]
     end loop;

Example:
     for J in Buffer'Range loop      --legal even with a null range
       if Buffer(J) /= Space then
         Put(Buffer(J));
       end if;
     end loop;

@  3
```

Figure 30: Screen Dump of Problem Two.

As soon as the student entered the carriage return the CLIPS rules detected the error and notified the student. At this point the student knows of the error and can fix the problem prior to completing the solution and having it checked by the parser. Figure 30 is a clean dialogue. It does not contain any debugging information from the system. Note that @ is the system prompt and all user input is shown in italics.

Another example shows the facts that are present as each line of code is entered. This is part of a run from problem four which has the following description: As long as the input value is greater than or equal to zero, sum the numbers. The expert solution is:

```
Get(Value);
Sum := 0;
loop
    exit when Value < 0;
    Sum := Sum + Value;
    Get(Value);
end loop;
```

The expert solution fact base is shown in Figure 31. This error is another common error on the survey. Many novice programmers attempted to combine constructs to make their own. Specifically a combination of the "if" and basic "loop" constructs.

```
(expert token loop)
(expert token exit when)
(expert boolean-cond value < 0)
(expert stmnt sum := sum + value)
(expert token end loop)
```

Figure 31: Facts Contained in **prob4.clp**.

Again, as soon as the student enters the carriage return the error is detected and the student receives feedback (see Figure 32). This error would not have been detected until the parser was called by the student once the solution was completed.

A final example is one that is trivial in terms of detection but is a semantic error and would not be caught at all by the parser (see Figure 34). This is the infinite loop error. Again using problem four this time without an exit condition.

```
 ┌─────────────────────────────────────────────────────────────────────┐
 │ ⊠ grus ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓ 凹│
 │  WHAT IS YOUR NAME / USER ID:      hoppe                              │
 │  Enter problem number:   4                                           │
 │  @  1  if input_value >= 0 loop                                      │
 │  f-0      (initial-fact)                                             │
 │   13 Rules Fired                                                     │
 │  f-0      (initial-fact)                                             │
 │  f-1      (no error)                                                 │
 │  ERROR: if input_value >= 0 loop                                     │
 │  The Ada "if" statement does not have a loop construct               │
 │  (LRM 5.3.2).  You are trying to write a looping construct.  Recommend│
 │  you review the basic loop (LRM 5.5.9), the "while loop" (LRM 5.5.10)│
 │  and the "for loop" (LRM 5.5.11).                                    │
 │   4 Rules Fired                                                      │
 └─────────────────────────────────────────────────────────────────────┘
```

Figure 32: Screen Dump of Partial Solution to Problem Four.

The key is the pattern matching of the student's solution and the knowledge that the student is solving problem four. This allows the rules to be written without the need for complex analysis of the loop invariant or the remainder of the code (see Figure 33).

```
;Infinite loop in basic loop construct.  No exit condition in the loop.
(defrule infinite-loop-in-basic-loop
 (declare (salience 10))
 ?init <- (initial-fact)
 ?error-status <- (no error)
 ?token1 <- (token loop)
 ?token2 <- (token end loop)
 (and (not (token exit))
    (not (token exit when)))
=>
 (printout model "ilibl" crlf)
 (printout t "You have ended the loop construct without an exit
condition.
This will cause an INFINITE loop once entered.  You may want to
re-evaluate the loop conditions." crlf)
 (retract ?error-status ?init)
 (assert (concept error)
     (syntax error)
     (lrm 5.5.4))
 (close))
```

Figure 33: CLIPS Error Rule **infinite-loop-in-basic-loop**.

```
WHAT IS YOUR NAME / USER ID:      hoppe
Enter problem number:   4
@   1   loop

@   2   sum := sum + value;

@   3   end loop;
You have ended the loop construct without an exit condition.
This will cause an INFINITE loop once entered.  You may want to re-evaluate
the loop conditions.
Do you want to see an explanation and example (yes or no)? yes


According to the LRM a loop statement without an iteration
scheme specifies repeated execution of the sequence of statements.
Execution of the loop statement is complete when the loop is left as a
consequence of the execution of an exit statement, or as a consequence
of some other transfer of control(LRM 5.1).

The syntax of the basic loop is:
      loop                            loop
        [sequence of statements]        [sequence of statements]
        exit;                           exit when [boolean condition];
        [sequence of statements]        [sequence of statements]
      end loop;                       end loop;

Example:
      loop
        Get(Current_Character);
        exit when Current_Character = '*';
      end loop;

@   4   .
This is your solution:
1   loop
2   sum := sum + value;
3   end loop;
```

Figure 34: Screen Dump of Partial Solution to Problem Four with an Infinite Loop.

The obvious limitation of the system is the robustness of the bug library. However, keeping in mind the target domain of this system, the novice Ada programmer, the bug library does not need to be exhaustive. The most common and most instructive bugs are the ones that need to be catalogued.

VIII. CONCLUSIONS

A. ACCOMPLISHMENTS

The CLIPS enhancement (*clips.el*) for the emacs editing environment has been placed on-line and opened to the general public for use. It has found a considerable amount of use in this project and in several other classes which use CLIPS.

To this point we have a working prototype that emulates the interaction of *ITS Ada* and the student. This prototype was necessary in order to prove that CLIPS and Ada could be coupled in a manner suitable for use in this research. CLIPS has been successfully integrated with Verdix Ada with the Ada program operating as the outer control program.

The control problems of Ada invoking the CLIPS interpreter and regaining control have been overcome. Addition of several CLIPS commands to the Ada package *clips.a* made these commands available to be used in the Ada source code.

For a very selective group of Ada statements the CLIPS rules have been successful in determining limited semantic errors for documented error types. The student model has been extended from a relatively flat, very structured representation to a more dynamic, somewhat more robust and more detailed representation of the student's actual knowledge within the domain.

B. FUTURE RESEARCH AREAS

A complete and detailed survey similar to the survey conducted during the development of PROUST [JONS 85] needs to be done for the Ada programming language. This would facilitate the classification and development of a significantly larger and more complete bug library.

Integration of *ITS Ada* to the completed bug library would fully extend the student model over the entire spectrum of the language topics covered. This would allow more of the potential of the tutor to be realized.

Integration of an on-line Ada language reference manual (LRM) with an index and retrieval system would preclude having to rewrite the LRM in the form of explanation

rules. This index and retrieval system would then allow us to specify which sentences of the LRM to print to the screen thus allowing any combination of explanation from the LRM references.

The addition of the remaining CLIPS functions to the interface package *clips.a* would make the entire CLIPS environment and all CLIPS constructs available to the Ada programmer. This would extend the functionality of CLIPS within the Ada program and could present new methods of control that currently are not available because the functionality is not present.

The main control unit of this thesis was the Ada main program calling the CLIPS interpreter when needed. Another alternative would be to re-engineer the entire system with CLIPS as the main control unit calling Ada functions. The advantage to be gained from this approach would be the increase in potential for capturing more information about the student and the cogintive process. The major difficulting in this approach is the communication of C and Ada. Currently the communication is from Ada to C. Having CLIPS as the main program would require C to communicate with Ada. This is a compiler specific problem.

There is clearly an object oriented approach that can be taken in terms of the information gained about the student. Utilizing the OOP paradigm and CLIPS Object Oriented Language (COOL) this entire process could be reworked. This would require an extension of the interface package to include all COOL functions. It would require a redesign of the extended student model and a rewrite of many of the rules.

CLIPS version 5.1 has been released with extended features that integrate the rule based side of CLIPS and the object oriented side of CLIPS. Extending this project under the newer language release could provide a simpler solution than using CLIPS version 5.0.

APPENDIX A. EMACS CLIPS MODE

```
;; CLIPS mode, CLIPS editing support package in GNU elisp. v 1.0
;; Written by William C. Hoppe, CPT, IN, US Army <hoppe@cs.nps.navy.mil>
;; March 1992.
;; Borrows logic and code fragments from lisp-mode.el and prolog.el packages
;; for GNU Emacs

;; GNU Emacs is distributed in the hope that it will be useful,
;; but WITHOUT ANY WARRANTY. No author or distributor
;; accepts responsibility to anyone for the consequences of using it
;; or for whether it serves any particular purpose or works at all,
;; unless he says so in writing. Refer to the GNU Emacs General Public
;; License for full details.

;; Everyone is granted permission to copy, modify and redistribute
;; GNU Emacs, but only under the conditions described in the
;; GNU Emacs General Public License. A copy of this license is
;; supposed to have been given to you along with GNU Emacs so you
;; can know your rights and responsibilities. It should be in a
;; file named COPYING. Among other things, the copyright notice
;; and this notice must be preserved on all copies.

;; In order to autoload or use auto-mode-alist to check for .clp
;; files add the following lines of code to your .emacs file

;;load-path for clips.el
;; (setq load-path (cons "Thesis/clips-mode" load-path))
;; (require 'shell "shell")
;;load clips.el
;; (autoload 'clips-mode "clips" "edit a clips file" t nil)
;; (autoload 'run-clips "clips" "run CLIPS 5.0 in a subwindow" t nil)
;; (load "Thesis/clips-mode/clips.el")

;; set auto-mode-alist to run *.clp files in clips-mode
;; (setq auto-mode-alist
;; (cons
;; '("\\.clp" . clips-mode) ;Filename ends in .clp
;; auto-mode-alist))


(defvar clips-mode-syntax-table nil
  "Syntax table used in CLIPS-mode")

(defvar clips-mode-abbrev-table nil
  "Abbrev table used in CLIPS-mode")

(defvar clips-mode-map nil
```

```
   "Keymap used in CLIPS-mode")

;Standard indentation level for CLIPS
(defvar clips-indent-width 2)

;define the clips-mode-syntax-table
(if clips-mode-syntax-table
 () ;clips-mode-syntax-table already loaded, do not change
 (let ((table (make-syntax-table)))
 (modify-syntax-entry ?# "_" table)
 (modify-syntax-entry ?& "" table)
 (modify-syntax-entry ?' "_" table)
 (modify-syntax-entry ?, "_" table)
 (modify-syntax-entry ?. "_" table)
 (modify-syntax-entry ?: "" table)
 (modify-syntax-entry ?| "" table)
 (modify-syntax-entry ?~ "" table)
 (modify-syntax-entry ?? "_" table)
 (setq clips-mode-syntax-table table) ) )

;Abbreviation table for CLIPS mode
(define-abbrev-table 'clips-mode-abbrev-table '(
 ("ld" "(load" nil 1)
 ("dfa" "(deffacts" nil 1)
 ("dru" "(defrule" nil 1)
 ("dte" "(deftemplate" nil 1)
 ("dfu" "(deffunction" nil 1)
 ("dgl" "(defglobal" nil 1)
 ("dmsg" "(defmessage-handler" nil 1)
 ("prt" "(printout" nil 1)
 ("prtt" "(printout t" nil 1)
 ("asr" "(assert" nil 1)
 ("rtr" "(retract" nil 1)
 ("dcl" "(defclass" nil 1)
 ("dscl" "(describe-class" nil 1)
 ("slt" "(slot" nil 1)
 ("snd" "(send" nil 1)
 ("bnd" "(bind" nil 1)
 ("rst" "(reset)" nil 1)
 ("clr" "(clear)" nil 1)
 ("mkins" "(make-instance" nil 1)
 ("ins" "(instances)" nil 1)
 ("din" "(definstances" nil 1)
 ("dft" "(default" nil 1)
 ("dftdy" "(default-dynamic" nil 1)
 ("isa" "(is-a" nil 1)
 ("abst" "(abstract)" nil 1)
 ("mtpl" "(multiple)" nil 1)
 ("rdly" "(read-only)" nil 1)
 ("cmpt" "(composite)" nil 1)
 ("gsm" "(gensym)" nil 1)
```

```
("gsmm" "(gensym*)" nil 1)
("inily" "(initialize-only)" nil 1)
("da" "=>" nil 1)
("la" "<-" nil 1)
("anyinsp" "(any-instancep" nil 1)
("fndins" "(find-instance" nil 1)
("fndinss" "(find-all-instances" nil 1)
("dins" "(do-for-instance" nil 1)
("dinss" "(do-for-all-instances" nil 1)
("dlydinss" "(delayed-do-for-all-instances" nil 1)
("sym2ins" "symbol-to-instance-name" nil 1)
("ins2sym" "instance-name-to-symbol" nil 1)
("lscl" "(list-defclasses)" nil 1)
("supclp" "(superclassp" nil 1)
("subclp" "(subclassp" nil 1)
("brcl" "(browse-classes)" nil 1)
("ppcl" "(ppdefclass" nil 1)
("dscl" "(describe-class" nil 1)
("lsins" "(list-definstances)" nil 1)
("ppins" "(ppdefinstances" nil 1)
("lsfun" "(list-deffunctions)" nil 1)
("ppfun" "(ppdeffunction" nil 1)
                                    ) )

;Local variables used while in CLIPS mode
(defun clips-mode-variables ()
 (set-syntax-table clips-mode-syntax-table)
 (setq local-abbrev-table clips-mode-abbrev-table)
 (make-local-variable 'paragraph-start)
 (setq paragraph-start (concat "^;\\|^$\\|" page-delimiter)) ; ';....'
 (make-local-variable 'paragraph-separate)
 (setq paragraph-separate paragraph-start)
 (make-local-variable 'paragraph-ignore-fill-prefix)
 (setq paragraph-ignore-fill-prefix t)
 (make-local-variable 'indent-line-function)
 (setq indent-line-function 'clips-indent-line)
 (make-local-variable 'comment-start)
 (setq comment-start ";")
 (make-local-variable 'comment-start-skip)
 (setq comment-start-skip ";+ *")
 (make-local-variable 'comment-column)
 (setq comment-column 48)
 (make-local-variable 'comment-indent-hook)
 (setq comment-indent-hook 'clips-comment-indent) )

;Key bindings specific for CLIPS mode
(defun clips-mode-commands (map)
 (define-key map "\t" 'clips-indent-line)
;; (define-key map "\C-m" 'newline-and-indent)
 (define-key map "\C-c\C-p" 'run-clips)
 (define-key map "\C-c\C-b" 'clips-reconsult-buffer-and-go)
```

```
  (define-key map "\C-c\C-r" 'clips-consult-region-and-go)
  )


(if clips-mode-map
 nil ; already in CLIPS-mode, change nothing
 (setq clips-mode-map (make-sparse-keymap))
 (clips-mode-commands clips-mode-map) )

(defun clips-mode ()
 "Major mode for editing CLIPS 5.0 code for CLIPS. ';' starts
comments. Normally blank lines separate structures and blocks.
Use LF for automatic indentation. RTN will reset that lines
indentation to 0.
Commands:
\\{clips-mode-map}
Entry to this mode call the value of clips-mode-hook if that
value is non-nil."
 (interactive)
 (kill-all-local-variables)
 (use-local-map clips-mode-map)
 (set-syntax-table clips-mode-syntax-table)
 (setq major-mode 'clips-mode)
 (setq mode-name "CLIPS")
 (clips-mode-variables)
 (run-hooks 'clips-mode-hook) )

(defconst clips-indent-offset nil "")
(defconst clips-indent-hook 'clips-indent-hook "")

(defun clips-indent-line (&optional whole-exp)
 "Indent current line as CLIPS code. With arguments,
indent any additional lines of the same clause
rigidly along with this one."
 (interactive "p")
 (let ((indent (calculate-clips-indent)) shift-amount begin end
         (pos (- (point-max) (point))) )
 (beginning-of-line)
 (setq begin (point))
 (if (looking-at ";")
         ;;Comment lines should be indented as comment lines not code
         (progn
          (indent-for-comment)
          (forward-char -1) )
 (if (listp indent) (setq indent (car indent)))
 (setq shift-amount (- indent (current-column)))
 (if (zerop shift-amount)
         nil
         (delete-region begin (point))
         (indent-to indent) ) )
 ;;if initial point was within line's indentation,
```

```
;;position after the indentation. Else stay at same point in text.
(if (> (- (point-max) pos) (point))
        (goto-char (- (point-max) pos)) )
;;If desired, shift remaining lines of expression the same amount.
(and whole-exp (not (zerop shift-amount))
        (save-excursion
        (goto-char begin)
        (forward-sexp 1)
        (setq end (point))
        (goto-char begin)
        (forward-line 1)
        (setq begin (point))
        (> end begin))
        (indent-code-rigidly begin end shift-amount) ) ) )


(defun calculate-clips-indent (&optional parse-start)
  "Return appropriate indentation for current line as CLIPS code.
In usual case returns an integer: the column to indent to.
Can instead return a list, whose car is the column to indent to.
This means that following lines at the same level of indentation
should not necessarily be indented the same way.
The second element of the list is the buffer position
of the start of the containing expression."
 (save-excursion
 (beginning-of-line)
 (let ((indent-point (point))
        state paren-depth
        ;;setting desired-indent to a number inhibits calling hook
        (desired-indent nil)
        (retry t)
        last-sexp containing-sexp) ;closing paren for 'let'
 (if parse-start
        (goto-char parse-start)
        (beginning-of-defform) )
;;Find outermost containing sexp
(while (< (point) indent-point)
        (setq state (parse-partial-sexp (point) indent-point 0)) )
(while (and retry
                state
                (> (setq paren-depth (elt state 0)) 0))
        (setq retry nil)
        (setq last-sexp (elt state 2))
        (setq containing-sexp (elt state 1))
        ;;Position following last unclose open.
        (goto-char (1+ containing-sexp))
        ;;Is there a complete sexp since then?
        (if (and last-sexp (> last-sexp (point)))
        ;;Yes, but is there a containing sexp after that?
        (let ((peek (parse-partial-sexp last-sexp indent-point 0)))
        (if (setq retry (car (cdr peek)))
```

```
                  (setq state peek) ) ) ) )
(if retry
          nil
          ;;Innermost containing sexp found
          (goto-char (1+ containing-sexp))
          (if (not last-sexp)
          ;;indent point immediately follows open paren.
          ;;Don't call hook.
          (setq desired-indent (current-column))
          ;;Find the start of first element of containing sexp.
          (parse-partial-sexp (point) last-sexp 0 t)
          (cond ((looking-at "\\s(")
                  ;;First element of containing sexp is a list.
                  ;;Indent under that list
                  )
                ((> (save-excursion (forward-line 1) (point))
                last-sexp)
                ;;This is the first line to start within the containing sexp.
                (if (= (point) last-sexp)
                ;;containing sexp has nothing before this line
                ;;except the first element. Indent under that element.
                (progn (forward-sexp 1)
                        (parse-partial-sexp (point) last-sexp 0 t) ) )
                (backward-prefix-chars) )
                (t
                ;;Indent beneath first sexp on same line as last-sexp.
                (goto-char last-sexp)
                (beginning-of-line)
                (parse-partial-sexp (point) last-sexp 0 t)
                (backward-prefix-chars) ) ) ) )
;;Point is at the point to indent under unless we are inside a string.
;;Call indentation hook except when overriden by clips-indent-offset
;;or if the desired indentation has already been computed.
(let ((normal-indent (current-column)))
        (cond ((looking-at "=") ;check for the '=>' , implies
          (forward-char 1)
          (if (looking-at ">")
                  (setq normal-indent clips-indent-width)
                  nil) )
        ((elt state 3)
        ;;Inside a string, don't change indentation.
        (goto-char indent-point)
        (skip-char-forward " \t")
        (current-column) )
        ((and (integerp clips-indent-offset) containing-sexp)
        ;;Indent by constant offset
        (goto-char containing-sexp)
        (+ normal-indent clips-indent-offset) )
        (desired-indent)
        ((and (boundp 'clips-indent-hook)
                clips-indent-hook
```

```lisp
                 (not retry) )
            (or (funcall clips-indent-hook indent-point state)
                 normal-indent) )
        (t
        normal-indent) ) ) ) ) )



(defun clips-indent-hook (indent-point state)
 (let ((normal-indent (current-column)))
 (goto-char (1+ (elt state 1)))
 (parse-partial-sexp (point) last-sexp 0 t)
 (if (and (elt state 2)
 (not (looking-at "\\sw\\|\\s_")))
 ;; car of form doesn't seem to be a a symbol
 (progn
 (if (not (> (save-excursion (forward-line 1) (point))
 last-sexp))
 (progn (goto-char last-sexp)
 (beginning-of-line)
 (parse-partial-sexp (point) last-sexp 0 t)))
 ;; Indent under the list or under the first sexp on the
 ;; same line as last-sexp. Note that first thing on that
 ;; line has to be complete sexp since we are inside the
 ;; innermost containing sexp.
 (backward-prefix-chars)
 (current-column))
 (let ((function (buffer-substring (point)
                                   (progn (forward-sexp 1) (point))))
          method)
         (setq method (get (intern-soft function) 'clips-indent-hook))
         (if (or (eq method 'defrule)
                 (and (null method)
                  (> (length function) 3)
                  (string-match "\\'def" function)))
          (clips-indent-defform state indent-point)
          normal-indent) ) ) ) )

(put 'deffacts 'clips-indent-hook 'defrule)
(put 'defglobal 'clips-indent-hook 'defrule)
(put 'deftemplate 'clips-indent-hook 'defrule)
(put 'deffunction 'clips-indent-hook 'defrule)


(defun clips-indent-defform (state indent-point)
 (goto-char (car (cdr state)))
 (forward-line 1)
 (if (> (point) (car (cdr (cdr state))))
 (progn
         (goto-char (car (cdr state)))
         (+ clips-indent-width (current-column)) ) ) )
```

```lisp
(defun end-of-clips-clause ()
 "Go to end of clause in this line."
 (interactive) ;;
 (beginning-of-line 1)
 (let* ((eolpos (save-excursion (end-of-line) (point))))
 (if (re-search-forward comment-start-skip eolpos 'move)
        (goto-char (match-beginning 0)) )
 (skip-chars-backward " \t") ) )

(defun clips-comment-indent ()
 "Compute CLIPS comment indentation."
 (cond ((looking-at ";;;") 0)
        ((looking-at ";;") (clips-indent-level))
        (t (save-excursion
         (skip-chars-backward " \t")
         (max (1+ (current-column)) ;insert at least one space
               comment-column) ) )
        ) )

(defun beginning-of-defform (&optional arg)
 "Move backward to next beginning-of-defform. With argument,
do this 'arg' number of times.
Returns t unless search stops due to end of buffer."
 (interactive "p")
 (and arg (< arg 0) (forward-char 1))
 (and (re-search-backward "^\\s(" nil 'move (or arg 1))
 (progn (beginning-of-line) t) ) )


;; {Mon Apr 27 20:19:20 1992 - ylee}
;; Adopted from prolog.el
;;
;;
(defun clips-consult-region-and-go ()
 "Send the region to the inferior CLIPS, and switch to *clips* buffer."
 (interactive)
 (copy-region-as-kill (mark) (point))
 (switch-to-buffer-other-window "*clips*")
 (end-of-buffer)
 (yank)
 (end-of-buffer)
;; (newline-and-indent)
 )

(defun clips-reconsult-buffer-and-go ()
 "Send buffer to *clips* for reconsultation."
 (interactive)
 (copy-region-as-kill (point-min) (point-max))
 (switch-to-buffer-other-window "*clips*")
 (end-of-buffer)
```

```
 (yank)
 (end-of-buffer)
;; (newline-and-indent)
)



;;;
;;;
;;; Inferior clips mode
;;;
;;;
(defvar inferior-clips-mode-map nil )

;; (defvar inferior-clips-mode-abbrev-table clips-mode-abbrev-table
;; "Abbrev table used in inferior-CLIPS-mode")

;; (set 'inferior-clips-mode-abbrev-table
;; clips-mode-abbrev-table)

(defun inferior-clips-mode ()
 "Major mode for interacting with an inferior CLIPS process.
This CLIPS version is 5.0. Version 5.1 has been released
and will require modification of this program to execute 5.1 instead
of 5.0 once it is installed. The minimum modification is in the
definition of 'run-clips'.
Instead of (switch-to-buffer (make-shell \"clips\" \"clips5\")),
 (switch-to-buffer (make-shell \"clips\" \"new executable\").
The following commands are available:
\\{inferior-clips-mode-map}

Entry to this mode calls the value of clips-mode-hook with no arguments,
if that value is non-nil. Likewise with the value of shell-mode-hook.
clips-mode-hook is called after shell-mode-hook.

You can send text to the inferior CLIPS from other buffers
using the commands send-region, send-string and \\[clips-consult-region].

Commands:
Tab indents for CLIPS; with no arguments, shifts rest of expression
rigidly with the current line.
Paragraphs are separated only by blank lines and ';'. ';' starts comments.

Return at end of buffer sends line as input.
Return not at end of buffer copies rest of line to end and sends it.
\\[shell-send-eof] sends end-of-file as input.
\\[kill-shell-input] and \\[backward-kill-word] are kill commands, imitating
normal Unix input editint.
\\[interrupt-shell-subjob] interrupts the shell or its current subjob if any.
\\[stop-shell-subjob] stops, likewise. \\[quit-shell-subjob] sends quit
signal, likewise."
 (interactive)
 (kill-all-local-variables)
 (setq major-mode 'inferior-clips-mode)
```

```
(setq local-abbrev-table clips-mode-abbrev-table) ;;
(setq mode-name "Inferior CLIPS")
(setq mode-line-process '(": %s"))
(clips-mode-variables)
(require 'shell)
(if inferior-clips-mode-map
nil
(setq inferior-clips-mode-map (copy-alist shell-mode-map))
(clips-mode-commands inferior-clips-mode-map) )
(use-local-map inferior-clips-mode-map)
(make-local-variable 'last-input-start)
(setq last-input-start (make-marker))
(make-local-variable 'last-input-end)
(setq last-input-end (make-marker))
(make-variable-buffer-local 'shell-prompt-pattern)
;; (setq shell-prompt-pattern "^|[ C][L ][! ][P ][S ][> ] *") ;set clips prompt
(setq shell-prompt-pattern "^CLIPS> ") ;set clips prompt
(run-hooks 'shell-mode-hook 'clips-mode-hook)
)


(defun run-clips ()
"Run an inferior CLIPS process, input and output via buffer *clips*."
(interactive)
(require 'shell)
(switch-to-buffer (make-shell "clips" "clips5"))
(inferior-clips-mode) )

;; (defun clips-consult-region (beg end)
;; "Send the region to the CLIPS process made by M-x run-clips."
;; (interactive "P\nr")
;; (save-excursion
;; (send-string "clips" clips-consult-string)
;; (send-region "clips" beg end)
;; (send-string "clips" "\n") ;ensure carriage return sent
;; (if clips-eof-string
;;        (send-string "clips" clips-eof-string)
;; (process-send-eof "clips") ) ) )
;;
;; (defun clips-consult-region-and-go (beg end)
;; "Send the region to the inferior CLIPS, and switch to *clips* buffer."
;; (interactive "P\nr")
;; (clips-consult-region beg end)
;; (switch-to-buffer "*clips*") )
```

APPENDIX B. CLIPS/ADA INTERFACE PACKAGE

```
--<><><><><><><><><><><><><><><><><><><><><><><><><>
--Title          :  Package Clips
--Author         :  William C. Hoppe
--Date           :  7 November 1991
--Revised        :  21 July 1992
--Course         :  Thesis
--System         :  SUN-3 UNIX
--Compiler       :  Verdix, VADS, Ver 6.0
--Description     :  This package contains the specifications and body
--              for some of the CLIPS (Ver 5.0) functions used in
--              embedded CLIPS systems.  This code is a modification
--              of the DEC Ada compiler version found on page 133 of
--              CLIPS Reference Manual, Volume II, "Advanced Programming
--              Guide", dtd 11 January 91.
--              This package is needed in order to embed CLIPS in Ada
--              applications or to run Ada programs from CLIPS.
--<><><><><><><><><><><><><><><><><><><><><><><><><>

with System;
use System;

package CLIPS is

  --Initialize the CLIPS environment upon program startup.

  procedure xInitializeCLIPS;

  --Clears the CLIPS environment.

  procedure xClearCLIPS;

  --Resets the CLIPS environment.

  procedure xResetCLIPS;

  --Loads a set of constructs into the CLIPS database.  If there are syntactic
  --errors in the constructs, xLoadConstructs will still attempt to read the
  --entire file, and error notices will be sent to werror.
  --Returns:  an integer, zero if an error occurs.

  function xLoadConstructs(File_Name : in STRING) return INTEGER;

  --Allows Run_Limit rules to fire (execute). -1 allows to fire until
  --the agenda is empty.
  --Returns:  Number of rules that were fired.

  function xRunCLIPS (Run_Limit : in INTEGER := -1) return INTEGER;
```

--Allows the dribble function of CLIPS to be turned on.
--Returns: an integer, zero if an error occurs otherwise a one.

function xOpenDribble(File_Name : in STRING) return INTEGER;

--Turns off the storing of dribble information.
--Returns: an integer, zero if an error occurs otherwise a one.

function xCloseDribble return INTEGER;

--Closes all open files in CLIPS

procedure xCloseAllFiles;

--Lists the facts in the fact-list.

procedure xListFacts;

--Turns the watch facilities of CLIPS on and off.

function xSetWatchItem (Watch_Item : in STRING;
                Value   : in INTEGER) return INTEGER;

--Asserts a fact into the CLIPS fact-list. The function version returns the
--Fact_Pointer required by xRetractFact.

function xAssertString (Pattern : in STRING) return INTEGER;

--Causes a fact asserted by the xAssertString function to be retracted.
--Returns: FALSE if fact has already been retracted, else TRUE.
--Input other values not returned by xAssertString will cause CLIPS to
--abort.

function xRetractFact (Fact_Pointer : in INTEGER) return INTEGER;

--Queries all active routers until it finds a router that recognizes the
--logical name associated with this I/O request to print a string. It then
--call the print function associated with that router.

function xPrintCLIPS (Log_Name,
                Str   : in string) return INTEGER;

--Finds a rule within the knowledge base.
--Returns: A pointer to a rule required by function such as
--xDeleteDefrule.

function xFindDefrule (Rule_Name : in STRING) return INTEGER;

--Removes a rule from CLIPS.
--Returns: FALSE if rule not found, else TRUE.

```
    function xDeleteDefrule (Rule_Pointer : in INTEGER) return INTEGER;

private

    pragma INTERFACE (C, xInitializeCLIPS);
    pragma INTERFACE_NAME (xInitializeCLIPS, "_InitializeCLIPS");

    pragma INTERFACE (C, xClearCLIPS);
    pragma INTERFACE_NAME (xClearCLIPS, "_ClearCLIPS");

    pragma INTERFACE (C, xResetCLIPS);
    pragma INTERFACE_NAME (xResetCLIPS, "_ResetCLIPS");

    function cLoadConstructs (File_Name : in ADDRESS) return INTEGER;
    pragma INTERFACE (C, cLoadConstructs);
    pragma INTERFACE_NAME (cLoadConstructs, "_LoadConstructs");

    pragma INTERFACE (C, xRunCLIPS);
    pragma INTERFACE_NAME (xRunCLIPS, "_RunCLIPS");

    function cOpenDribble (File_Name : in ADDRESS) return INTEGER;
    pragma INTERFACE (C, cOpenDribble);
    pragma INTERFACE_NAME (cOpenDribble, "_OpenDribble");

    function cCloseDribble return INTEGER;
    pragma INTERFACE (C, cCloseDribble);
    pragma INTERFACE_NAME (cCloseDribble, "_CloseDribble");

    pragma INTERFACE (C, xListFacts);
    pragma INTERFACE_NAME (xListFacts, "_ListFacts");


    pragma INTERFACE (C, xCloseAllFiles);
    pragma INTERFACE_NAME (xCloseAllFiles, "_CloseAllFiles");

    function cSetWatchItem (Item        : in ADDRESS;
                    Active_Value : in INTEGER) return INTEGER;
    pragma INTERFACE (C, cSetWatchItem);
    pragma INTERFACE_NAME (cSetWatchItem, "_SetWatchItem");

    function cAssertString (Pattern : in ADDRESS) return INTEGER;
    pragma INTERFACE (C, cAssertString);
    pragma INTERFACE_NAME (cAssertString, "_AssertString");

    function cRetractFact (Fact_Pointer : in INTEGER) return INTEGER;
    pragma INTERFACE (C, cRetractFact);
    pragma INTERFACE_NAME (cRetractFact, "_RetractFact");

    function cPrintCLIPS (Log_Name,
                    Str      : in ADDRESS) return INTEGER;
```

```
        pragma INTERFACE (C, cPrintCLIPS);
        pragma INTERFACE_NAME (cPrintCLIPS, "_PrintCLIPS");

        function cFindDefrule (Rule_Name : in ADDRESS) return INTEGER;
        pragma INTERFACE (C, cFindDefrule);
        pragma INTERFACE_NAME (cFindDefrule, "_FindDefrule");

        function cDeleteDefrule (Rule_Pointer : in INTEGER) return INTEGER;
        pragma INTERFACE (C, cDeleteDefrule);
        pragma INTERFACE_NAME (cDeleteDefrule, "_DeleteDefrule");

    end CLIPS;

    --***************************************************************

    with Text_IO;

    package body CLIPS is

        function Ada_to_C_String (Input_String : in STRING) return ADDRESS is
          Out_String : string(1..Input_String'last+1);
        begin
          for I in Input_String'range loop
            if (Input_String(I) in ' '..'~' or
                Input_String(I) = ASCII.Cr or
                Input_String(I) = ASCII.Lf ) then
              Out_String(I) := Input_String(I);
            else
              Out_String(I) := ASCII.Nul;
            end if;
          end loop;
          Out_String (Out_String'last) := ASCII.Nul;
          return Out_String'Address;
        end Ada_to_C_String;

        function xLoadConstructs(File_Name : in STRING) return INTEGER is
          package Integer_Inout is new Text_IO.Integer_IO(Integer);
        begin
          return cLoadConstructs(Ada_to_C_String(File_Name));
        end xLoadConstructs;

        function xOpenDribble(File_Name : in STRING) return INTEGER is
          int : Integer;
          package Integer_Inout is new Text_IO.Integer_IO(Integer);
        begin
    --    return cOpenDribble(Ada_to_C_String(File_Name));
          Text_IO.Put_Line("Inside the Package CLIPS, function xOpenDribble");
          int := cOpenDribble(Ada_to_C_String(File_Name));
          Text_IO.Put("The value from the cOpenDribble function is:  ");
          Integer_Inout.Put(int, width =>1);
          Text_IO.New_line;
```

76

```ada
      return int;
    end xOpenDribble;

    function xCloseDribble return INTEGER is
      int : Integer;
      package Integer_Inout is new Text_IO.Integer_IO(Integer);
    begin
--    return cCloseDribble(Ada_to_C_String(File_Name));
      Text_IO.Put_Line("Inside the Package CLIPS, function xCloseDribble");
      int := cCloseDribble;
      Text_IO.Put("The value from the cCloseDribble function is:  ");
      Integer_Inout.Put(int, width =>1);
      Text_IO.New_line;
      return int;
    end xCloseDribble;

    function xSetWatchItem (Watch_Item : in STRING;
                    Value     : in INTEGER) return INTEGER is
    begin
      return cSetWatchItem(Ada_to_C_String (Watch_Item), Value);
    end xSetWatchItem;


    function xAssertString (Pattern : in STRING) return INTEGER is
    begin
      return cAssertString(Ada_to_C_String (Pattern));
    end xAssertString;


    function xRetractFact (Fact_Pointer : in INTEGER) return INTEGER is
    begin
      return cRetractFact(Fact_Pointer);
    end xRetractFact;


    function xPrintCLIPS (Log_Name,
                    Str      : in string) return INTEGER is
    begin
      return cPrintCLIPS(Ada_to_C_String (Log_Name),
                    Ada_to_C_String (Str));
    end xPrintCLIPS;


    function xFindDefrule (Rule_Name : in STRING) return INTEGER is
    begin
      return cFindDefrule(Ada_to_C_String(Rule_Name));
    end xFindDefrule;


    function xDeleteDefrule (Rule_Pointer : in INTEGER) return INTEGER is
    begin
```

```
        return cDeleteDefrule(Rule_Pointer);
    end xDeleteDefrule;

end CLIPS;
```

APPENDIX C. CLIPS RUN-TIME EXAMPLE

```
(deffacts test
  (no error)
  (token for)
  (variable I)
  (statement I := I + 1))

(defrule for-loop-concept-error1
  ?error-status <- (no error)
  ?token1 <- (token for)
  ?variable1 <- (variable ?v)
  (statement ?v := $?)
=>
  (printout t "The Ada for loop automatically increments its counter.
Incrementing the counter manually inside a for loop may produce invalid
results." crlf)
  (retract ?error-status)
  (assert (concept error)))
```



```
        CLIPS (V5.0 3/05/91)
CLIPS> (load test1.clp)
Defining deffacts: test
Redefining defrule: for-loop-concept-error1 +j+j+j+j
CLIPS> (reset)
CLIPS> (facts)
f-0    (initial-fact)
f-1    (no error)
f-2    (token for)
f-3    (variable I)
f-4    (statement I := I + 1)
For a total of 5 facts.
CLIPS> (run)
The Ada for loop automatically increments its counter.
Incrementing the counter manually inside a for loop may produce invalid
results.
CLIPS>
```

APPENDIX D. ADA EMULATOR SOURCE CODE

```
--<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
--Title      : Editor Emulator for ITS ADA
--Author     : William C. Hoppe
--Date       : 12 May 1992
--Revised    : 7 Aug 1992
--Course     : Thesis
--System     : SUN-3 UNIX
--Compiler   : Verdix Ada, VADS, Ver 6.0
--Description : This program emulates the editor functions of the Ada
--             tutor ITS ADA.  It interacts with CLIPS in order to
--             conduct semantic checking of the student solution as
--             the solution is typed in.  This is basically a prototype
--             to learn where to break the control of ITS ADA and to
--             work out control issues between ADA and CLIPS.
--<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>

with CLIPS, Text_IO;
use CLIPS, Text_IO;

procedure clips_ada_editor is

  Dribble_File : String(1..9) := "dribl.out";
  Student_Model_File_Name,
  User_Name,
  File_Name  : String(1..20) := (others => ' ');
  File_Open_Status,
  Rules_Fired      : Integer;

  stmnt : string(1..79);

  Name_Len,
  Index    : Integer :=1;
  Problem_Number,
  stmnt_len : Integer := 0;

  type Line is record
    statement : string(1..79) := (others => ' ');
    statement_length : integer:= 0;
  end record;

  type Response is (yes, no);
  type Solution is array(1..20) of Line;

  Answer  : Response;
  Code_Segment : Solution;

  package Integer_Inout is new Integer_IO(Integer);
```

```ada
package Enum_Inout is new Enumeration_IO(Response);

--Gets the log in name of the student that should match any
--previous student modeling data maintained by the system.
--This will work as long as the student uses the same log in
--name every time.  Otherwise a new file will be created each
--time the student log in under a different name.
procedure Get_User_Name (Name     : in out String;
                         Name_Length : in out Integer) is separate;


--This function takes as input the Log in name of the student and the
--name length.  It builds the filename *.mod and attempts to open it.
--If the file is already open a STATUS_ERROR will occur which will
--execute that portion of the exception handler.  If the file does
--not exist then a NAME_ERROR will occur and the file is created. Any
--other condition will return a value of false.
function Check_for_Existing_Student_Model (File_Name    : in String;
                    File_Name_Len : in Integer)
                              return BOOLEAN is separate;


--This function takes as input the Log in name of the student and the
--name length.  It builds and returns the filename *.mod.
function Get_Student_Model(Name     : in String;
                Name_Len : in Integer)
                return String is separate;


--This procedure takes as input an integer that represents the
--problem number the student is attempting to solve.  It creates
--a wrapper in the form of a CLIPS fact with the appropriate
--.clp file.  Example:  Problem 4 would build the following fact:
--(current-problem prob4.clp) .  The expert solution for problem
--4 is in prob4.clp.  This allows CLIPS to load the expert solution.
procedure Write_Current_Problem(Prob_Num : in Integer) is separate;


--Writes the portions of the solution out to "temp.dat" or
--the entire student solution out to a file "stdnt.dat".  The
--target file is passed in as a parameter.  Solutions being
--written to "temp.dat" are preprocessed to replace the ';' with
--the token 'semicolon'.  This is because the semicolon character
--is a delimeter in CLIPS and is not written to the file.
procedure Write_Student_Solution (Code_Segment : in out Solution;
                   Line_Count   : in Integer;
                   Output_File   : in String;
                   File_Name_Len: in Integer ) is separate;


--This procedure takes as input the *.mod student model filename
--and the length of the filename.  It first initializes the CLIPS
--runtime environment.  This function should only be called once
--per procedure invocation.  It then loads the CLIPS file IO.clp
--which preprocesses the student solution into a form for CLIPS
--to use.  It then loads the student model file and the file
```

```
--parser.clp which is the rule base for the CLIPS diagnostic
--package and error rules.
procedure Invoke_CLIPS (Student_Model_File : in String;
               Name_Length      : in Integer) is separate;

--This procedure takes as input the *.mod filename of the student
--and writes out any facts that were written out to the file
--model.out by CLIPS.  It builds the CLIPS deffacts construct as
--a wrapper around the facts that are in model.out and gives it
--the log in name of the student .mod.
procedure Update_Student_Model(Output_File : in String;
               File_Name_Len : in Integer) is separate;


begin
  Get_User_Name(User_Name, Name_Len);
  if Check_for_Existing_Student_Model(User_Name, Name_Len) then
    Student_Model_File_Name(1..Name_Len + 4) :=
               Get_Student_Model(User_Name, Name_Len);
  else
    Put_Line("Can not access Current Student Model");
  end if;
  Put("Enter problem number:  ");
  Integer_Inout.Get(Problem_Number);
  Write_Current_Problem(Problem_Number);
  Skip_Line;
  Put("@  1  ");
  Get_Line(Stmnt, Stmnt_Len);
  loop
    exit when stmnt(1) = '.';
    Code_Segment(Index).Statement(1..Stmnt_Len) := Stmnt(1..Stmnt_Len);
    Code_Segment(Index).Statement_Length := Stmnt_Len;
    Write_Student_Solution(Code_Segment, Index,"temp.dat",8);
    Write_Student_Solution(Code_Segment, Index,"stdnt.dat",9);
    Invoke_CLIPS(Student_Model_File_Name, Name_Len + 4);
    Update_Student_Model(Student_Model_File_Name, Name_Len + 4);
    New_Line;
    Index := Index + 1;
    Put("@  ");
    Integer_Inout.Put(Index, width => 1);
    Put("  ");
    Get_Line(Stmnt, Stmnt_Len);
  end loop;
  Put_Line("This is your solution:");
  for I in 1..(Index-1) loop
    Integer_Inout.Put(I, width => 1);
    Put("  ");

Put_line(Code_Segment(I).Statement(1..Code_Segment(I).Statement_Length)
);
  end loop;
```

```ada
      New_Line(2);
      Put("Do you wish to modify any of the lines of code? ");
      Enum_Inout.Get(Answer);
      case Answer is
        when yes =>
          null;
        when no =>
          Index := Index - 1;
          --writing final solution to stdnt.dat
          Write_Student_Solution(Code_Segment, Index,"stdnt.dat",9);
      end case;
end clips_ada_editor;
```

```
--<><><><><><><><><><><><><><><><><><><><><><><><><><><>
--Title       : Separate Compilation Files
--Author      : William C. Hoppe
--Date        : 20 June 1992
--Revised     : 8 Aug 1992
--Course      : Thesis
--System      : SUN-3 UNIX
--Compiler    : Verdix Ada, VADS, Ver 6.0
--Description  : This file contains the separate compilation
--              units that belong to the main procedure
--              clips-ada-editor.
--<><><><><><><><><><><><><><><><><><><><><><><><><><><>


separate (clips_ada_editor )
--Gets the log in name of the student that should match any
--previous student modeling data maintained by the system.
--This will work as long as the student uses the same log in
--name every time.  Otherwise a new file will be created each
--time the student log in under a different name.
procedure Get_User_Name (Name   : in out String;
                         Name_Length : in out Integer) is
begin
  Put("WHAT IS YOUR NAME / USER ID:    ");
  Get_Line(Name,Name_Length);
end Get_User_Name;


separate (clips_ada_editor )
--This function takes as input the Log in name of the student and the
--name length.  It builds the filename *.mod and attempts to open it.
--If the file is already open a STATUS_ERROR will occur which will
--execute that portion of the exception handler.  If the file does
--not exist then a NAME_ERROR will occur and the file is created. Any
--other condition will return a value of false.
function Check_for_Existing_Student_Model (File_Name    : in String;
                         File_Name_Len : in Integer)
                                 return BOOLEAN is
  Outfile   : File_Type;
  Student_File : String(1..File_Name_Len + 4) :=
           File_Name(1..File_Name_Len) & ".mod";

begin
Open(Outfile, Out_File, Student_File);
return true;
exception
  when STATUS_ERROR =>
    Close(Outfile);
    Open(Outfile, Out_File, Student_File);
    return true;
  when NAME_ERROR =>
```

84

```ada
      Create(Outfile, Out_File, Student_File);
      return true;
   when others =>
      return false;
end Check_for_Existing_Student_Model;


separate (clips_ada_editor )
--This function takes as input the Log in name of the student and the
--name length.  It builds and returns the filename *.mod.
function Get_Student_Model(Name     : in String;
               Name_Len : in Integer) return String is

   Student_Model : String(1..Name_Len + 4) := Name(1..Name_Len) & ".mod";

begin
   return Student_Model;
end  Get_Student_Model;

separate (clips_ada_editor)
--This procedure takes as input an integer that represents the
--problem number the student is attempting to solve.  It creates
--a wrapper in the form of a CLIPS fact with the appropriate
--.clp file.  Example:  Problem 4 would build the following fact:
--(current-problem prob4.clp) .  The expert solution for problem
--4 is in prob4.clp.  This allows CLIPS to load the expert solution.
procedure Write_Current_Problem(Prob_Num : in Integer) is
   Outfile : File_Type;
   File_Name : String(1..16) := "current_prob.dat";
   Problem_Number : Integer := Prob_Num;
begin
   Open(Outfile, Out_File, File_Name);
   Put(Outfile, "(current-problem prob");
   Integer_Inout.Put(Outfile, Problem_Number, width => 1);
   Put(Outfile, ".clp)");
   Close(Outfile);
end Write_Current_Problem;


--Writes the portions of the solution out to "temp.dat" or
--the entire student solution out to a file "stdnt.dat".  The
--target file is passed in as a parameter.  Solutions being
--written to "temp.dat" are preprocessed to replace the ';' with
--the token 'semicolon'.  This is because the semicolon character
--is a delimeter in CLIPS and is not written to the file.
separate (clips_ada_editor)
procedure Write_Student_Solution (Code_Segment : in out Solution;
               Line_Count   : in Integer;
               Output_File    : in String;
               File_Name_Len: in Integer ) is
   Outfile  : File_Type;
```

85

```ada
   File_Name : String(1..File_Name_Len) := Output_File(1..File_Name_Len);
   Count      : Integer := Line_Count;
   Last_Char,
   End_of_Line,
   Statement_Length : Integer := 0;
   Statement : string(1..79) := (others => ' ');
   Writing_Temp_File : Boolean := False;

begin
  Open(Outfile, Out_File, File_Name);
  if Output_File(1..4) = "temp" then
    Writing_Temp_File := True;
  end if;
  for I in 1..Count loop
    if Writing_Temp_File then
      Statement := Code_Segment(I).Statement;
      Statement_Length := Code_Segment(I).Statement_Length;
      --check to see if the last character in the string is a semicolon
      if Statement(Statement_Length) = ';' then
        End_of_Line := Statement_Length + 9;
        Statement(1..End_of_Line) :=
                  Statement(1..Statement_Length -1) & " semicolon";
        Put_Line(Outfile, Statement(1..End_of_Line));
      else
      Put_Line(Outfile,
          Code_Segment(I).Statement(1..Code_Segment(I).Statement_Length));
      end if;
    else
      Put_Line(Outfile,
          Code_Segment(I).Statement(1..Code_Segment(I).Statement_Length));
    end if;
  end loop;
  Close(Outfile);
end Write_Student_Solution;


separate (clips_ada_editor)
--This procedure takes as input the *.mod student model filename
--and the length of the filename.  It first initializes the CLIPS
--runtime environment.  This function should only be called once
--per procedure invocation.  It then loads the CLIPS file IO.clp
--which preprocesses the student solution into a form for CLIPS
--to use.  It then loads the student model file and the file
--parser.clp which is the rule base for the CLIPS diagnostic
--package and error rules.
procedure Invoke_CLIPS (Student_Model_File : in String;
                Name_Length        : in Integer) is

begin
  xInitializeCLIPS;
  File_Name(1..6) := "IO.clp";
```

```ada
   File_Open_Status := xLoadConstructs(File_Name(1..6));
   if File_Open_Status = 0 then
     xResetCLIPS;
--   xListFacts;
     Rules_Fired := xRunCLIPS(-1);
--    Put(Integer'Image(Rules_Fired));
--    Put_Line(" Rules Fired");
     xCloseAllFiles;
   else
     Put_Line("Unable to open rules file.");
   end if;
   File_Open_Status :=
xLoadConstructs(Student_Model_File(1..Name_Length));
   File_Name(1..10) := "parser.clp";
   File_Open_Status := xLoadConstructs(File_Name(1..10));
   if File_Open_Status = 0 then
     xResetCLIPS;
--   xListFacts;
     Rules_Fired := xRunCLIPS(-1);
--    Put(Integer'Image(Rules_Fired));
--    Put_Line(" Rules Fired");
     xCloseAllFiles;
   else
     Put_Line("Unable to open rules file.");
   end if;
end Invoke_CLIPS;


separate (clips_ada_editor)
--This procedure takes as input the *.mod filename of the student
--and writes out any facts that were written out to the file
--model.out by CLIPS.  It builds the CLIPS deffacts construct as
--a wrapper around the facts that are in model.out and gives it
--the log in name of the student .mod.
procedure Update_Student_Model(Output_File : in String;
                    File_Name_Len : in Integer) is
   Line : String(1..79) := (others => ' ');
   Line_Length : Integer := 0;
   Student_Model,
   CLIPS_File  : File_Type;
   File_Name : String(1..File_Name_Len) := Output_File(1..File_Name_Len);
begin
   Open(CLIPS_File, In_File,"model.out");
   Open(Student_Model, Out_File, File_Name);
   Put_Line(Student_Model, "(deffacts Student_Model");
   While not END_OF_FILE(CLIPS_File) loop
    Put(Student_Model, "  (");
    Get_Line(CLIPS_File,Line,Line_Length);
    Put(Student_Model, Line(1..Line_Length));
    Put(Student_Model, ')');
   end loop;
```

```
   Put_Line(Student_Model, " )");
   Close(CLIPS_File);
   Close(Student_Model);
end Update_Student_Model;
```

APPENDIX E. CLIPS RULES

```
;;<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
;;Title      : IO.clp
;;Author     : William C. Hoppe
;;Date       : 26 June 1992
;;Revised    : 22 August 1992
;;Course     : Thesis
;;System     : SUN-4 UNIX
;;Compiler   : CLIPS version 5.0 Interpreter
;;Description : These CLIPS rules create a loop that reads each
;;               "token" out of a file named temp.dat and writes
;;               out a token per line to a file temp.out.  This
;;               file can be done without for efficiency purposes.
;;               It was initially written to get output from the Ada/
;;               CLIPS interface.
;;<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>


(defrule open-files
  (initial-fact)
=>
  (open temp.dat raw-data "r") ;student input file is raw data
  (open temp.out parsed-data "w") ;input file is data token
  (assert (read-file)))

(defrule read-input-file
  ?read-file <- (read-file)
=>
  (retract ?read-file)
  (assert (input-read =(read raw-data))))

(defrule write-output
  ?input-read <- (input-read ?input&~EOF)
=>
  (retract ?input-read)
  (printout parsed-data ?input crlf)
; (printout t "Data written:  " ?input crlf)
  (assert (read-file)))

(defrule close-files
  ?close-files <- (input-read EOF)
=>
  (retract ?close-files)
  (close))
```

```
;;<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
;;Title      : parser.clp
;;Author     : William C. Hoppe
;;Date       : 26 June 1992
;;Revised    : 25 August 1992
;;Course     : Thesis
;;System     : SUN-4 UNIX
;;Compiler   : CLIPS version 5.0 Interpreter
;;Description : These CLIPS rules both parse and diagnose the lines
;;             of code found in the file temp.out.  If one of the
;;             error rules fires it will write a code out to a file
;;             named model.out for the Ada program to convert to the
;;             enhanced student model.
;;<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>


;Control fact necessary to start up the system since no connection is
;available yet to the tutor ITS Ada.
(deffacts start-facts
  (no error))

;Global variable that holds the current line of text from the file
;temp.dat.  This is necessary in order to print out the error message
;and in order to do some matching on sub-strings within the code segment.
(defglobal
  ?*current-statement* = (mv-append EOF))

;Opens the student solution file and the token file which will be
;derived from the student solution.  Then asserts the control
;fact that begins reading the files.
(defrule open-files
  (initial-fact)
=>
  (load-facts current_prob.dat)
  (open temp.dat data "r") ;student input file is raw data
  (open temp.out input "r") ;input file is data token
  (open model.out model "w") ;student model facts to be asserted
  (assert (read-data)))

;Loads the facts from the expert solution for the problem the student
;is solving.
(defrule load-expert-solution
  (declare (salience 15))
  (current-problem ?number)
=>
  (load-facts ?number)
  (bind ?num (eval (sub-string 5 5 ?number)))
  (assert (problem ?num)))


;Gets a line of code from the student's solution.  Asserts it as a
```

```clips
;fact in the fact base for string comparison and multifield comparisons.
;Then is asserts the control fact which enables the parsing of the tokens.
(defrule read-screen-input
  ?read-data <- (read-data&~EOF)
=>
  (retract ?read-data)
  (bind ?string (readline data))
  (bind ?*current-statement* ?string)
  (if (neq ?*current-statement* EOF)
   then
   (str-assert ?string)
   else
   (assert (data-read EOF)))
  (assert (read-file)))

;Closes the student solution file when the end of file (EOF) is reached
;and both read-data and read-file facts are on the fact list.
(defrule close-files
  (declare (salience 20))
  ?close-files <- (data-read EOF)
  ?read-data <- (read-data)
  ?read-file <- (read-file)
  ?initial <- (initial-fact)
=>
  (retract ?close-files ?read-data ?read-file ?initial)
 ; (printout t "Closing the input files from inside CLIPS" crlf)
  (close))

;Closes the student solution file when the end of file (EOF) is reached
;and only the read-data fact is on the fact list.
(defrule close-files2
  (declare (salience 20))
  ?close-files <- (data-read EOF)
  ?read-data <- (read-data)
  ?initial <- (initial-fact)
=>
  (retract ?close-files ?read-data ?initial)
 ; (printout t "Closing the input files from inside CLIPS" crlf)
  (close))

;Closes the student solution file when the end of file (EOF) is reached
;and only read-file fact is on the fact list.
(defrule close-files3
  (declare (salience 20))
  ?close-files <- (data-read EOF)
  ?read-file <- (read-file)
  ?initial <- (initial-fact)
=>
  (retract ?close-files ?read-file ?initial)
 ; (printout t "Closing the input files from inside CLIPS" crlf)
  (close))
```

```
;Gets the next token from the input file temp.out
(defrule read-input-file
  (declare (salience 1))
  (no error)
  ?read-file <- (read-file)
=>
  (retract ?read-file)
  (assert (data-read =(read input))))


;************
; for loop  *
;************

;For token identifier
(defrule for-loop-token
  (no error)
  ?data-read <- (data-read for)
=>
  (retract ?data-read)
  (assert (token for)
        (need loop parameter spec)
        (read-file)))

(defrule loop-parameter-specification
  (no error)
  (token for)
  ?flag <- (need loop parameter spec)
  ?data-read <- (data-read ?variable)
=>
  (retract ?data-read ?flag)
  (assert (identifier ?variable)
        (need reserved word in)
        (read-file)))

(defrule reserved-word-in
  (no error)
  ?flag1 <- (need reserved word in)
  ?data-read <- (data-read in)
=>
  (retract ?flag1 ?data-read)
  (assert (token in)
        (need discrete range)
        (read-file)))

(defrule discrete-range
  (no error)
  ?flag1 <- (need discrete range)
  ?data-read <- (data-read $?range )
  (? ? ? ?range ?)
=>
```

```
(retract ?flag1 ?data-read)
(assert (range ?range)
      (need reserved word loop)
      (read-file)))



;**************
;
; while loop  *
;**************
;

;While token identifier
(defrule while-loop-token
  (no error)
  ?data-read <- (data-read while)
=>
  (retract ?data-read)
  (assert (token while)
      (need boolean expression)
      (read-file)))

(defrule while-loop-boolean-expression
  (no error)
  (token while)           .
  ?flag <- (need boolean expression)
  (while $?variable loop)
=>

  (assert (while-boolean-exp $?variable)
      (token loop)
          (sequence of statements)
          (need reserved word end loop)
      (read-data)
          (dump-read-file)))

;Reads the file temp.out until it reads the reserved word
;loop.  The tokens to this point have already been determined
;and added to the fact list.  This allows the read file marker
;to catch up to the program.
(defrule dump-read-file1
  (declare (salience 2))
  ?dump-flag <- (dump-read-file)
  (or (token while)
      (token for))
  ?data-read <- (data-read ?var)
  (test (neq ?var loop))
=>
  (retract ?data-read ?dump-flag)
  (assert (dump-read-file)
      (read-file)))
```

```
;Terminates the read cylce on this line of code and allows
;the normal continuation of the diagnosis.  After the loop
;reserved word is read the dump stops.  The read file marker
;has caught up to the program.
(defrule dump-read-file2
  (declare (salience 2))
  ?dump-flag <- (dump-read-file)
  (or (token while)
      (token for))
  ?data-read <- (data-read ?var)
  (test (eq ?var loop))
=>
  (retract ?data-read ?dump-flag)
  (assert (read-data)))

(defrule reserved-word-loop
  (no error)
  ?flag1 <- (need reserved word loop)
  ?data-read <- (data-read loop)
=>
  (retract ?flag1 ?data-read)
  (assert (token loop)
        (need reserved word end loop)
        (read-file)
        (read-data)))

;*************
; basic loop *
;*************
(defrule basic-loop
  (no error)
  ?data-read <- (data-read loop)
  (and (not (token for))
      (not (token while)))
  (not (end loop semicolon))
=>
  (retract ?data-read)
  (assert (token loop)
        (check for exit condition)
            (need reserved word end loop)
            (read-data)
            (read-file)))

(defrule exit-when-condition
  (declare (salience 5))
  ?flag <- (check for exit condition)
  (exit when $?boolean-cond semicolon)
=>
  (retract ?flag)
  (assert (statements)
            (token exit when)
```

```
                    (loop-exit-cond $?boolean-cond)
                    (sequence of statements)
                    (dump-read-file)
                    (read-file)))

(defrule exit-condition
  (declare (salience 5))
  ?flag <- (check for exit condition)
  (exit $?boolean-cond semicolon)
  (not (exit when $?boolean-cond semicolon))
=>
  (retract ?flag)
  (assert (statements)
          (token exit)
              (loop-exit-cond $?boolean-cond)
          (sequence of statements)
          (dump-read-file)
          (read-file)))

(defrule end-loop
  (declare (salience 5))
  ?flag <- (need reserved word end loop)
  (end loop semicolon)
=>
  (retract ?flag)
  (assert (token end loop)
        (read-file)))
;          (read-data)))

;***************
;
;if expression *
;***************
;

(defrule if-boolean-expression
  (no error)
  ?data-read <- (data-read if)
  (if $?boolean-exp then)
  (not (end if semicolon))
=>
  (assert (token if)
        (if-boolean-exp $?boolean-exp)
        (token then)
          (sequence of statements)
          (need reserved word end if)
          (check for elsif)
          (check for else)
        (read-data)
          (dump-read-file)))

(defrule elsif-expression
  ?flag <- (check for elsif)
```

```
      (elsif $?boolean-exp then)
=>
  (retract ?flag)
  (assert (token elsif)
       (check for repeated elsif)
       (elsif-boolean-exp $?boolean-exp)
       (token then)
           (sequence of statements)
           (read-data)
           (dump-read-file)))

(defrule repeated-elsif-expression
 ?flag <- (check for repeated elsif)
 (elsif $?boolean-exp then)
 (elsif-boolean-exp $?previous-bexp)
 (test (neq $?boolean-exp $?previous-bexp))
=>
  (retract ?flag)
  (assert (token elsif)
       (elsif-boolean-exp $?boolean-exp)
       (token then)
           (sequence of statements)
           (read-data)
           (dump-read-file)))

(defrule else-guard
 ?flag <- (check for else)
 (else $?statement)
=>
  (retract ?flag)
  (assert (token else)
       (sequence of statements)
           (read-data)))

;Reads the file temp.out until it reads the reserved word
;then.  The tokens to this point have already been determined
;and added to the fact list.  This allows the read file marker
;to catch up to the program.
(defrule dump-read-file3
  (declare (salience 2))
  ?dump-flag <- (dump-read-file)
  ?data-read <- (data-read ?var)
  (not (statements))
  (test (neq ?var then))
=>
  (retract ?data-read ?dump-flag)
  (assert (dump-read-file)
       (read-file)))

;Terminates the read cylce on this line of code and allows
;the normal continuation of the diagnosis.  After the then
```

```
;reserved word is read the dump stops.  The read file marker
;has caught up to the program.
(defrule dump-read-file4
  (declare (salience 2))
  ?dump-flag <- (dump-read-file)
  ?data-read <- (data-read ?var)
  (not (statements))
  (test (eq ?var then))
=>
  (retract ?data-read ?dump-flag)
  (assert (read-file)))


(defrule sequence-of-statements
  (initial-fact)
  ($?statement semicolon)
=>
  (assert (statements)
       (read-data)
       (dump-read-file)))

;Reads the file temp.out until it reads the reserved word
;semicolon.  The tokens to this point have already been determined
;and added to the fact list.  This allows the read file marker
;to catch up to the program.
(defrule dump-read-file5
  (declare (salience 2))
  ?dump-flag <- (dump-read-file)
  ?stmnt-flag <- (statements)
  ?data-read <- (data-read ?var&~EOF)
  (test (neq ?var semicolon))
=>
  (retract ?data-read ?dump-flag ?stmnt-flag)
  (assert (dump-read-file)
       (statements)
       (read-file)))

;Terminates the read cylce on this line of code and allows
;the normal continuation of the diagnosis.  After the semicolon
;reserved word is read the dump stops.  The read file marker
;has caught up to the program.
(defrule dump-read-file6
  (declare (salience 2))
  ?dump-flag <- (dump-read-file)
  ?stmnt-flag <- (statements)
  ?data-read <- (data-read ?var)
  (test (eq ?var semicolon))
=>
  (retract ?data-read ?dump-flag ?stmnt-flag)
  (assert (read-file)
       (read-data)))
```

```
(defrule end-if
  (declare (salience 5))
  ?flag <- (need reserved word end if)
  (end if semicolon)
=>
  (retract ?flag)
  (assert (token end if)
      (read-file)))


;*********************
;concept error rules *
;*********************

;Infinite loop in basic loop construct.  No exit condition in the loop.
(defrule infinite-loop-in-basic-loop
  (declare (salience 10))
  ?init <- (initial-fact)
  ?error-status <- (no error)
  ?token1 <- (token loop)
  ?token2 <- (token end loop)
  (and (not (token exit))
      (not (token exit when)))
=>
  (printout model "ilibl" crlf)
  (printout t "You have ended the loop construct without an exit condition.
This will cause an INFINITE loop once entered.  You may want to re-evaluate
the loop conditions." crlf)
  (retract ?error-status ?init)
  (assert (concept error)
      (syntax error)
      (lrm5.5.4))
  (close))

;Exit condition instead of exit when condition.  Loop will only execute
;one iteration down to the exit condition and terminate the loop.
(defrule faulty-exit-from-basic-loop
  (declare (salience 10))
  ?init <- (initial-fact)
  ?error-status <- (no error)
  (problem 4)
  (token exit)
  (not (token exit when))
=>
  (printout model "fefbl" crlf)
  (printout t "This loop will only execute one time because the exit
statement will terminate the loop on the first pass.  Check the exit
conditions of the problem." crlf)
  (retract ?error-status ?init)
  (assert (concept error)
      (syntax error)
```

```
      (lrm5.7))
  (close))

;Boolean condition for exit condition does not meet specifications of the
;problem statement.
(defrule faulty-boolean-cond-basic-loop
  (declare (salience 10))
  ?init <- (initial-fact)
  ?error-status <- (no error)
  (problem 4)
  (loop-exit-cond $?condition)
  (expert boolean-cond $?exp-cond)
  (test (neq $?exp-cond $?condition))
=>
  (printout model "fbcbl" crlf)
  (printout t "Your boolean condition "$?condition " will not meet the
criteria for exiting the loop.  You may want to look at the exit condition
again." crlf)
  (retract ?error-status ?init)
  (assert (concept error)
        (syntax error))
  (close))

;Concept error in the construction of a for loop.  In the body of the loop
;the loop variable appears on the left side of an assignment statement.
(defrule for-loop-concept-error1
  (declare (salience 10))
  ?init <- (initial-fact)
  ?error-status <- (no error)
  ?token1 <- (token for)
  ?identifier <- (identifier ?v)
  ?statement <- (?v := $?)
=>
  (printout model "flce1" crlf)
  (printout t "ERROR: " ?*current-statement* crlf)
  (printout t "The Ada for loop automatically increments its counter.
Within the sequence of statements the loop parameter is a constant. Hence
a for loop parameter is not allowed as a variable on the LHS of an
assignment statement (LRM 5.5.6)." crlf)
  (retract ?error-status ?init)
  (assert (concept error)
        (syntax error)
        (lrm5.5.6-8))
  (close))

;Concept error in the construction of the basic loop.  Attempt to
;concatentate the if construct and the loop construct.
(defrule loop-concept-error1
  (declare (salience 10))
  ?init <- (initial-fact)
  ?error-status <- (no error)
```

```
    ?statement <- (if $? loop)
=>
  (printout model "Ice1" crlf)
  (printout t "ERROR: " ?*current-statement* crlf)
  (printout t "The Ada \"if\" statement does not have a loop construct
(LRM 5.3.2).  You are trying to write a looping construct.  Recommend
you review the basic loop (LRM 5.5.9), the \"while loop\" (LRM 5.5.10)
and the \"for loop\" (LRM 5.5.11)." crlf)
  (retract ?error-status ?init)
  (assert (concept error))
  (close))

;Concept error in the building of a for loop.  Using a boolean condition
;as the loop test instead of a loop parameter specification.
(defrule iteration-scheme-error1
  (declare (salience 10))
  ?init <- (initial-fact)
  ?error-status <- (no error)
  ?statement <- (for $?loop-para-spec loop)
  (test (not(member in $?loop-para-spec)))
=>
  (printout model "ise1" crlf)
  (printout t "ERROR: " ?*current-statement* crlf)
  (printout t "The Ada \"for loop\" (LRM 5.5.6) uses a loop parameter
specification (LRM 5.5.2) not a boolean condition.  If you are going to
use a \"for loop\" construct you need to re-evaluate the loop parameter."
 crlf)
  (retract ?error-status ?init)
  (assert (concept error)
       (syntax error))
  (close))

;Concept error in the building of a while loop.  Using a loop parameter
;specification as the loop test instead of a boolean condition.
(defrule iteration-scheme-error2
  (declare (salience 10))
  ?init <- (initial-fact)
  ?error-status <- (no error)
  ?statement <- (while $?boolean-cond loop)
  (test (member in $?boolean-cond))
=>
  (printout model "ise2" crlf)
  (printout t "ERROR: " ?*current-statement* crlf)
  (printout t "The Ada \"while loop\" (LRM 5.5.5) uses a boolean condition
(LRM 5.5.5) not a loop parameter specification.  If you are going to
use a \"while loop\" construct you need to re-evaluate the boolean
condition:  while <boolean condition> loop ." crlf)
  (retract ?error-status ?init)
  (assert (concept error)
       (syntax error))
  (close))
```

```
;Given that the student is solving problem number 2, the best construct
;to use is the for loop.  Forcing any other construct may work but is
;inefficient.
(defrule loop-efficiency-error1
  (declare (salience 10))
  ?init <- (initial-fact)
  ?error-status <- (no error)
  (problem 2)
  (token loop)
  (not (token for))
=>
  (printout model "lee1" crlf)
  (printout t "Current Statement:  " ?*current-statement* crlf)
  (printout t "This loop may work but you will have to increment a counter
within the code.  Ada provides a looping construct that automatically
increments the loop counter (\"for loop\" LRM 5.5.6).  You may want to look
at how to use it." crlf)
  (assert (efficiency error)))

;Given that the student is solving problem 3, the best construct
;to use is the while loop.  There is insufficient information in the
;problem statement to use the for loop as a substitute.  The basic
;loop will work given the proper exit condition.
(defrule loop-efficiency-error2
  (declare (salience 10))
  ?init <- (initial-fact)
  ?error-status <- (no error)
  (problem 3)
  (token loop)
  (not (token while))
=>
  (printout model "lee2" crlf)
  (printout t "Current Statement:  " ?*current-statement* crlf)
  (printout t "The problem statement did not give you enough information
to determine how many accounts to loop through.  A \"for loop\" will not
meet the criteria of the problem.  The \"while loop\" is the most
appropriate choice of constructs however, the basic loop with the proper exit
condition is a valid alternative solution." crlf)
  (assert (efficiency error)))
```

```
;;<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
;;Title      : explanation.clp
;;Author     : William C. Hoppe
;;Date       : 28 August 1992
;;Revised    :
;;Course     : Thesis
;;System     : SUN-4 UNIX
;;Compiler   : CLIPS version 5.0 Interpreter
;;Description : These CLIPS rules are the Ada language reference
;;             manual (LRM) explanations.  They are used to tutor
;;             the student on errors identified during problem
;;             solving.
;;<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>

;Response rule that allows the student to decide whether or not to view
;an explanation of the rule violated and an example of the correct code.
(defrule answer-yes-or-no
  (declare (salience 20))
  ?error-flag <- (concept error)
  (lrm ?)
=>
  (retract ?error-flag)
  (printout t "Do you want to see an explanation and example (yes or no)? ")
  (bind ?answer (read))
  (if  (eq ?answer yes)
   then
   (assert (explain)))
  (if (and (neq ?answer no)
        (neq ?answer yes))
   then
   (printout t "Answer yes or no." crlf)
   (assert (concept error))))


;LRM reference for the basic loop construct
(defrule LRM-5.5.4
  ?LRM-para <- (lrm 5.5.4)
  ?explanation-flag <-(explain)
=>
  (retract ?LRM-para ?explanation-flag)
  (printout t crlf)
  (printout t crlf)
  (printout t "According to the LRM a loop statement without an iteration
scheme specifies repeated execution of the sequence of statements.
Execution of the loop statement is complete when the loop is left as a
consequence of the execution of an exit statement, or as a consequence
of some other transfer of control(LRM 5.1)." crlf)
  (printout t crlf)
  (printout t "The syntax of the basic loop is:
     loop                    loop
     [sequence of statements]     [sequence of statements]
```

```
      exit;                    exit when [boolean condition];
      [sequence of statements]     [sequence of statements]
      end loop;                end loop;" crlf)
  (printout t crlf)
  (printout t "Example:
      loop
        Get(Current_Character);
        exit when Current_Character = '*';
      end loop;" crlf))


;LRM reference for the while loop construct
(defrule LRM-5.5.5
  ?LRM-para <- (lrm 5.5.5)
  ?explanation-flag <-(explain)
=>
  (retract ?LRM-para ?explanation-flag)
  (printout t crlf)
  (printout t crlf)
  (printout t "According to the LRM a loop statement with a \"while\" iteration
scheme, the condition is evaluated before each execution of the sequence of
statement; if the value of the condition is TRUE, the sequence of statements
is executed, if FALSE the execution of the loop statement is complete." crlf)
  (printout t crlf)
  (printout t "The syntax of the while loop is:
      while [boolean expression] loop
        [sequence of statements]
      end loop;" crlf)
  (printout t crlf)
  (printout t "Example:
      while Bid(N).Price < Cut_Off.Price loop
        Record_Bid(Bid(N).Price);
        N := N + 1;
      end loop;" crlf))


;LRM reference for the for loop construct
(defrule LRM-5.5.6-8
  ?LRM-para <- (lrm 5.5.6-8)
  ?explanation-flag <-(explain)
=>
  (retract ?LRM-para ?explanation-flag)
  (printout t crlf)
  (printout t crlf)
  (printout t "According to the LRM a loop statement with a \"for\" iteration
scheme, the loop parameter specification is the declaration of the loop
parameter with the given identifier.  The loop parameter is an object whose
type is the base type of the discrete range (see 3.6.1). ")
  (printout t " Within the seqence of statements, the loop parameter is a
constant.  Hence a loop parameter must not be given as an \"out\" or \"in out\"
parameter of a procedure or entry call statement, or as an \"in out\" parameter
```

of a generic instantiation." crlf)
    (printout t crlf)
    (printout t "For the execution of a loop statement with a \"for\" iteration
scheme, the loop parameter specification is first elaborated. This elaboration
creates the loop parameter and evaluates the discrete range." crlf)
    (printout t crlf)
    (printout t "If the discrete range is a null range, the execution of the loop
statement is complete. Otherwise, the sequence of statements is executed
once
for each value of the discrete range. Prior to each such iteration, the
corresponding value of the discrete range is assigned to the loop parameter."
crlf)
    (printout t crlf)
    (printout t "The syntax of the for loop is:
        for [loop parameter specification] loop
          [sequence of statements]
        end loop;" crlf)
    (printout t crlf)
    (printout t"Example:
        for J in Buffer'Range loop   --legal even with a null range
          if Buffer(J) /= Space then
            Put(Buffer(J));
          end if;
        end loop;" crlf))

;LRM reference for the for exit statements
(defrule LRM-5.7
  ?LRM-para <- (lrm 5.7)
  ?explanation-flag <-(explain)
=>
    (retract ?LRM-para ?explanation-flag)
    (printout t crlf)
    (printout t crlf)
    (printout t "According to the LRM an exit statement is used to complete the
execution of an enclosing loop statement; the completion is conditional if the
exit statement includes a condition." crlf)
    (printout t crlf)
    (printout t "An exit statement applies to the innermost enclosing loop." crlf)
    (printout t crlf)
    (printout t "For the execution of an exit statement, the condition, if present,
is first evaluated. Exit from the loop then takes place if the value is TRUE or
if there is no condition." crlf)
    (printout t crlf)
    (printout t "The syntax of the exit statement is:
        exit [when boolean condition];" crlf)
    (printout t"Example:
        for N in 1..Max_Num_Items loop   --This code will only get one
          Get_New_Item(New_Item);   --New_Item because of the exit
          Merge_Item(New_Item, Storage_File); --statement.
          exit;

```
    end loop;" crlf)
(printout t crlf)
(printout t "    for N in 1..Max_Num_Items loop        --This code will execute to
    Get_New_Item(New_Item);          --Max_Num_Items or when the
    Merge_Item(New_Item, Storage_File); --New_Item equals the
    exit when New_Item = Terminal_Item; --Terminal_Item.
    end loop; " crlf))
```

Expert Solution Fact Files

prob1.clp

```
(expert ibe value1 - value2 > 0)
(expert eibe  value1 - value2 < 0)
(expert token else)
(expert token end if)
```

prob2.clp

```
(expert range 1..100)
(expert stmnt sum := sum + I)
(expert token end loop)
```

prob3.clp

```
(expert be account_balance >= 500.0)
(expert stmnt account_balance := fc Compound_Interest fcp account_balance)
(expert token end loop)
```

prob4.clp

```
(expert token loop)
(expert token exit when)
(expert boolean-cond value < 0)
(expert stmnt sum := sum + value)
(expert token end loop)
```

APPENDIX F. ADA SURVEY

This is a short survey to collect and catalog bugs in Ada. We request that you answer the questions with your first impulse. The short coding questions should be done without the use of any references. Once you have written your answer **don't go back** to it even if you realize you made a mistake. It is these types of errors we are trying to catalog. Thank you for your assistance.

How would you catagorize your level of programming experience?

☐ Novice (Never Programmed before)   ☐ Beginner (Some home programmin

☐ Intermediate (Some Undergraduate Classes)   ☐ Advanced

If you have previous programming experience, in what languages?

_____

_____

_____

_____

In the next four questions you will be asked to write Ada code segments. Make the assumption that any variable you need has already been declared. Do not concern yourself with anything except the code segment. For example, if you were asked to write a code segment that uses a **case statement** to print out a different message depending on the day of the week your answer might look like this:

```
case Day_of _Week is
    when Monday =>
        Put_Line("Let's hope its not one of those days!");
    when Tuesday =>
        Put_Line("Two Mondays in a row is not fair!");
    when Wednesday =>
        Put_Line("Hump day!  Halfway to another weekend.");
    when Thursday =>
        Put_Line("Hang on, the weekend starts tomorrow!");
    when Friday =>
        Put_Line("TGIF!");
    when Saturday =>
        Put_Line("Party time!");
    when Sunday =>
        Put_Line("Gotta enjoy today, tomorrow is Monday!");
end case;
```
Thank you again for your cooperation.

1. Use the Ada **if** construct to code the following senario. You have three alternatives: if the value of the stock goes up, wait till tomorrow to decide to sell or not, if the value of the stock remains the same, buy more stock, otherwise, sell.

```
if Todays_Value - Yesterdays_Value  > 0 then
      Wait := True;
elsif Todays_Value - Yesterdays_Value  = 0 then
      Buy := True;
else
      Sell := True;
end if;
```

There are three basic Ada looping constructs: For ,While, and the basic loop which may or may not have an exit condition. In the following three questions use each of the looping constructs only once but use all three constructs to code the questions. Chose the construct you feel is appropriate for that question and code the question. Remember to use each construct only once.

2. Write a loop that sums the numbers from 1 to 100 inclusive.

```
Sum := 0;
for I in 1..100 loop
      Sum := Sum + I;
end loop;
```

3. As long as the daily account balance is above $500.00, calculate compound interest. Assume there is a function **Compund_Interest** available to you with what ever parameters you need.

```
while Account_Balance >= 500.0 loop
      Account_Balance := Compound_Interest(Account_Balance);
end loop;
```

4. As long as the input value is greater than or equal to zero, sum the numbers.

```
Get(Value);
Sum := 0;
loop
      exit when Value < 0;
      Sum := Sum + Value;
      Get(Value);
end loop;
```

REFERENCES

[ADAM 80] Adam, A., and Lautent, J., "LAURA, A System to Debug Student Programs", *Artificial Intelligence*, No. 15, 1980.

[ALLE 91] Allemang, D., "Using Functional Models in Automatic Debugging", *IEEE Expert*, December 1991.

[ANDE 85] Anderson, J. and Reiser, B., "The LISP Tutor", *Byte*, April 1985.

[ANDE 88] Anderson, J., "The Expert Module", in M. Polson and J. Richardson (Ed.), *Foundations of Intelligent Tutoring Systems*, Lawrence Erlbaum Assoc. Publishers, 1988.

[ANDE 90] Anderson, J., Boyle, C. F., Corbett, A., and Lewis, M., "Cognitive Modeling and Intelligent Tutoring", *Artificial Intelligence*, No. 42, 1990.

[BARN 89] Barnes, J., *Programming in Ada, Third Edition*, Addison-Wesley Publishing Co., 1989.

[BARR 82] Barr, A. and Feigenbaum, E., *The Handbook of Artificial Intelligence, Volume II*, Addison-Wesley Publishing Co., 1982.

[BOOC 87] Booch, G., *Software Engineering with Ada, Second Edition*, The Benjamin/ Cummings Publishing Company Inc., 1987.

[BURN 88] Burns, H. and Capps, C., "Foundations of Intelligent Tutoring Systems: An Introduction", in M. Polson and J. Richardson (Ed.), *Foundations of Intelligent Tutoring Systems*, Lawrence Erlbaum Assoc. Publishers, 1988.

[BURN 91a] Burns, H. and Parlett, J., "The Evolution of Intelligent Tutoring Systems: Dimensions of Design", in H. Burns, J. Parlett, and C. Redfield (Ed.), *Intelligent Tutoring Systems, Evolutions in Design*, Lawrence Erlbaum Assoc. Publishers, 1991.

[BURN 91b] Burns, H., Parlett, J., and Redfield, C., *Intelligent Tutoring Systems, Evolutions in Design*, Lawrence Erlbaum Assoc. Publishers, 1991.

[CLAN 87a] Clancey, W. J., *Knowledge Based Tutoring: The Guidon Program*, The MIT Press, 1987.

[CLAN 87b] Clancey, W. J., "Methodology for Building an Intelligent Tutoring System", in G.P. Kearsley (Ed.), *Artificial Intelligence and Instruction: Applications and Methods*, Addison-Wesley Publishing Co., 1987.

[COUS 90]   Cousot, P., "Methods and Logic for Proving Programs", in J. van Leeuwen (Ed.), *Handbook of Theoretical and Computer Science, Volume B, Informal Models and Semantics*, The MIT Press/Elsevier, 1990.

[DELO 91]   DeLooze, L., *ITS Ada: An Intelligent Tutoring System for the Ada Programming Language*, Masters Thesis, Naval Postgraduate School, Monterey, CA, 1991.

[DERS 90]   Dershowitz, N. and Jouannaud, J., "Rewrite Systems", in J. van Leeuwen (Ed.), *Handbook of Theoretical and Computer Science, Volume B, Informal Models and Semantics*, The MIT Press/Elsevier, 1990.

[FEUR 88]   Feurzeig, W. and Ritter F., "Understanding Reflective Problem Solving", in J. Psotka, L. Massey, and S. Mutter (Ed.), *Intelligent Tutoring Systems, Lessons Learned*, Lawrence Erlbaum Assoc. Publishers, 1988.

[GIAR 89]   Giarratano, J. and Riley, G., *Expert Systems, Principles and Programming*, PWS-Kent Publishing Co., 1989.

[GIAR 91]   Giarratano, J., *CLIPS User's Guide, Volume I, Rules*, NASA, Lyndon B. Johnson Space Center, Information Systems Directorate, Software Technology Branch, January 1991.

[GONZ 91]   Gonzalez, D., *Ada LRM, Programmer's Handbook and Language Reference Manual*, The Benjamin/Cummings Publishing Company Inc., 1991.

[HARR 78]   Harrison, M., *Introduction to Formal Language Theory*, Addison-Wesley Publishing Co., 1978.

[HOLM 91a]   Holmes, W., *A Structure for Maturing Intelligent Tutoring Systems Student Models*, Internal Memo AMSMI-RD-SS-92-07, U.S. Army Missile Command, Redstone Arsenal, AL, 1991.

[HOLM 91b]   Holmes, W. and Piper, *U.S. Army Materiel Command's Intelligent Tutoring System Technology Base Plan*, Interservice/Industry Training System Conference, Orlando, FL, 1991.

[HOPC 69]   Hopcroft, J. and Ullman, J., *Formal Languages and Their Relationship to Automata*, Addison-Wesley Publishing Co., 1969.

[HOPC 79]   Hopcroft, J. and Ullman, J., *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley Publishing Co., 1979.

[JONA 88]   Jonassen, D., *Instructional Designs for Microcomputer Courseware*, Lawrence Erlbaum Assoc. Publishers, 1988.

[JONS 85]   Johnson, W. and Soloway, E., "PROUST: Knowledge-Based Program Understanding", *IEEE Transactions on Software Engineering*, March 1985.

[JONS 90]   Johnson, W., "Understanding and Debugging Novice Programs", *Artificial Intelligence*, No. 42, 1990.

[KEAR 87]   Kearsley, G., *Artificial Intelligence and Instruction*, Addison-Wesley Publishing Co., 1987.

[KNIG 89]   Knight, K., "Unification: A Multidisciplinary Survey", *ACM Computing Surveys*, March 1989.

[LEE 89]   Lee, Y., *A Knowledge Based Approach to Program Debugging*, Technical Paper, NPS52-89-060, Naval Postgraduate School, Monterey, CA, 1989.

[LEWI 85]   Lewis, W. and Soloway, E., "PROUST", *Byte*, April 1985.

[LEWI 90]   Lewis, B., Laliberte, D., and the GNU Manual Group, *GNU Emacs Lisp Reference Manual*, Free Software Foundation, Cambridge, MA, 1990.

[LOEC 72]   Loeckx, J., "Computability and Decidability, An Introduction for Students of Computer Science", *Lecture Notes in Economics and Mathematical Systems*, No. 68, 1972.

[MACK 88]   Mackay, W., "Tutoring, Information Databases, and Interactive Design", in D. Jonassen (Ed.), *Instructional Designs for Microcomputer Courseware*, Lawrence Erlbaum Assoc. Publishers, 1988.

[MASS 90]   Masson, V. and Quiniou, R., "Application of Artificial Intelligence to Aphasia Treatment", *Proceedings of the Third International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, Volume II*, Charleston, SC, 1990.

[MILL 88]   Miller, J., "The Role of Human-Computer Interaction in Intelligent Tutoring Systems", in M. Polson and J. Richardson (Ed.), *Foundations of Intelligent Tutoring Systems*, Lawrence Erlbaum Assoc. Publishers, 1988.

[MURR 85]   Murray, W., "Heuristic and Formal Methods in Automatic Program Debugging", *Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Volume 1*, Los Angeles, CA, 1985.

[MURR 86]   Murray, W. *Automatic Program Debugging for Intelligent Tutoring Systems*, Doctoral Dissertation, The University of Texas at Austin, Austin, TX, 1986.

[NASA 91a] NASA, *CLIPS Reference Manual, Volume I, Basic Programming Guide*, Software Technology Branch, Lyndon B. Johnson Space Center, Houston, TX, 1991.

[NASA 91b] NASA, *CLIPS Reference Manual, Volume II, Advanced Programming Guide*, Software Technology Branch, Lyndon B. Johnson Space Center, Houston, TX, 1991.

[NASA 91c] NASA, *CLIPS Reference Manual, Volume III, Utilities and Interfaces Guide*, Software Technology Branch, Lyndon B. Johnson Space Center, Houston, TX, 1991.

[PARK 87] Park, O., Perez, R., and Seidel, R., "Intelligent CAI: Old Wine in New Bottle, or a New Vintage?" in G. Kearsley (Ed.), *Artificial Intelligence and Instruction: Applications and Methods*, Addison-Wesley Publishing Co., 1987.

[POLS 88] Polson, M. and Richardson, J., *Foundations of Intelligent Tutoring Systems*, Lawrence Erlbaum Assoc. Publishers, 1988.

[PSOT 88] Psotka, J., Massey, L., and Mutter, S., *Intelligent Tutoring Systems, Lessons Learned*, Lawrence Erlbaum Assoc. Publishers, 1988.

[SCAN 88] Scandura, J. and Scandura, A., "A Structured Approach to Intelligent Tutoring", in D. Jonassen (Ed.), *Instructional Designs for Microcomputer Courseware*, Lawrence Erlbaum Assoc. Publishers, 1988.

[SEVI 87] Seviora, R., "Knowledge-Based Program Debugging Systems", *IEEE Software*, May 1987.

[SHAP 83] Shapiro, E., *Algorithmic Program Debugging*. MIT Press, 1983.

[SHRO 88] Shrode, H., "Exploring Artificial Intelligence", Morgan Kaufman Publishers, Inc., 1988.

[SIMM 88] Simmons, R., "A Theory of Debugging Plans and Interpretations", *Proceedings of the Seventh National Conference on Artificial Intelligence, Volume 1*, Saint Paul, MN, 1988.

[SKAN 88] Skansholm, J., *Ada from the Beginning*, Addison-Wesley Publishing Co., 1988.

[SLEE 82] Sleeman, D. and Brown, J., *Intelligent Tutoring Systems*, Academic Press, 1982.

[VANL 88] VanLehn, K., "Student Modeling", in M. Polson and J. Richardson (Ed.), *Foundations of Intelligent Tutoring Systems*, Lawrence Erlbaum Assoc. Publishers, 1988.

[VANL 90] van Leeuwen, J., *Handbook of Theoretical and Computer Science, Volume B, Informal Models and Semantics*, The MIT Press/Elsevier, 1990.

[VERD 90]   Verdix Corporation, *VADS, Verdix Ada Development System, Version 6.0-Sun4 OS*, Verdix Corporation, 1990.

[WAGE 88]   Wager, W. and Gagne, R., "Designing Computer-Aided Instruction", in D. Jonassen (Ed.), *Instructional Designs for Microcomputer Courseware*, Lawrence Erlbaum Assoc. Publishers, 1988.

[WENG 87]   Wenger, E., *Artificial Intelligence and Tutoring Systems*, Morgan Kaufman Publishers, Inc., 1987.

[WERT 87]   Wertz, H., *Automatic Correction and Improvement of Programs*, Ellis Horwood Limited, 1987.

[WOOL 88]   Woolf, B., "Intelligent Tutoring Systems: A Survey", in H. Shrode (Ed.), *Exploring Artificial Intelligence*, Morgan Kaufman Publishers, Inc., 1988.

[WOOL 91]   Woolf, B., "AI in Education", in S. Shapiro (Ed.), *Encyclopedia of Artificial Intelligence, Second Edition*, John Wiley & Sons Inc., 1992.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center   2
    Cameron Station
    Alexanderia, VA 22304-6145

2. Dudley Knox Library   2
    Code 52
    Naval Postgraduate School
    Monterey, CA 93943-5002

3. Chairman   1
    Computer Science Department
    Naval Postgraduate School
    Monterey, CA 93943-5000

4. Dr. Yuh-jeng Lee   6
    Code CS/Le
    Assistant Professor, Computer Science Department
    Naval Postgraduate School
    Monterey, CA 93943-5000

5. Dr. Man-Tak Shing   1
    Code CS/Sh
    Associate Professor, Computer Science Department
    Naval Postgraduate School
    Monterey, CA 93943-5000

6. Dr. Sehung Kwak   1
    Code CS/Kw
    Adjunct Professor, Computer Science Department
    Naval Postgraduate School
    Monterey, CA 93943-5000

7. HQDA, OCSA, AI Center   1
    ATTN: CSDS-AI-COL Teter
    The Pentagon, Room 1D659
    Washington, DC 20310-6900

8. CPT William C. Hoppe   1
    Software Development Center-Washington
    ATTN: ASQB-IWI (Stop H-23)
    Ft. Belvoir, VA 22060-5456

GAYLORD S