ITSY: an automated programming adviser

Thesis

ITSY
AN AUTOMATED PROGRAMMING ADVISER

John Domingue

Thesis submitted in partial fulfillment of requirements for Ph.D in Psychology, April 1987.

Human Cognition Research Laboratory
The Open University
Milton Keynes MK7 6AA
U.K.

Date of Submission: April 1987
Date of Award: 19th June 1987

ABSTRACT

This thesis presents an automated programming adviser. This system (called ITSY) tutors students in Lisp. This is from the viewpoint of automated program debugging of novice programs. Work within HCRL [Eisenstadt et al, Hasemer, Lewis] has shown that novice programming students can benefit from relatively small changes to the environment and from help via (intelligent) debugging tools. This thesis investigates the use of these debugging techniques in tutoring. The debugging techniques described here rely totally on detecting patterns in the student's code which represent erroneous concepts the student may have.

The thesis is divided into three parts. Each part describes a separate area of investigation.

The first part provides a detailed description of the types of errors that professional programmers make when using a 'traditional' (i.e. glass teletype) Lisp environment.

In the second part the concept of a programming cliche has been inverted and used as a basis for a system designed to help overcome the difficulties described in the first part of the thesis. This approach can be used in the design of computing systems built to help novices in certain domains. The constraint on the domain is that students' answers are complex enough to contain patterns of errors (so one word answers would not suffice). This would include domains where students are learning procedural skills - such as arithmetic, algebra or mechanics.

The third part describes a study involving professional programmers using the system.

CONTENTS

I BACKGROUND

III EVALUATION OF ITSY

PREFACE

This thesis is divided into three parts. Part I both introduces and provides a background for the project. Part II describes the implementation of ITSY in detail. Part III contains an evaluation of ITSY as well as concluding remarks.

ACKNOWLEDGEMENTS

I'd like to thank the following:

First and foremost Marc Eisenstadt. Few students can have a supervisor able to inspire confidence and motivate as he can.

Tony Hasemer for his support and encouragement.

Rick Evertsz for interrupting his own PhD work to help port and maintain his object-oriented package. This saved a significant amount of time in the construction of the system.

My parents deserve thanks for providing the perfect environment while drafts of this thesis were prepared.

My subjects for giving up a considerable amount of their spare time. Special thanks go to Anne, Cathy, Claire, Louise, Steve and Simon. I'd like to thank Claire for getting some subjects when I was in desperate need.

This research is supported by a Science and Engineering Research Council CASE award in collaboration with International Computers Ltd and has been carried out in the Discipline of Psychology.

PART I

1. INTRODUCTION

1.1 Motivation

Today a growing number of commercial companies are becoming interested in both the development and use of Artificial Intelligence (AI) systems. In the Human Cognition Research Laboratory there are collaborative projects with International Computers Ltd, British Telecom, Expert Systems International, Sperry and British Petroleum. Because of its built-in facilities for symbol-processing Lisp [McCarthy, Abrahams, Edwards, Hart & Levin, 1962] is one of the most widely used languages of AI. An understanding of Lisp is vital not only to build (and in some cases to use) AI software, but to understand a substantial amount of the available literature. Because of this there are an increasing number of conventional programmers who want to become competent Lisp programmers. Unfortunately, because AI is still a young science, there are relatively few expert Lisp programmers and therefore few Lisp tutors. This thesis describes a system, called ITSY, to aid conventional computer programmers to learn Lisp. It is hoped that ITSY will help fill the gap.

1.2 Influences

This thesis draws from work from two areas of Artificial Intelligence - Intelligent Tutoring and Automated Program Understanding/Debugging. The overall structure of the system and some of its components are based on existing Intelligent Tutoring Systems. One of the components common to all Intelligent Tutoring systems is the *domain expert*. In ITSY the domain expert is a Lisp debugger for novice Lisp programmers.

The debugger is based on work carried out on the Programmer's Apprentice project at MIT [Waters , 1982]. The aim of this project was to build a knowledge based editor. The programmer would be able to converse with the editor in terms of programming concepts instead of text strings. In order to do this the Programmer's Apprentice would have a library of common programming *cliches*. A cliche is a standard form of code. These cliches represented a common knowledge base amongst computer programmers. The

Programmer's Apprentice would be able to construct programs using these cliches and analyse raw code into these cliches.

1.3 Approach

The first step in building ITSY was to study novice Lisp programmers. About 130 hours of data were collected from professional programmers learning Lisp. During this pilot study we noticed two things:

a) the errors made by the subjects fell into relatively few categories,

b) a substantial amount (12%) of the errors were caused by the environment.

We decided that the best way to cure b) was to improve the environment. The fact that the errors fell into few categories led us to believe that novice Lisp programmers share a common knowledge base of misconceptions and these misconceptions manifest themselves as 'similar' segments of code. We call these 'similar' segments of code *error cliches.* As we said earlier a cliche is a standard form of code. By *error cliche* we mean a standard form of code which is incorrect. We decided that ITSY should trap these error cliches and explain the misunderstood concept to the student.

1.4 The System

ITSY has been designed to be used with a set text, "Lisp" [Winston & Horn, 1984]. Many of the 'traditional' (unintelligent) CAI packages produced in the seventies were based on the 'branching text' concept. That is pages of text with several branching points. It would be a duplication of effort to produce long pieces of text and numerous exercises when there are so many 'teach yourself Lisp' books available [Hasemer, 1984; Winston, 1984; Touretszky, 1984; Wilensky, 1984].

ITSY provides help in two different ways:

1) Tutorial advice when a student makes an error.

2) A 'friendly' environment - including coaching on available Lisp tools.

These different sources of help will be explained in chapter 2.

1.5 Target Users

ITSY is intended for use by professional programmers, that is programmers who are currently employed to program in a 'conventional' language and have had at least two years' experience doing so . This makes a difference to the questions we have to address. Students who were computer naive would have a different set of problems - one example is the problem of students imputing too much "intelligence" to the machine. The fact that the students will be professional programmers may cause problems as the students will carry over knowledge that is not applicable to a language such as Lisp. Most conventional programming languages do not have an *interpreter toplevel* present in most Lisp systems.

The fact that a debugger is used as the domain expert gives ITSY advantages over the frame-based CAI systems. As Johnson says [Johnson, 1985 p 10]:

> Frame-based CAI systems may work in some domains, but they are ill-suited for teaching programming ...

Incorrect answers from students provide an awkward problem for such systems - they may have remedial frames but if the student continues to make the error they have no further action that they can take. Mistakes are often made by novices learning to program in a new language - in one of the studies conducted, 35% of the of the lines typed in by subjects contained an error.

The main questions raised in this thesis are:

1. What misconceptions do professional programmers have when learning Lisp?

2. How can the misconceptions best be categorised into *error cliches*?

3. Will explaining these misconceptions when an error cliche is found help novice programmers?

In describing the debugging system SNIFFER Daniel Shapiro comments [1981 p. 7]:

> This approach defines an initial theory of bug recognition. It considers errors to be positive entities around which knowledge can be organised, as opposed to representing them as differences from an established norm.

As the approach used in SNIFFER defines a theory of bug recognition so error cliches are a theory of the buggy knowledge that novice programmers are assumed to have.

The approach used here can be extended to other domains where students make the same types of error. One such area is children learning about arithmetic. Burton and Brown [1978] studied 1300 school children attempting basic mathematical problems. They found that rather than not following the correct procedures perfectly, the children were following *incorrect procedures.* Each of these different incorrect procedures meant that the child had a particular misconception.

The basic method used, in this thesis, is to collect a library of error cliches from students learning in the domain. Then to determine what misconception a student must have in order to produce a particular error cliche. The next step is to build a system that is able to match error cliches against student input and then explain the misconception that the student must have.

1.6 Guide to the Reader

The next chapter gives an overview of ITSY. Chapter 3 contains a scenario using screen snapshots from the running system. Chapter 4 describes ITSY in the context of related work. The last chapter in this part describes a pilot study, which involved looking at professional programmers learning Lisp.

2. OVERVIEW

An overview of the structure of ITSY is given in figure 2-1. The arrows leaving the main box show how the student can interact with ITSY. When the student types input at the enhanced lisp environment the code analyser transforms the student's code into an internal form (a surface plan Waters [1978]). If the input does not cause an error the student model is updated. The student model is represented as a set of nodes. Updating the student model involves changing the state of some of the nodes. The student model is used to determine whether a particular mistake, made be the student, is due to a misconception that the student has or is trivial. If the input does cause an error the error cliche finder tries to match one of the error cliches in the error cliche library against part of the student's transformed code.

The remainder of figure 2.1 will be explained later in this chapter.

2.1 An Example

The following is a description of what happens 'behind the scenes' as a bug is detected and a tutorial is presented. Suppose that the student has typed in the following function definition, in an attempt to create a synonym for CAR:

```
(defun buggy-first (l)
  (car (l)))
```

Once the student has loaded the function BUGGY-FIRST and typed in the form:

```
(buggy-first '(a b c))
```

Lisp gives an error (the reason for the error is that the L in (CAR (L)) should not be surrounded by brackets). Instead of entering the debugger ITSY displays the Lisp error message and tries to spot the error.

The first part of spotting the error is transforming the students' code into an internal

form. The internal form used in ITSY is close to the *surface plans* used the Programmer's Apprentice [Waters, 1985]. The toplevel form is translated into the network shown in figure 2-2.

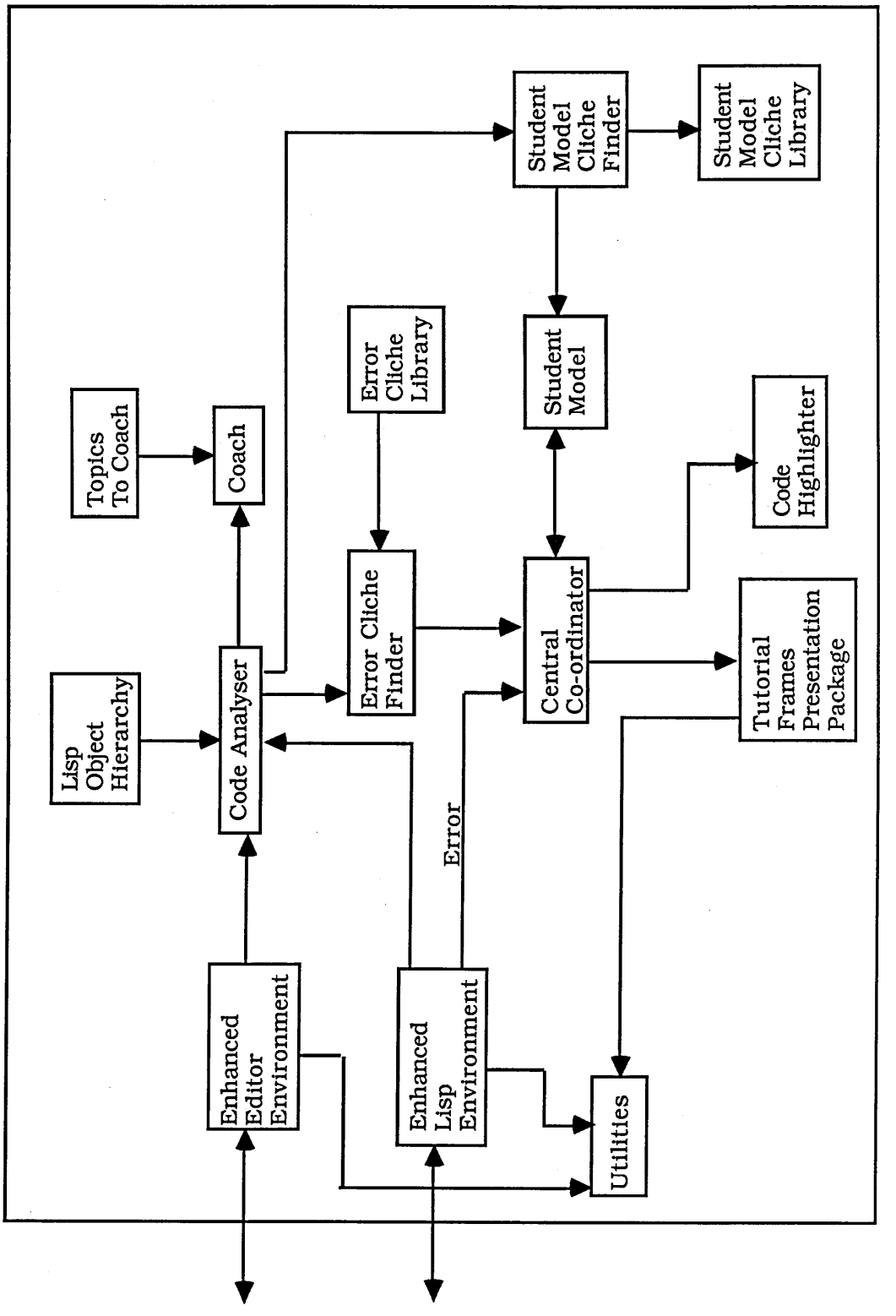


Figure 2-1 Overview of ITSY's Architecture

The following two figures show the internal representation used by ITSY. Each labelled box represents a segment of code (usually a function). The arrows show dataflow.

Figure 2-2

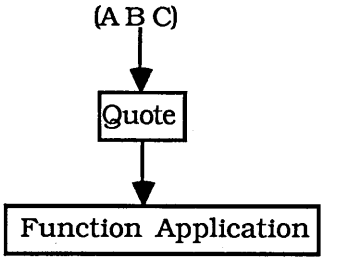
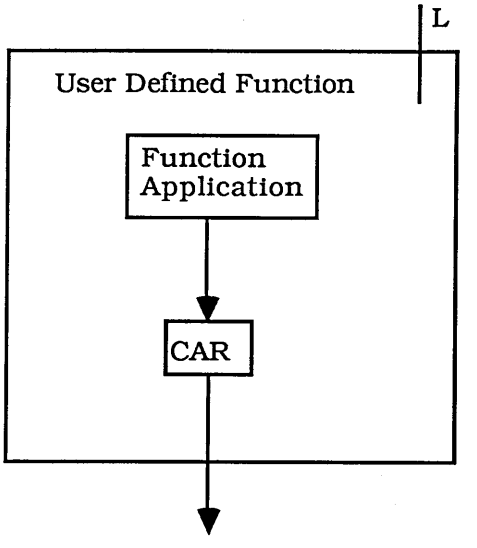


Figure 2-2 shows the internal representation of the form (BUGGY-FIRST '(A B C)). The box labelled *Function Application* represents the application of the function BUGGY-FIRST. The input to this function is the QUOTE function - represented by the box labelled *Quote*. The input to QUOTE is the list (A B C).

As ITSY analyses the toplevel form it also analyses any functions, defined by the student, that are called. The function buggy-first is analysed into the network:

Figure 2-3



The network shown in figure 2-3 represents the definition of the function BUGGY-FIRST. The box labelled *User Defined Function* represents the function BUGGY-FIRST. The input to the function is represented by the line labelled L. Notice this line does not lead to any of the internal boxes as the parameter L is not used within the function body. The function CAR is represented by the box labelled *CAR*. The input to this function is the box labelled *Function Application*. The *Function Application* box represents the function call L, in the student's code, which of course is undefined.

The error cliche matcher then tries to match one of the error cliches against this network (comprising of figures 2-2 and 2-3), starting from the toplevel form and working in the same way as the evaluator. As the error cliche matcher traverses the network each error cliche actively tries to match itself against a section of the network. The surface plan representation and the error cliches are implemented in an object oriented manner. The nodes in the network are implemented as objects, the labels on the boxes refer to the type or class of the object. The error cliches are implemented as messages.

The error cliche which is called *brackets around a variable* matches. This error cliche has the form:

Error Cliche Name:	Brackets Around a Variable
Surface Code Segment:	Function Application
Criteria:	Function is undefined The 'name' of the function is the same as one of the input ports of the function definition the segment occurs in.
Other Checks:	The 'name' of the function is not a function.

Each error cliche has four parts. The first is the name of the error cliche. The second is the 'type' of object that the error cliche can match against. The third and fourth parts contain tests that the object must satisfy in order to match against the error cliche. The Criteria and Other Checks differ in that the former contain criteria that need to be

satisfied in order for the error cliche to match and the latter contain tests that prevent false alarms.

The error cliche shown above matches against the function application object in figure 2-3. This error cliche matches against the function application object as follows:

a) The type of the object (function application) is the same as the Surface Code Segment,

b) the two Criteria are true,

c) the Other Checks are true.

As indicated earlier each surface code segment (shown as a labelled box in figures 2-2 and 2-3) is represented as an object. Each object has various slots which can be filled in. The Criteria and Other Checks test the slots of the object. Three of the function application object's slots are used. One of these slots contains the name of the function being applied. The second slot contains the type of the function call, which can be one of normal, recursive and undefined. The third slot that is used contains a pointer to the surface plan representation of the function that the object appears in. The first Criteria checks that the type of function call slot has the value 'undefined'.

The second criteria checks that the 'first atom' in the name of the function slot is the same as one of the input ports of the function definition the function application appears in. The input ports of a function definition represent the possible ways that data can flow into a function. There are two possible ways this can happen, values can be passed via variables in the parameter list or via global variables referenced within the body of the function. The function definition shown in figure 2-3 has one input port L (represented by the line labelled *L*). Because the student may put more than one set of brackets around a variable the 'name' of a function may be a list. By 'first atom' we mean the first atom in a list, if the list is traversed in a depth first manner. For example, if instead of writing (CAR (L)) the student had written (CAR (((L)))) the name slot of the function in the function application object would have contained ((L)). The 'first atom' in the name slot would be L.

The Other Checks uses the name slot and checks that the 'first atom' is not the name of a

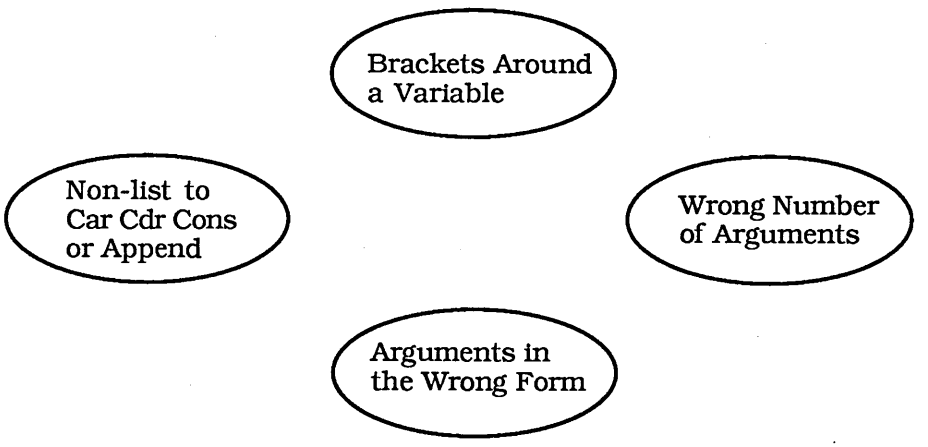
function. The use of the Other Checks can best be described by an example. In function definition below:

```
(defun buggy-fun (list)
  ((list list)))
```

The error cliche described above would match against the s-expression ((LIST LIST)) if the Other Check did not exist. This error is caught by the *extra bracket around a function call* error cliche.

Once the error has been found ITSY checks the student model. In this case the model indicates that the student needs to be tutored (the second scenario (3.2) shows the action ITSY takes when the student model indicates that the student does not need tutoring). The student model consists of an unconnected graph. Each node in the graph can have one of several states. Some of the nodes are shown below in figure 2-4.

Figure 2-4



The code highlighter receives information including the buggy piece of code and the function the buggy piece of code occurs in. The code highlighter brings the function to the top of the screen and highlights the buggy piece of code.

ITSY displays a set of message frames to explain the source of the error to the student. There is a set of message frames for each error cliche. Information returned by the error

cliche finder is used to fill in various slots in the messages. In the example the name of the function that the error occurred in (BUGGY-FIRST) and the name of the variable surrounded by brackets (L) are inserted into different slots in the messages.

The remaining sections of this chapter provide an overview of how ITSY works.

2.2 Spotting Errors

ITSY spots errors using a variant of the code analyser in the Programmer's Apprentice Waters [1978] (see section 4.2.1). A certain part of the analysis within the Programmer's Apprentice involves replacing segments of code by a smaller number of segments, or by a single segment. This involves matching segments of code against a pre-stored *cliche*. In an analogous fashion ITSY attempts to match segments of code against a pre-stored *error cliche*. This is carried out in two steps:

1. Transform the code into an internal form that abstracts out certain surface features such as data and control flow. The internal form used in ITSY is analogous to surface plans described in Waters [1978 p. 44] and consists of a network of objects (see figures 2-2 and 2-3).

2. Try and match each part of the network against an error cliche.

Notice that the matching process can occur at either a low or high level of code abstraction. In the Programmer's Apprentice code is transformed into surface plans and then by matching these against cliches, plan diagrams are produced. The plan diagrams can be matched against cliches to produce further (more abstract) plan diagrams. This process can occur as many times as necessary, each time a deeper understanding of the code is achieved. The same can occur with error cliches. When matching occurs more than once each new round of matching produces the source of deeper errors.

Error cliches are the central concept in ITSY. They represent *typical* student errors. A pilot study of novice Lisp programmers was carried out at the start of this project (see chapter 5). During this study we noticed that the errors made by novice Lisp programmers fell into relatively few categories. From this we concluded that Lisp

novices share a small set of misconceptions about the Lisp evaluator and that these misconceptions lead novices to writing the same *incorrect forms* which we call *error cliches.*

2.2.1 Transforming the Code

The code analyser (see figure 2-1) uses an object hierarchy. This contains the specific knowledge about Lisp. The hierarchy is described in detail in Chapter 8. The analyser uses the following algorithm:

Let O be the object to be analysed

If O is an atom then replace O with the surface plan representation of the value of the atom.

If O is a list then it is of the form:

```
(f arg1 ... argn)
```

One of the following actions is carried out:

1. If the function f is a special form[1] replace f with the surface plan representation of f and use the special form's analyser to analyse the arguments. ITSY contains a separate analyser for the special forms COND and DEFUN. This is needed because special forms do not evaluate their arguments in the same way as normal Lisp functions.

2. If the function f is a Lisp function then replace f with the surface plan representation of f and analyse f's arguments.

3. The function f is not a special form or a Lisp function, so it must be a user defined function. Analyse the arguments. If the function f has been analysed before then retrieve the surface plan representation. If the function has not been analysed then analyse and store the function definition. Create a surface plan representation of the

[1] A special form is a 'special' Lisp function, each special form has its own idiosyncratic syntax

function application.

2.2.2 The Matching Process

Once the relevant code has been converted into surface plan form the next step is to find an error cliche. A set of candidate error cliches are chosen. Which error cliches are included in the set depends on the type of error. ITSY then traverses the network of objects created by the code analyser attempting to match each of the chosen error cliches.

2.3 Presenting the Tutorial

A tutorial involves explaining a concept. The explanation involves four parts. Before the explanation is presented ITSY first checks that it has correctly diagnosed the cause of the bug. ITSY does this by asking a question. If the student does not understand the question s/he can ask for a rewording. These five parts are presented as frames, the top half containing a message, and the bottom a menu. The five parts are:

a) The *question* - used to check the diagnosis.

b) The *reworded question* - in case the student does not understand the question.

c) The *main explanation* - this is a expanded version of the error message using terms that a novice can understand.

d) A *deeper explanation* - this is an explanation of the error in terms of the evaluator. From the first study (chapter 5) it is clear that students do not understand how the Lisp evaluator works.

e) *Examples* - this is a set of concrete examples illustrating the misunderstood concept.

Each message contains slots which are filled with specific information about the error such as the function the error occurred in.

The student moves through the frames using a menu. Included as an extra item on this

menu is an option to look up the definition of any technical term used in the explanations.

2.4 The Student Model

The student model is used to determine whether or not a student requires a tutorial (this can be seen in the second scenario (3.2)). Instead of launching into the tutorial immediately ITSY just notifies the student that the tutorial is available on the Lisp menu.

The student model is closely linked to the error cliche library. The student model is represented as a graph. There is a node in the graph for every error cliche in the library. Each error cliche can be thought of as detecting a Lisp concept that a student may lack. Each node indicates whether or not the particular student lacks that particular concept.

The student model is updated every time a student types in a toplevel form. Updating the student model in ITSY is similar to finding an error cliche. The student model uses the surface plan representation of the student code (see fig 2-1). If the student's input to Lisp toplevel is correct ITSY traverses the surface plan representation of the code attempting to match the student model cliches against segments of the student's code. The student model cliches are derived from error cliches. A student model cliche will match against a correct segment of code where the student *could* have made an error but didn't. For example, the error cliche *wrong type argument* would match against the surface plan representation of the code segment:

```
(car 'a)
```

The corresponding student model cliche would match against the surface plan representation of the code segment:

```
(car '(a))
```

Every time a student model cliche matches against a segment of the student's code the corresponding node in the network is updated.

2.5 The Environment

When building any environment in which users are expected to learn, certain aspects of the whole environment need to be considered. The aim is to minimise the amount of knowledge that the students will need in order to use the system. The tools available to the students should be functionally, logically and syntactically simple (Du Boulay, O'Shea and Monk [1981]). There are two main ways of providing the tools: via menus or via user commands. Norman [1983] gives a table of trade-offs between these two systems. Menu-based systems are easier for novices, but are not as fast for the expert user. In ITSY, the tools are available on both pop-up menus and on single keys, the commands on the single keys acting as short cuts.

ITSY's environment can be seen in the scenario. The basic configuration has three panes. The top pane contains an editor, the middle a Lisp interpreter and the bottom status information. The environment was designed to prevent some of the errors that occurred in the pilot study (see Chapter 5). ITSY also provides limited coaching on the tools.

3. SCENARIO

In this chapter we describe five different scenarios. Each shows a different aspect of ITSY's behaviour. The scenarios are screen snapshots from a running implementation of ITSY. Each screen shown is divided into three main areas.

The top area consists of an editor window, this is used by students to write and edit Lisp functions. The editor window is divided into three sub-areas. The top part of the editor window contains the title. The middle part of the editor window contains the area where students can type their code. The bottom part of the editor window is used both to display information such as the file a buffer has been saved to, and for students to supply information such as the name of a file the student wishes to load into the editor.

The middle area is a Lisp toplevel window. Lisp forms can be typed into this window. The forms are evaluated and a result returned as soon as they are typed in. The bottom area is a status window. This give students information about what ITSY is currently doing.

Students can carry out various actions, such as saving and loading files, by using menus. A menu is brought up whenever a mouse button is pressed. There are two menus. Which menu is brought up depends on which window the mouse is in when a mouse button is pressed. If the mouse is within the Lisp toplevel window the *ITSY Lisp Menu* is brought up. This menu enable the student to carry out actions such as loading a file or obtaining documentation on a function. If the mouse is within the editor the *ITSY Editor Menu* is brought up. This menu enables the student to carry out the same actions the *ITSY Lisp Menu* as well as certain actions specific to the editor, such as saving a buffer.

When a student makes an error ITSY provides a tutorial via a set of frames. These frames have a message in the top half and a menu in the bottom half. The menu is used to either bring up more frames or leave the tutorial.

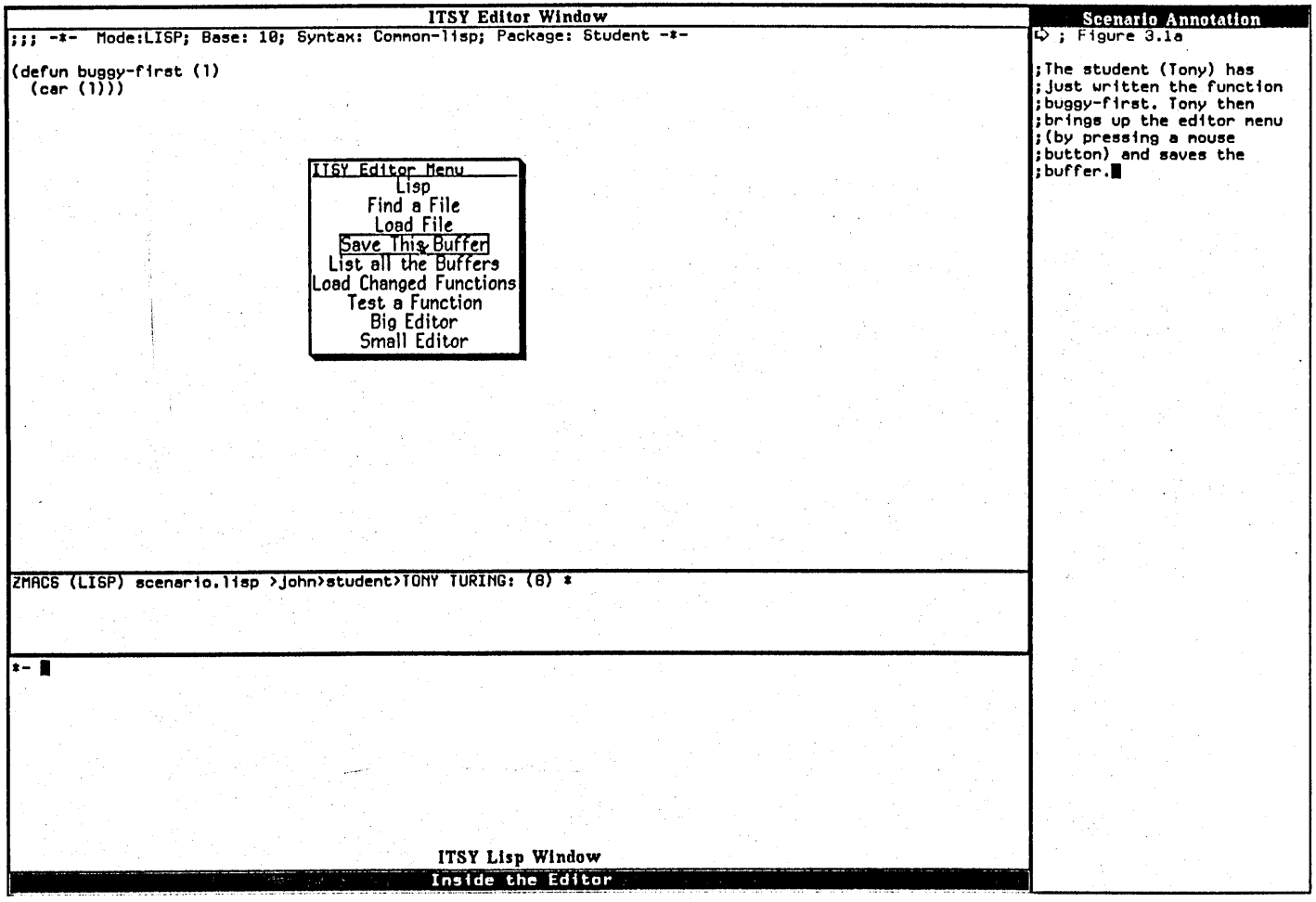
The scenario contains two extensions to the original design of ITSY which are not described until chapter 13. The first extension consists of an extra frame, the *fix* frame. The second extension is shown in the fifth scenario. This is the *test a function* tool.

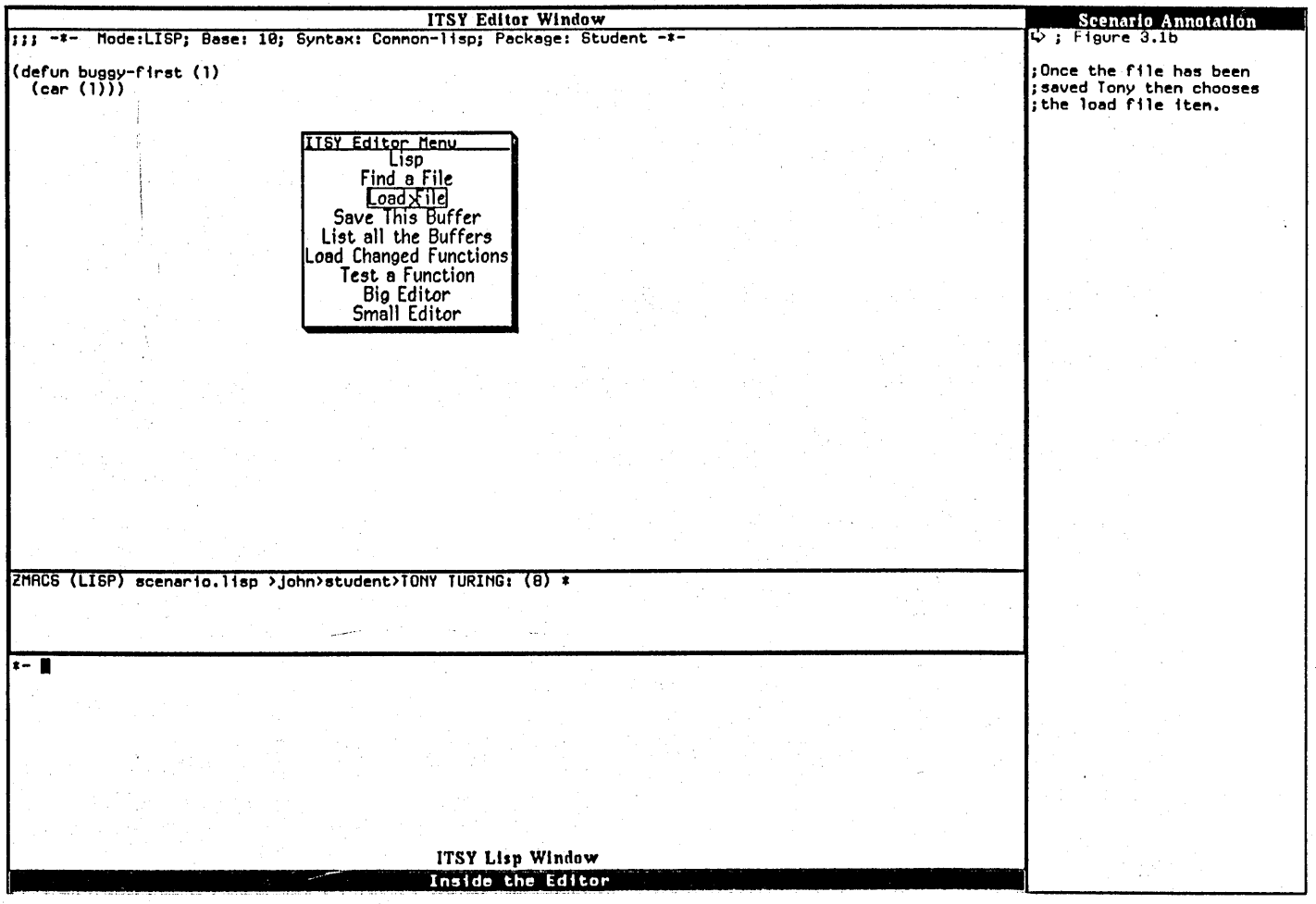
These two extensions were added as a result of a pilot evaluation of ITSY. They are described in chapter 13.

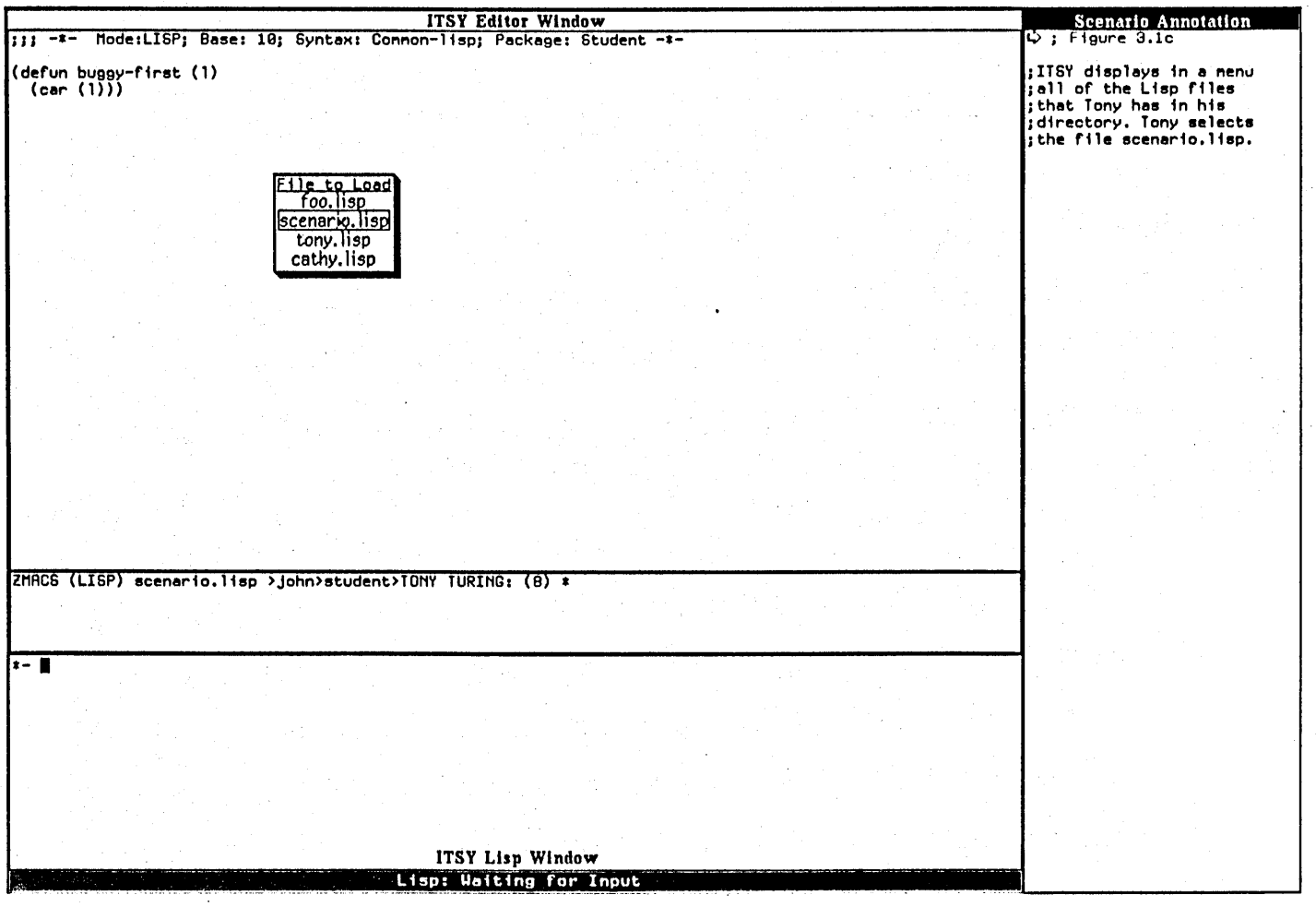
Each screen snapshot has been annotated using an extra window. This window appears in the right hand side of the snapshot and has the title *Scenario Annotation.* This window is **not** part of ITSY.

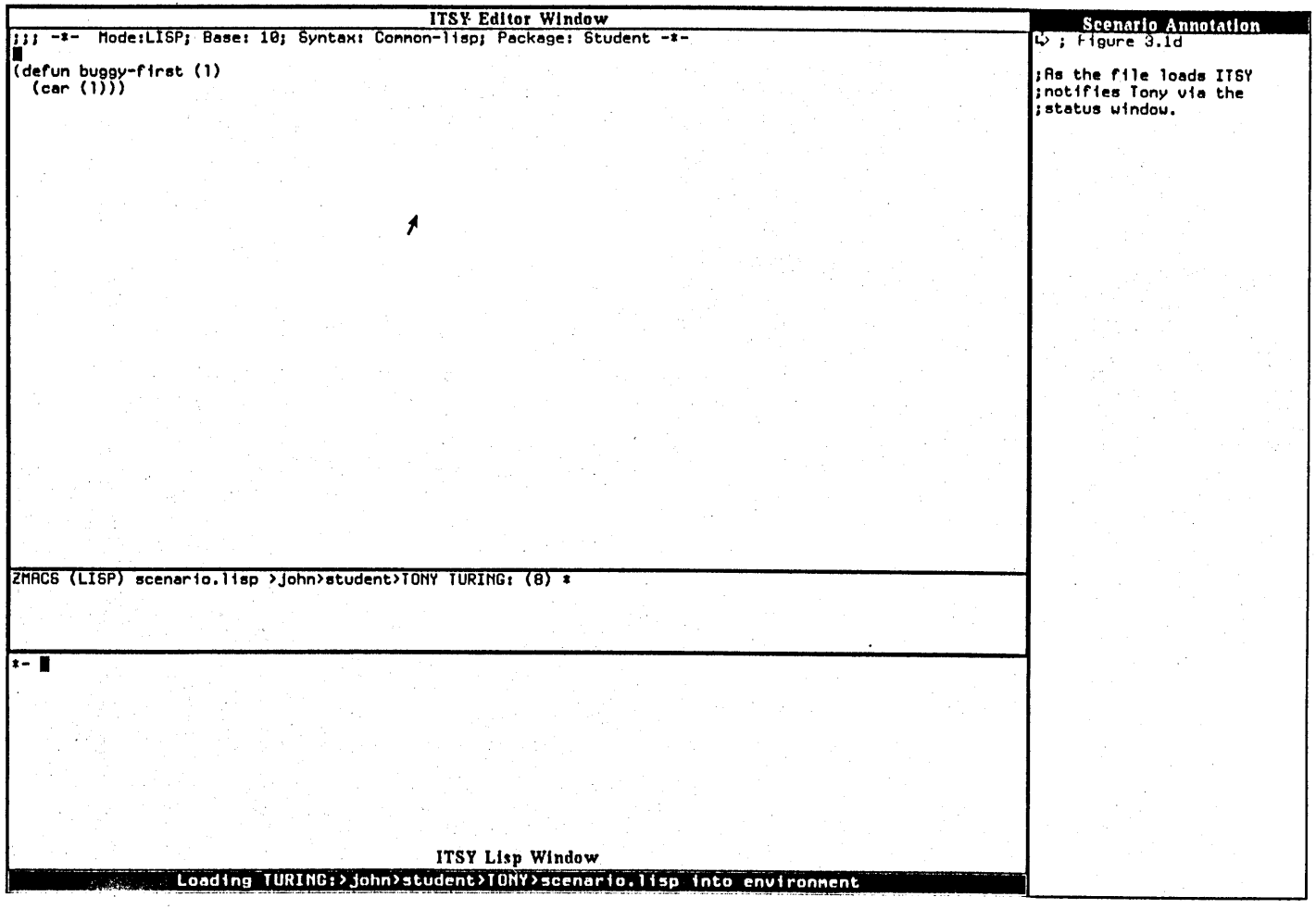
3.1 First Scenario

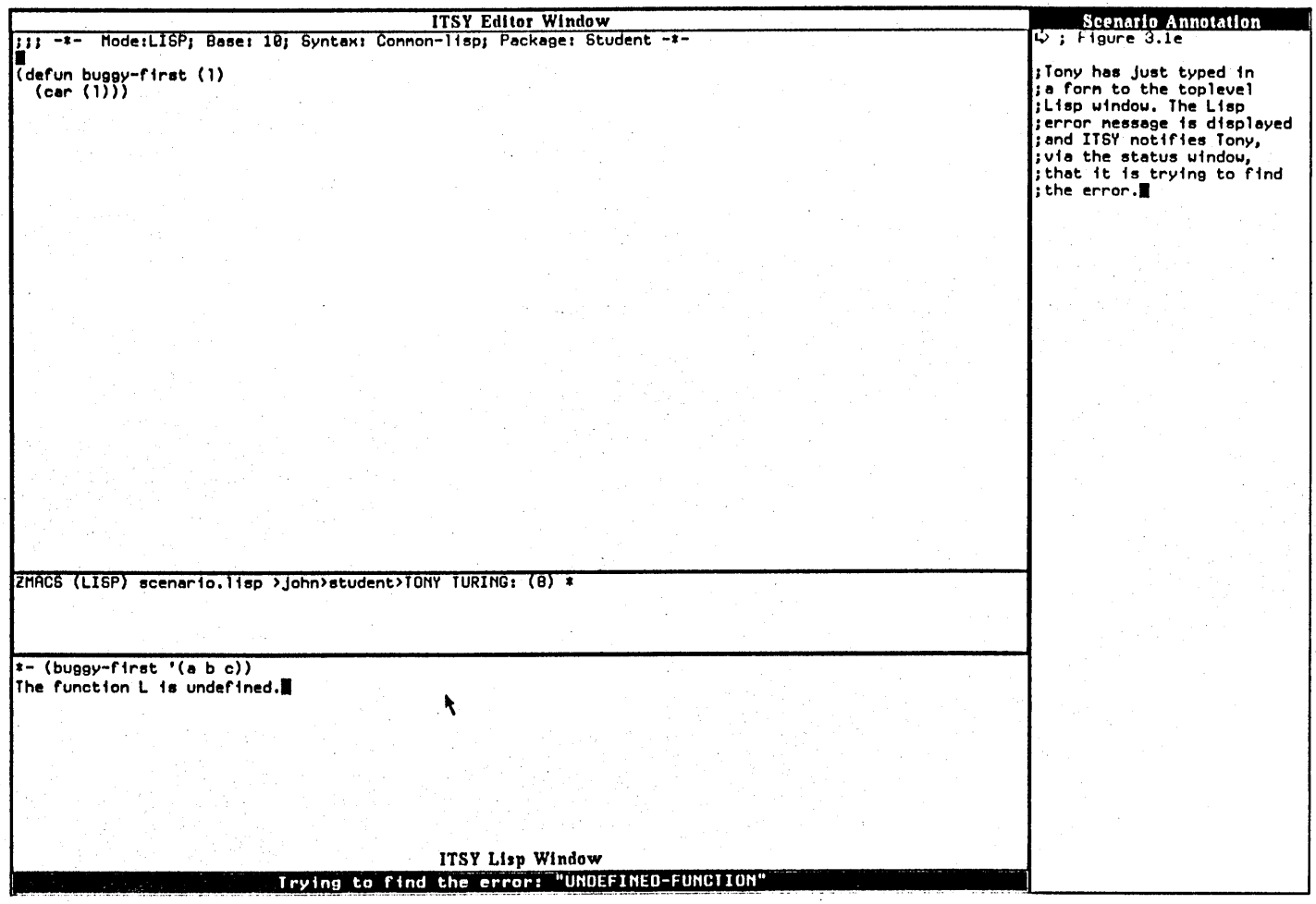
The first scenario shows the student defining a (buggy) function and ITSY's actions when the function is called.

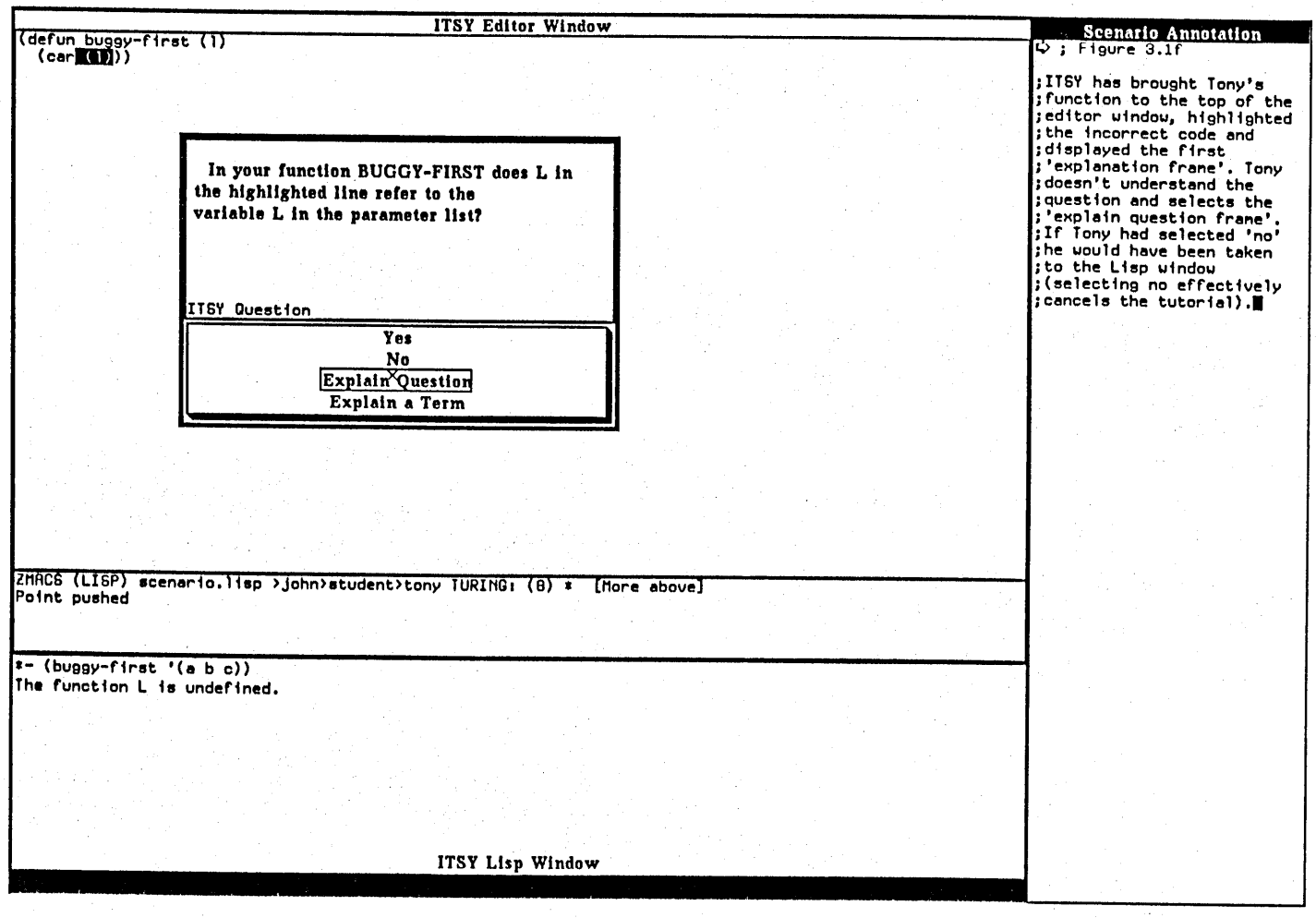

ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun buggy-first (1)
(car (1)))
ITSY Editor Menu
Lisp
Find a File
Load File
Save This Buffer
List all the Buffers
Load Changed Functions
Test a Function
Big Editor
Small Editor
ZMACS (LISP) scenario.lisp >John>student>TONY TURING: (8) *
*-
ITSY Lisp Window
Inside the Editor
Scenario Annotation
; Figure 3.1a
;The student (Tony) has
;just written the function
;buggy-first. Tony then
;brings up the editor menu
;(by pressing a mouse
;button) and saves the
;buffer.

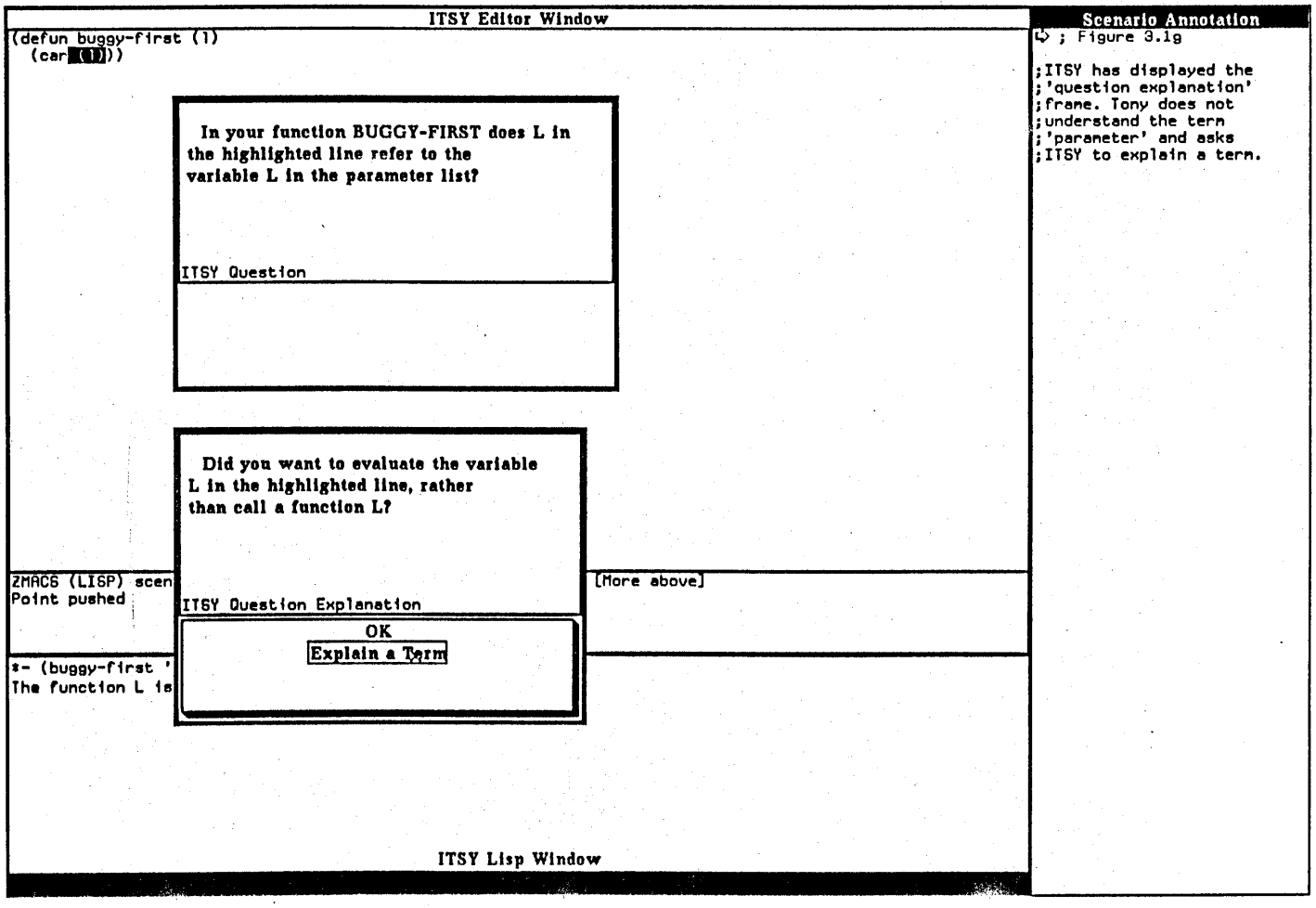

ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun buggy-first (l)
(car (l)))
ITSY Editor Menu
Lisp
Find a File
Load File
Save This Buffer
List all the Buffers
Load Changed Functions
Test a Function
Big Editor
Small Editor
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (8) *
*-
ITSY Lisp Window
Inside the Editor
Scenario Annotation
; Figure 3.1b
;Once the file has been
;saved Tony then chooses
;the load file item.

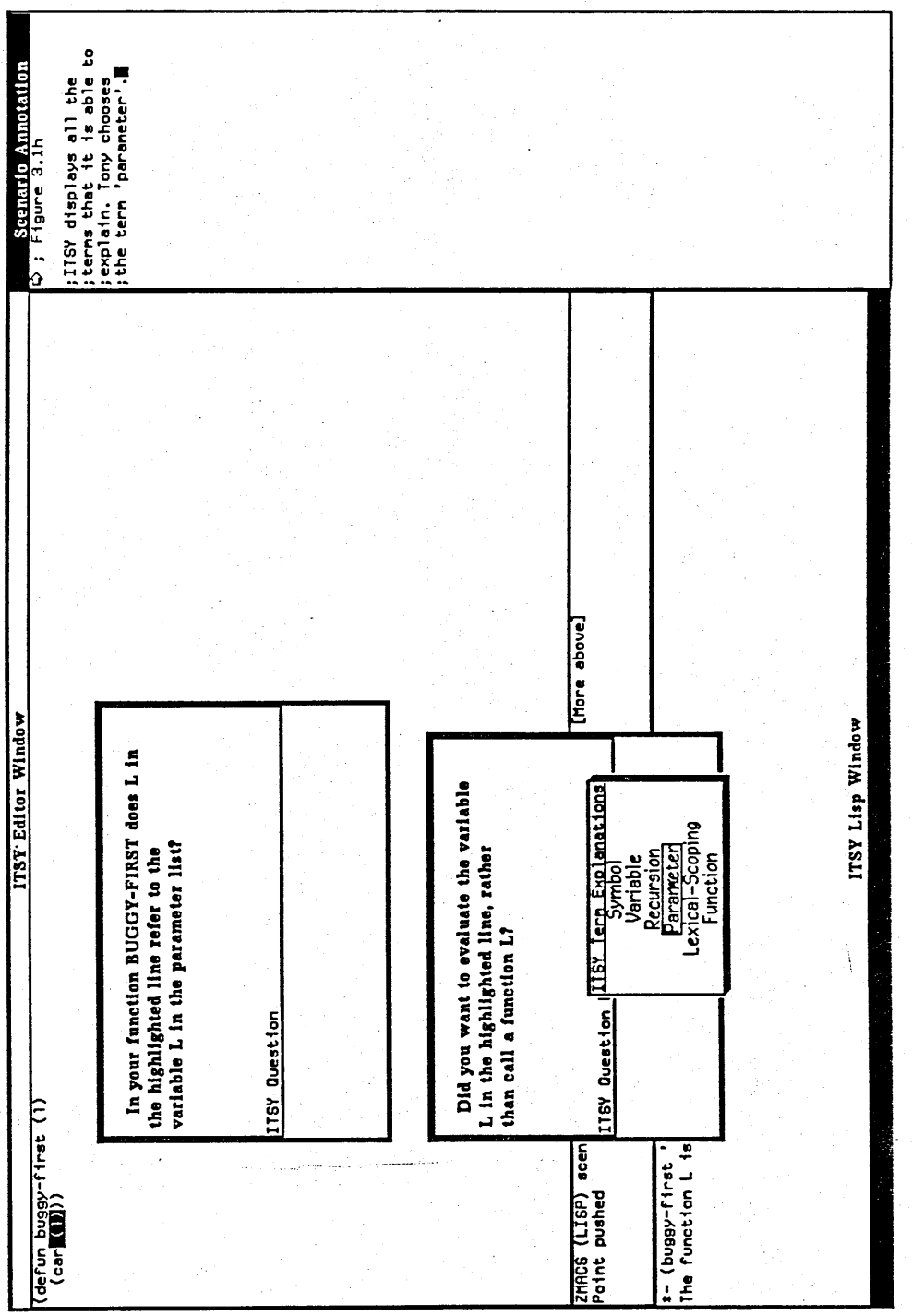

ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun buggy-first (l)
(car (l)))
File to Load
foo.lisp
scenario.lisp
tony.lisp
cathy.lisp
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (8) *
*-
ITSY Lisp Window
Lisp: Waiting for Input
Scenario Annotation
; Figure 3.1c
;ITSY displays in a menu
;all of the Lisp files
;that Tony has in his
;directory. Tony selects
;the file scenario.lisp.

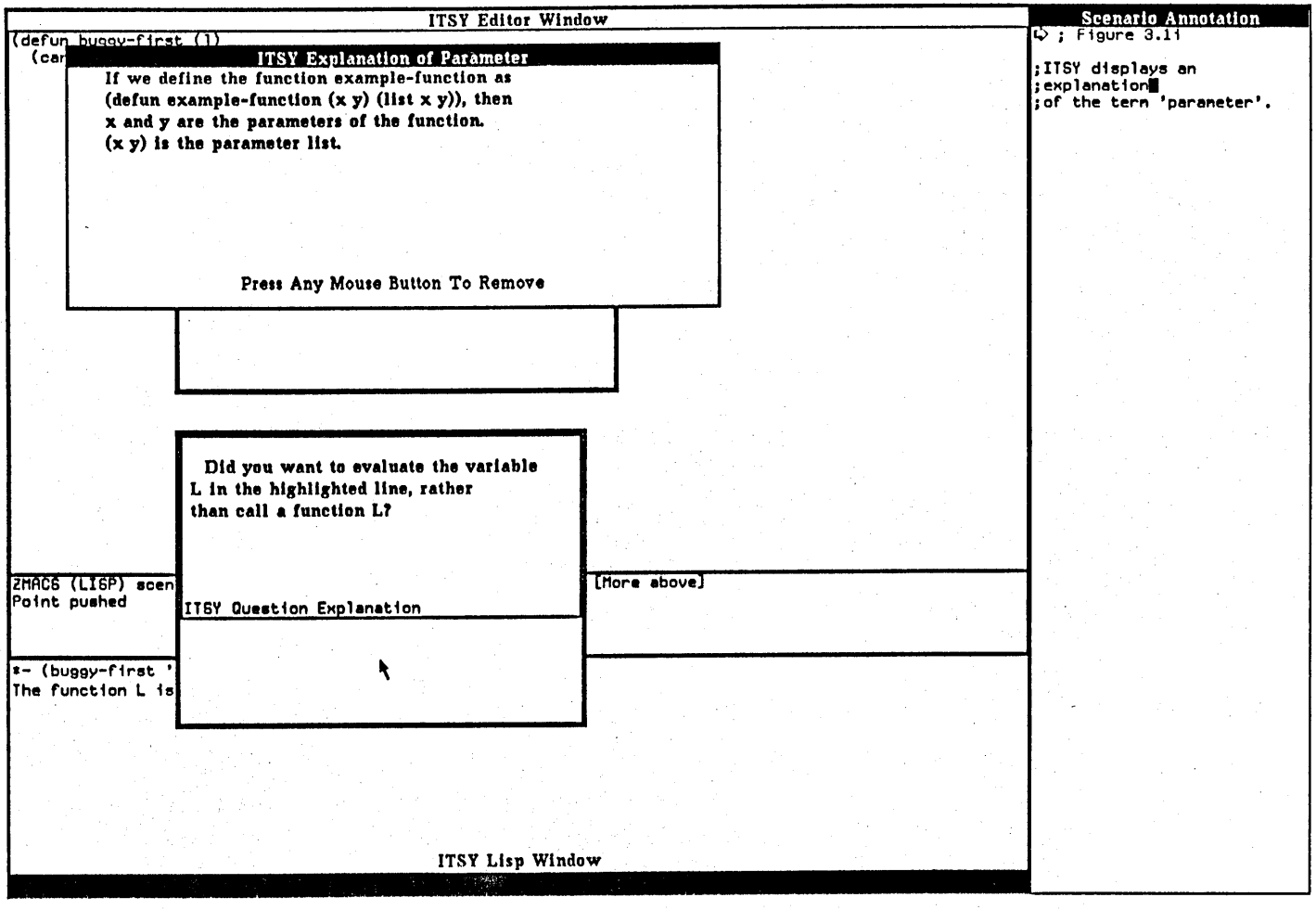

ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun buggy-first (l)
(car (l)))
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (8) *
*-
ITSY Lisp Window
Loading TURING:>john>student>TONY>scenario.lisp into environment
Scenario Annotation
; Figure 3.1d
;As the file loads ITSY
;notifies Tony via the
;status window.

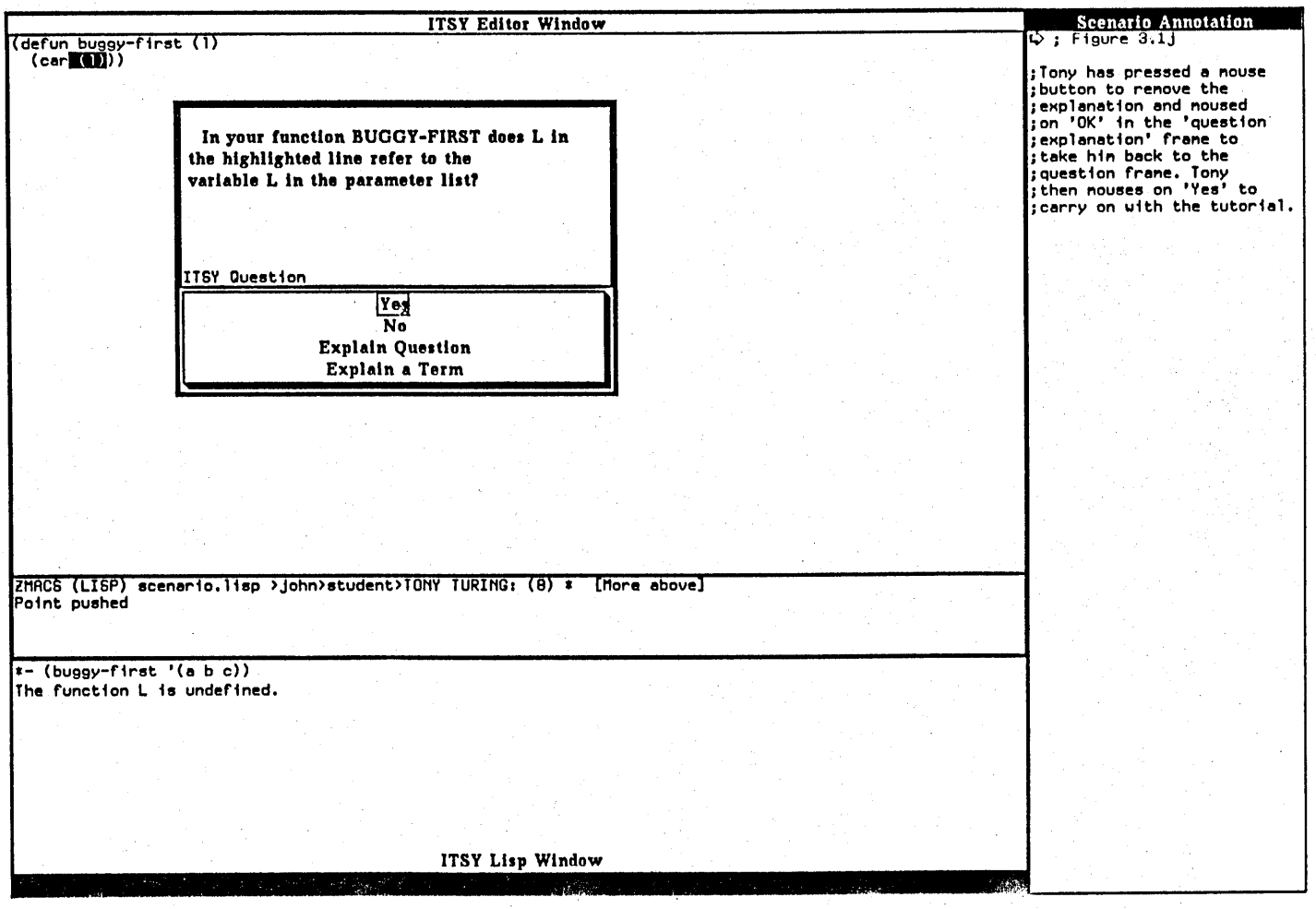

ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun buggy-first (l)
(car (l)))
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (8) *
*- (buggy-first '(a b c))
The function L is undefined.
ITSY Lisp Window
Trying to find the error: "UNDEFINED-FUNCTION"
Scenario Annotation
; Figure 3.1e
;Tony has just typed in
;a form to the toplevel
;Lisp window. The Lisp
;error message is displayed
;and ITSY notifies Tony,
;via the status window,
;that it is trying to find
;the error.

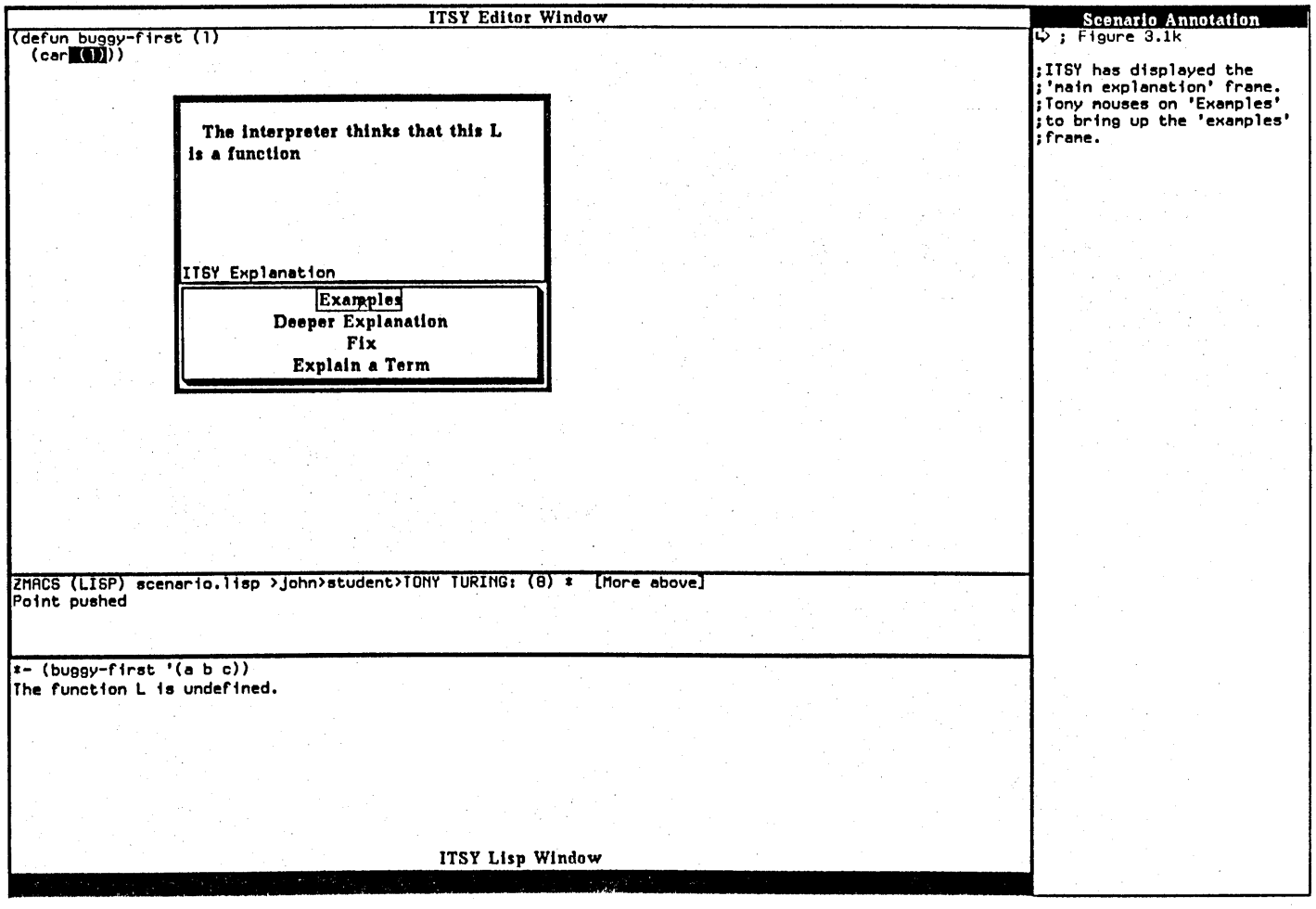

ITSY Editor Window
(defun buggy-first (l)
(car (l)))
In your function BUGGY-FIRST does L in the highlighted line refer to the variable L in the parameter list?
ITSY Question
Yes
No
Explain Question
Explain a Term
ZMACS (LISP) scenario.lisp >john>student>tony TURING: (8) * [More above]
Point pushed
*- (buggy-first '(a b c))
The function L is undefined.
ITSY Lisp Window
Scenario Annotation
; Figure 3.1f
;ITSY has brought Tony's
;function to the top of the
;editor window, highlighted
;the incorrect code and
;displayed the first
;'explanation frame'. Tony
;doesn't understand the
;question and selects the
;'explain question frame'.
;If Tony had selected 'no'
;he would have been taken
;to the Lisp window
;(selecting no effectively
;cancels the tutorial).

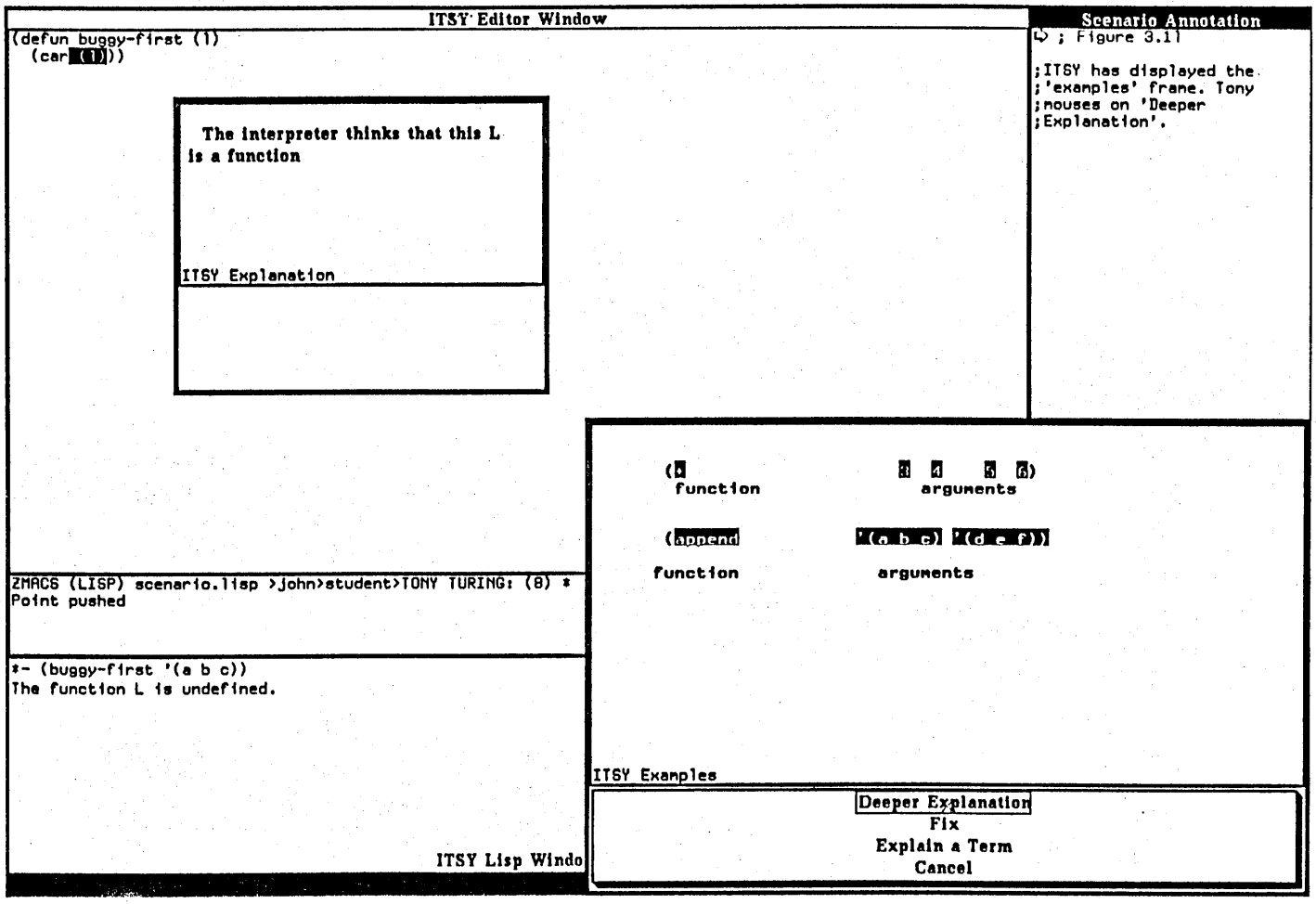

ITSY Editor Window
(defun buggy-first (l)
(car (l)))
In your function BUGGY-FIRST does L in the highlighted line refer to the variable L in the parameter list?
ITSY Question
Did you want to evaluate the variable L in the highlighted line, rather than call a function L?
ITSY Question Explanation
OK
Explain a Term
ZMACS (LISP) scen
Point pushed
[More above]
*- (buggy-first '
The function L is
ITSY Lisp Window
Scenario Annotation
; Figure 3.1g
;ITSY has displayed the
;'question explanation'
;frame. Tony does not
;understand the term
;'parameter' and asks
;ITSY to explain a term.

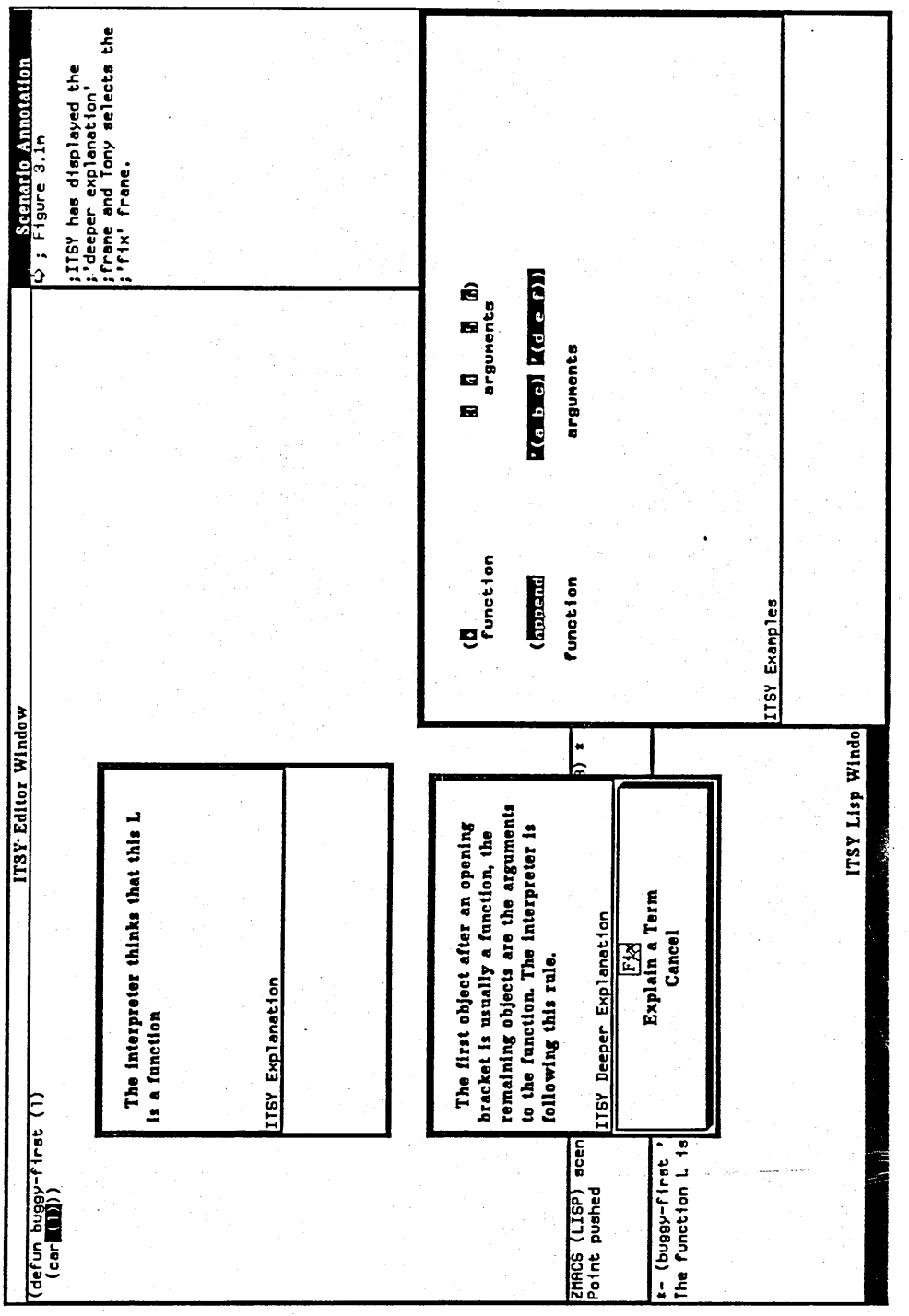

Scenario Annotation
; Figure 3.1h
;ITSY displays all the
;terms that it is able to
;explain. Tony chooses
;the term 'parameter'.
ITSY Editor Window
(defun buggy-first (l)
(car (l)))
In your function BUGGY-FIRST does L in
the highlighted line refer to the
variable L in the parameter list?
ITSY Question
Did you want to evaluate the variable
L in the highlighted line, rather
than call a function L?
ITSY Question
ITSY Term Explanations
Symbol
Variable
Recursion
Parameter
Lexical-Scoping
Function
(More above)
ITSY Lisp Window

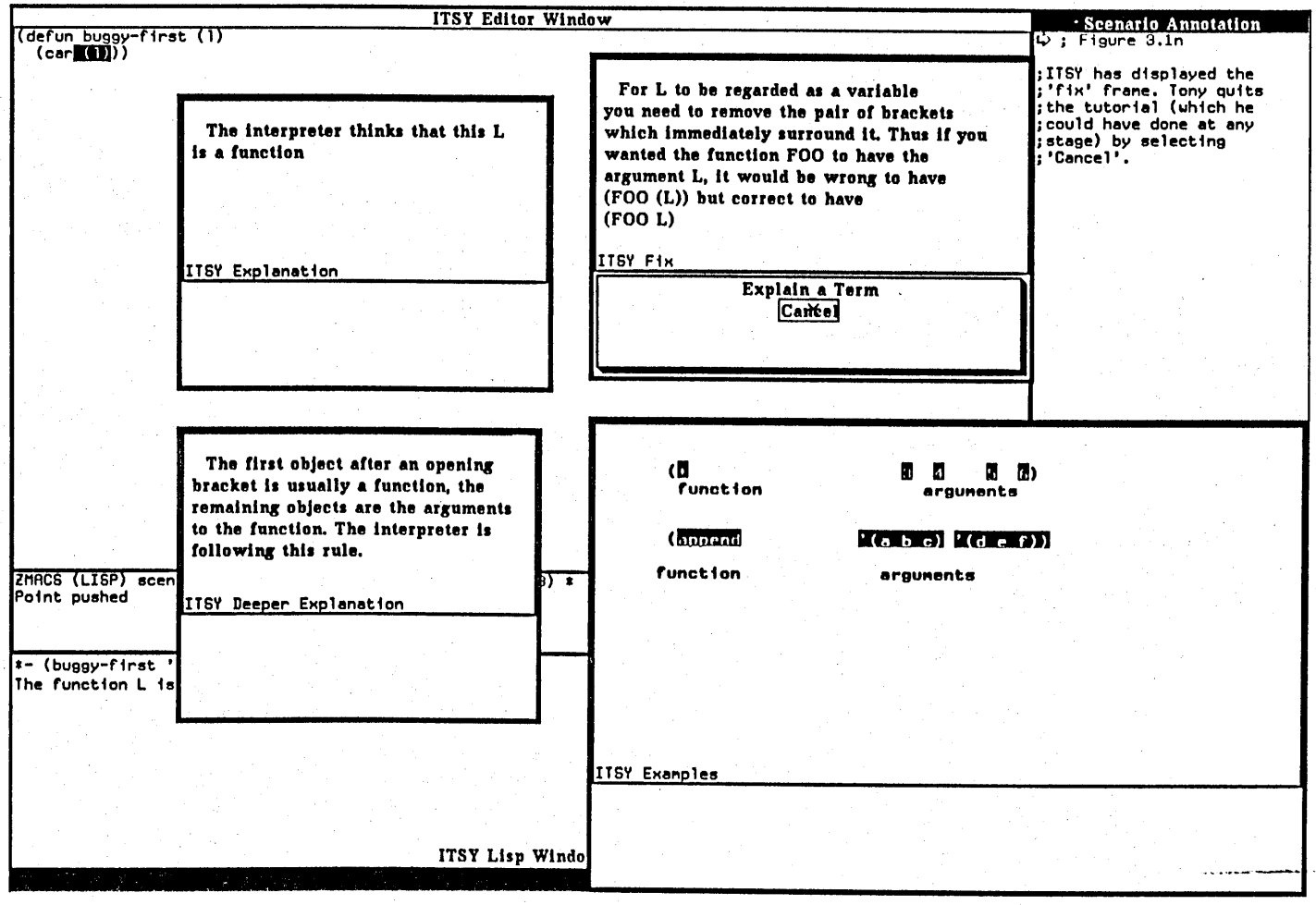

ITSY Editor Window
Scenario Annotation
; Figure 3.11
;ITSY displays an
;explanation
;of the term 'parameter'.
(defun buggy-first (l)
(car
ITSY Explanation of Parameter
If we define the function example-function as
(defun example-function (x y) (list x y)), then
x and y are the parameters of the function.
(x y) is the parameter list.
Press Any Mouse Button To Remove
Did you want to evaluate the variable
L in the highlighted line, rather
than call a function L?
ITSY Question Explanation
ZMACS (LISP) scen
Point pushed
[More above]
:- (buggy-first '
The function L is
ITSY Lisp Window


ITSY Editor Window
(defun buggy-first (l)
(car (l)))
In your function BUGGY-FIRST does L in the highlighted line refer to the variable L in the parameter list?
ITSY Question
Yes
No
Explain Question
Explain a Term
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (8) * [More above]
Point pushed
*- (buggy-first '(a b c))
The function L is undefined.
ITSY Lisp Window
Scenario Annotation
; Figure 3.1j
;Tony has pressed a mouse
;button to remove the
;explanation and moused
;on 'OK' in the 'question
;explanation' frame to
;take him back to the
;question frame. Tony
;then mouses on 'Yes' to
;carry on with the tutorial.


ITSY Editor Window
(defun buggy-first (l)
(car (l)))
The interpreter thinks that this L is a function
ITSY Explanation
Examples
Deeper Explanation
Fix
Explain a Term
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (8) * [More above]
Point pushed
*- (buggy-first '(a b c))
The function L is undefined.
ITSY Lisp Window
Scenario Annotation
; Figure 3.1k
;ITSY has displayed the
;'main explanation' frame.
;Tony mouses on 'Examples'
;to bring up the 'examples'
;frame.


ITSY Editor Window
(defun buggy-first (l)
(car (l)))
The interpreter thinks that this L is a function
ITSY Explanation
Scenario Annotation
; Figure 3.11
;ITSY has displayed the
;'examples' frame. Tony
;mouses on 'Deeper
;Explanation'.
(l
function
3 4 5 6)
arguments
(append
'(a b c) '(d e f))
function
arguments
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (8) *
Point pushed
*- (buggy-first '(a b c))
The function L is undefined.
ITSY Lisp Windo
ITSY Examples
Deeper Explanation
Fix
Explain a Term
Cancel

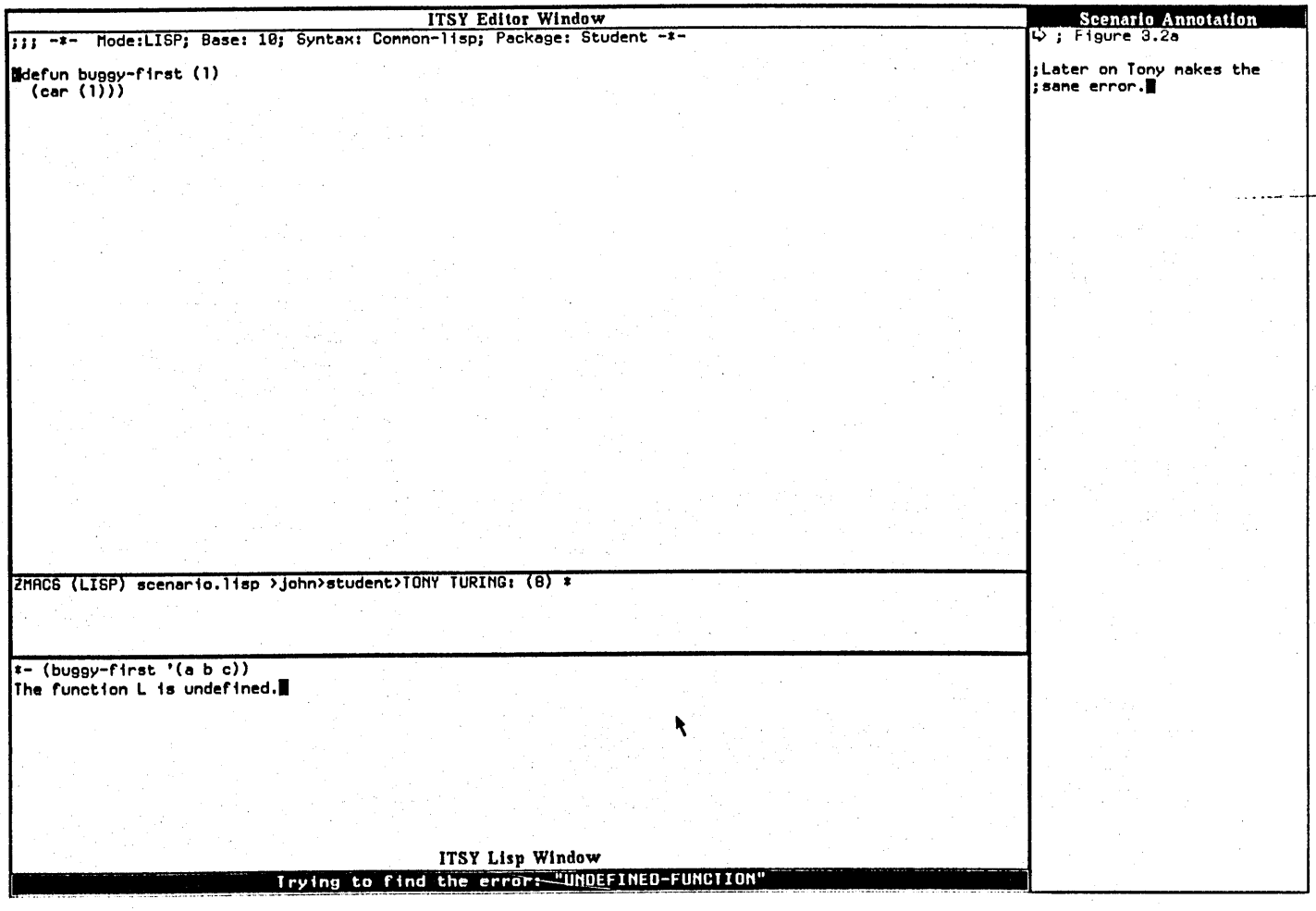

Scenario Annotation
; Figure 3.1n
;ITSY has displayed the
;'deeper explanation'
;frame and Tony selects the
;'fix' frame.
ITSY Editor Window
(defun buggy-first (l)
(car (l)))
The interpreter thinks that this L
is a function
ITSY Explanation
The first object after an opening
bracket is usually a function, the
remaining objects are the arguments
to the function. The interpreter is
following this rule.
ITSY Deeper Explanation
Fix
Explain a Term
Cancel
(l function arguments
(append (a b c) '(d e f))
function arguments
ITSY Examples
ITSY Lisp Windo

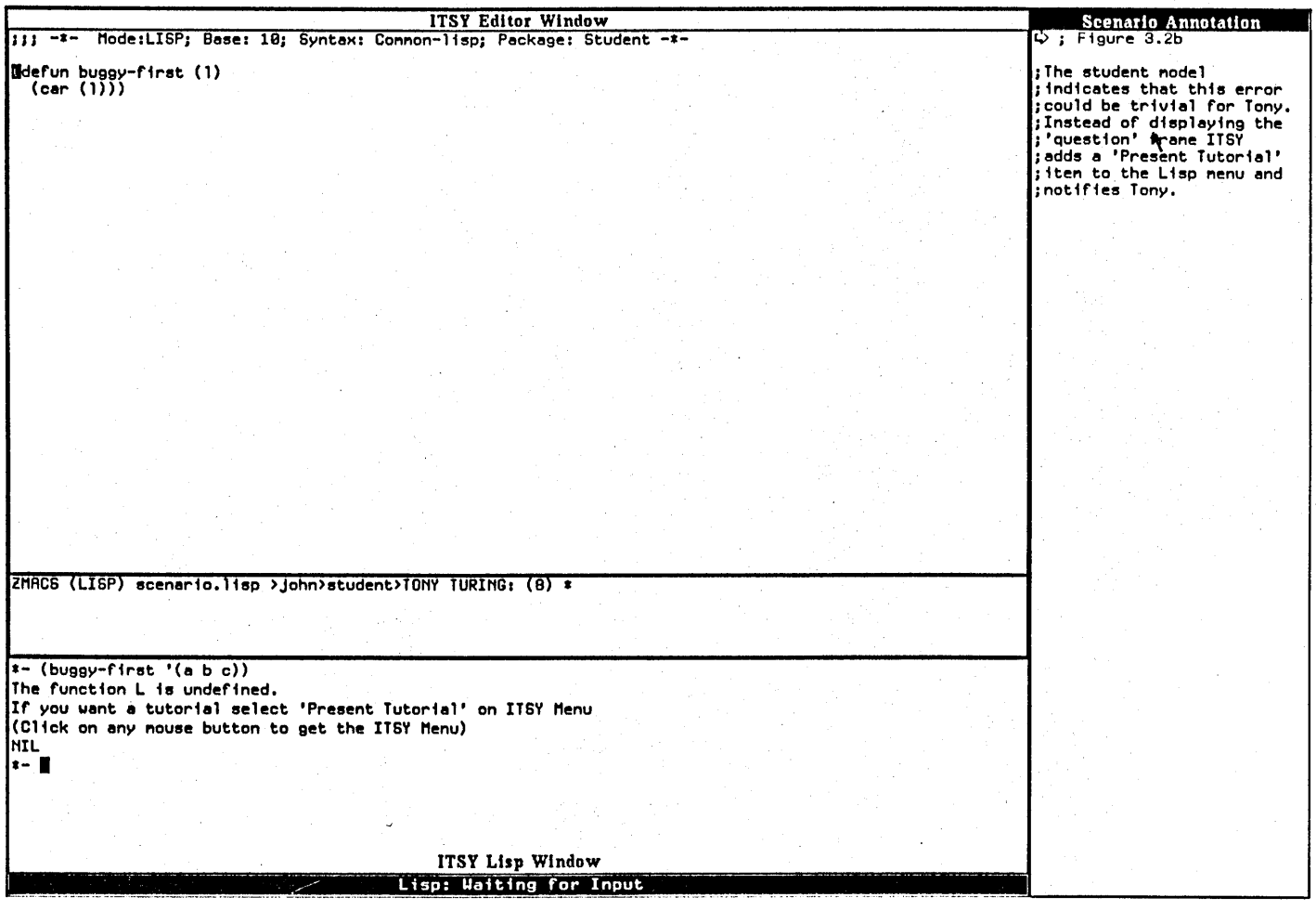

ITSY Editor Window
(defun buggy-first (l)
(car (l)))
The interpreter thinks that this L is a function
ITSY Explanation
For L to be regarded as a variable you need to remove the pair of brackets which immediately surround it. Thus if you wanted the function FOO to have the argument L, it would be wrong to have (FOO (L)) but correct to have (FOO L)
ITSY Fix
Explain a Term
Cancel
Scenario Annotation
; Figure 3.1n
;ITSY has displayed the
;'fix' frame. Tony quits
;the tutorial (which he
;could have done at any
;stage) by selecting
;'Cancel'.
The first object after an opening bracket is usually a function, the remaining objects are the arguments to the function. The interpreter is following this rule.
ITSY Deeper Explanation
ZMACS (LISP) scen
Point pushed
*- (buggy-first '
The function L is
function
arguments
(append
'(a b c)
'(d e f))
function
arguments
ITSY Examples
ITSY Lisp Windo

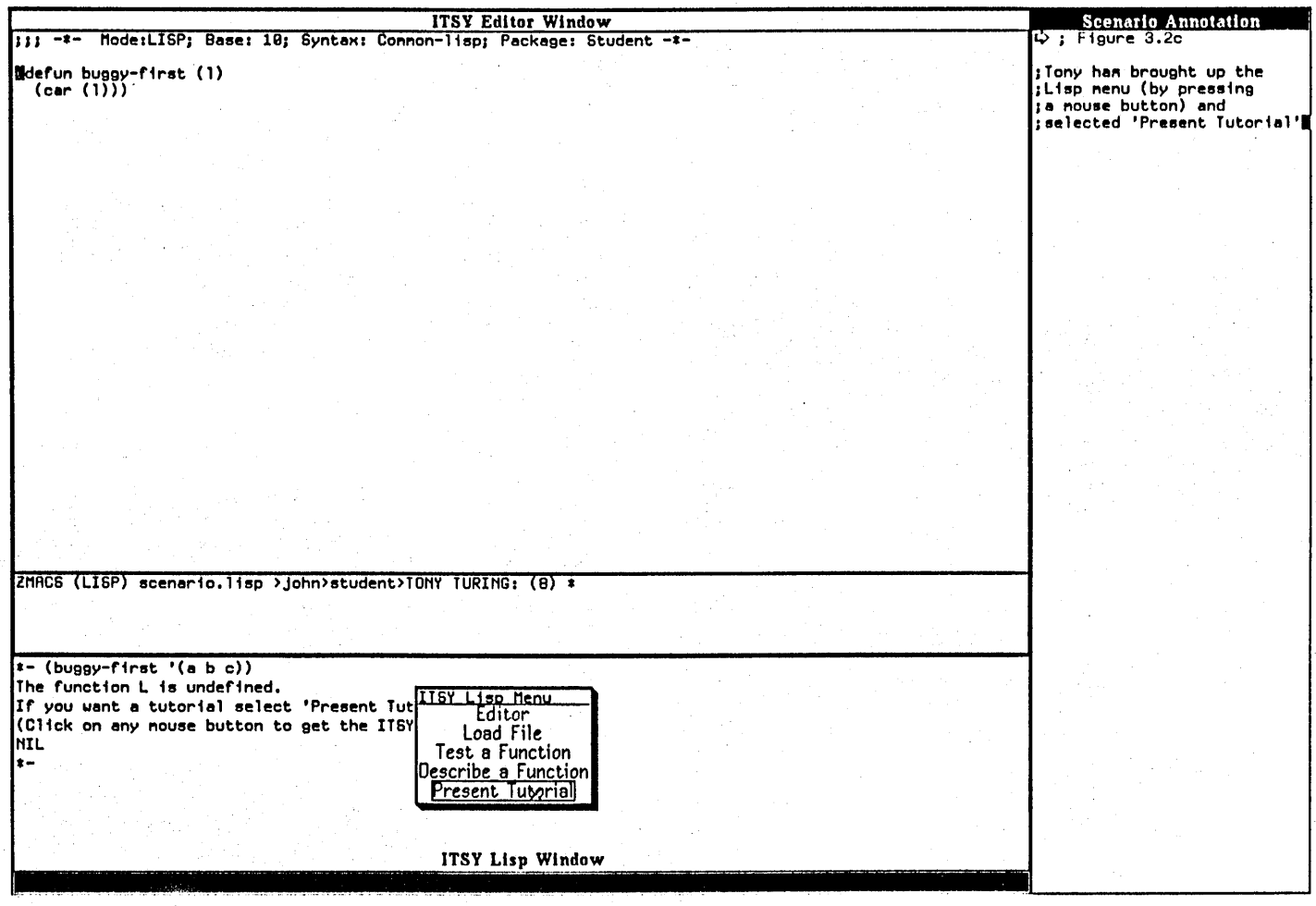
The error in the first scenario would have changed the *Bracket Around a Variable* node in the student model. This node would have moved from the *Concept has not yet been Encountered* state to the *Concept has been seen but not Learnt* state.

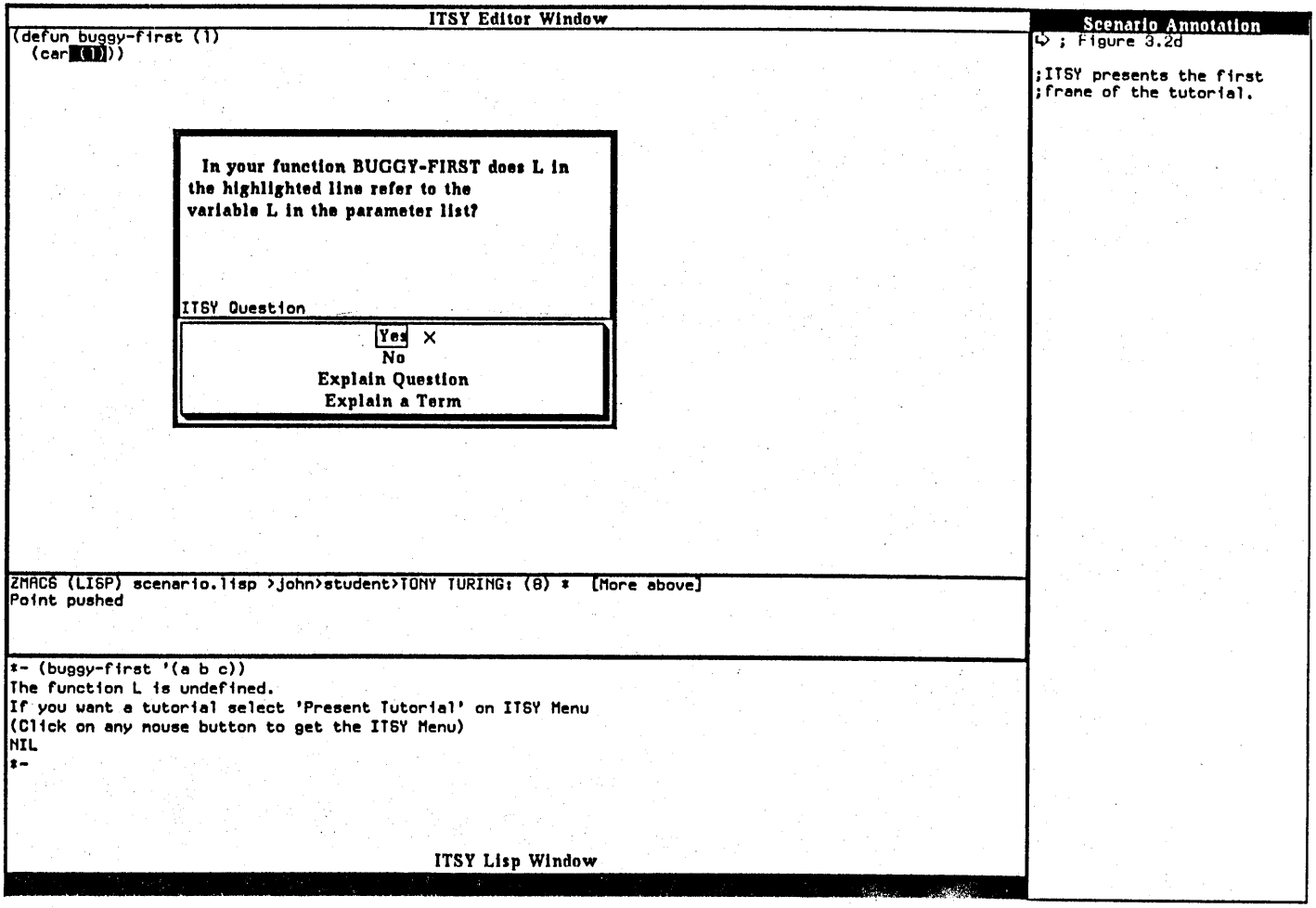
3.2 Second Scenario

The second scenario shows Tony making the same error as in the previous scenario some time later. The *Bracket Around a Variable* node in the student model is now in the *Concept has been fully Learnt* state and ITSY changes its action according to this.


ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun buggy-first (l)
(car (l)))
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (8) *
*- (buggy-first '(a b c))
The function L is undefined.
ITSY Lisp Window
Trying to find the error: "UNDEFINED-FUNCTION"
Scenario Annotation
; Figure 3.2a
;Later on Tony makes the
;same error.

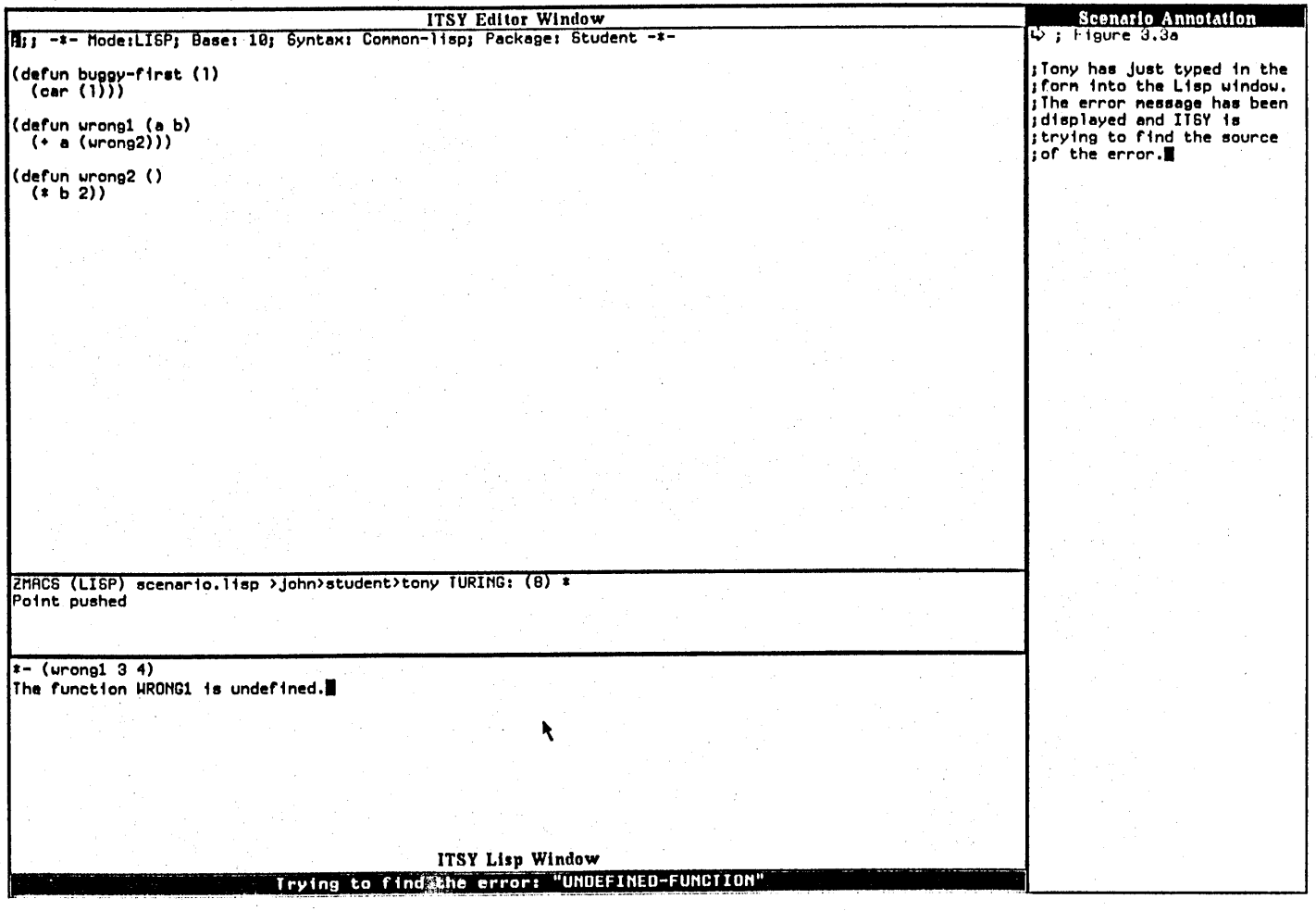

ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun buggy-first (l)
(car (l)))
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (8) *
*- (buggy-first '(a b c))
The function L is undefined.
If you want a tutorial select 'Present Tutorial' on ITSY Menu
(Click on any mouse button to get the ITSY Menu)
NIL
*-
ITSY Lisp Window
Lisp: Waiting for Input
Scenario Annotation
; Figure 3.2b
;The student model
;indicates that this error
;could be trivial for Tony.
;Instead of displaying the
;'question' frame ITSY
;adds a 'Present Tutorial'
;item to the Lisp menu and
;notifies Tony.

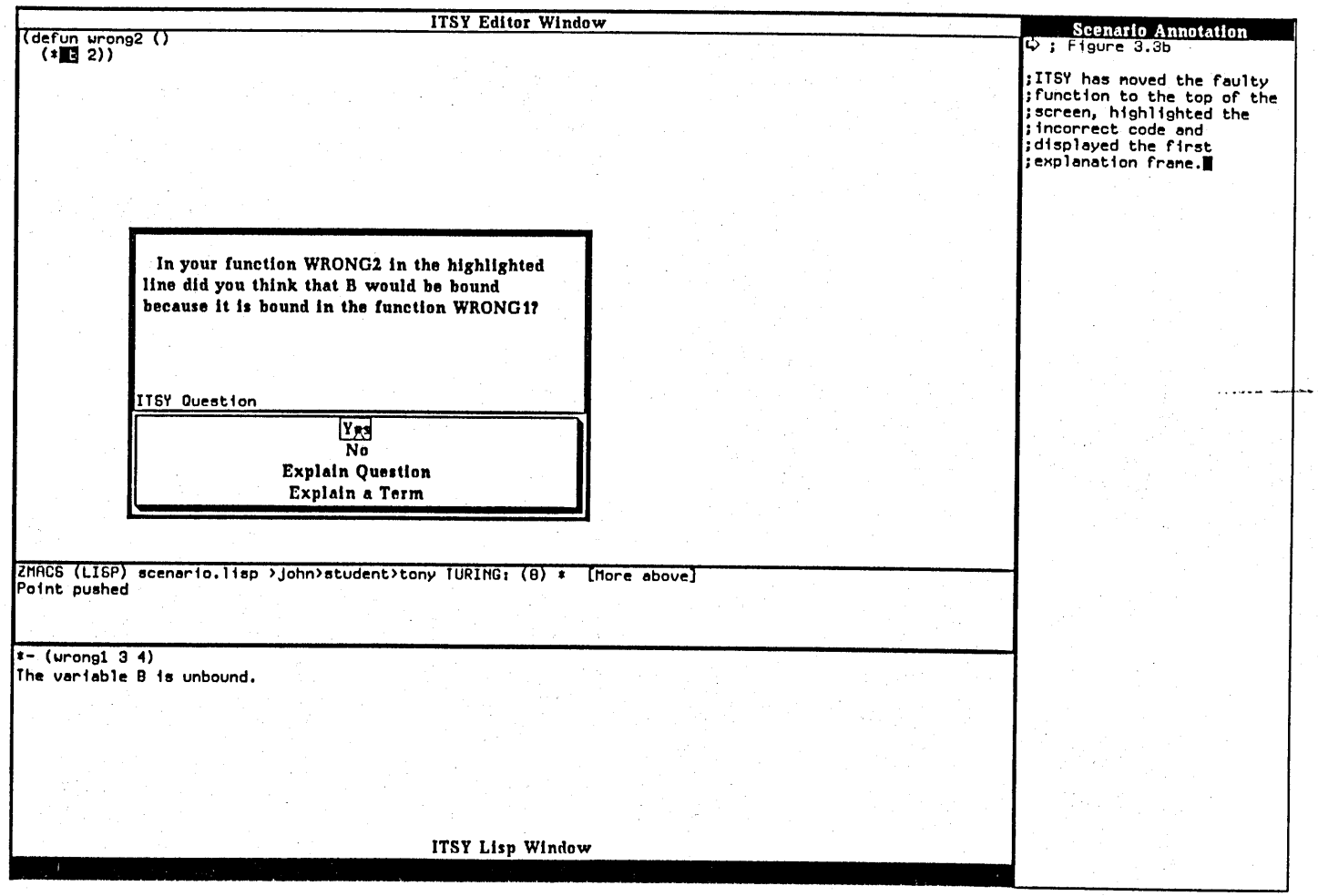

ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun buggy-first (l)
(car (l)))
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (8) *
*- (buggy-first '(a b c))
The function L is undefined.
If you want a tutorial select 'Present Tut
(Click on any mouse button to get the ITSY
NIL
*-
ITSY Lisp Menu
Editor
Load File
Test a Function
Describe a Function
Present Tutorial
ITSY Lisp Window
Scenario Annotation
; Figure 3.2c
;Tony has brought up the
;Lisp menu (by pressing
;a mouse button) and
;selected 'Present Tutorial'

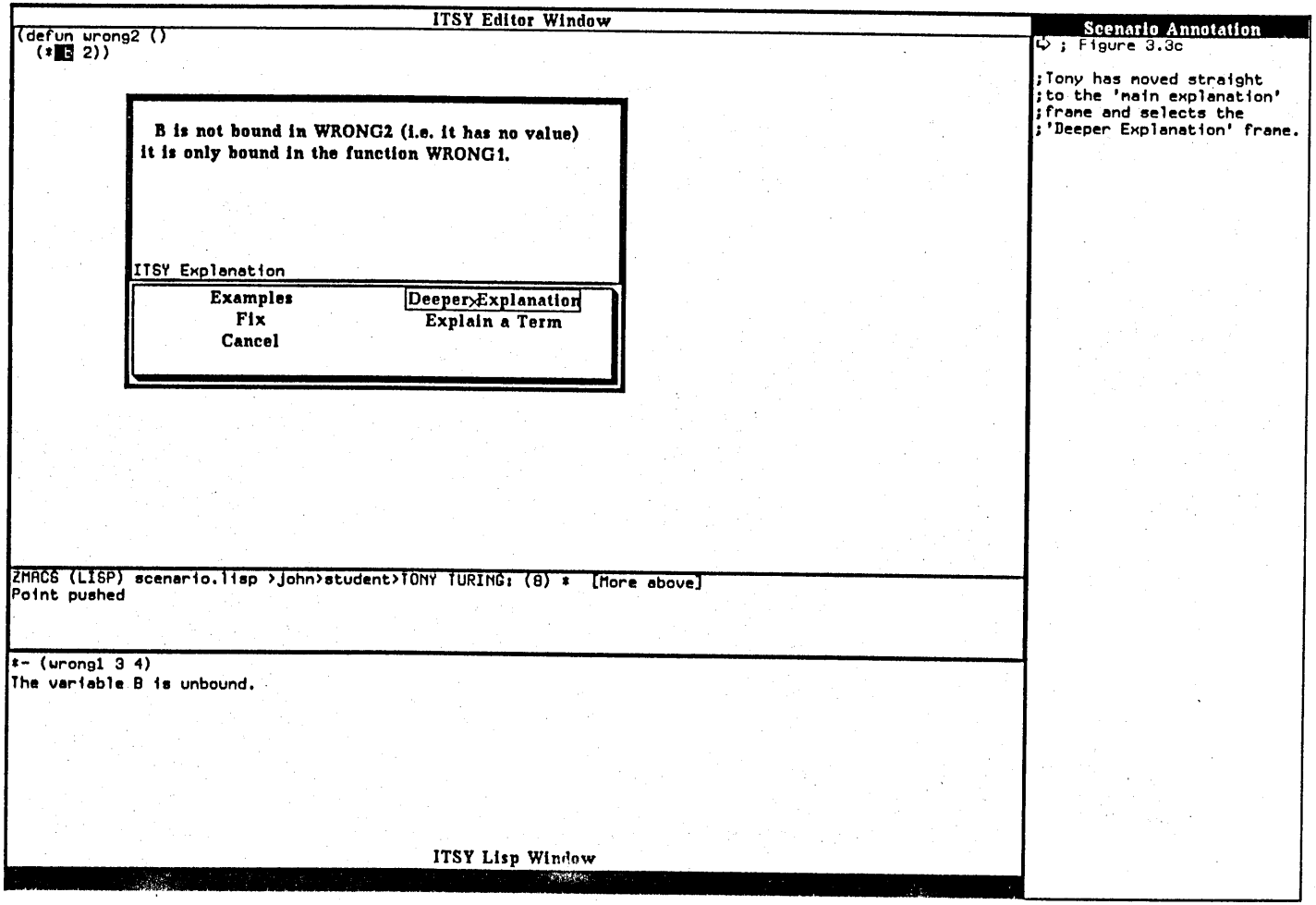

ITSY Editor Window
(defun buggy-first (l)
(car (l)))
In your function BUGGY-FIRST does L in the highlighted line refer to the variable L in the parameter list?
ITSY Question
Yes
No
Explain Question
Explain a Term
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (8) * [More above]
Point pushed
*- (buggy-first '(a b c))
The function L is undefined.
If you want a tutorial select 'Present Tutorial' on ITSY Menu
(Click on any mouse button to get the ITSY Menu)
NIL
*-
ITSY Lisp Window
Scenario Annotation
; Figure 3.2d
;ITSY presents the first
;frame of the tutorial.

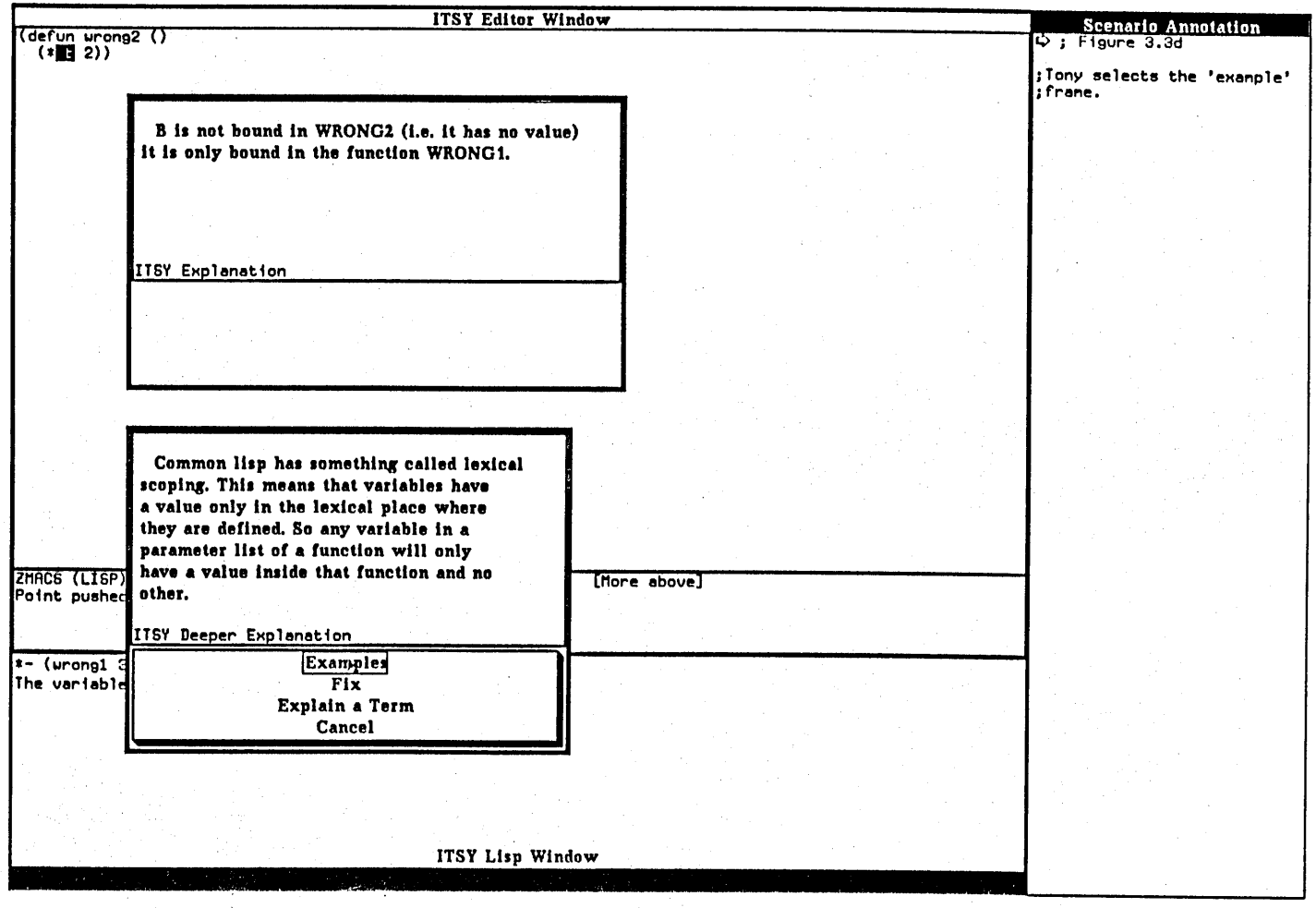
3.3 Third Scenario

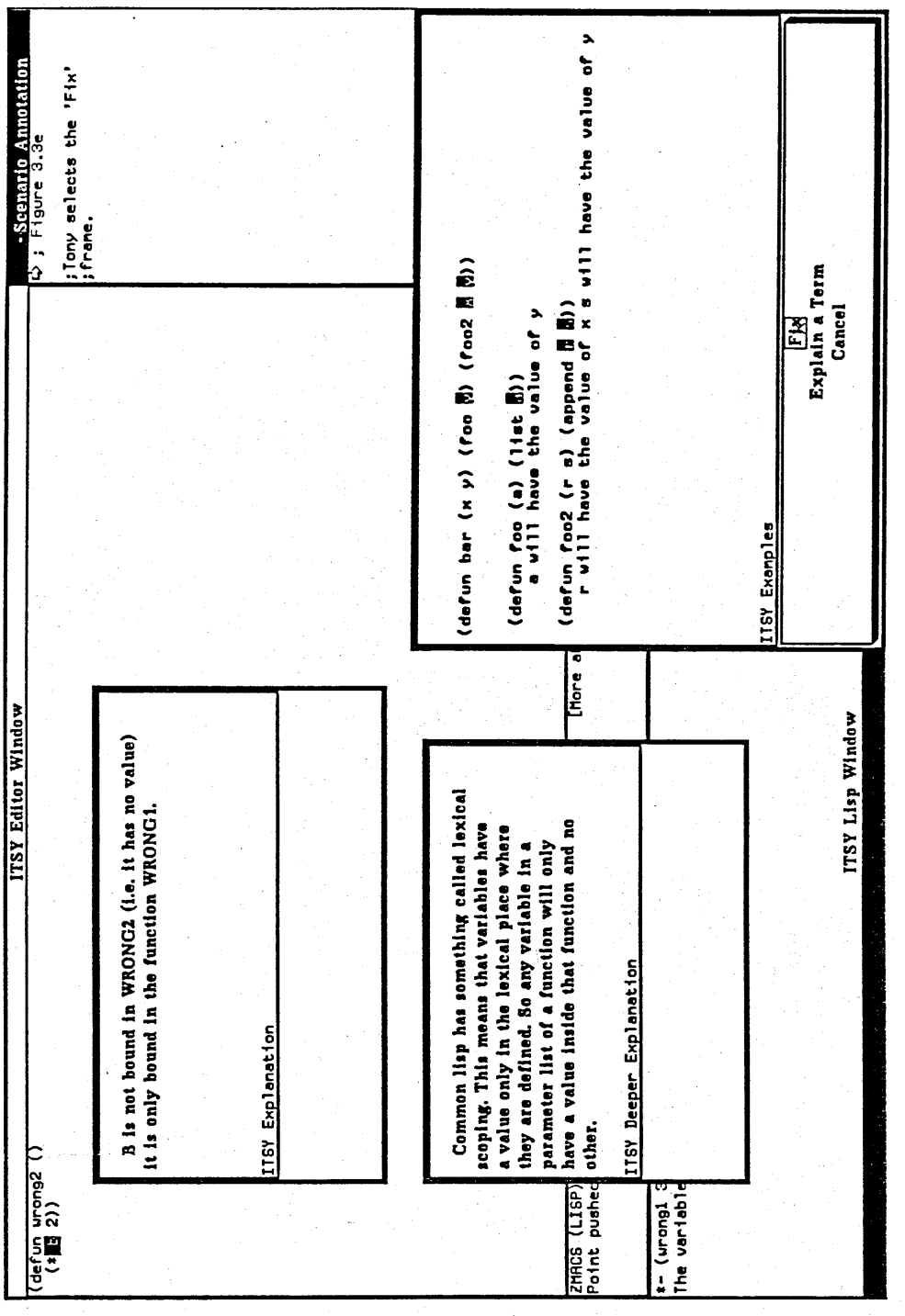
The third scenario shows the student making a different error.

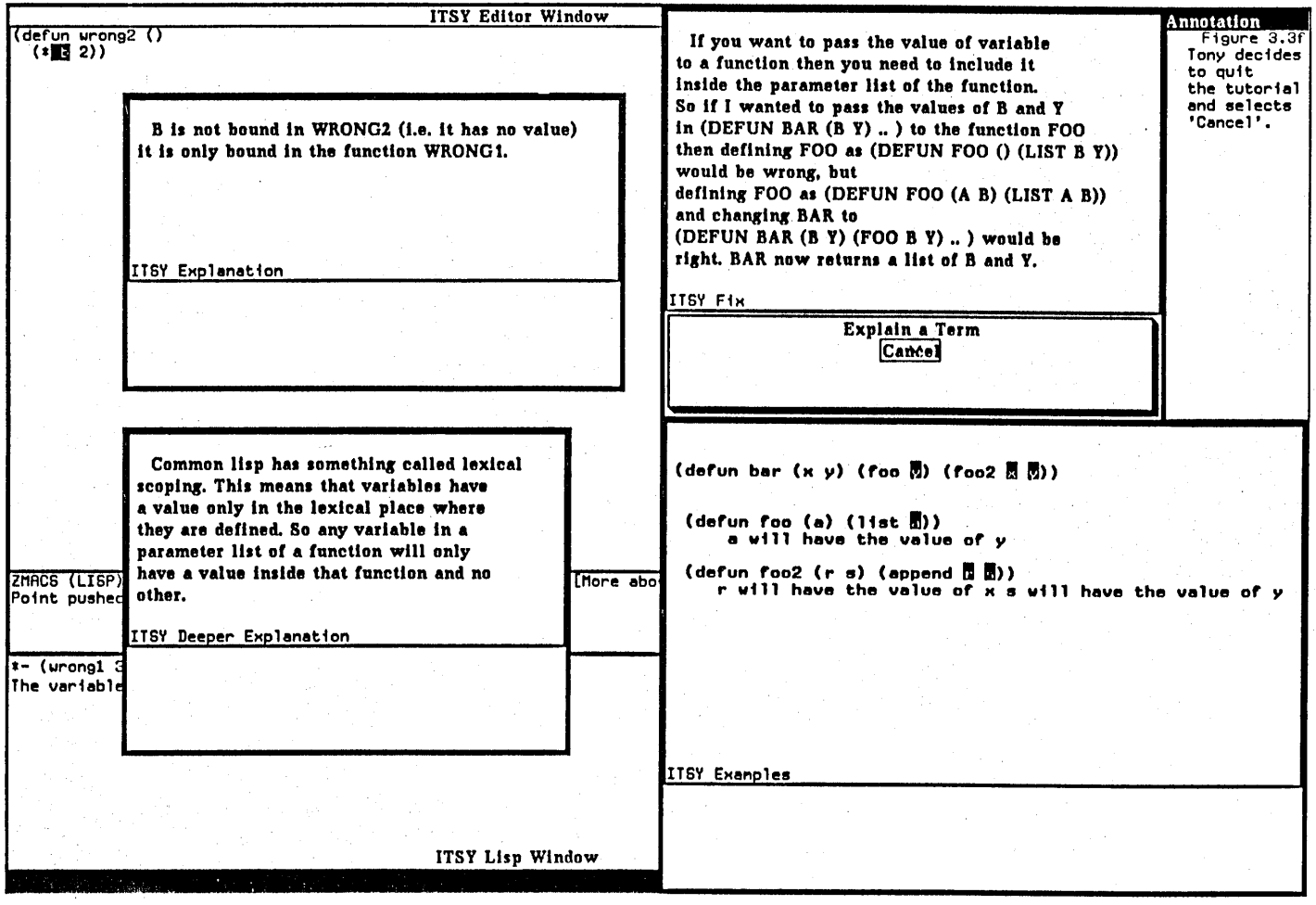

ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun buggy-first (l)
(car (l)))
(defun wrong1 (a b)
(+ a (wrong2)))
(defun wrong2 ()
(* b 2))
ZMACS (LISP) scenario.lisp >john>student>tony TURING: (8) *
Point pushed
*- (wrong1 3 4)
The function WRONG1 is undefined.
ITSY Lisp Window
Trying to find the error: "UNDEFINED-FUNCTION"
Scenario Annotation
; Figure 3.3a
;Tony has just typed in the
;form into the Lisp window.
;The error message has been
;displayed and ITSY is
;trying to find the source
;of the error.

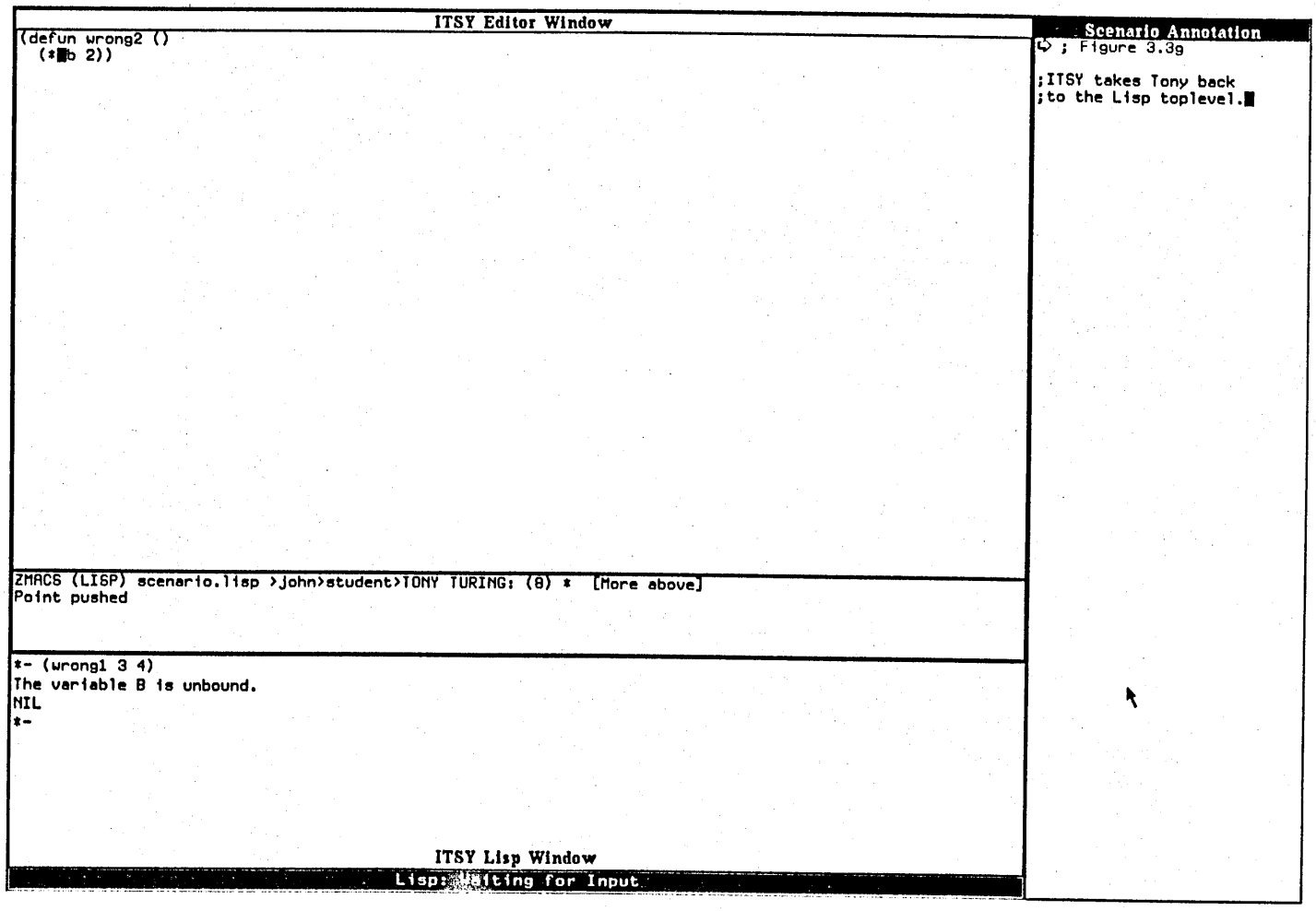

ITSY Editor Window
(defun wrong2 ()
(* B 2))
In your function WRONG2 in the highlighted line did you think that B would be bound because it is bound in the function WRONG1?
ITSY Question
Yes
No
Explain Question
Explain a Term
ZMACS (LISP) scenario.lisp >john>student>tony TURING: (8) * [More above]
Point pushed
*- (wrong1 3 4)
The variable B is unbound.
ITSY Lisp Window
Scenario Annotation
; Figure 3.3b
;ITSY has moved the faulty
;function to the top of the
;screen, highlighted the
;incorrect code and
;displayed the first
;explanation frame.


ITSY Editor Window
(defun wrong2 ()
(* B 2))
B is not bound in WRONG2 (i.e. it has no value)
it is only bound in the function WRONG1.
ITSY Explanation
Examples
Fix
Cancel
Deeper Explanation
Explain a Term
ZMACS (LISP) scenario.lisp >John>student>TONY TURING: (8) * [More above]
Point pushed
*- (wrong1 3 4)
The variable B is unbound.
ITSY Lisp Window
Scenario Annotation
; Figure 3.3c
;Tony has moved straight
;to the 'main explanation'
;frame and selects the
;'Deeper Explanation' frame.


ITSY Editor Window
(defun wrong2 ()
(* B 2))
B is not bound in WRONG2 (i.e. it has no value)
it is only bound in the function WRONG1.
ITSY Explanation
Common lisp has something called lexical scoping. This means that variables have a value only in the lexical place where they are defined. So any variable in a parameter list of a function will only have a value inside that function and no other.
ITSY Deeper Explanation
Examples
Fix
Explain a Term
Cancel
ZMACS (LISP)
Point pushed
[More above]
*- (wrong1 3
The variable
ITSY Lisp Window
Scenario Annotation
; Figure 3.3d
;Tony selects the 'example'
;frame.

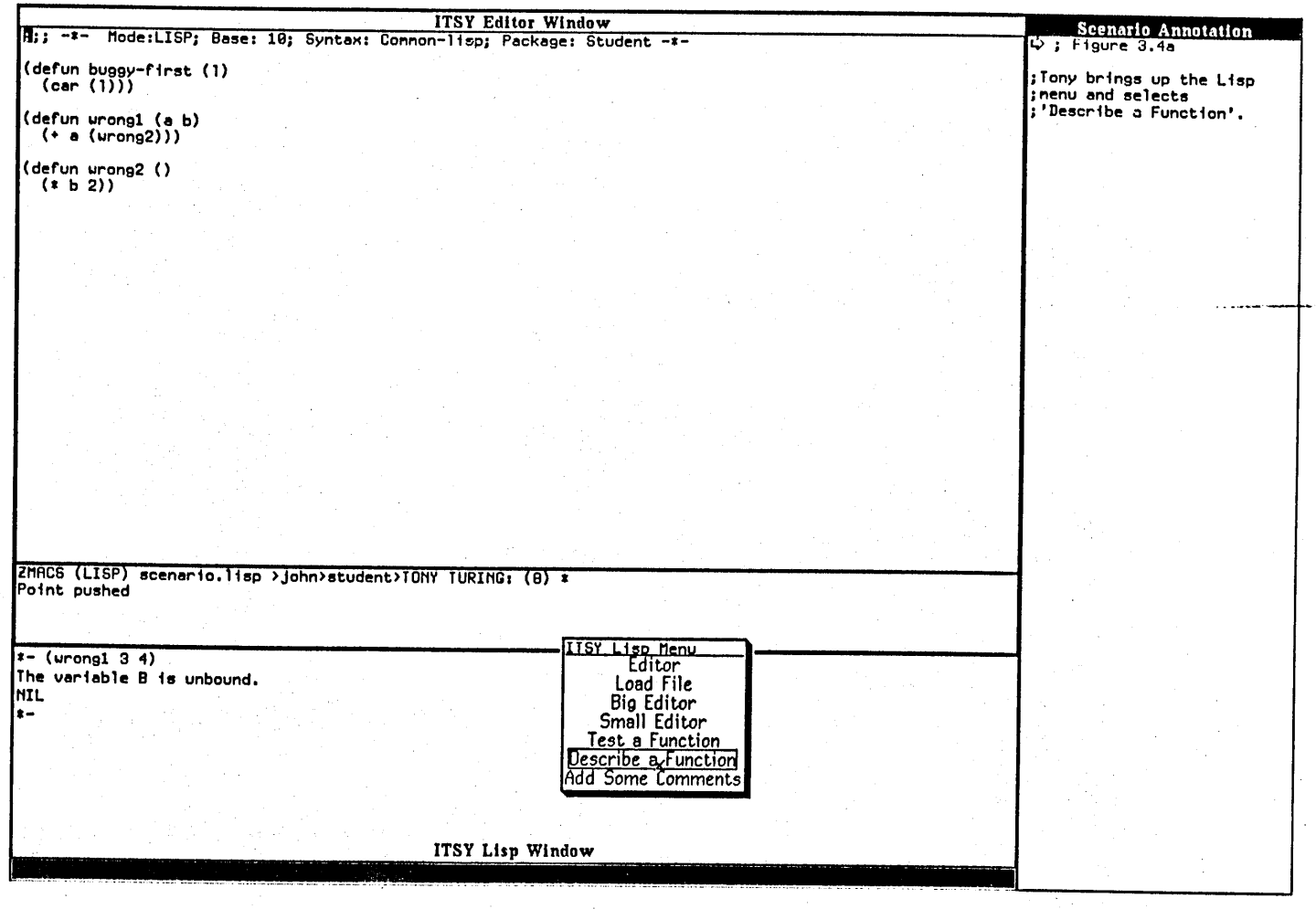

-Scenario Annotation
; Figure 3.3e
;Tony selects the 'Fix'
;frame.
ITSY Editor Window
(defun wrong2 ()
(* 2))
B is not bound in WRONG2 (i.e. it has no value)
It is only bound in the function WRONG1.
ITSY Explanation
(defun bar (x y) (foo) (foo2))
(defun foo (a) (list))
a will have the value of y
(defun foo2 (r s) (append))
r will have the value of x s will have the value of y
ITSY Examples
Fix
Explain a Term
Cancel
Common lisp has something called lexical scoping. This means that variables have a value only in the lexical place where they are defined. So any variable in a parameter list of a function will only have a value inside that function and no other.
ITSY Deeper Explanation
ITSY Lisp Window

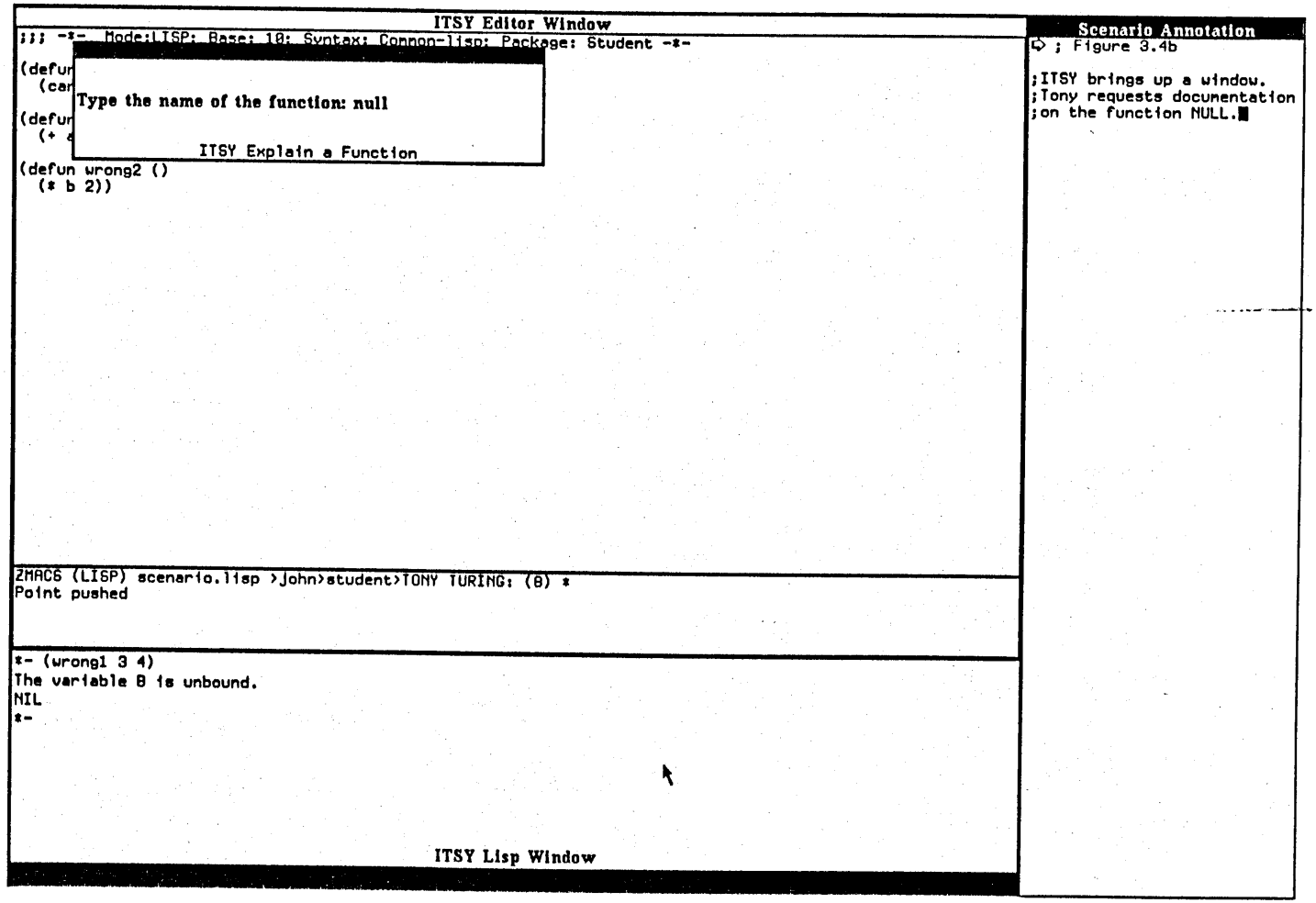

ITSY Editor Window
(defun wrong2 ()
(* 2))
B is not bound in WRONG2 (i.e. it has no value)
it is only bound in the function WRONG1.
ITSY Explanation
Common lisp has something called lexical
scoping. This means that variables have
a value only in the lexical place where
they are defined. So any variable in a
parameter list of a function will only
have a value inside that function and no
other.
ITSY Deeper Explanation
ZMACS (LISP)
Point pushed
[More abo
*- (wrong1 3
The variable
ITSY Lisp Window
If you want to pass the value of variable
to a function then you need to include it
inside the parameter list of the function.
So if I wanted to pass the values of B and Y
in (DEFUN BAR (B Y) ..) to the function FOO
then defining FOO as (DEFUN FOO () (LIST B Y))
would be wrong, but
defining FOO as (DEFUN FOO (A B) (LIST A B))
and changing BAR to
(DEFUN BAR (B Y) (FOO B Y) ..) would be
right. BAR now returns a list of B and Y.
ITSY Fix
Explain a Term
Cancel
Annotation
Figure 3.3f
Tony decides
to quit
the tutorial
and selects
'Cancel'.
(defun bar (x y) (foo y) (foo2 x y))
(defun foo (a) (list a))
a will have the value of y
(defun foo2 (r s) (append r s))
r will have the value of x s will have the value of y
ITSY Examples

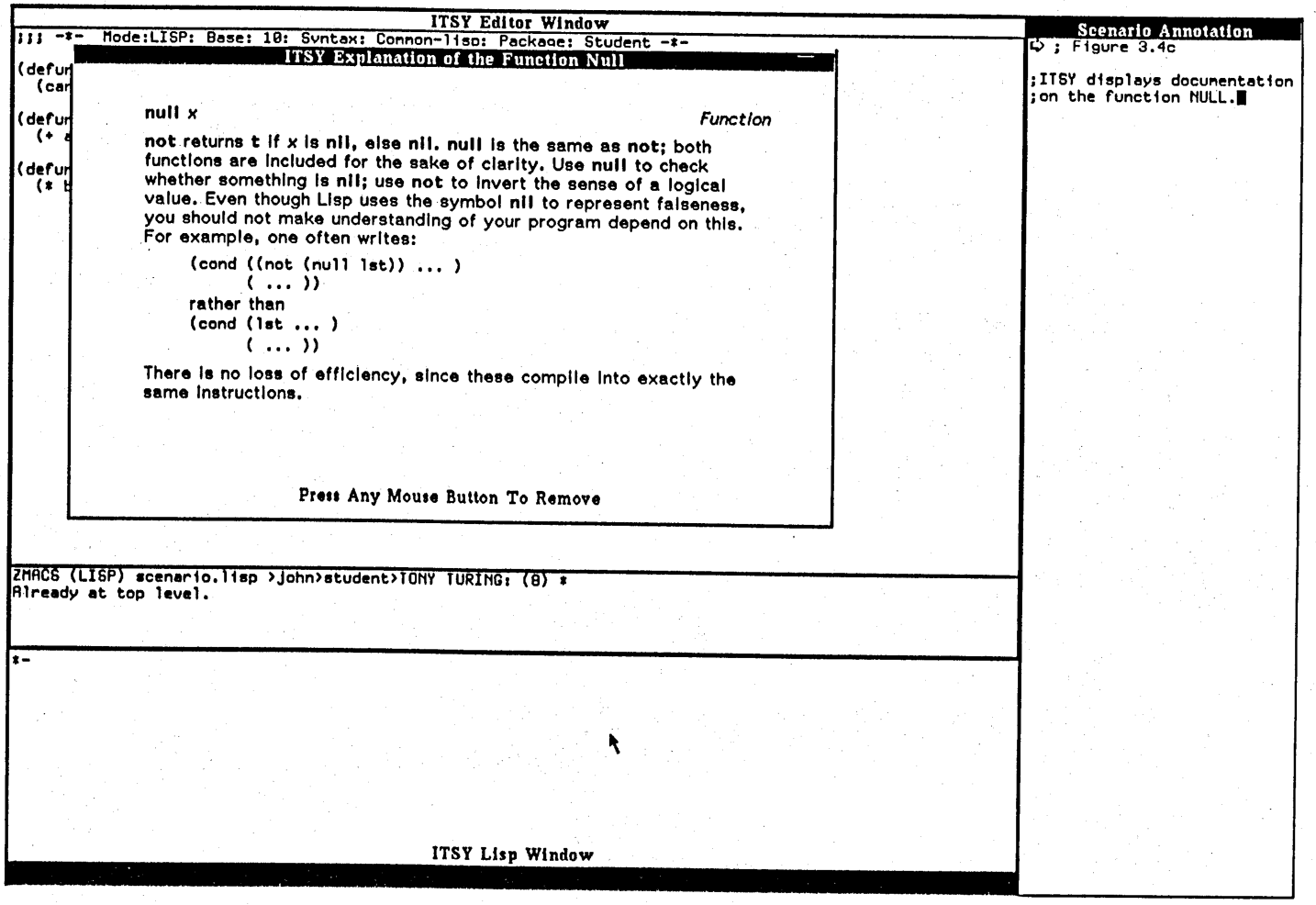

ITSY Editor Window
(defun wrong2 ()
(*b 2))
Scenario Annotation
; Figure 3.3g
;ITSY takes Tony back
;to the Lisp toplevel.
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (8) * [More above]
Point pushed
*- (wrong1 3 4)
The variable B is unbound.
NIL
*-
ITSY Lisp Window
Lisp: Waiting for Input

3.4 Fourth Scenario

The fourth scenario shows how students can call up documentation on Lisp functions.

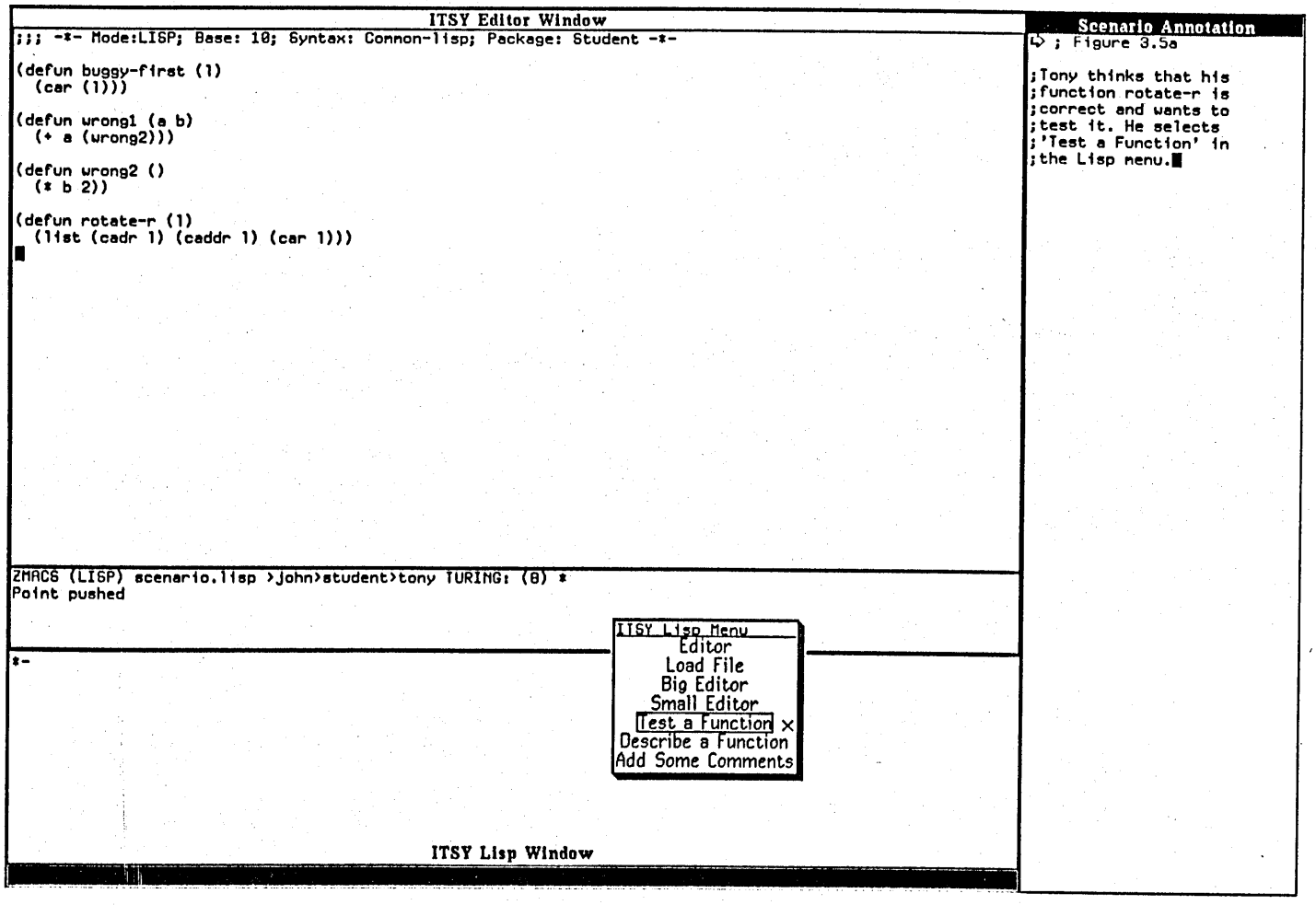

ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun buggy-first (l)
(car (l)))
(defun wrong1 (a b)
(+ a (wrong2)))
(defun wrong2 ()
(* b 2))
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (0) *
Point pushed
*- (wrong1 3 4)
The variable B is unbound.
NIL
*-
ITSY Lisp Menu
Editor
Load File
Big Editor
Small Editor
Test a Function
Describe a Function
Add Some Comments
ITSY Lisp Window
Scenario Annotation
; Figure 3.4a
;Tony brings up the Lisp
;menu and selects
;'Describe a Function'.

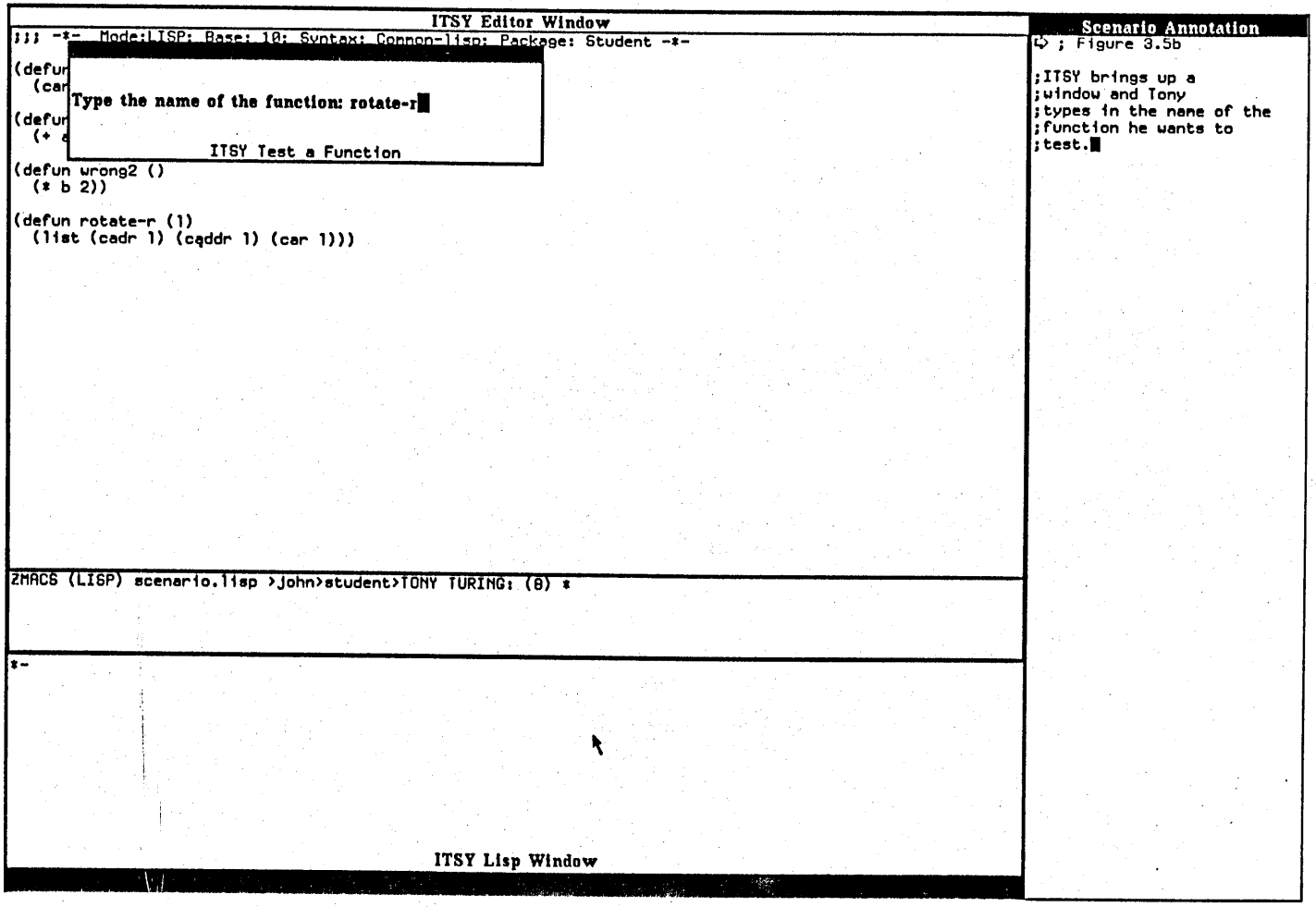

ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun
(car
(defun
(+ a
Type the name of the function: null
ITSY Explain a Function
(defun wrong2 ()
(* b 2))
ZMACS (LISP) scenario.lisp >John>student>TONY TURING: (8) *
Point pushed
*- (wrong1 3 4)
The variable B is unbound.
NIL
*-
ITSY Lisp Window
Scenario Annotation
; Figure 3.4b
;ITSY brings up a window.
;Tony requests documentation
;on the function NULL.

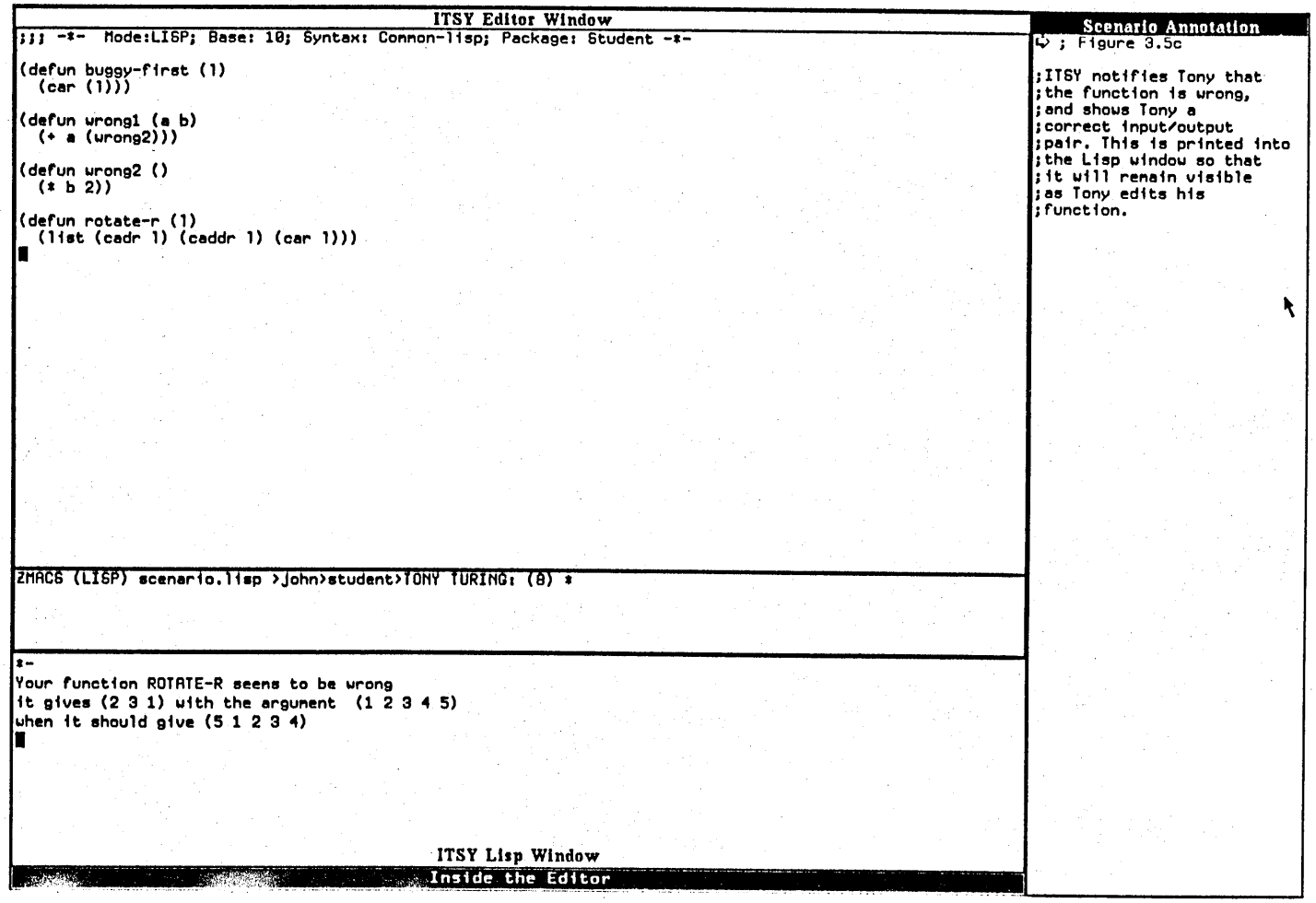

ITSY Editor Window
;;; -*- Mode:LISP: Base: 10: Syntax: Common-lisp: Package: Student -*-
ITSY Explanation of the Function Null
null x
Function
not returns t if x is nil, else nil. null is the same as not; both functions are included for the sake of clarity. Use null to check whether something is nil; use not to invert the sense of a logical value. Even though Lisp uses the symbol nil to represent falseness, you should not make understanding of your program depend on this. For example, one often writes:
(cond ((not (null lst)) ...)
(...))
rather than
(cond (lst ...)
(...))
There is no loss of efficiency, since these compile into exactly the same instructions.
Press Any Mouse Button To Remove
ZMACS (LISP) scenario.lisp >John>student>TONY TURING: (8) *
Already at top level.
*-
ITSY Lisp Window
Scenario Annotation
; Figure 3.4c
;ITSY displays documentation
;on the function NULL.

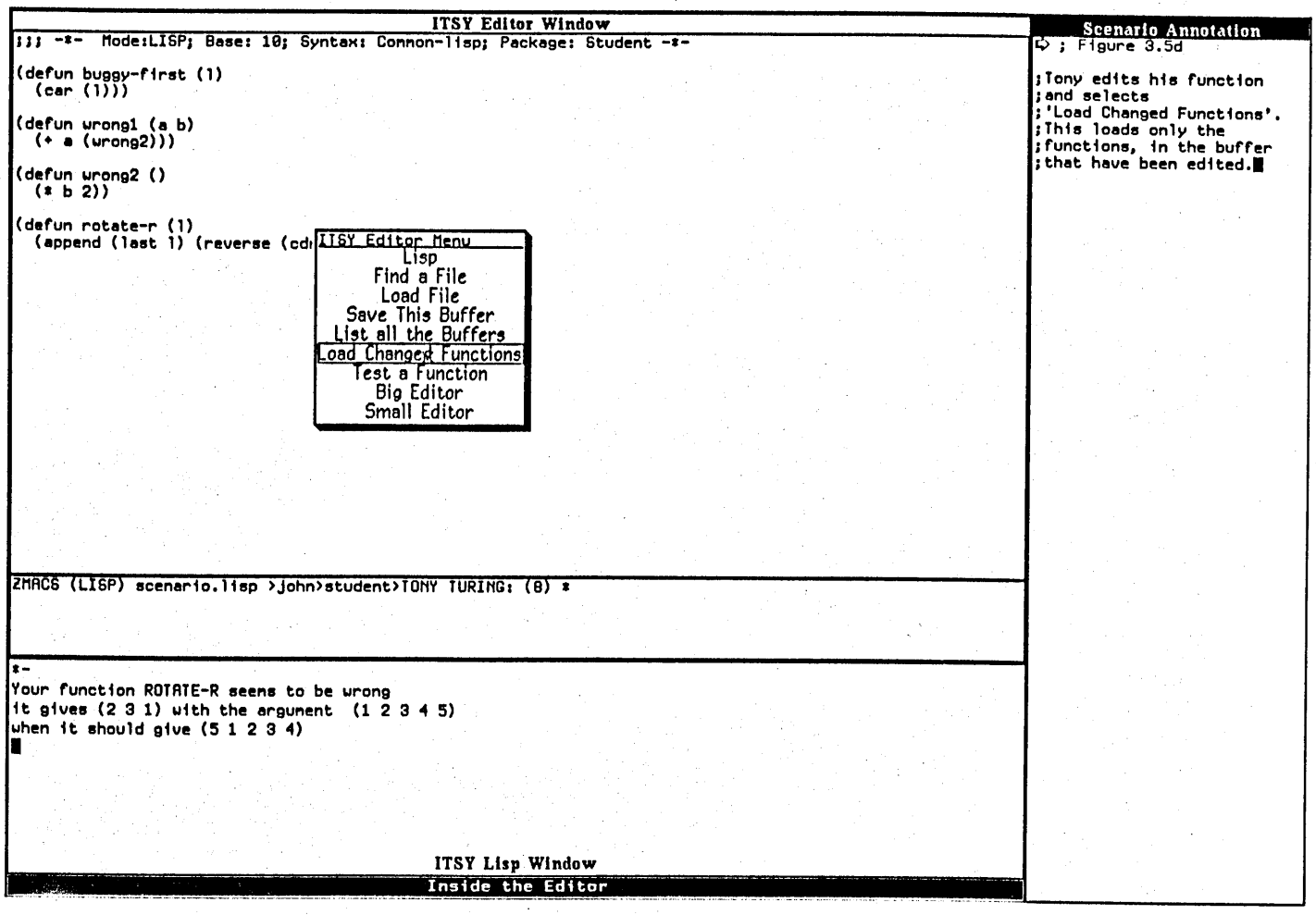
3.5 Fifth Scenario

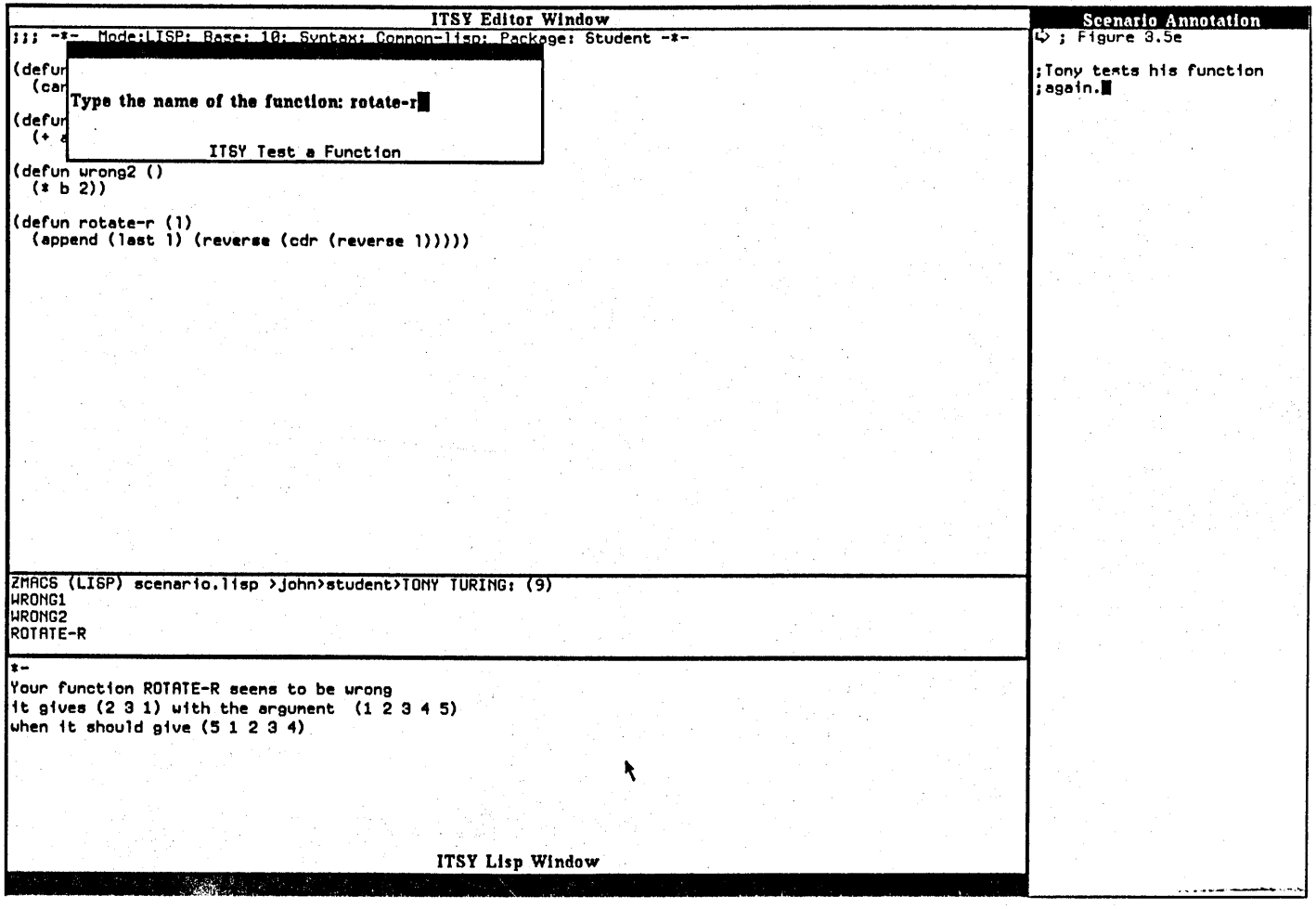
The fifth scenario shows the student using ITSY's test facility. ITSY has a set of pre-stored examples for each exercise. The function rotate-r (exercise 3-3 in "Lisp" [Winston & Horn]) should rotate *any* list right but often student's solutions only cope with lists with three elements. This scenario also shows how students can quickly reload their functions.

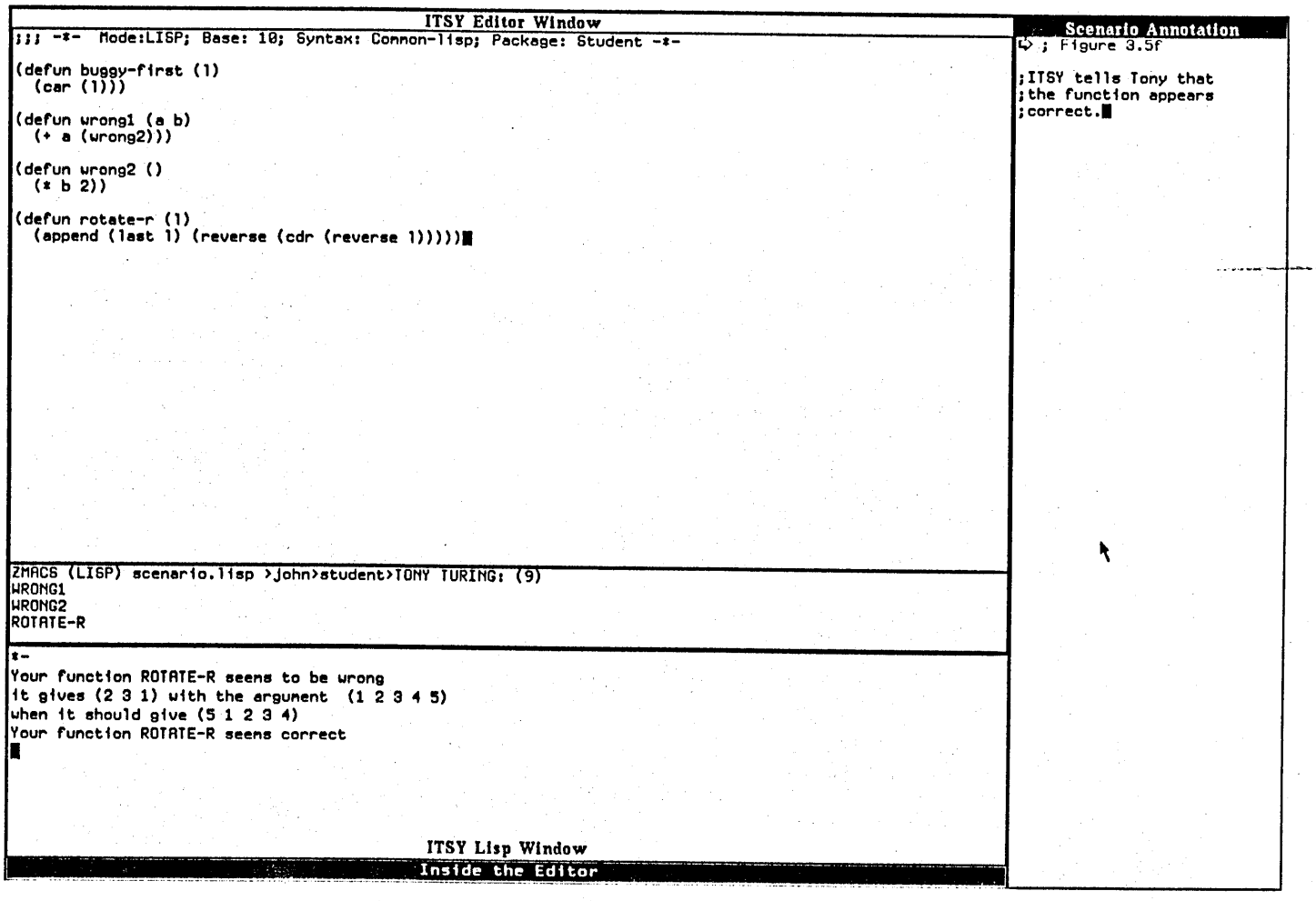

ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun buggy-first (l)
(car (l)))
(defun wrong1 (a b)
(+ a (wrong2)))
(defun wrong2 ()
(* b 2))
(defun rotate-r (l)
(list (cadr l) (caddr l) (car l)))
Scenario Annotation
; Figure 3.5a
;Tony thinks that his
;function rotate-r is
;correct and wants to
;test it. He selects
;'Test a Function' in
;the Lisp menu.
ZMACS (LISP) scenario.lisp >john>student>tony TURING: (8) *
Point pushed
ITSY Lisp Menu
Editor
Load File
Big Editor
Small Editor
Test a Function
Describe a Function
Add Some Comments
*-
ITSY Lisp Window


ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun
(car
(defun
(+ a
Type the name of the function: rotate-r
ITSY Test a Function
(defun wrong2 ()
(* b 2))
(defun rotate-r (l)
(list (cadr l) (cqddr l) (car l)))
ZMACS (LISP) scenario.lisp >John>student>TONY TURING: (8) *
*-
ITSY Lisp Window
Scenario Annotation
; Figure 3.5b
;ITSY brings up a
;window and Tony
;types in the name of the
;function he wants to
;test.


ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun buggy-first (l)
(car (l)))
(defun wrong1 (a b)
(+ a (wrong2)))
(defun wrong2 ()
(* b 2))
(defun rotate-r (l)
(list (cadr l) (caddr l) (car l)))
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (8) *
*-
Your function ROTATE-R seems to be wrong
it gives (2 3 1) with the argument (1 2 3 4 5)
when it should give (5 1 2 3 4)
ITSY Lisp Window
Inside the Editor
Scenario Annotation
; Figure 3.5c
;ITSY notifies Tony that
;the function is wrong,
;and shows Tony a
;correct input/output
;pair. This is printed into
;the Lisp window so that
;it will remain visible
;as Tony edits his
;function.


ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun buggy-first (l)
(car (l)))
(defun wrong1 (a b)
(+ a (wrong2)))
(defun wrong2 ()
(* b 2))
(defun rotate-r (l)
(append (last l) (reverse (cd
ITSY Editor Menu
Lisp
Find a File
Load File
Save This Buffer
List all the Buffers
Load Changed Functions
Test a Function
Big Editor
Small Editor
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (8) *
*-
Your function ROTATE-R seems to be wrong
it gives (2 3 1) with the argument (1 2 3 4 5)
when it should give (5 1 2 3 4)
ITSY Lisp Window
Inside the Editor
Scenario Annotation
; Figure 3.5d
;Tony edits his function
;and selects
;'Load Changed Functions'.
;This loads only the
;functions, in the buffer
;that have been edited.


ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun
(car
(defun
(+ a
Type the name of the function: rotate-r
ITSY Test a Function
(defun wrong2 ()
(* b 2))
(defun rotate-r (l)
(append (last l) (reverse (cdr (reverse l)))))
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (9)
WRONG1
WRONG2
ROTATE-R
*-
Your function ROTATE-R seems to be wrong
it gives (2 3 1) with the argument (1 2 3 4 5)
when it should give (5 1 2 3 4)
ITSY Lisp Window
Scenario Annotation
; Figure 3.5e
;Tony tests his function
;again.


ITSY Editor Window
;;; -*- Mode:LISP; Base: 10; Syntax: Common-lisp; Package: Student -*-
(defun buggy-first (1)
(car (1)))
(defun wrong1 (a b)
(+ a (wrong2)))
(defun wrong2 ()
(* b 2))
(defun rotate-r (1)
(append (last 1) (reverse (cdr (reverse 1)))))
Scenario Annotation
; Figure 3.5f
;ITSY tells Tony that
;the function appears
;correct.
ZMACS (LISP) scenario.lisp >john>student>TONY TURING: (9)
WRONG1
WRONG2
ROTATE-R
*-
Your function ROTATE-R seems to be wrong
it gives (2 3 1) with the argument (1 2 3 4 5)
when it should give (5 1 2 3 4)
Your function ROTATE-R seems correct
ITSY Lisp Window
Inside the Editor

4. ITSY IN THE CONTEXT OF RELATED WORK

This chapter presents a review of related work. This review is split into three basic parts which correspond to the three broad fields that this work is founded on. The first part describes work on empirical studies of novice programmers. The second part describes work on Intelligent Tutoring Systems/Computer Aided Instruction. The third part describes work in the area of Automatic Program Analysis/Debugging.

4.1 Empirical Studies of Novice Programmers

Numerous studies on both expert and novice programmers have been carried out. These studies have varied on the type of data collected. Boies and Gould [1974] collect syntactic errors, caught at compile time, in FORTRAN, PL/1 and an Assembler Language. More recent studies have concentrated on the deeper cause of the error, rather than the symptom.

Various studies conducted a Yale university have used the idea of *Programming Plans* [Spohrer & Soloway, 1986; Johnson, Draper & Soloway, 1982; 1983]. Programming Plans are stereotypic sequences of code that accomplish some *Programming Goal.* A Programming Goal states what must be accomplished in order to solve a particular problem. A Programming Plan states how a particular goal can be achieved. Generally, there will be more than one Plan for a particular Goal as there is often more than one way of implementing a specification.

Spohrer and Soloway [1986] analysed students on a undergraduate PASCAL programming course. 158 syntactically correct PASCAL programs generated by 61 students. The bugs were analysed using Programming Goals and Plans. The bugs were identified as the difference between one of the correct Plans for achieving a Goal and the observed Plans for achieving a Goal. The following table shows the results of the analysis.

Type of Statistic	Problem 1	Problem 2	Problem 3
Number of Subject Analysed	55	46	57
Total Number of Bugs	85	46	57
Total Number Bug Types	28	46	27
Average Number of Bugs per Syntactically Correct Version	1.5	3.0	1.0
Average Number of Lines per Program	27.8	73.1	68.6
Assignment Number in Sequence	2	3	8
Percentage of Errors per Line	5.6	4.2	1.5

The assignment number indicates where the assignment occurred in the sequence of course assignments.

The average percentage of errors per line, in the study described above, is 3.2. In the studies outlined in chapters 5, 11 and 12 the average percentage of errors per line varied between 13 and 45. In these studies only the number of lines typed to the Lisp interpreter were counted. The lines in the actual program were discarded. This means that one line in these studies is equivalent to 'trying out a program'. If the number of errors in the study conducted by Spohrer et. al is taken per try out of the program, the average becomes 180. The increase from in the percentage between the studies outlined in this thesis and those of Spohrer et al. could be due to the increase in complexity of the assignments (the solutions to the exercises used in the studies carried out in this thesis are less than 10 lines in length).

Spohrer and Soloway concluded two things from this study:

a) misconceptions about language constructs do not seem to be as widespread or troublesome as typically believed,

b) just a few bug types account for a large percentage of program bugs.

Spohrer and Soloway analysed the top 10% of bug types (which accounted for well over a third of the bugs). Out of the 11 bug types only one was definitely caused by a programming construct.

The spread of the bugs found can be seen in the following table

Percentage of bug types	Percentage of bugs accounted for		
	Problem 1	Problem 2	Problem 3
10	44	46	32
20	55	64	46
25	62	69	56
50	80	84	77

As can be seen the top 10% of bug types account for over a third of the actual bugs. In the study outlined in chapter 5 91% of the errors fell into 19 categories.

A comparison of errors made by novice SOLO and LOGO programmers is given in [Eisenstadt & Lewis, 1985] . The SOLO studies were carried out by Eisenstadt and Lewis [1985] and the LOGO studies by Du Boulay [1979]. Eisenstadt and Lewis re-computed Du Boulay's collection of errors taking into account the differences between the two systems used in the studies. One of the differences was that the SOLO environment contained a spelling checker whereas the LOGO environment did not. The spelling errors caught by the spelling checker in SOLO environment were counted as errors.

The following table shows the four largest SOLO and LOGO errors side by side.

Symptom	% of all errors	
	LOGO	SOLO
1. Spelling/Typing/misquoting	28	34
2. Wrong Number of arguments passed	18	18
3. No line Number	12	9
4. Call to undefined procedure	12	9

As can be seen the top four culprits for both languages are the same and occur in the same relative order.

Pre-programming knowledge is a major source of programming errors. Bonar & Soloway [1985] developed a model to account for novices programming errors. They propose 2 kinds plan knowledge that novices have:

a) Knowledge of Step-by-Step Natural Language procedures - this is called SSK (Step-by-Step natural language Knowledge).

b) Novice knowledge of the programming language under study (PASCAL) - this is called PK (Pascal programming Knowledge).

Examples of a) are looping (eg. adding lists of figures), making choices and specifying sequences of actions.

Bonar and Soloway have characterised similarities exist between SSK and PK:

1. Functional similarities - both SSK and PK are concerned with repeated actions, choice between conditions, counting, etc.

2. Surface similarities - programming languages such as PASCAL share many words with Natural Language.

Bonar and Soloway outline the following as what happens when a novice programmer produces a bug:

1. While solving a programming problem novices will encounter some aspect of the problem they don't understand - an impasse.

2. In order to move beyond the impasse, novices cast about for a way to resolve the aspect of the problem they don't understand - a patch. Frequently that resolution involves an appeal to their knowledge of Natural Language step-by-step procedures that would be applicable in a similar situation.

3. In implementing the patch, a bug is introduced.

Kahney and Eisenstadt [1982] also showed that inappropriate pre-programming knowledge is used when novices write programs, in this case SOLO programs.

Anderson, Pirolli and Farrell [1984] studied novice Lisp programmers writing recursive programs. They gave the following reasons why recursive programming in Lisp is difficult:

a) Very difficult if not impossible for humans to execute recursive mental procedures.

b) Recursion is unfamiliar.

c) Recursion is difficult because of imprecise instructions. Textbooks do not explain how to write recursive functions. Writing recursive functions is not recursive. Text books explain what recursion is, explain how recursion works, examples of recursive functions, give traces of recursive functions but do not explain how recursion works.

d) Students try to solve recursion problems iteratively.

e) Recursion is complex. There are different types of recursion for example, CDR recursion and CAR-CDR recursion.

f) Other non-recursive aspects of Lisp complicate learning about recursion.

4.2 Intelligent Tutoring/Computer Aided Instruction

This section is divided into five sections corresponding to the five sections of ITSY.

4.2.1 Spotting Errors

Traditional CAI packages compare a student's answer with a correct version. Typically the answers are simple yes/no type or multiple choice. The BIP system [Barr, Beard & Atkinson, 1976] simply compared the output of student programs with a "correct" version. A more interesting method was used in WEST [Burton & Brown, 1976]. WEST compared the students' answers with those of an "expert", then assumed that the students lacked all of the skills needed to produce the optimum answer, unless they had used that skill recently. WEST would then tutor the student on one of these skills.

GREATERP [Anderson, 1985] uses production rules to implement both an expert and a "buggy" novice LISP programmer. These rules are implemented in GRAPES (Goal Restricted Production System). Every time a student types a LISP symbol GREATERP decides what rule would have to fire in order to duplicate the input. If the "duplicating" rule is in the "expert" set then GREATERP does nothing, but if the "duplicating" rule is in the "buggy" set then GREATERP gives the student a short tutorial. As long as the student writes "ideal" code GREATERP stays in background. There is a flaw in this strategy however. If a student is writing a variation of the solution that GREATERP does not know about then the student will get confusing advice.

The approach taken by GREATERP is different to that of ITSY. GREATERP inspects each symbol as the student types it in and proceeds to tutor the student if the symbol is incorrect. ITSY is less restrictive. The student is allowed to write the whole of his/her program before ITSY examines it. GREATERP has the advantage that it is able to tutor the student as soon as the error occurs. One of the disadvantages is that the student does

not have a chance to right a whole program at once. Nor does the student practice debugging programs.

Ideally the expert should be "glass box" or articulate [Goldstein & Papert, 1977]; non-articulate or "black-box" experts only being used where no theory exists on how to build one, or for reasons of efficiency.

As discussed in section 2.1 ITSY uses some of the techniques used by the Intelligent Debuggers described in section 4.2, rather than those used in traditional CAI packages.

4.2.2 Presentation Method

There are two main methods that CAI packages use in order to communicate with the student:

a) displaying stored chunks of text,

b) using natural language generators to create the text.

As Barr and Beard said [Barr & Beard, 1976 p 570, 571], storing chunks of text limits the amount of branching possible and so makes the system inflexible whereas natural language generators have a limited vocabulary and consequently their communications with students tend to be dry and unmotivating.

ITSY uses a combination of stored chunks of text and annotated graphic examples, because natural language generation is considered beyond the scope of this project.

4.2.3 Student Model

O'Shea and Self [O'Shea & Self, 1983 p. 143] describe a student model as follows:

> ... any information which a teaching program has which is specific to the particular student being taught. The reason for maintaining such information is to help the program to decide on appropriate teaching actions.

In ITSY a student model is needed to determine when a student has made an interesting

error, as any error that occurs may be due to a fundamental misconception, or may just be a trivial slip up. BIP [Barr & Beard, 1976] used a counter for each elementary skill in BASIC programming as a student model. Each time a student successfully completed a task the counters for all the skills needed in that task were incremented. If the student failed to complete the task the corresponding counters were decremented. A problem that Wescourt et al. [1977] found with this scheme was that the faster students sometimes "leap-frogged" over the simpler tasks and then failed on a difficult task. Because they missed out a lot of the simpler tasks, they may have met some of the simpler skills only once. If such a skill was then needed in a difficult task, and they failed on this task, the counter for this skill would be decremented to zero. The model then contained the inaccuracy that the student did not have this skill. This student modelling problem was overcome in BIP-2 [Wescourt et al, 1977]. Skills were represented by finite state machines rather than counters. The finite state model used was hierarchical, each skill being on a different level. The model had five possible states, the current state depending on how well a particular skill had been learned.

Guidon [Clancey, 1979] modelled the student as a subset of the system's 'expert', this is called an *overlay model.* The expert knowledge was represented as a set of rules (actually the expert was Mycin [Shortliffe, 1976]). Guidon's student model has three parts:

a) a record of the rules that the student knows,

b) a probability that a student will apply a particular rule in a specific case

c) a probability that a student would mention a rule if asked to support a partial solution.

In LMS (Leeds Modelling System) [Sleeman & Smith, 1981] have grouped problems into levels. If a student is be able to complete problems at a level I and then fails at a problem at level I + 1 LMS generates, from the existing student model, a set of new *possible* student models. This set is narrowed down by presenting the student with more problems from level I and deleting from the set those that did not predict the student's answer.

ITSY uses a non-hierarchical graph to represent the student's knowledge. Each node in this graph relates to a particular Lisp concept. There are two reasons why the model is

non-hierarchical. Firstly, students who use ITSY may have encountered some parts of Lisp before, each students may have some knowledge about a different part of Lisp. Secondly, students using ITSY are free to try the exercises in any order. This will mean that they will encounter concepts in a different order (see section 6.3 for a detailed description of the student model).

4.2.4 Path Selection

In traditional CAI packages the student is given a selection of tasks to complete. The student can proceed along a number of fixed routes through the system.

In early CAI packages the students followed a rigid path through the system with occasional branching. In both WEST [Burton & Brown, 1976] and BLOCKS [Brown & Burton, 1977] the students totally controlled their path through the system. In BIP-1 [Barr & Beard, 1976] the system chose the next problem for the student. The method by which to choose a problem for a student raises questions such as:

a) How many new skills should be present in the next task? Should the system try to find a task with the maximum number of new skills at a particular level or just one?

b) Should the number of new skills presented change as the student progresses, or depend on the type of student? If so how?

Wescourt et al [1977] tried to answer these questions by using various types of simulation. The method they used in BIP-2 changed the number of new skills presented according to how well the student was doing.

TRILL [Cerri, Fabrizzi, & Marsili, 1984] uses a *socratic* search strategy to find the erroneous concept that caused the student's error (TRILL asks the student). In order to carry out this search TRILL uses a semantic network representing the syntactic knowledge needed to correctly use Lisp concepts such as ATOM, LIST, CAR, CDR. This network guides control in the search for the concepts that the student needs in order to avoid making the mistake.

IMPART [Elsom-Cook, 1984] embodies a specific model of teaching interaction. Elsom-

Cook states that the goal of any teacher is to provide the student with a model of the domain which is at least as powerful as that of the teacher. Since the student has the clearest understanding of her state of knowledge and the teacher is the expert in the domain, in IMPART teaching is carried out by a negotiation between the student and teacher.

IMPART contains a teaching program which monitors the interaction between the student and the environment and attempts to contribute to the interaction. IMPART has the goal of detecting the skills which the user is ready to learn and encouraging the exploration of those skills either by manipulating the environment or by making direct "teaching statements" to the student.

In order to accomplish this IMPART has three knowledge sources:

a) Knowledge about problem domain - in this case the problem domain is Lisp. The semantics of Lisp are represented in declarative form. Lisp statements are represented in terms of preconditions for the statements application plus a body of commands to execute in order to achieve the statement's effect.

b) Knowledge about the student - this includes:

- the *apparent language:* the student's view of what statements in the language mean.

- how well students can react to error messages

c) Knowledge about interaction - IMPART participates in a structured interaction with the student - this uses psycholinguistic models of conversation. This is controlled by three subunits:

- *general interaction skills* - these maintain the consistency and smoothness of the interaction.

- *descriptors of the domain* - each concept about which the system is able to talk has a descriptor. Each descriptor contains outlines of various ways to present and discuss a topic. Also each descriptor contains mechanisms to assess its own importance at the current point of interaction (this can be thought to include the tutorial goals of the

teacher).

- *teaching strategies* - these represent different ways of tutoring such as Socratic tutoring or giving examples.

The students will choose their own tasks when using ITSY. Because of this they will not have to follow a fixed route through the system.

4.2.5 Environments As A Whole

The environment provided by ITSY will follow the principles outlined below.

Novices need a simple notional machine that corresponds to the language syntax and semantics. The language should be simple. Du Boulay, O'Shea and Monk [1981] describe two important characteristics that a programming language should have; visibility and simplicity. Visibility is concerned with providing methods for novices to observe certain parts of the notional machine working. Du Boulay, O'Shea and Monk [1981] describe three types of simplicity that a language can have:

1. Functional Simplicity

2. Logical Simplicity

3. Syntactic Simplicity.

Functional simplicity means that each instruction, in the programming language, can be broken down into a small number of basic instructions that are easy to understand. Logical simplicity means that the basic instructions in the language are suited to the job, so that problems of interest to the novice can be solved with relatively small programs. Logical simplicity, in a programming language, enables students to tackle interesting problems in their area, at a relatively early stage without having to spend weeks learning the language first. An example of a language that is logically simple is SOLO [Eisenstadt, 1983]. Using SOLO, cognitive science students are able to tackle problems on searching data bases and simple protocol simulations within a very short space of time.

The third type, syntactic simplicity, implies that the rules for writing instructions are uniform and have well chosen names.

The names of the basic instructions are important as novices tend to make inferences about the notional machine from these names. Examples of this are the LOAD and STORE instructions, used in assembler languages, that have real world connotations [Du Boulay, O'Shea & Monk, 1981 p243.] [Kahney & Eisenstadt, 1982].

The use of surrogate models helps novices, but it is generally hard to find a model that is simple and covers all of the system that it is modelling. LOGO specifically has a lot of pseudo-English in its syntax to provide novices with a surrogate model. Boxer [di Sessa, 1982] uses a spatial metaphor to represent functions because it is easy for novices (and experts) to relate to.

ITSY will provide a "pleasant" environment using these principles. This is discussed further in section 6.1.

A problem with several of the CAI systems described above (eg. GREATERP) is the fact that they are restrictive. In order to know exactly what the student is doing they constrain the student to work in a top down fashion. For example, problem 4-3 on page 57 of "Lisp" [Winston & Horn, 1981] is:

"Problem 4-3: Define SQUASH, a function that takes an s-expression as its argument and returns a nonnested list of all atoms found in the s-expression. Here is an example

(SQUASH '(A (A (A (A B))) (((A B) B) B) B))
(A A A A B A B B B B)"

The ideal solution (given on page 323 of "Lisp") that a CAI system would have stored is:

```
(defun squash (s)
  (cond ((null s) nil)
        ((atom s) (list s))
        (t (append (squash (car s))
```

```
                                    (squash (cdr s))))))
```

Suppose the student were attempting to write the following (nearly ideal and working) solution:

```
(defun squash (s)
  (cond ((null s) nil)
        ((atom (car s)) (cons (car s) (squash (cdr s))))
        (t (append (squash (car s))
                   (squash (cdr s))))))
```

A student may arrive at the above solution as follows:

1. The student codes the first clause and believes it to be the only terminating clause necessary.

2. The student begins to code the last clause and believes it to be the only recursive case necessary.

3. While coding the last clause the student realises that if the (car s) is an atom then an error will occur.

4. The student stops coding the last clause and codes the second clause to deal with the case above.

5. The student finishes coding the final clause.

If the CAI system did not have the student's solution it would complain about a "missing terminating case" as the student started to code the recursive case. If the student started to code the second clause as step two, the system would complain that the test clause "should test s and not (car s)". If the CAI system did have the second solution it would require the student to code the recursive cases separately, whereas the student would not realise that two recursive cases were required until s/he was actually coding the final clause. The student would be unable to code the second clause until s/he had finished coding the third. By allowing the student to finish coding a function before interrupting

with advice ITSY avoids this problem.

4.3 Intelligent Program Analysers And Debuggers

When students make an error ITSY will have to classify the error, so that the right tutorial package can be chosen. ITSY will use several of the techniques used in intelligent automatic code debuggers and analysers to classify the bugs.

Automatic debuggers fall into three broad categories:

a) Debuggers that work in a limited context. Examples of this type are [Ruth, 1976], [Adam & Laurent, 1980], [Eisenstadt & Laubsch, 1980], [Johnson & Soloway, 1985] and [Hasemer, 1983].

b) Debuggers that work in a general context, but need a program specification as well, examples of these are PUDSY [Lukey, 1980] and MYCROFT [Goldstein, 1975].

c) Debuggers that work in a general context without using a program specification, an example of this being PHENARETE [Wertz, 1982].

Those debuggers that work in a limited context, have a high level description of the task that the students are attempting stored internally. Adam and Laurent [1980 p. 78, 79] say that there are two possible ways of describing the task. Either statically, using a set of assertions, or dynamically having some general encoding of the algorithm. Ruth [1976] uses a dynamic description called a program generating model, which is described below. Adam and Laurent [1980] describe the program solution dynamically using graphs; transformations of the graph are used to prove the equivalence of the student program and the correct program. These transformations are similar to but more powerful than those used by Ruth [1976]. Any irreducible mismatches between the high level description and the student's code are taken to mean that there a bug in the code.

PROUST [Johnson, 1985] is an intention based PASCAL debugger. Johnson claims that debugging requires knowledge of the *intentions* of the programmer. Currently ITSY has no knowledge of the intentions of the programmer. This is because over 80% of the

errors found in the pilot study (see chapter 5) required **no** knowledge of context in order to be fixed. There are two reasons why such a large percentage of the errors required no context information in order to be fixed. Firstly, a quarter of the errors were caused by the environment. Secondly, the subjects used invalid Lisp forms such as:

```
((car '(a b c)))
```

It is possible to give ITSY context knowledge however and this is discussed in chapter 13.

Programming knowledge in PROUST is frame based and is contained in *problem descriptions.* Problem descriptions in PROUST consist of programming goals and sets of data objects. Programming goals are the principal requirements that must be satisfied and the sets of data objects are the data manipulated by the program. Data objects can either be constant-valued or variable-valued. Goal statements consist of a name of a type of goal followed by arguments. The problem descriptions describe what the programs must do but not how they are supposed to do it, these are described by *plans.*

PLANs [Waters, 1978] and the plans used in PROUST are similar but there is a subtle difference. The plans used in PROUST are derived from a psychological theory of programming plans being developed at Yale whereas PLANs are a program representation optimised for its utility for automatic systems. The main goal of PLANs is to represent a program completely, making as much information as possible explicit.

Plans are stereotypic methods for implementing goals. Plans are compared to the students program to determine which fits best. Plans contain a *template* slot which describe the form the PASCAL code should take. Plan templates consist of PASCAL statements, subgoals and labels. This representation of is low-level and PASCAL dependent. Johnson's reason for this is [Johnson, 1985 pp. 85]:

> If concrete plan and program representations are used, then some high-level errors are harder to identify, because the syntax gets in the way. If abstract representations are used, some low-level errors are impossible to identify, because relevant evidence has been abstracted away. Given the choice, a concrete representation must be used, since PROUST must be able to identify as wide a range of bugs possible.

ITSY uses PLANs, these can be as concrete or as abstract as needed, so that both low and high level errors can be spotted.

PROUST parses a student's program into a tree. It then selects, from the problem description, one goal at a time. The values of any data objects known at this point are substituted into the goal description. PROUST then tries to match each of the goal's plans in turn with the parse tree, using the plan's template slot. This is analysis by synthesis; PROUST generates possible implementations and matches these against the student's. If PROUST is unable to match a plan with the student's code then a bug is present. PROUST tries to interpret these plan differences using *bug rules.* Each bug rule has a test part which matches against the differences if the rule applies, and an action part which explains the plan differences.

The debuggers which need a program specification use the specification as a static description of the program, to try and spot any inconsistencies between this description and programs written by students. The specification takes the form of assertions (given by the student) in propositional calculus. This, whilst bringing in additional information, can lead to problems if the students give incorrect assertions, and also requires that students learn an additional "language" (the program specification language). MYCROFT [Goldstein, 1975] also uses the output of the program (i.e. the pictures that the program drew), to gain extra information.

Debuggers like PHENARETE [Wertz, 1982] that do not work in a small domain or use any sort of specification, look for a certain class of errors. These errors are typically syntax errors, unreachable statements, endless recursion and non-terminating loops. The errors that are spotted are not deep semantic or conceptual errors; finding such errors requires knowledge about the actual task being attempted. Programs that will run, but do not give the correct output fall into this latter category.

There are two distinct stages to debugging: analyzing the code and either fixing or reporting the errors. These are discussed in turn below.

4.3.1 Analysing The Code

Most systems break up the analysis stage into several passes. The first pass involves looking for syntactic and simple semantic errors (i.e. potential run-time errors detectable syntactically such as 3 ÷ 0, or calling an undefined function), Wertz [1982]

calls these surface errors. Compilers for languages such as PASCAL, ALGOL, Lisp and C are able to detect errors of this type.

A typical debugger takes one of three actions on spotting the error.

1. The system, like a compiler, just reports the error.

2. The system interacts with the user, suggesting a simple change in the code such as substitution of a word, deletion of an argument or addition of brackets. The user usually answers yes or no, depending on whether s/he agrees with the change.

3. The system makes simple changes to the code and then notifies the user eg. PHENARETE [Wertz, 1982].

One problem with the PHENARETE philosophy is that it can give naive users the impression that the system is more intelligent than it actually is.

The next stage in analysing involves producing a canonical form of the code. This can then be compared with either a library of cliches [Brotsky, 1981; Hasemer, 1983] or a library of plans [Rich, Shrobe, Waters, Sussman & Hewitt, 1978], or a general form of the solution in the limited context debuggers. Another method of analysing the code involves trying to generate the code from the high level description i.e. analysing by synthesis [Johnson & Soloway, 1985].

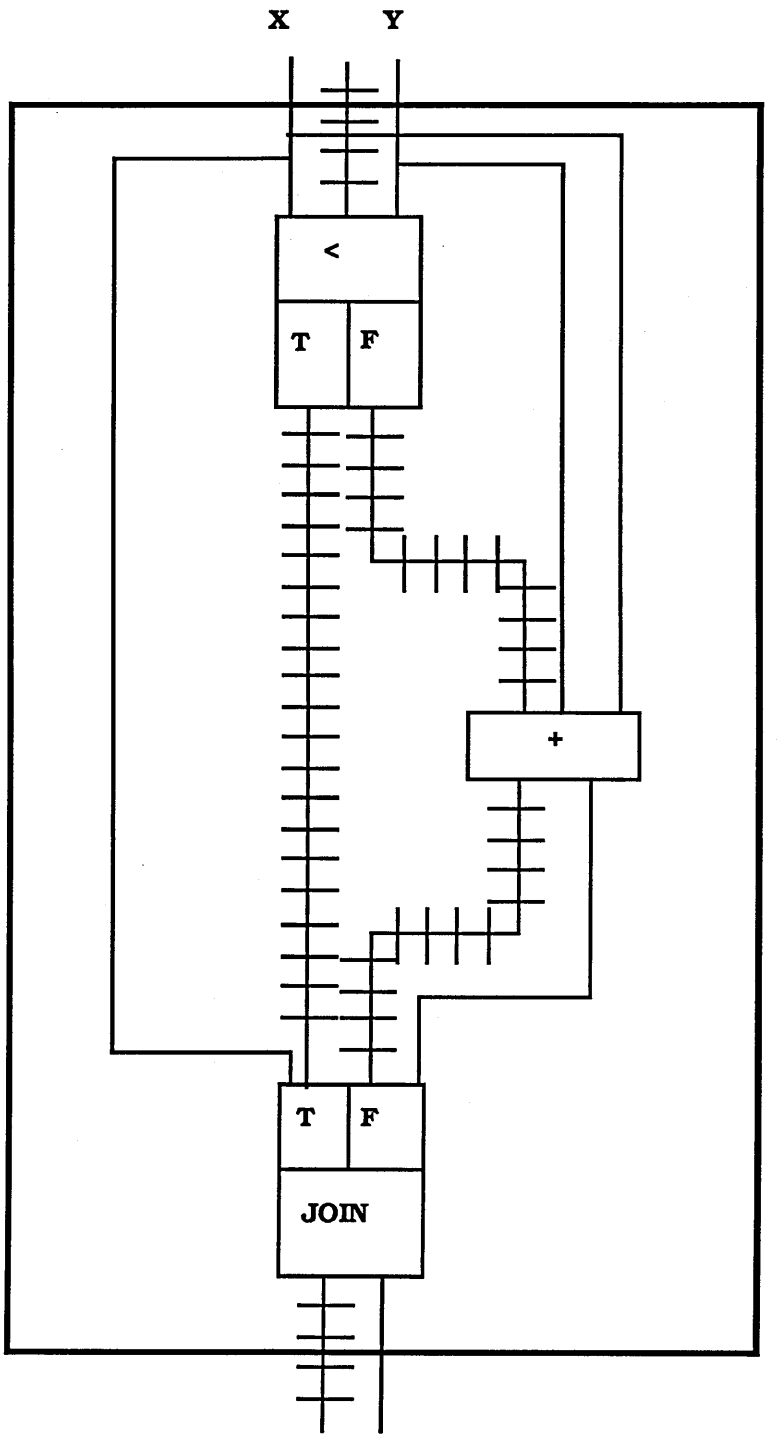
Analysing the program into plans or cliches originated at the Massachusetts Institute of Technology (M.I.T.). The canonical representation of code used was the language independent 'plan diagram' representation of the code developed by [Rich, Shrobe, Waters, Sussman & Hewitt, 1978]. This representation has been used and extended by Eisenstadt & Laubsch [1980], Lutz [1984], Rich [1981], Shapiro [1981], Waters [1979, 1982 & 1985] and Zelinka [1986].

The plan diagram represents code segments as boxes, each box giving a specification for the code segment. Control flow and data flow are represented by hashed and solid lines respectively. So for example, the LISP code:

```
(cond ((< x y) x)
      (t (+ x y)))
```

would be represented as figure 4-1.

Figure 4-1 An Example of a Plan Diagram

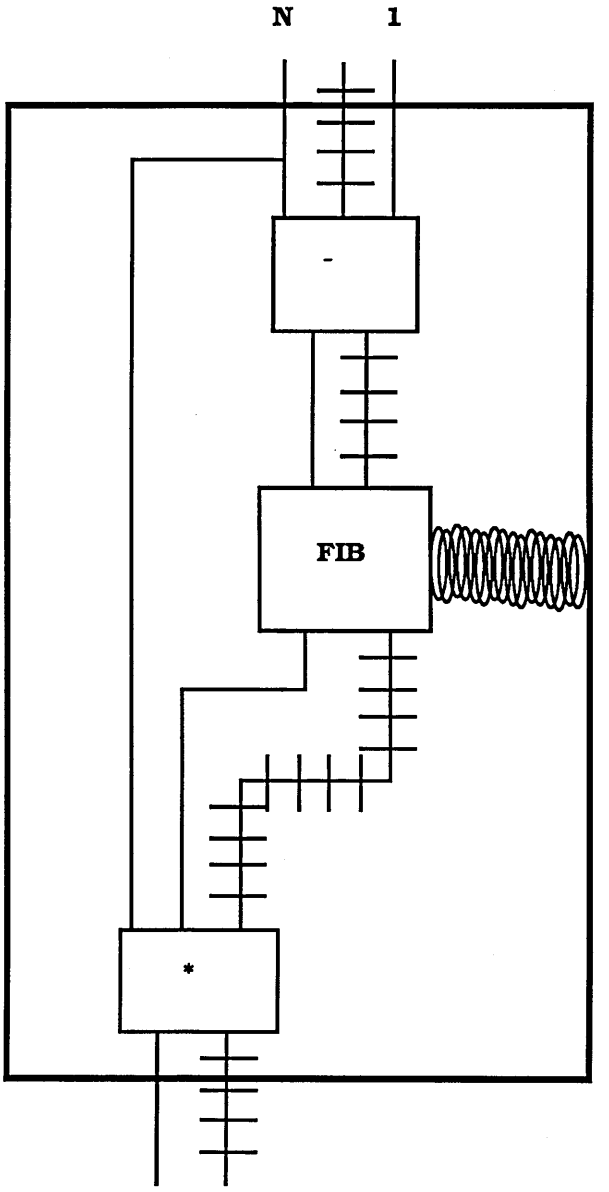


The function + is shown as a box. The two arguments, x and y, are represented by the two solid arcs connected to the top of this box. The output of the function is represented by the solid arc coming out of the bottom of the box. The predicate < is represented by the box containing the symbols <, T and F. The two possible paths for control flow, after the predicate, are represented by the two hashed lines from the T and F sections of the box. The lowest box represents a join. A join specification is a mirror image of a predicate specification. Unlike the predicate specification, however, the join does not represent any real computation. Joins are used to rejoin the two control-flow branches of a predicate block.

Recursion is represented as a looping line to the outside of the box. Figure 4-2 represents the (infinitely recursive) code:

```
(defun fib (n)
  (* n (fib (- n 1))))
```

Figure 4-2 Representation of Recursion in Plan Diagrams



Iterative loops are converted into their tail recursive counterparts, and temporal decomposition [Waters, 1979] is then applied. Temporal decomposition is a technique for abstracting iterative loops or tail recursive functions. Each operation in the loop becomes a vector operation that acts on a vector of data objects. A vector of data objects is a vector where each element contains the values of all the variables of the loop for a particular iteration (Laubsch and Eisenstadt [1982] used temporal abstraction to analyse a subset of recursive SOLO programs written by novices). The boxes in the plans can also be high level plans. This allows plans to be as abstract as need be.

The Recognizer [Zelinka, 1986] is a system that performs the program recognition by parsing. Programs are converted into a PLAN like graph representation. The library of structures to be recognised are translated into a graph grammar (currently performed by hand) and the program is parsed using the grammar. The graph parser is an extension of Brotsky's flow graph parser [Brotsky, 1984]. The extensions cope with some of the features of PLANs. Other features of PLANs, which cannot be dealt with by these extensions, have been transferred to attributes on the nodes and edges of the flow graph.

The first stage in translating programming code into plan diagram form is to translate the code into a *surface plan.* A surface plan can be thought of as an abstraction of data and control flow in a program, without abstracting data structures and operations. Once the surface plan has been constructed programming cliches from the cliche library are matched against chunks of the surface plan and more abstract plans are created. ITSY translates student programs into surface plan form before finding any bugs.

Another method of analysing code is to transform the code into a graph and then normalize the graph. Normalization transforms the graph into a standard form. An example of a transformation used might be - if a variable is used for two different purposes then a new variable is generated. This makes the matching process easier. This technique is primarily used by debuggers that store a version of the answer to the particular exercise attempted by the student eg. LAURA [Adam & Laurent, 1980].

Code may also be understood by means of meta or symbolic evaluation. PHENARETE uses meta-evaluation to analyse the students' code. The main difference between evaluation and meta-evaluation is that in meta-evaluation every possible branch of the code is taken. PHENARETE meta-evaluates the code, until every branch has either terminated, or has come to a repetition. This method is often combined with others, to work on pieces of code that another method has failed to analyse.

4.3.2 Finding The Bugs

Once the code has been transformed into a suitable form, the next step is to find the bugs. Bugs are found by looking for mismatches between a high level description of the student code, and a high level description of the correct code.

MYCROFT [Goldstein, 1975] and PUDSY [Lukey, 1980] used a specification as the high level description of the correct code, and matched the output of the students' programs against a specification. MYCROFT was able to find errors in LOGO programs which drew shapes. The specification used in debugging described the relationships between the components of the shapes drawn. The bug was then deemed to be in the section of code that constructed the part of the drawing that conflicted with the specification. PUDSY also used debugging clues such as re-assignments to variables before the old value was used. Ruth [1976] used a program generating model to try and generate the student program. The generator was a high level description of the correct program, and was only able to generate the student's code if the code was correct. If it could not do so it then tried to match on simple variations of the code. Variations might include code with the arms of conditionals swapped, or the signs in algebraic expressions switched. Laurent's system LAURA matched the normalized graph against the model answer. As in Ruth [1976] if it was not possible to obtain a perfect match then the student graph was altered until either a match was made or no more variations were left.

SNIFFER [Shapiro, 1981] uses a *cliche finder,* a *time rover* and *sniffers* to find bugs in a program's execution history. Each sniffer contains information about a particular type of bug. This information is represented by a set of rules. The program's execution history is recorded by a *time rover.* This stores all the intermediate states of variables and the effects of side effecting functions (enough information is stored so that the program could be run backwards if required). The cliche finder identifies algorithms by recognising patterns in a plan diagram representation of the code. The cliche finder acts in the same way as the code analyser in the Programmers' Apprentice [Waters, 1985].

Each sniffer uses the cliche finder and the time rover. A sniffer will use the time rover to obtain the value of a variable at different times during the evaluation. The recognition of typical algorithms by the cliche finder gives the sniffers a context for identifying errors, and raises the level at which SNIFFER can describe code.

ITSY will find bugs by trying to match sections of a plan-diagram-like description of the code against error cliches.

4.3.3 Finding and Fixing The Errors

One of the major problems when fixing a bug is that the edit may interfere with another part of the program, so causing another bug. Goldstein [1975] specified a certain order in which to fix bugs in programs that drew pictures so as to cause the least interference. Some examples of the heuristic order rules are:

1) Fix the bugs in the properties of the picture parts before bugs in relation between the picture parts,

2) Fix the bugs in the intrinsic properties of a picture part before the bugs in the extrinsic properties,

3) Use the edit that has the maximally beneficial side effects. This is because several errors can be caused by the same bug.

4) Use the edit that causes the minimum changes to the user's code.

PUDSY [Lukey, 1980] tested each possible edit to make sure that the edit did not cause another bug to appear in the program. This test consisted of comparing the amended program with the program specification.

TALUS [Murray, 1986] is a Lisp debugger able to detect and correct errors at the algorithmic, functional and implementation level. TALUS takes student program and a reference program and tries to prove them equivalent using a theorem prover. TALUS is a debugger that works in a limited context (see 4.2 a) and has eighteen task descriptions stored in a task library. Each task description has the following information:

a) *The task assignment* - instructions to the student,

b) *Algorithms* - identifiers naming acceptable algorithms for the solution of the task,

c) *Algorithm Representations* - frame representations of the above algorithms,

d) *Reference Functions* - functions that correctly implement the algorithm they are associated with.

Debugging takes place in four stages: program simplification, algorithm recognition, bug detection and bug correction.

Program Simplification

TALUS uses a theorem prover to prove various conjectures involving Lisp code. Because the theorem prover can only deal with a subset of Lisp TALUS uses a sequence of program simplification transforms to reduce students solutions. These transforms eliminate CONDs, PROGs, LAMBDAs and mapping functions.

Algorithm Recognition

The simplified code is parsed into frames. These frames are matched against the frame representations of the various algorithms stored in the task structure. A heuristic evaluation function computes how closely the frame slots match up. The algorithm with the highest score is chosen. This stage also pairs reference and student functions.

Bug Detection

The equivalence of the reference and student program forms a *conjecture.* If the conjecture cannot be proved then the student's program is considered buggy. Conjectures are first checked by a conjecture disprover. This contains a pre-stored set of counter-examples. If a conjecture passes all the examples (which are in fact sets of bindings of formal variables for each function in a stored task algorithm) then it is matched against a reference function. Functions are represented as binary trees, the nonterminal nodes representing conditional tests. The collection of terms that must be true or false for a terminal node to be reached are the terms *governing* the node. Each set of terms governing a terminal node is a *case.* Each case of the student and reference code is compared by symbolic evaluation. A theorem prover is used to check the equivalence of symbolic values. If the student and reference values cannot be proved equivalent the student's program is considered buggy, the bug occuring in the case where

the proof of equivalence breaks down. This means buggy implementations are always detected but some false alarms are generated.

Bug Correction

Before comparing student and reference code, the reference code is normalised. This means the variable and function names and the order of formal parameters are changed to those used by the student. Once the bug has been detected TALUS inserts the minimum amount of normalised reference code to restore the proof of equivalence.

Limits

Because TALUS uses a theorem prover which can only deal with a subset of Lisp, student programs are first simplified. There are some constructs which TALUS cannot simplify however:

- Free variables in function definitions

- Side effects in conditional tests

- Side effects in the actual arguments of lambda expressions.

- Destructive functions such as NCONC. TALUS replaces NCONC with APPEND when the arguments are *fresh list structures* (that is lists that have been CONSed up within the program). When the arguments are not fresh list structures TALUS has to rely on heuristics.

In the debuggers there seems to be a trade-off between generality and complexity. In order to find deep semantic and teleological errors context knowledge is needed. In areas such as programming we cannot expect novices to provide this knowledge so it must be built into the system. Builtin context knowledge limits the generality of a debugger as it will only be able to debug those programs the builtin context knowledge covers.

5. CATEGORISING ERRORS IN A TRADITIONAL LISP ENVIRONMENT

5.1 Motivation

Before undertaking the construction of an intelligent debugging environment for Lisp novices, it is critical to find out exactly what kinds of errors they make. In particular, I was interested in the errors made by professional programmers (i.e programmers who are currently employed to program in a 'conventional' language and have had at least two years experience doing so). I wanted to try to categorise the errors they made when using a fairly standard Lisp environment. The study described in this chapter therefore had two main objectives:

1. To find out exactly what problems computer programmers have learning Lisp, and so determine what hand-holding aids they need. This information helped determine the overall shape of the environment that ITSY provides.

2. To build up a bug taxonomy to be used by ITSY in recognising students' bugs. This taxonomy was built up by including any error cliches that could be found from the data produced.

5.2 Methods

Nine COBOL programmers were used as subjects. They each sat at a terminal, for two hours a week, over a ten week period, reading from Winston and Horn's book "Lisp" (first edition) and attempting the exercises. The subjects were also able to take the book home to study in private. While at the terminal the subjects were able to phone me for help. I would immediately hang up the phone, and then advise the subject via a "keyboard dialogue" conducted at the terminal. The subjects were placed, when possible, in separate offices. I was always in a separate office. This prevented any communication between myself and the subjects, other than via the terminal. These conditions simulated, as closely as possible, the conditions in which the students would be working, in an industrial environment using ITSY.

The study was carried out on a DEC-20 using MACLISP. The subjects were given information sheets, describing how to log on and how to use EMACS (see appendix A). EMACS is a screen editor that "knows" about LISP, carrying out automatic bracket balancing and automatic code indenting. The subjects were also given a short tutorial about the editor. To simplify the learning of EMACS, subjects were given an initialisation file. This gave the subjects the basic cursor moving operations on single keys; these were kept simple so as not to overload the subjects with too many new editor commands. The basic operations given were: to move up and down a line; forward and backward a character; and forward and backward a word. After a few weeks some of the more advanced subjects were introduced to the "zap" key. This key loads a single LISP function from the editor into the LISP environment. All of the subjects' interactions were recorded using the LISP "dribble" system. This sends all of the input and output at the LISP top level to a specified file (see appendix C). A list of the number of attendances and the number of lines typed at top level (a measure of the amount of work carried out), is included in section 5.4.

5.3 Method Of Analysis

The errors were classified according to the cause of the error, rather than the error message given when it arose (the symptom of the error). The error messages that MACLISP gives are not a reliable indicator of the type of error made. Consider the message ";UNDEFINED FUNCTION". There are several different causes for this "symptom"

1. (CAR (A B C)) instead of (CAR '(A B C)): missing out a quote mark.

2. (DEFUN FOO (X) (CAR (X)) instead of (DEFUN FOO (X) (CAR X)): not fully understanding what brackets mean.

3. ((APPEND '(A B C) '(D E F))) instead of (APPEND '(A B C) '(D E F)). Not knowing how to call a function.

4. (APEND '(A B C) '(D E F)) this is a misspelling.

5. (COND ((NULL X) NIL)) (T.... The subject does not understand the syntax of the cond

form.

6. Forgetting to load a function.

7. Not realising that a function has not loaded from a file because the brackets do not balance.

The dribble files were first analysed to obtain a comprehensive list of the causes of the errors, from which categories were produced. The data was re-analysed using these categories. Three policies were adhered to in counting the errors:

1. If a subject typed (LIST A B C) followed, after the corresponding error message, by (LIST 'A 'B 'C) as his/her second attempt, then this was counted as only one (quoting) error, not three. However if a subject typed (LIST A B C), then (LIST 'A B C), then (LIST 'A 'B C) and then (LIST 'A 'B 'C), this was counted as three errors. The number of errors attributed to any particular section of code depended on how the subject corrected the code.

2. Subjects sometimes corrected the wrong part of an incorrect function. They would then re-load the function, leaving the faulty part of the function intact. Each re-load was counted as a separate error.

3.Non-lists given as the second argument to either of the functions CONS and APPEND were also counted as errors. This does not produce an error, but a dotted pair[1]. Dotted pairs are not covered in Winston and Horn's book until chapter nine. The subjects should only be giving list arguments to these functions. The definitions of CONS and APPEND given by Winston and Horn are:

"CONS takes a list and inserts a new first element ...(CONS <new first element> <some list>)" (pages 24 and 25)

"APPEND strings together the elements of all lists supplied as arguments" (page 24)

[1]In LISP there are two ways of printing lists (or cons cells). One way of representing lists is by using dotted notation. In dotted notation the result of evaluating (CONS A (CONS B (CONS C NIL))) or (LIST A B C) would be (A . (B. (C . NIL))). The LISP print functions usually represents lists without the dots, so the above is written as (A B C). However there are certain lists that cannot be written in the normal way; an example of this is (A . (B . C)). When this happens the dot is printed. For example, (A . (B . C)) is written as (A B . C). A pair such as (A . B) is called a dotted pair.

5.4 Results

A great proportion of the errors that subjects made fell into a relatively small number of groups. Ninety-one percent of the total number of errors fell into nineteen categories.

The following table gives the number of sessions attended, the number of lines input, the total number of errors, and the percentage of errors compared to the number of input lines for each subject.

Subject	Number of Sessions Attended	Number of Lines Input	Number of Errors	Percentage of Errors per line
K	4	249	31	12
J	10	1184	150	13
A	7	968	187	19
B	3	227	41	18
D	9	443	56	13
E	5	284	36	13
J2	5	306	26	8
B2	11	761	73	10
P	7	410	45	11
Total	61	4832	645	13.3

The number of sessions attended only gives a rough indication of the time a subject spent in the study. This is due to two reasons. Firstly, some sessions were shorter than two hours because the subjects had to wait to log onto the computer. Secondly subjects sometimes spent longer than two hours at sessions.

The types of errors that the subjects made can be divided into three sections: the errors that were caused by the environment; algorithmic errors; and the errors caused by the language.

All of the categories containing more than 1.5% of all the errors are presented here. Some of the notable sub-categories and examples have also been given. In some cases a simple cure is proposed, in others the "cure" is to include a cliche, in the error cliche library, to match against this type of error.

5.4.1 Problems Caused By The Environment

1. Problems Caused by Written Materials

(a) Writing / instead of //. In MACLISP the / character is special, and needs the escape character (/) before it. On page 59 of "Lisp", Winston and Horn write: "Binary trees can be used to represent arithmetic expressions, as for example:

(* (+ A B) (- C (/ D E)))

One can write a compiler, or program for translating such an arithmetic expression into the machine language of some computer, using LISP ..."

The fact that the slash character is special is only mentioned in the appendix.

(b) When the subjects finished a session, I wanted them to type *(stop)*. This was a LISP function which I had written, that closed the dribble file, then exited from LISP. I explained this on the hand out sheet (see appendix A) with the sentence: "Once you have finished type "(stop)" to leave the LISP top level." Some of the subjects typed *"(stop)"*.

(c) Winston and Horn give examples of functions defined at the LISP top level. For example, page 34 of "Lisp" reads:

```
(DEFUN F-TO-C (TEMP)
   (QUOTIENT (DIFFERENCE TEMP 32) 1.8))
F-TO-C
```

because the function definition is written in capitals, it is not immediately obvious that "F-TO-C" was returned by LISP interpreter. Two subjects wrote the names of the functions after each definition in the file.

Percentage of the total number of errors: 3

Number of subjects affected: 5

Cure:

> Change the text. In case (a) the fact that the slash character is special should be mentioned. In cases (b) and (c) different fonts could be used.

2. The Computing Environment

(a) Control-s freezes the vt-100 terminal. Control-q unfreezes the terminal. There is also a "noscroll" button that toggles the freezing and unfreezing of the terminals. During the course of the study this caused all of the subjects, at one time or another, to get stuck.

(b) In MACLISP there is an autoload feature, that allows a function not present in the environment to be loaded in automatically from a file, the first time it is called. This caused problems when the function that the subject was working on was not in the environment, and the function name was the same as another in a different (autoloadable) LISP file. This happened when either the subject had forgotten to load the function, or the function contained unbalanced brackets and was not loaded. The subjects did not realise that the function had been automatically loaded, and thought that their (usually incorrect) attempts were correct.

(c) Four of the subjects tried to use control-h and backspace as a rubout key. This is what they use in their habitual working environment.

Percentage of the total number of errors: 9

Number of subjects affected: 9

Cure:

> Provide a protective environment for the students, shielding them from the "harmful" aspects of the LISP environment. The environment that ITSY provides has no autoload feature. Keys that students do not need have been disabled.

5.4.2 Algorithmic Errors

These errors are only detectable if the problem that the student is attempting is known. A "cure" is only possible if a version of the solution is stored.

1. Using the wrong combination of CARs and CDRs to pick out an element of a list.

Percentage of the total number of errors: 6

Number of subjects affected: 7

2. Using the wrong function. About seventy percent of the errors in this category were due to the subjects picking the wrong function out of CONS, LIST and APPEND.

Percentage of the total number of errors: 3

Number of subjects affected: 5

3. Errors in recursion. There were two main types of recursive errors found in the study:

(a) Not altering the "terminating argument" in the recursive call. By "terminating argument", I mean the argument that is tested in the exit part of the recursive function. For example, one of the subjects wrote the following function:

```
(defun s2 (l)
  (cond ((null l) nil)
        ((atom l) l)
        (t (s2 (cons (car l) (cdr l))))))
```

the argument l (the "terminating argument") is passed unaltered in the recursive call.

(b) Missing out the recursive call. Two of the subjects wrote functions that missed out the recursive call. One of the functions that a subject wrote was:

```
(defun uparam (i list1 list2)
  (cond ((equal i (length list1)) list2)
        ((not (member (car list1) list2))
         (set 'list2 (cons (car list1) list2)))
        (t (uparam (add1 i) (rotate-l list1) list2))))
```

The function should rotate the list list1, adding the head of the list to list2, if it is not already a member. The second clause should be

```
((not (member (car list1) list2))
 (uparam (add1 i) (rotate-l list1) (cons (car list1) list2)))
```

The subject asked for help and said that he could not understand why the function "stops after inserting the first non-matching atom".

Percentage of the total number of errors: 3

Number of subjects affected: 4

4. Not realising that solution is incorrect. There are two main reasons for this.

(a) The subjects did not always try their solutions on the right input. For example, one subject wrote:

```
(defun mobile-p (m)
  (cond ((atom m) m)
        ((not (equal (mobile (caddr m))
         (mobile (cadr m)))) nil)
        (t (plus (car m) (mobile (cadr m))
           (mobile (caddr m))))))
```

as a solution to problem 4-10 in "Lisp". This function should return a value if the input is a list of the right form, otherwise it should return NIL. The function worked with all of the inputs that the subject tried. However if both (mobile (caddr m)) and (mobile (cadr m)) are NIL, then the t clause is executed, and the function plus receives two non-numeric arguments, giving an error.

(b) The subjects had missed the generalisation of an example. One subject defined the following function to rotate a list right:

```
(defun rotate-r (exp-1)
 (append (cdr (cdr (exp-1)) (list (car exp-1) (car (cdr exp-1))))))
```

this function only works on lists of three elements. The subject probably did this because the two examples given for the previous exercise, rotate left, are:

```
(ROTATE-L '(A B C))
  (B C A)

(ROTATE-L (ROTATE-L '(A B C)))
  (C A B)
```

Percentage of the total number of errors: 3

Number of subjects affected: 2

5. Other algorithmic errors. One of the simpler algorithmic errors found in the study was:

```
(defun merge (x y)
  (cond ((null x) y)
        ((lessp (car x) (car y)) (merge (cdr x) y))
        (t (cons (car x) (merge (cdr x) y)))))
```

the function should merge two sorted lists of numbers into one ascending list. The code has several algorithmic errors. One algorithmic error in code is there is no clause to deal with the case when the heads of both lists are equal. A correct version of the code is:

```
(defun merge (x y)
  (cond ((null x) y)
        ((lessp (car x) (car y))
         (cons (car x) (merge (cdr x) y)))
        ((equal (car x) (car y))
         (cons (car x) (merge (cdr x) (cdr y))))
        (t (cons (car y) (merge (cdr y) x)))))
```

Percentage of the total number of errors: 4

Number of subjects affected: 4

5.4.3 Problems With The Language

1. Simple errors or slip ups. The three main sub-categories are:

(a) Spelling errors.

(b) Simple bracketing errors. Only the "obviously" simple bracketing errors were counted as such. If there was any doubt then the mistake was taken to be conceptual. For example, the following code, written by one of the subjects:

```
(cond ((and (null lst1) (null lst2)) .....
      ((and (null lst1) (not(null lst2))) ....
      ((and (not (null lst1) (null lst2))) ....
.......
```

(There should be an extra closing bracket after the (null lst1) in the third clause.) counted as a simple bracketing error.

(c) Forgetting to load a function from a file.

Percentage of the total number of errors: 12

Number of subjects affected: 9

Cure:

> In order to cure (b) ITSY has an entry in the error cliche library. Cures to (a) and (c) are currently not implemented. (a) could be cured with a spelling checker. (c) could be cured by checking the last file edited for the missing function.

2. Incorrectly putting a pair of brackets around an atom. Sometimes it is correct to put brackets around atom, as in (defun foo (x) ... and in (cond (a) (t nil ...

Percentage of the total number of errors: 3

Number of subjects affected: 6

Cure:

> There is an entry in the cliche library to match against this type of error.

3. Stuck at top-level because there are not enough closing brackets. MACLISP on the DEC-20 responds once an s-expression followed by <RETURN> has been typed in. If the user types <RETURN> before closing all the brackets, the interpreter assumes that the rest of the expression is to continue on the next line. Eight of the subjects thought that they

had typed enough closing brackets, and when they hit <RETURN> they thought that the machine was stuck. Six of the subjects tried to re-enter the s-expression, on the next line, and three of the subjects had to dial for help.

Percentage of the total number of errors: 2

Number of subjects affected: 8

Cure:

The top level of ITSY includes the following features:

(a) A bracket balancing feature.

(b) The evaluation of forms as soon as the last closing bracket is typed, instead of waiting for the return key to be pressed.

(c) The use of a prompt

4. Incorrectly putting an extra set of brackets around a function call. This includes attempts to "listify" objects, for example writing ((foo x)) instead of (list (foo x)). Sometimes it is correct to put two pairs of brackets around a function call, as in (cond ((null l)) ...

Percentage of the total number of errors: 3

Number of subjects affected: 2

Cure:

Included in the cliche library

5. Not putting brackets around a function call.

Percentage of the total number of errors: 5

Number of subjects affected: 6

Cure:

Included in the cliche library

Errors in category 4 and 5 do not include errors that occurred in a COND form, these have been separated out and are given in category 9.

6. Wrong number of arguments given to a function. For example:

```
(add1 3 4)
```

Percentage of the total number of errors: 2

Number of subjects affected: 5

Cure:

Included in the cliche library

7. Wrong number of arguments given to a function, because the arguments are in the wrong form. This category has been separated from the one above because in eighty percent of cases, the arguments were present but in the wrong form. Typical examples of this type of error are:

(f-to-c '(10)) needs one numeric argument

(roots '(2 4 8)) needs three numeric arguments

(CONS 'a 'b c d e) needs an element and a list.

Percentage of the total number of errors: 6

Number of subjects affected: 6

Cure:

Included in the cliche library

8. Arguments of the wrong type given to a function. Non-list arguments given to CONS and APPEND have been counted as this type of error, as dotted pairs are not introduced in "Lisp" until chapter nine. Ninety-five percent of the errors in this category involved non-list arguments given to one of CONS, APPEND, CAR and CDR.

Percentage of the total number of errors: 9

Number of subjects affected: 7

Cure:

Included in the cliche library

9. Errors in a COND form. All errors that occurred in a COND form have been separated out. Ninety percent of these errors were due to:

(a) Not putting a pair of brackets around a function call,

(b) Putting an extra pair of brackets around a function call,

(c) Putting a pair of brackets around an atom.

Subjects made errors because of the different shapes that a COND form produces. For example, if the first (test) clause in a COND form is a function then the clause list begins with two left brackets. The subjects' experience of double brackets at this stage of their study suggests that they cause errors, and they may therefore leave out one of them.

Percentage of the total number of errors: 7

Number of subjects affected: 5

Cure:

Included in the cliche library

10. Quoting an object that shouldn't be quoted.

Percentage of the total number of errors: 2

Number of subjects affected: 3

Cure:

Included in the cliche library

11. Not quoting an object that should be quoted.

Percentage of the total number of errors: 7

Number of subjects affected: 9

Cure:

Included in the cliche library

12. Function not loaded because not enough closing brackets in the file.

Percentage of the total number of errors: 4

Number of subjects affected: 2

Cure:

ITSY warns students if they try to save a file with unbalanced parentheses.

5.5 Error Messages

The error messages caused problems for the subjects. Whether or not the messages were understood depended on the context of the error. Most of the subjects had no problem understanding error messages concerned with top level errors, but could not understand the same message when the error occurred inside a function. Error messages concerned with the loading of files gave the most problems. The messages were so incomprehensible that none of the subjects even realised that an error had occurred. Below are some examples of error messages and typical subjects' reactions to them.

1. "function received 1 arg wanted 3" This message was by far the easiest to understand causing few problems.

2. "a undefined function object" The subjects' abilities to understand this message depended on how the "a" had been derived. If the subject had typed at top level `(first (a b c))`, and received this message, then usually the subject would correct the error immediately. If the bug was embedded in a function, such as this first attempt at the function first:

```
(defun first (a)
  (car (a)))
```

then there was less chance that the subject would understand the message. If the "a" was the result of some evaluation as in:

```
(defun (list1)
    .....(cdr ((reverse list1)) (car ((...)))
```

(list1 is bound to (c b a)) then there was little chance, that the subject would know what the message meant.

3. Some of the messages such as:

(read-eof #file-in |RS<R.Lispclass.2>|)

and

pdl overflow - red pdl space exceeded

were not understood at all, by any of the subjects. The first message means that the file has not been loaded because there is a bracket missing (an 'end of file' character was read in the middle of an s-expression). The second message means that the stack has overflowed.

The first error message led to numerous problems. None of the subjects realised that an error had occurred. All errors that were reported after this were attributed to the code that had just been written rather than to the (still current) previous version.

The subjects were reluctant to use even the simplest environmental facilities available. One of the subjects never ever used the editor when writing definitions. Only four subjects used the facility which enables the user to examine function definitions in the environment. None of the subjects used the editor's automatic indenter.

5.6 Conclusions From The Study

Some of the errors produced by the subjects were due to the book used. That does not mean that there any great deficiencies in the book, just that some of the errors would not have been made, if another book had been used. The best example of this was the bewilderment of some the subjects when they encountered error messages about dotted pairs. If another book had been used the subjects might have known about dotted pairs before encountering the messages. In "Lisp: A Gentle Introduction to Symbolic Computation" [Touretzky, 1984], the internal structure of lists and dotted pairs are covered before any on-line exercises are given. To what extent the error taxonomy is tied to the book used by the subjects can only be found out by repeating the study with another book e.g. [Hasemer, 1984], [Touretzky, 1984].

All of the subjects were "clobbered" by the computing environment, at one time or another. This accounted for nearly one tenth of all the errors. Any system on which users have to learn should always give the student a "way out". It should always be obvious how to get back to the top level of the system. The student should always know what state the system is in, for example if it is waiting for input, if the stepper is on, if it is in a debugging mode etc. Students should be shielded from parts of the language and the various tools available until they are ready for them. ITSY will have to protect students from the environment if they are to use the system unaided.

Nearly one fifth of the errors were algorithmic in nature. To match against every instance of this type of error, the exercise being attempted needs to be known. Currently ITSY has no knowledge of the exercise being attempted. The addition of context knowledge to ITSY is discussed in chapter 13.

For some of the error categories, the code producing the error follows a general "shape". The error cliches are derived directly from these general "shapes". This is better shown with a couple of examples:

1. One of the error cliches that has been derived from Lisp error 4; putting too many brackets around a function call:

((<function> <any>))

This will match against any embedded list, where <function> is any defined function.

2. One of the error cliches that has been derived from Lisp error 7; wrong number of arguments given to a function because the arguments are in the wrong form:

(<function> <argument list>)

<function> is any defined function and <argument list> is (incorrectly) a list where each element is one of the arguments to <function>. This error cliche would match against code such as:

```
(+ '(1 2 3))
```

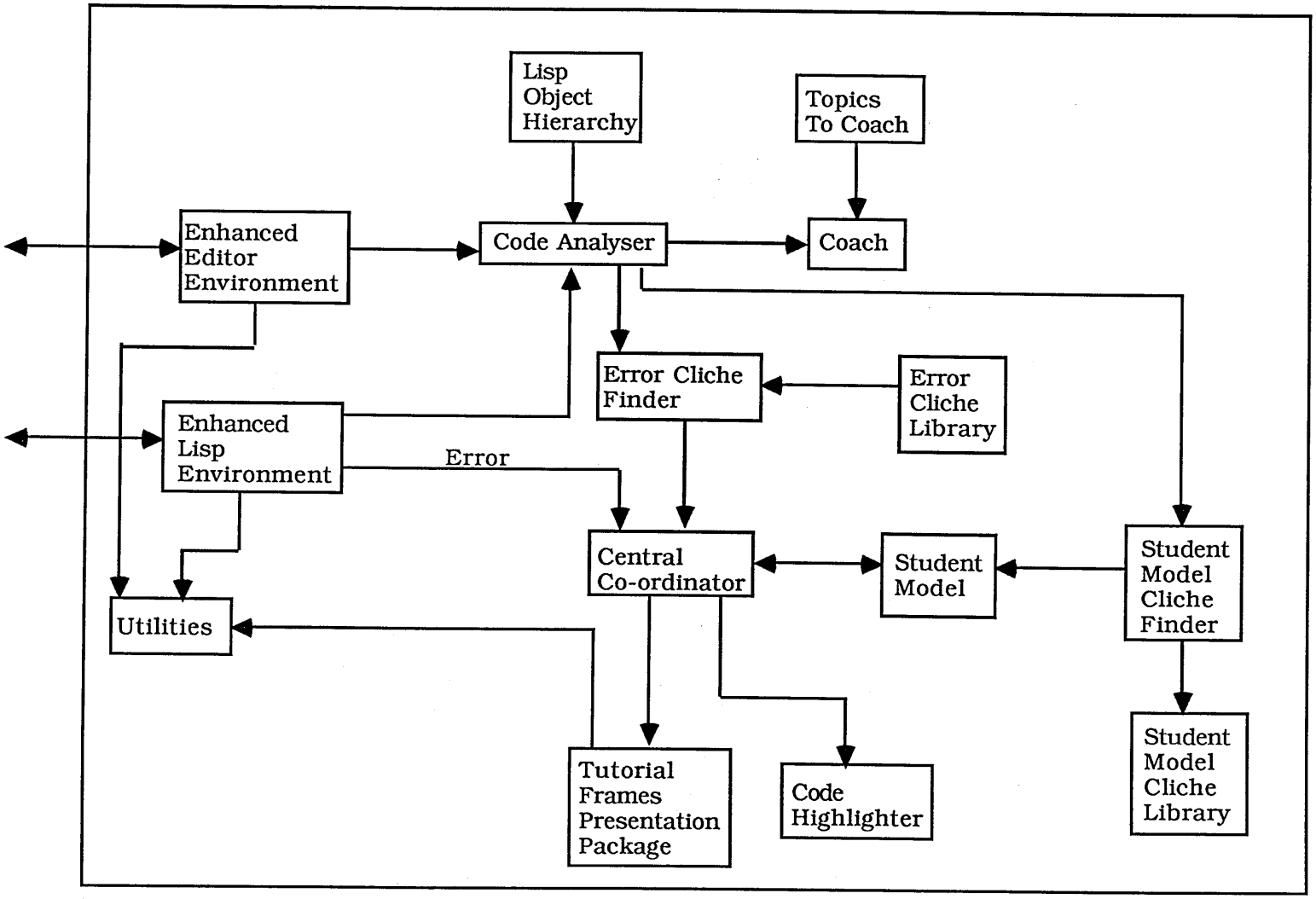
The results of this study were used in the design of ITSY. Implementation details are given in the next part of the thesis and the results of an evaluation study are given in the third part of the thesis.

PART II

This section describes, in detail, how ITSY has been implemented. Figure 2-1 from chapter 2 is reproduced below. The arrows in figure 2-1 show dataflow.

The enhanced Lisp and Editor environments and the Coach are described in chapter 6. The Code Analyser and the Lisp Object Hierarchy are described in chapter 7. The Error Cliche Finder and the Error Cliches are described in chapter 8. The Code Highlighter and the Tutorial Frames Presentation Package are described in chapter 9. The Student Model is described in chapter 10.

Figure 2-1 Overview of ITSY's Architecture



6. THE ENVIRONMENT

In this chapter we discuss the implementation details of the overall environment. ITSY's environment was designed using two basic guidelines:

a) To fulfil some of the principles outlined by O'Shea [Du Boulay, O'Shea & Monk, 1981]. That is to make the tools functionally, logically and syntactically simple.

b) To prevent the errors caused by the environment in the pilot study (chapter five).

6.1 Overall Environment

The overall environment has three different parts or *panes*. The three different panes are:

a) The Editor pane. This contains a modified Emacs [Stallman, 1981] style editor.

b) The Lisp pane. This contains a modified Lisp toplevel window.

c) The status pane. This gives the student information about the current state of ITSY.

The number and size of the panes depends on which *configuration* the environment is in. There are three different configurations.

a) starting configuration - in this configuration the environment has two panes. A Lisp pane and a status pane.

b) large editor configuration - in this configuration the environment has three panes. The editor pane occupies the top 75% of the screen, the status pane occupies the bottom line and the lisp pane occupies the rest of the screen.

c) medium editor configuration - this configuration also has three panes. The editor pane occupies 50% of the screen, the status pane occupies the bottom line, and the Lisp pane

occupies the rest of the screen (see figure 6-1).

It is possible to move between the lisp and editor panes by clicking on them using the left mouse button. Using the middle or right button brings up a menu. There are separate menus for the editor and lisp toplevel.

The following operations have been installed on both the editor and lisp menus.

a) Loading files

b) Moving to the large editor configuration

c) Moving to medium editor configuration

The overall environment changes as the student progresses. This is controlled by the coaching module (see 6.5).

6.2 Lisp Environment

The Lisp toplevel environment has been enhanced - this includes the addition of an Emacs [Stallman, 1981] style bracket balancer.

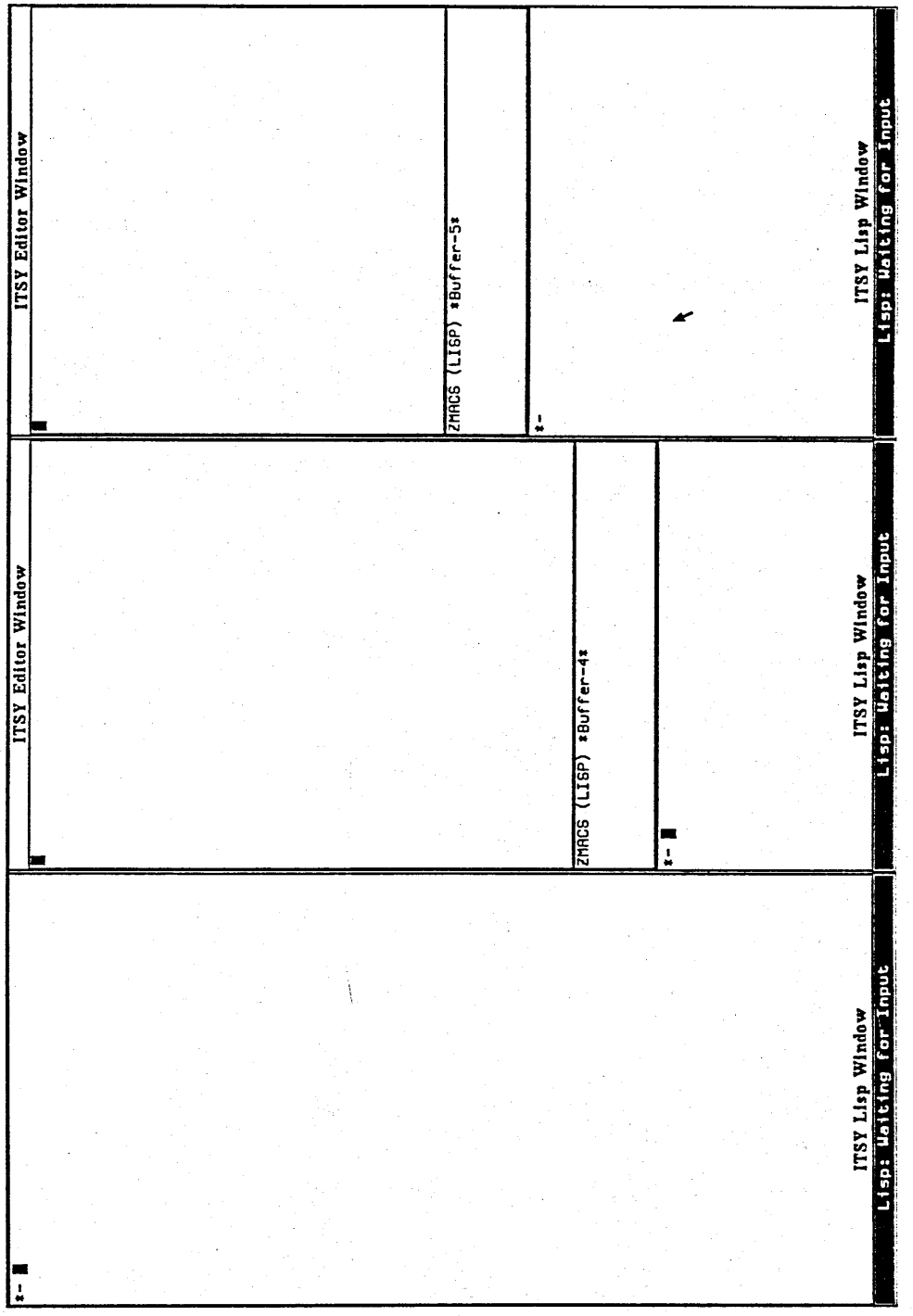
The following operations have been installed on the Lisp menu only:

a) Selecting the Editor Pane - this provides an alternative method of selecting the editor pane.

b) Adding Some Comments - sometimes the students require a short piece of advice, or I want to note something that has happened. If this is selected everything that is typed is recorded verbatim, until "end" is typed.

c) Function Documentation - the student can obtain documentation on Common Lisp functions. The Symbolics has an on-line document examination facility. ITSY uses this facility to present function documentation to the student. Currently not all Common Lisp

Figure 6-1


ITSY Editor Window
ZMACS (LISP) *Buffer-5*
ITSY Lisp Window
Lisp: Waiting for Input
ITSY Editor Window
ZMACS (LISP) *Buffer-4*
ITSY Lisp Window
Lisp: Waiting for Input
ITSY Lisp Window
Lisp: Waiting for Input

functions are documented.

d) The Prompt - the prompt is provided "for free" by the symbolics environment.

6.3 Editor Environment

Two distinct changes have been made to the editor.

The majority of Emacs commands are carried out by using control and meta keys. This led to subjects getting stuck when they accidentally hit one of these keys. Any keys that the students do not need have been disabled. The return key has been set to indent the next line as this emphasises the structure of lisp to the student.

The essential non-movement commands have been installed on the editor menu. These are:

a) Selecting a buffer

b) Saving a buffer

c) Loading any changed functions from a buffer.

There is also a menu choice to select the lisp pane.

6.4 The Status Line

The status line performs two distinct jobs. Firstly, it indicates which state ITSY is in. Secondly, it gives the student information about functions applied at toplevel.

The different states that are indicated are:

a) Waiting for input.

b) Evaluating current s-expression.

c) Currently inside the editor.

d) Loading a file into the lisp environment

e) Trying to find the source of an error

f) Found the source of an error

g) Trying to find the code in the buffer to highlight

h) Setting up the tutorial

Each time the student types the name of a function, the names of the parameters (given in the parameter list) of the function are displayed in the status pane. This tells the student two things. Firstly it says how many arguments the function takes. Secondly, it gives the student an indication of how each argument will be used and what type it should be. If no information appears the student knows that the function is undefined.

6.5 Coaching

ITSY has a coaching facility. This is used in a limited way at present. Coaching is carried out using *coaching events.* A coaching event has four parts:

a) a name,

b) the name of a *coaching trigger*,

c) when the trigger is to lead to coaching,

d) the name of a function that will carry out the coaching.

At present a *coaching trigger* can be the name of either a plan diagram segment or an error cliche. Whenever a coach trigger occurs a *coach demon* checks when this trigger should lead to coaching. At present, coaching can take place either *every* time or the

first time a trigger occurs.

Currently the only coaching event is the editor event. The trigger is the creation of a plan diagram for a user defined function. This trigger leads to coaching the first time it occurs. The coaching function first changes the environment from the starting configuration to the large editor configuration, then presents a short tutorial on the editor.

The first time a student defines a function the environment changes from having just the Lisp toplevel and status line to incorporating an editor.

7. TRANSFORMING THE CODE INTO PLAN DIAGRAM FORM

In this chapter we discuss how ITSY transforms the student's code into ITSY's surface plan form. At the instant when this transformation takes place, the student has written a piece of code, usually intended to test a user-defined function, directly into Lisp toplevel. Whilst evaluating this code the interpreter signals an error, and ITSY attempts to discover the cause of this error. In order to do so, ITSY transforms the student's code into surface plans (here implemented as *objects*) and compares the plans with its library of error cliches.

7.1 The End Product - Internal Representation of the Code

In the Programmer's Apprentice [Waters, 1985] code is first translated into a *surface plan.* A surface plan can be thought of as an abstraction of data and control flow in a program, without abstracting data structures and operations. Once the surface plan has been constructed programming cliches from the cliche library are matched against chunks of the surface plan to form more abstract plans. ITSY translates student programs into a surface plan form then tries to match Error Cliches from the Cliche library against segments of the surface plan.

Rich [1981, p. 65] describes the translation process as follows:

> In order to translate between a given programming language and surface plans, the primitives of the programming language are divided into two categories: connectives such as PROG, COND, SETQ, GO and RETURN in Lisp, which are concerned solely with implementing data and control flow; and the objects, relations and actions of the language, such as numbers, dotted pairs, arithmetic relations, CAR, CDR and CONS.

The differences between surface plan representation used in the Programmers' Apprentice and the internal representation used in ITSY are quite minor. In surface plan representation segments are represented in terms of lists. The control and data flow links are represented explicitly stating which segments are in the connected. In ITSY's internal representation segments are represented as objects. The control and data flow links are represented implicitly by setting a control or dataflow slot of an object to another object. In surface plan representation segments have a type. This type determines how the segment interacts with control flow. Splits in the control flow are

achieved by setting the type of a segment to 'split'. In ITSY's internal representation there is only one type of object that can split the data flow - the *pred.* Pred objects have a test slot. The test slot holds the predicate that will split the control flow.

7.1.1 Advantages of Using Plan Diagram Representation

The main reason for using a plan diagram form as the internal representation is the dataflow abstraction achieved. Abstracting the raw Lisp code simplifies the error cliche matcher. One of the simplifications achieved is that the error cliche matcher does not have to worry about scoping issues. In the following function:

```
(defun example ()
  (let ((a 1) (b 2))
    (let ((a '(1 2 3)) (b '(4 5 6)))
      (append a b))
    (append a b)))
```

The error cliche *Wrong Type Argument Given to a Function Call* should match against the second (APPEND A B) form and not the first. If the error cliche matcher used raw Lisp code then it would have to take the scope of LETs into account. Because the plan diagram representation abstracts dataflow the error cliche matcher does not have to worry about these type of scoping issues.

One of the problems when looking for bugs in novice Lisp programs is the level of abstraction to use. If the level of abstraction is too high then low level errors will not be caught because abstracting the code removes low level features. If the level of abstraction is too low then the debugger will have problems if the students' solutions are too varied. [Johnson, 1985] suggests the use of multiple systems as a cure for this. The power of PLAN formalism is the fact that PLANs can be as abstract or as concrete as needed. For example, consider the following code:

```
(prog ((x input-list) (count 0))
 lp
 (cond ((null x) (return count))
       (t (setq x (cons (car x) (cdr x))) (go 'lp))))
```

There are two errors in the above code, one low level the other high level. The low level error is that LP the argument to GO should not be quoted. The high level error is that the list X will never be NIL and the loop will not terminate. Once the code has been analysed into surface plan form the relatively low level error is easily found. Temporal abstraction [Waters, 1978] could then be applied. This abstracts out various parts of looping constructs. Once temporal abstraction has been carried out the high level looping error is easily detected. Note that once temporal abstraction has been carried out it would not be possible to detect the quoting error.

7.1.2 Representation of Lisp Objects

Different types of Lisp objects are represented in ITSY by different classes. The classes form a hierarchy: at the top of the tree there is the most general lisp object; at the bottom there are objects such as individual functions and constants. Figure 7.1 shows part of the complete tree of classes. The lightly filled in boxes show the parts where the analyser is partially implemented. The heavily filled in boxes show the parts where only the class exists ie. the analyser is not implemented.

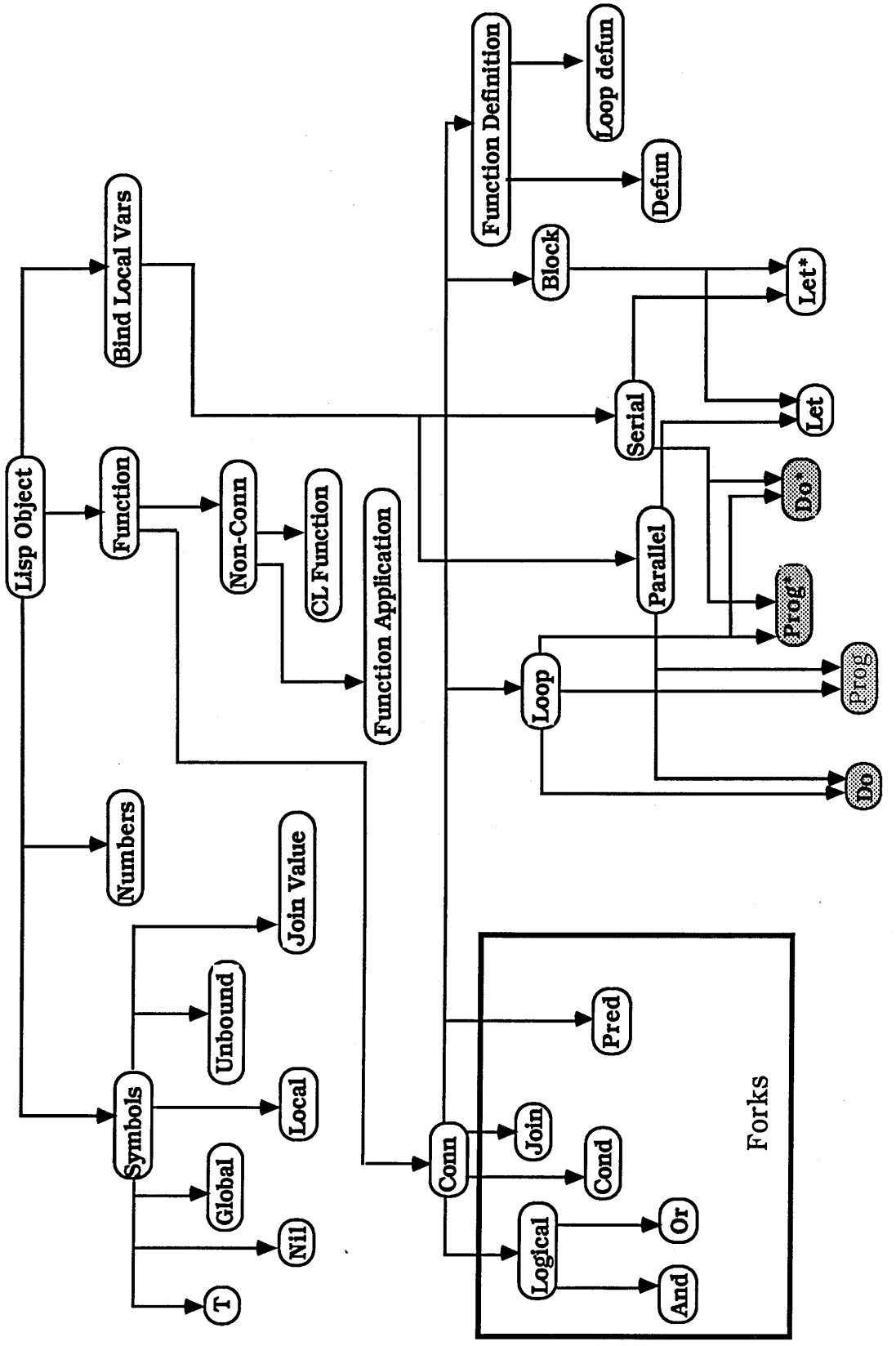


Figure 7-1a Lisp Object Hierarchy

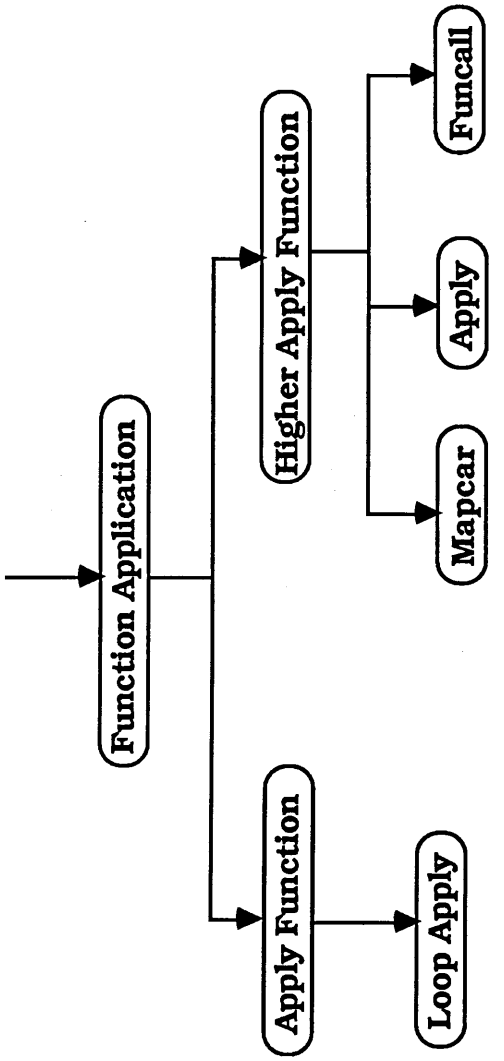


Figure 7-1b Lisp Object Hierarchy

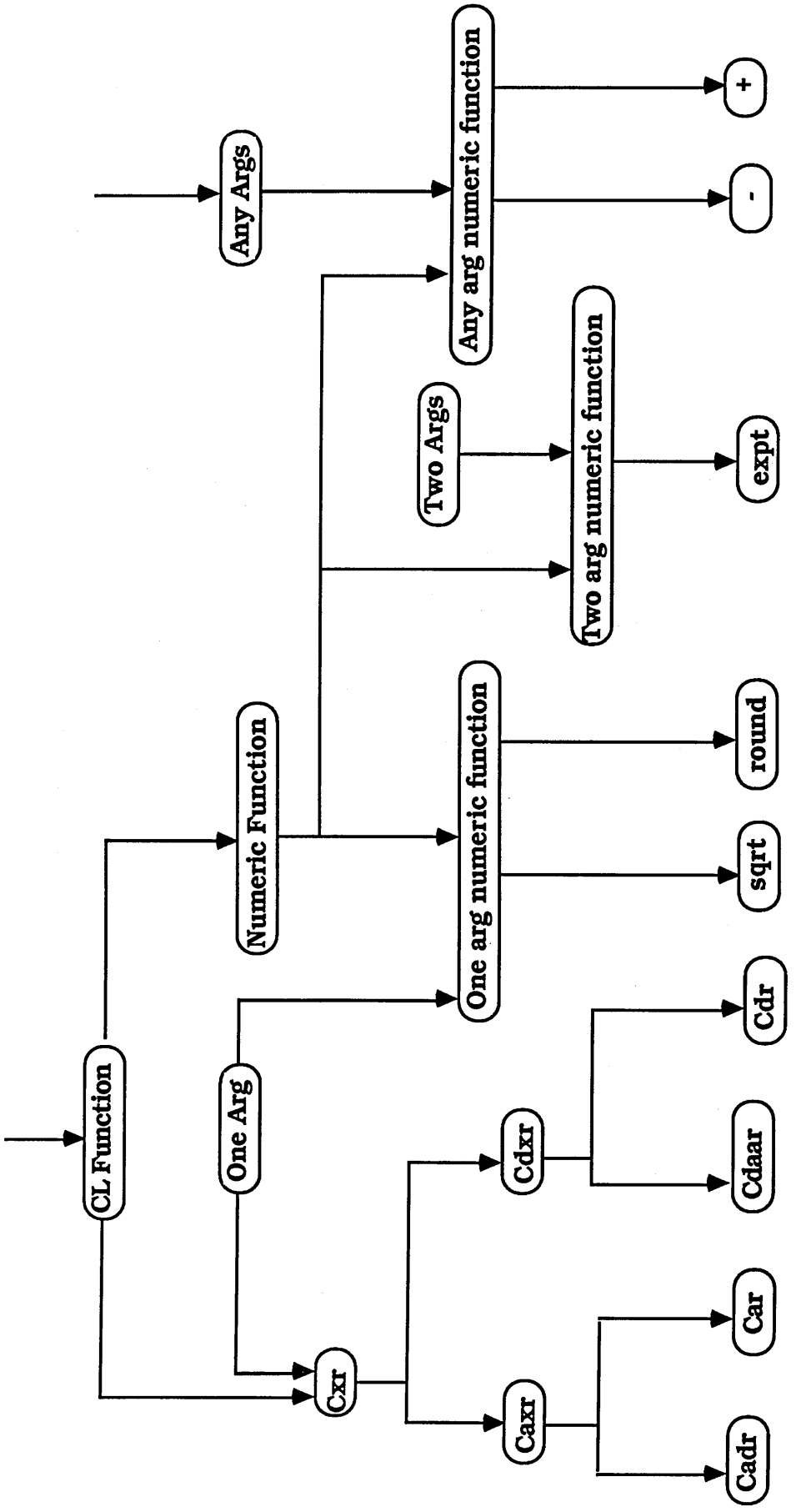


Figure 7-1c Lisp Object Hierarchy

Each object contains the following slots:

1. Name - holds the name of the object

2. Control-in - this is the controlflow input port.

3. Control-out - this is the controlflow output port.

4. Input - this is the dataflow input port.

5. Output - this is the dataflow input port.

6. Expectations - points to an entry in a hash table. This entry contains a prototypical object. This holds information such as the expected number and type of arguments and the type of output.

7. Code - this slot contains the code represented by the object. This is needed so that ITSY can give the tutorial in terms of the student's own code.

8. Code function name - the function, if any, the code occurs in. This is used by ITSY's highlighting module.

Just below the top of the tree are *symbols, numbers, functions,* and *bind local vars.* The first three denote the obvious types of objects. Bind local vars refers to objects that represent any Lisp function able to locally bind variables, such as LET or DO. Below the functions are the *connectives* (Conn in figure 7-1a) and *non-connectives* (Non-Conn in figure 7-1a). The connectives of a language are the primitives that implement control and dataflow (they are called connectives because they *connect* statements of the language together). In Lisp functions such as SETQ, COND and DO are connectives.

7.1.3 Data and Control Flow: Connectives

There are no variables as such in plan diagrams. Variables are replaced by a pointer to their value. SETQs change the position of the pointer. This removes variations due to

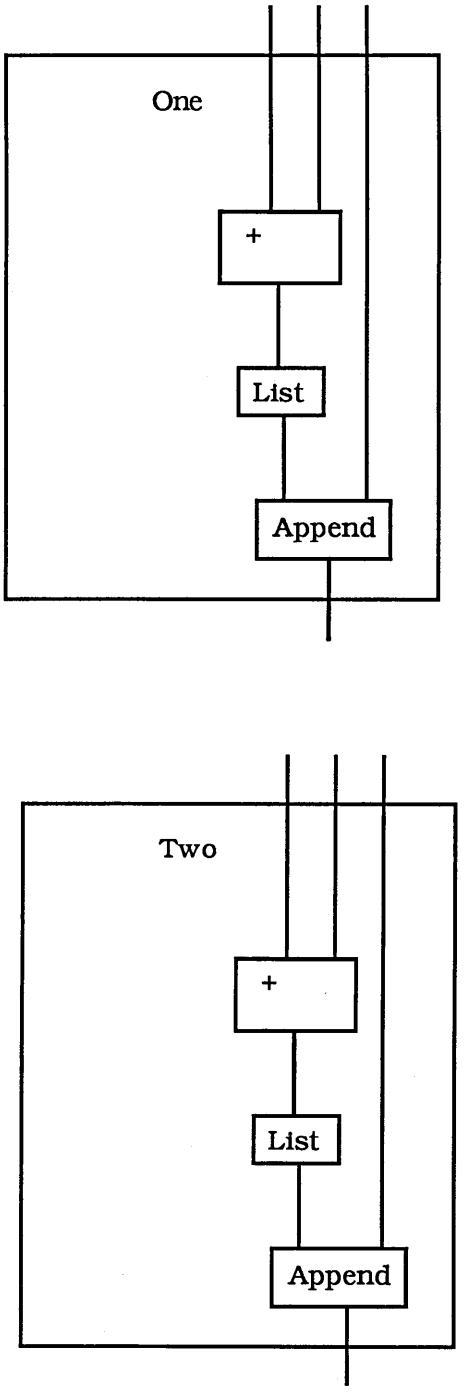
using variables to store temporary values. The two segments of code:

```
(defun one (x y z)
  (append (list (+ x y)) z))

(defun two (x y z)
  (setq x (list (+ x y)))
  (setq z (append x z))
  z)
```

would be represented as figure 7-2. Only the last s-expression in a function is represented in the surface plan representation. The other s-expressions in a function exist only for side effect purposes. The representation of function TWO is in fact the representation of the variable Z. The two previous s-expressions in TWO are there to side effect the values of the variables X and Z. Notice that the effect of SETQs (controlling dataflow) is abstracted away.

Figure 7-2



Below the connectives in figure 7-1a are *forks, function definitions, blocks* and *loops.*

Forks

Forks include the constructs concerned with branching control flow, such as COND and OR. These are changed into *pred* and *join* objects. Pred objects have three parts: a *test*, a *true output port* and a *false output port.* The test holds the predicate of the fork. Note that the value of a test slot can be *any* object, even if it represents a non-predicate function. In the representation of the following code:

```
(or (> x y) (print "X is greater") x)
```

two pred objects would be created. The first pred object's test slot would be connected to a > object and the second pred object's test slot would be connected to a PRINT object. The *true output port* holds the control flow path followed if the *test* is true. Similarly, the *false output port* holds the control flow path followed if the *test* is false. So, for example, the following code:

```
(cond ((> x y) (+ x y))
      ((= x y) (+ x 1)))
```

would be represented as figure 7-3.

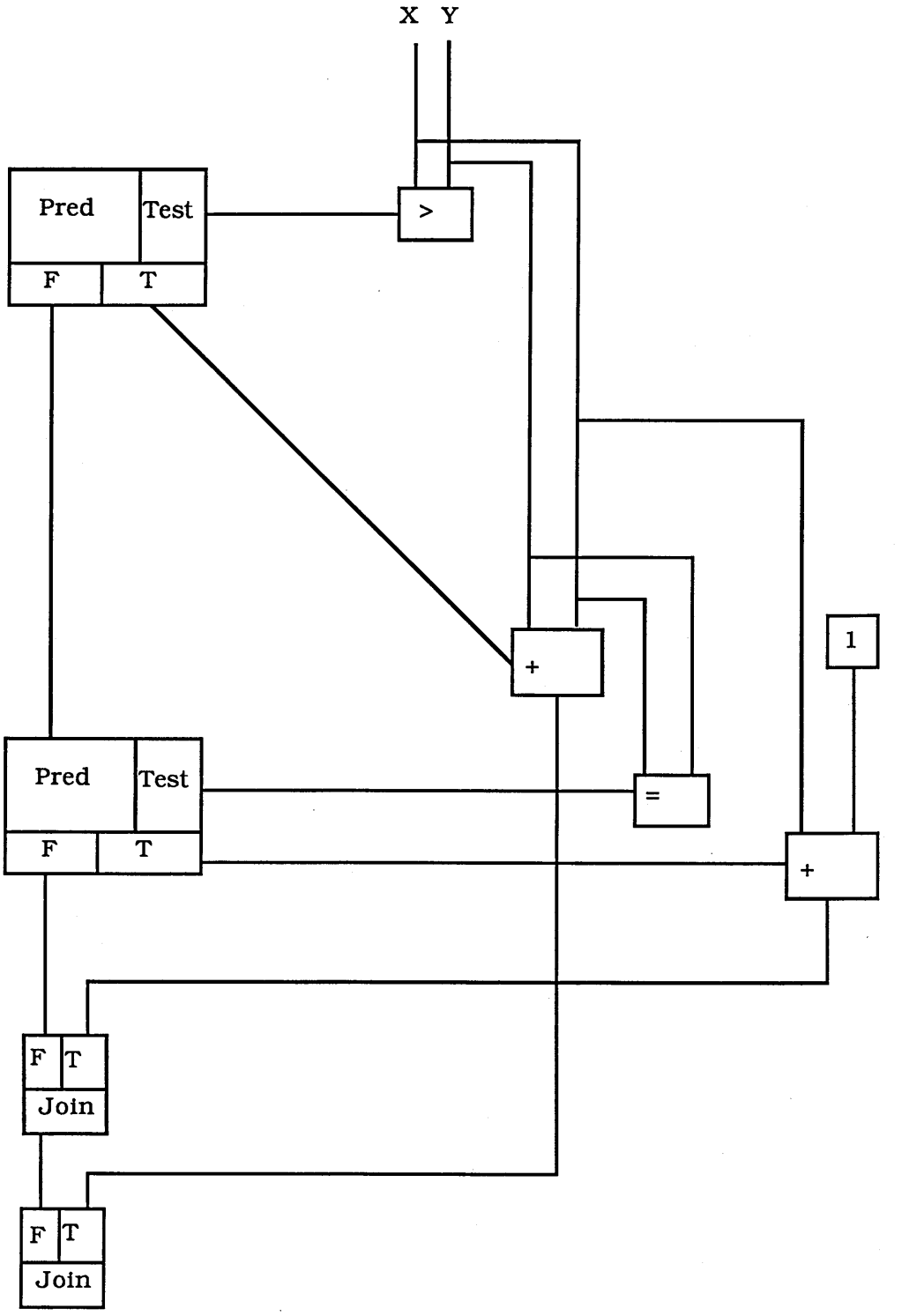
Notice there is no COND object in this diagram. Each test part of a clause has been replaced by a pred object, the test part connected an object representing the test function. The test for the first clause, (> X Y) is represented by the first pred object and the > object connected to the pred's test slot. The result part of the first clause, the s-expression (+ X Y) is represented by the + object. This is connected to the true output port of the pred object.

The test part of the second clause (= X Y) is represented by the second pred object and the = object connected to the pred object's test slot. The result part of the second clause (+ X 1) is represented by the + object with X and a 1 object connected to its input ports.

The top join object connects the two control flow paths of the lower pred object. The lower

join object connects the two control flow paths of the top pred object.

Figure 7-3



The following code:

```
(or (and (> x y) (+ x y))
    (and (= x y) (+ x 1)))
```

would be represented as figure 7-4. Notice that there are no OR or AND objects in the diagram. A pred object is created for every s-expression except the last in ORs and ANDs. This is because the control flow can split in every s-expression except the last in an OR or AND. The surface plan representation of each s-expression except the last is connected to the test slot of a pred object. The last s-expression in an OR clause is connected to the false output port of pred object representing the penultimate s-expression. The last s-expression in an AND clause is connected to the true output port of pred object representing the penultimate s-expression.

The top left pred object is created when the OR is analysed. The test slot of this pred object is connected to the surface plan representation of the first s-expression in the OR, (AND (< X Y) (+ X Y)). The false output port of this pred object is connected to the surface plan representation of the last s-expression in the OR, (AND (= X Y) (+ X 1)).

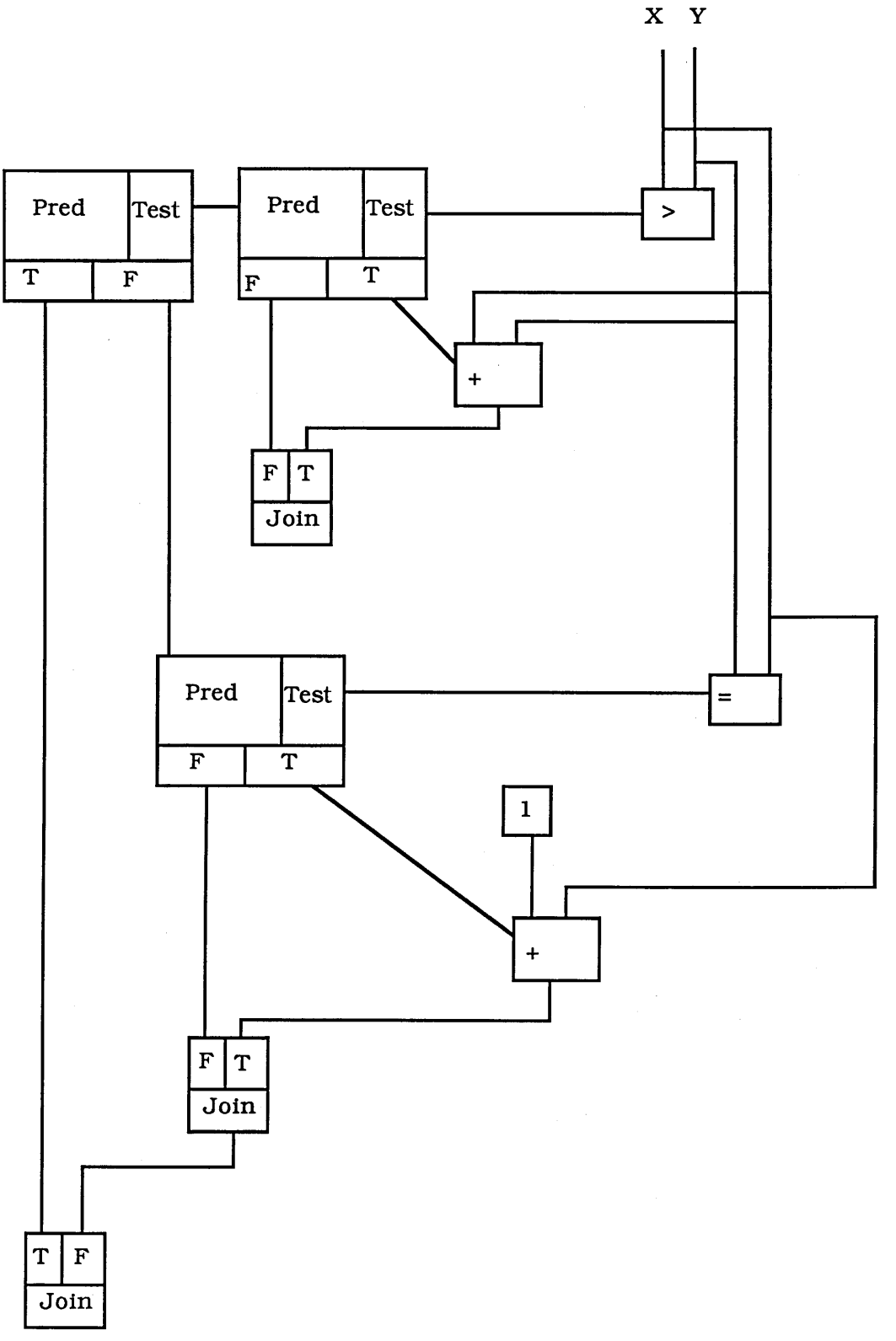
The top right pred object is created when the first AND is analysed. The test slot of this pred object is connected to the surface plan representation of the first s-expression in the first AND, (> X Y). The true output port of this pred object is connected to the surface plan representation of the last s-expression in the first AND, (+ X Y). The top join object connects the two control flow paths of the top right pred object.

The lowest pred object is created when the second AND s-expression is analysed. The test slot of this pred object is connected to the surface plan representation of the first s-expression in the second AND, (= X Y). The true output port of this pred object is connected to the surface plan representation of the last s-expression in the second AND, (+ X 1). The second lowest join object connects the two control flow paths of the lowest pred object.

The lowest join object connects the two control flow paths of the top left pred object. Note that if either a true output port or the false output port of a pred object is connected directly to the corresponding join then the output of the join is the *test* of the pred object. In figure 7-4

the true output port of the top left pred object is connected directly to the true input port of the lowest join object. The true output of this join is the test of the top left pred object, which is the surface plan representation of (AND (< X Y) (+ X Y)).

Figure 7-4


X
Y
Pred
Test
T
F
Pred
Test
F
T
>
+
F
T
Join
Pred
Test
F
T
=
1
+
F
T
Join
T
F
Join

Notice that many of the differences in the code have been abstracted away. The reason why not all the differences have been abstracted away is that the two pieces of code are not really isomorphic. If the s-expression (+ X Y) returned NIL then the COND expression would return NIL, and the second clause would not be tried. In the OR version the second clause would be then tried, as the first returned NIL.

Pred objects have two extra slots, *set-variables* and *inner-set-variables.* These two slots are used as temporary holders when forks are analysed. Set-variables holds the names of any variables set within the current fork being analysed. Inner-set-variables holds the names of any variables set within any fork inside the current fork being analysed. The reason these two slots are needed is explained in section 7.2.4.

Function Definitions

Function definitions include functions defined by DEFUN and *loop functions* (see figure 7-1a). User defined functions are analysed and stored away in a hash table. A user defined function is represented by an object with the following extra slots:

1. Function name.

2. Parameter list.

3. Global variable list.

4. Code and objects.

The function name (obviously) contains the name of the function. The important part of function definitions are the *parameter list* and the *global variable list* (could be called the port list or the io-port list). The parameter list contains the arguments (input ports) to the function. The global variable list contains the global variables side effected in the function (i.e. altered by SETQ). This is the only side effect a user defined function can cause in the subset of Lisp that we are considering. The code and objects slot contains a mapping between each object and the code it represents, this is used by the highlighting module (see section 9.1).

Loop functions are used in the analysis of loops. These are similar to user defined functions except that scoping within the function is dynamic and there are no local variables. So, for example, the following code:

```
(prog (a)
  (setq a 2)
  lp
  (setq a (+ a 2))
  (go lp))
```

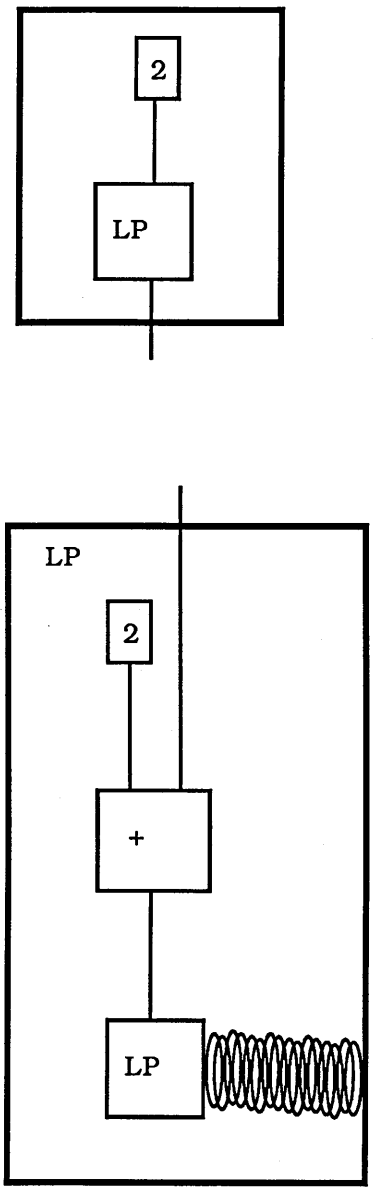
would first be translated into the following:

```
(prog (a)
  (setq a 2)
  (lp))

(loop-define lp ()
  (setq a (+ a 2))
  (lp))
```

The body of the loop (between the tag and the go tag) have been converted into a function. LOOP-DEFINE creates this function. The code is then converted to:

Figure 7-5



Loop function objects are used to represent the LOOP-DEFINE section of code - the second segment containing the + is a loop function object. The fact that the two LP objects are connected by a spring means that the inner LP object is a recursive call the the outer LP object.

Loop

Loop objects are Lisp constructs that allow looping. No objects of this type are actually created. This is used to differentiate between the Lisp constructs that allow loops and those that only allow linear control flow eg. PROG and LET. Currently PROG is the only Lisp looping construct that ITSY can analyse.

Blocks

Blocks represent constructs that allow linear sequences of code without loops. As with loop, objects of this type are not created. There are two such constructs that ITSY can currently analyse, LET and LET*.

Bind Local Vars

Bind local vars represents constructs that allow local variables to be bound. *Parallel* and *serial* represent the two different ways this can happen. If local variables are bound in parallel all the values for the variables are evaluated before binding any of them to the variables. If local variables are bound serially, the variables are bound as the values are evaluated one after another.

7.1.4 Non Connectives

The non-connectives consist of two parts *function applications* and *Common Lisp functions*. Function application represent the application of user defined functions to their arguments. *Loop apply* is used in the analysis of loops - the first segment in diagram 7-5 is a loop apply. *Higher apply function* represents the application of functions using a higher order function, that is a function that takes a function as an argument. Examples of higher order functions include MAPCAR and APPLY. Function application objects contain a type slot. This is set to the type of the function call. This slot can have one of three values; normal, recursive and undefined.

Common Lisp functions covers the non connective Common Lisp functions. *One arg*, *two args* and *any args* are used to specify the number of arguments a function requires.

7.2 The Transformation Process

The actions of this module of ITSY are in some ways similar to the Lisp evaluator. The transformation process begins with the form typed into ITSY's toplevel. At any time there is exactly one *active object.* Initially a *pointer* object is created and made the active object. The input to this object is the toplevel form. At this stage of the analysis the input slot of the object is just the code itself rather than any representation.

If the toplevel form is an atom, the input to the pointer is replaced by an object representing the value of the atom. When a variable is analysed the surface plan representation of the variable's value is stored in a hash table under the variable's name. This is manipulated when the Lisp environment changes, such as inside a function or a LET.

If the toplevel form is a list, the first element of the list is considered to be a function and the analyser creates an object that represents the function. The inputs to this new object are the arguments to the function.

If the function is user-defined the analyser creates a *Function Application* object. The analyser then checks if the function definition has been analysed. The analyser carries out this check because the definition of user-defined functions are only analysed once. As the function application appears in the toplevel form the function definition will not have been analysed. The definition is analysed and stored in a hash table under the function's name. If no definition exists for the function the type slot is set to 'undefined'.

The new object created then becomes the active object and the process starts again.

As the transformation process runs, the segments of raw code in the active object's input slots are replaced by ITSY's internal representation. As a code segment is translated into plan diagram form the code is stored, to be used by the tutorial frame presentation package (see chapter 9). After analysing all of the code we end up with a network of connecting objects.

As discussed in section 7.1.2 each object has an expectations slot. There is a hash table

containing prototype versions of each Lisp function. Whenever a Lisp construct is met this slot is filled with the prototype versions stored in the hash table. The prototype version is used carries information such as the number of arguments a function should be given. This is used by the error cliche matcher.

The transformation process described above is carried out in two independent *contexts.* The first context is the *toplevel context.* This is either *toplevel* or *inside a function.* When code is analysed within a function definition the toplevel context is inside a function, otherwise it is toplevel. The second context used is the *embedded context.* This context is either *normal* or *inside a fork.* This is used in the analysis of forks.

7.2.1 Application of Non-Connective Common Lisp Functions

This type of construct is the simplest to analyse. Whenever a non-connective common Lisp function is met, an object of the appropriate type is constructed. The input slot is set to the arguments of the function (the raw code) and the function object becomes the active object.

7.2.2 Application of User Defined Functions

A function application object is constructed to represent the function applied. The object is given a type depending on the type of function that is being applied. The function definition is analysed if necessary. Each of the arguments is then analysed in turn. The next step depends on the type the function application object was assigned. A function application has one of three types; *not defined, recursive* and *normal.* If the function type is *not defined* or *recursive* no further analysis takes place. If the function is not defined then there is nothing to analyse, if the function type is recursive the function is currently being analysed.

If the function type is *normal* the side effects of the function application, that is any global variables assigned are updated. The side effects of the application of user defined functions within the function body are also updated. This action, of course, would lead to endless recursions in mutually recursive functions. At the moment this is prevented by keeping track of the function definitions checked and restricting the number of times a

function can be checked in this way. Students rarely write mutually recursive functions and there are no cases of a global variable being assigned a value that depended on a recursive call such as in:

```
(defun strange-function (n)
  (cond ((= n 0) 0)
        (t (setq *var* (append (list n) *var*
                               (strange-function (1- n)))))))
```

the global variable *VAR* depends on a recursive call to STRANGE-FUNCTION. Simple FUNCALLs and APPLYs are converted into normal function application, for example:

```
(funcall '+ 1 2 3)
```

or

```
(apply '+ '(1 2 3))
```

is converted to:

```
(+ 1 2 3)
```

7.2.3 Function Definitions

When a function is analysed the two most important rules used are:

a) the output of a function is the last form, all the other forms inside the function are only used for storing temporary values or for side effects.

b) function definitions are only analysed once, even if there is more than one call to the function in the student's code. The necessary information needs to be stored for when the function is applied. The main information is the side effects that occur within the function.

The first thing that happens is that input ports are created for each of the function's parameters. A new hash table is created to store variables. The body of the function is then analysed with the toplevel context set to inside a function. This affects the way dataflow analysis is carried out. Normally when a globally scoped variable is met it is replaced by its value. Inside a function definition a global variable becomes an input port. If the value of a global variable is changed normally a new value is inserted in the hash table. If this happens inside a function this is kept in the side effect slot of the function definition. This saves ITSY having to re-analyse the function whenever it is applied.

Whenever a parameter of a user-defined function is used as an argument to a Common Lisp function the expected type of argument to the Common Lisp function is added to the list of expected types of the user-defined function. So in the following code:

```
(defun my-add (num1 num2)
  (+ num1 num2))
```

the function MY-FIRST would have *number* added to its list of expected types because the Common Lisp function + expects a number. This is used by the Wrong Type error cliche.

7.2.4 Forks

This is where most of the effort has been put during this research.

Conds

Each clause of a COND is analysed in turn. When analysing a clause first a *pred* is created. The test slot of the pred is filled with the surface plan representation of the test part of the clause. Each of the result subclauses from the COND are then analysed. The true control output of the pred is filled with the first of the result subclauses. The false control output of the pred is filled with the test of the pred corresponding to the next clause. The last result subclause is connected to a *join*. The false control output path meets up with this join after passing through the preds and joins corresponding to the clauses that follow the current clause.

And Or

Each element of the and/or is analysed in turn. When analysing an element a pred is created. The test slot of the pred is filled with the element. If an OR is being analysed the true control output port is connected to the join and the false control output port is connected to the next element. If an AND is being analysed the true control output port is connected to the next element and the false control output port is to the join.

Added to both of the above is a complication due to variables being set within various parts of a fork. Consider the following code:

```
(cond ((> x y) (setq min y) (setq max x))
      (t (setq min x) (setq max y))
(list max min)
```

How should we represent this? The representation used in ITSY is the *join value*. A join value contains all the possible values for a variable. Each possible value is represented in two parts; the test and the value. The expression (LIST MAX MIN) above would be represented as figure 7-6. The two inputs to the LIST object correspond to the two inputs to the function LIST, MAX and MIN.

The lowest join value object on the left has two inputs to its value slot. These two inputs correspond to the two possible values that MAX can have X and Y. The first input is the join value object with value X and preds the > object. This represents the fact that MAX will have the value X if (> X Y) is true. The second input is the join value object with value Y and preds the T object. This represents the fact that MAX will have the value Y if T is true (ITSY does not know that T is always true).

The lowest join value object on the right has two inputs to its value slot. These two inputs correspond to the two possible values that MIN can have X and Y. The first input is the join value object with value Y and preds the > object. This represents the fact that MIN will have the value Y if (> X Y) is true. The second input is the join value object with value X and preds the T object. This represents the fact that MIN will have the value X if T is true.

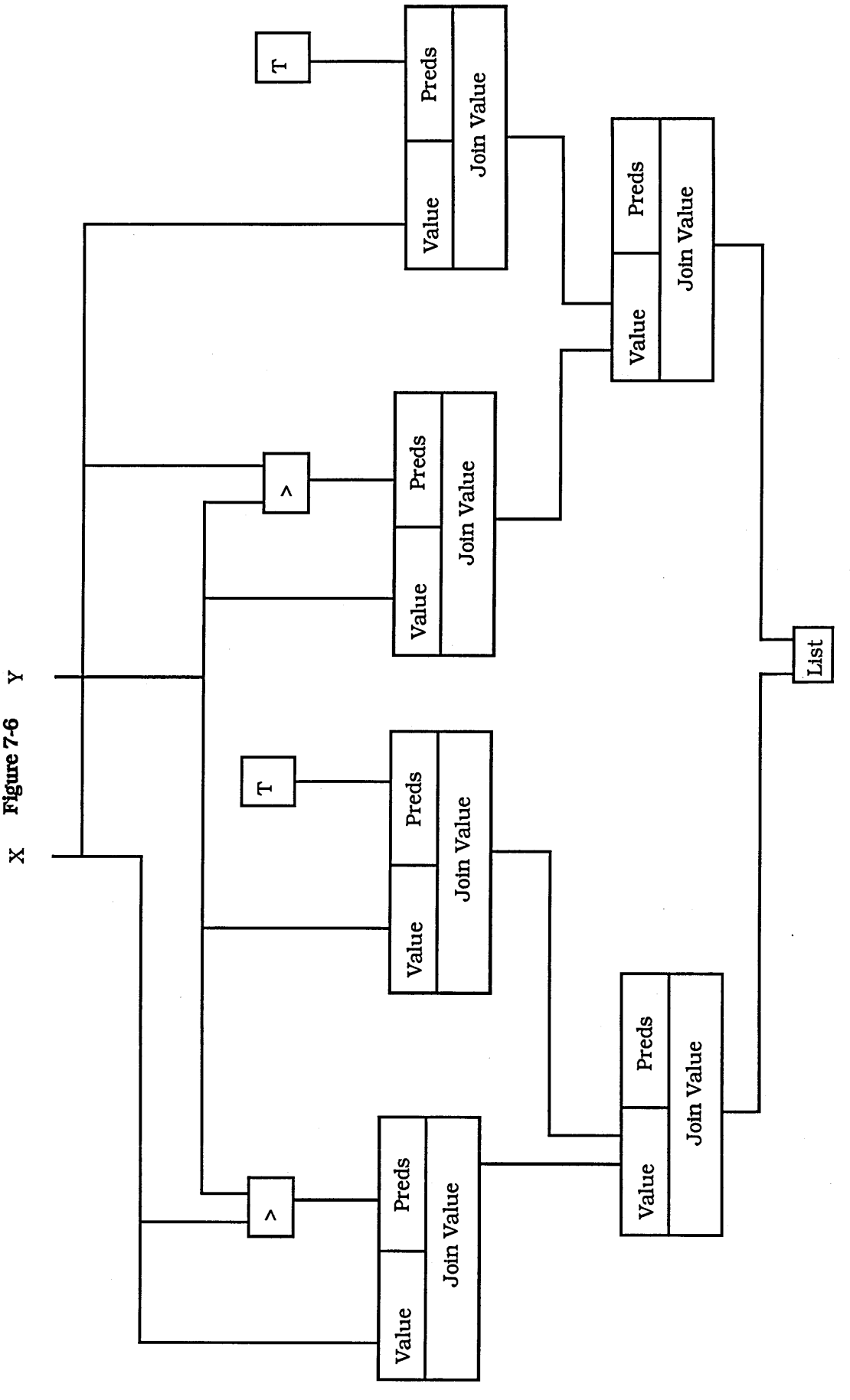

X
Y
T
Preds
Value
Join Value
>
Preds
Value
Join Value
T
Preds
Value
Join Value
>
Preds
Value
Join Value
Preds
Value
Join Value
Preds
Value
Join Value
List

Figure 7-6

This example would be further complicated if the COND was embedded in another fork such as:

```
(and (numberp x) (numberp y)
 (cond ((> x y) (setq min y) (setq max x))
       (t (setq min x) (setq max y)))
(list max min)
```

this would be represented as figure 7-7. This diagram is the same as figure 7-6 except that two extra inputs have been added to the preds slots of the join value objects representing the possible values of MAX and MIN. The two extra inputs correspond to the two new s-expressions whose values will determine the values of MAX and MIN, (NUMBERP X) and (NUMBERP Y). The two s-expressions (NUMBERP X) and (NUMBERP Y) both have to be true for MAX and MIN to be assigned a value.

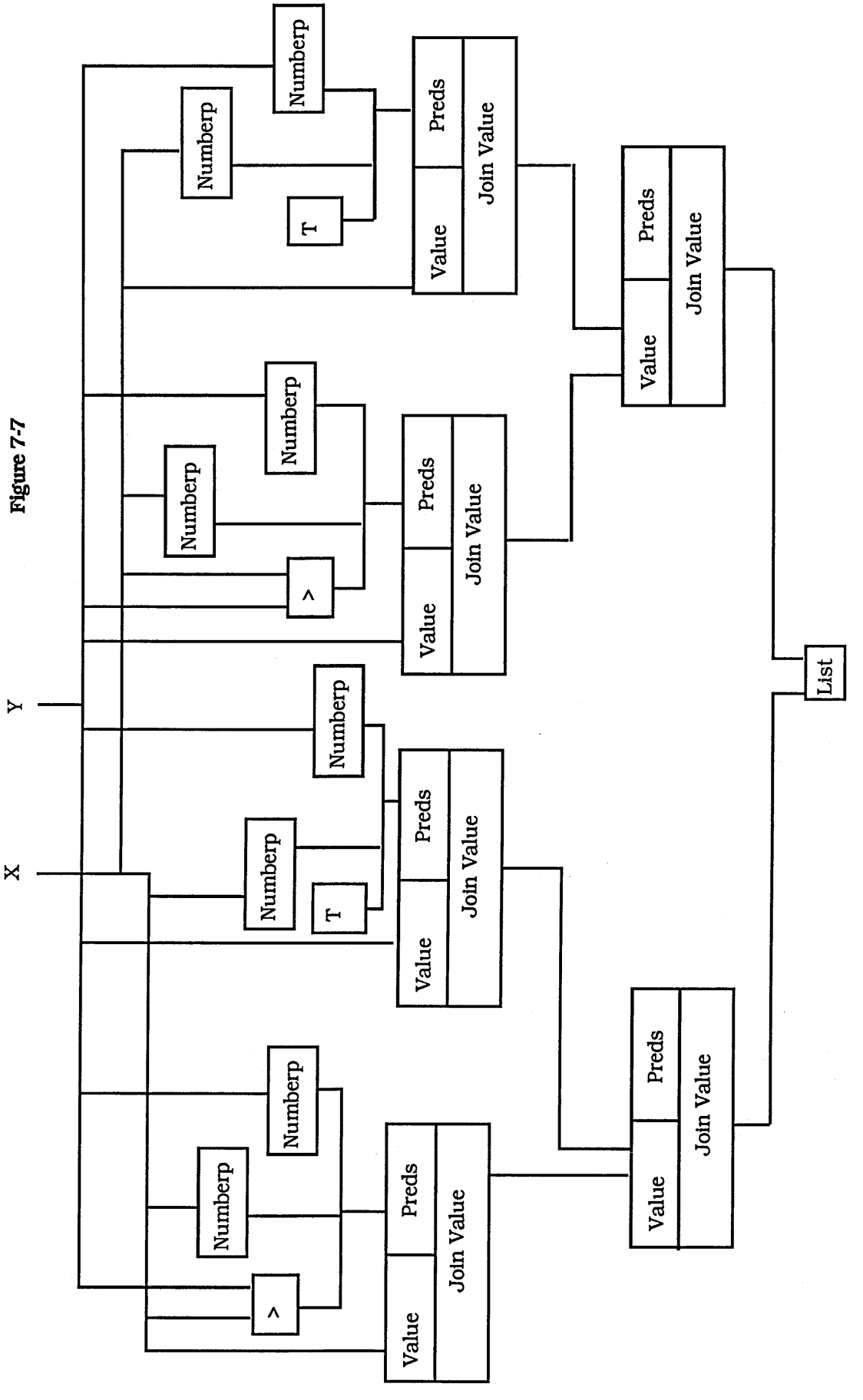

X
Y
Numberp
Numberp
T
Preds
Value
Join Value
Numberp
Numberp
>
Preds
Value
Join Value
Numberp
Numberp
T
Preds
Value
Join Value
Numberp
Numberp
>
Preds
Value
Join Value
Preds
Value
Join Value
Preds
Value
Join Value
List

Figure 7-7

The expression below:

```
(or (numberp x) (numberp y)
   (cond ((> x y) (setq min y) (setq max x))
         (t (setq min x) (setq max y)))
(list max min)
```

would be represented as figure 7-8. This is the same as figure 7-7 except that the numberp objects are now inputs into a *not* object. This represents the fact that the two s-expressions (NUMBERP X) and (NUMBERP Y) must not be true for MAX and MIN to be assigned values.

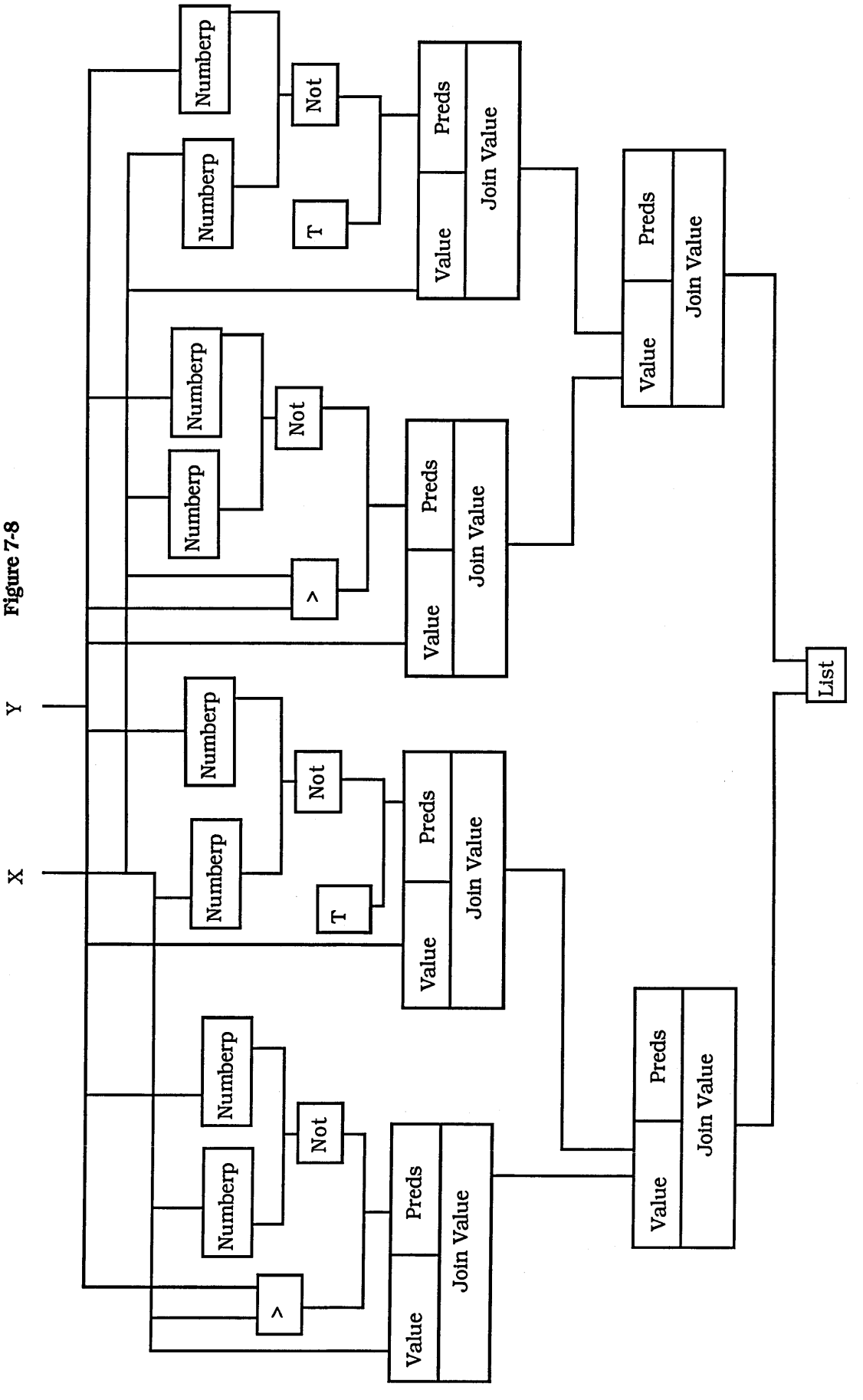

X
Y
Numberp
Numberp
Not
T
Preds
Value
Join Value
Numberp
Numberp
Not
>
Preds
Value
Join Value
Preds
Value
Join Value
Numberp
Numberp
Not
T
Preds
Value
Join Value
Numberp
Numberp
Not
>
Preds
Value
Join Value
Preds
Value
Join Value
List

Figure 7-8

These transformations use the set-variables and inner-set-variables mentioned in section 7.1.3. In all three of the above examples the pred object representing the COND uses the set-variables slot. This slot holds the two variables max and min. The analyser creates join values for these two variables and adds the appropriate predicates. The pred objects representing the AND and OR use the inner-set-variables slot. The analyser adds the appropriate predicates to the existing join values - no new join values are created. The values that a variable had before it was set inside a fork is stored a default value slot of a join value. As none of the variables had such a value it has not been shown, it is shown however in the example given at the end of this chapter.

7.2.5 Loops

DO's and DO*'s are converted to equivalent PROGs. So, for example, the following code:

```
(defun fact (n)
  (do ((i 1 (+ i 1))
       (result 1))
      ((> i n) result)
     (setq result (* result i))))
```

would be translated into the equivalent PROG:

```
(defun fact (n)
  (prog ((i 1) (result 1))
    loop
    (cond ((> i n) (return result)))
    (setq result (* result i))
    (setq i (+ i 1))
    (go loop)))
```

and then to:

```
(defun fact (n)
   (prog ((i 1) (result 1))
      (loop)))

(loop-define loop ()
   (cond ((> i n) result))
         (t (setq result (* result i))
            (setq i (+ i 1))
            (loop))))
```

which is then represented as:

Figure 7-9a

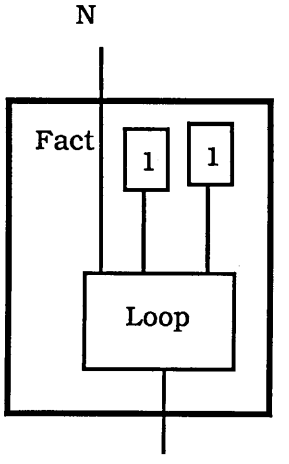
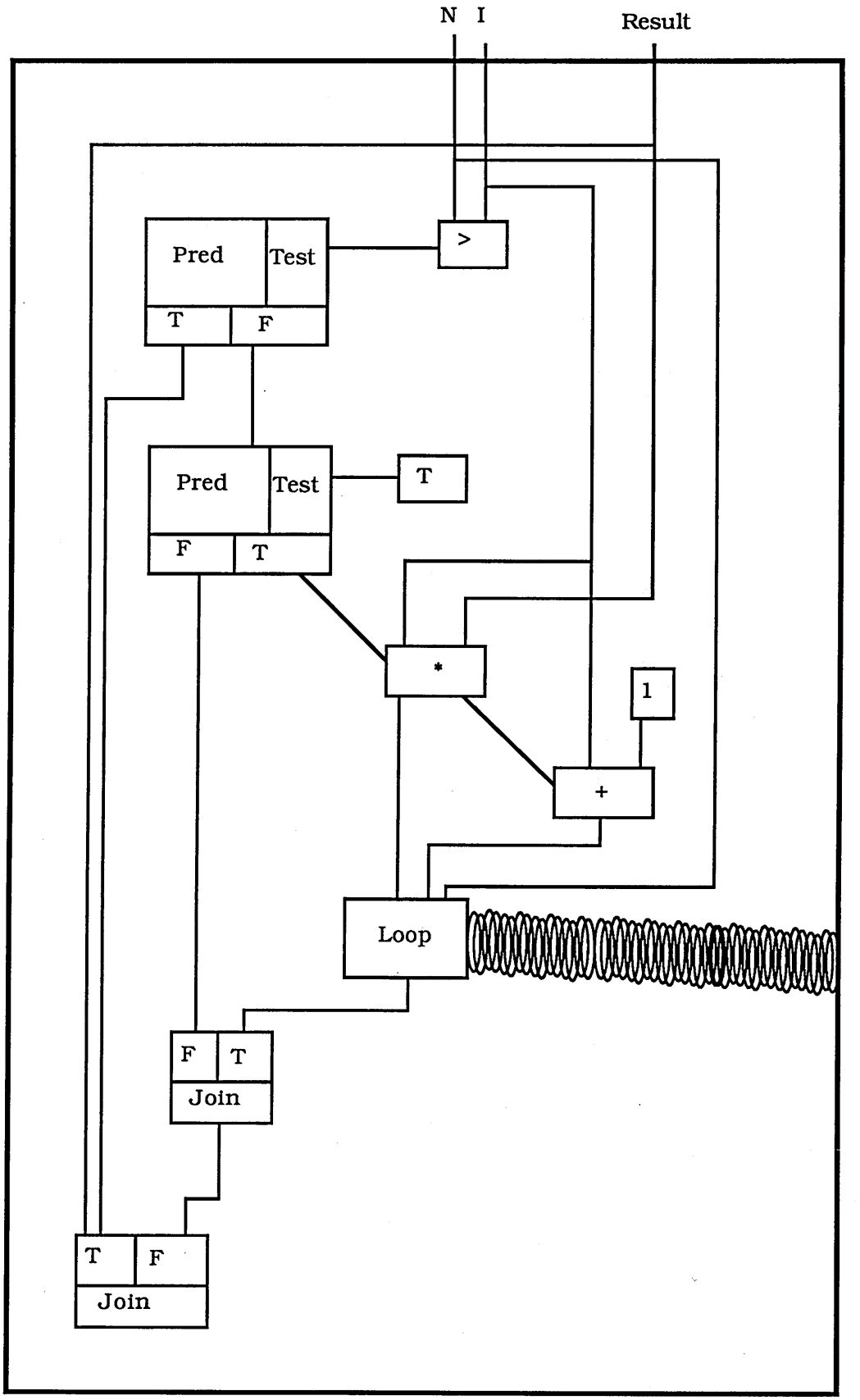


Figure 7-9b



Inside a loop construct the analyser analyses each of the sub-s-expressions until a tag (such as 'loop' in the above example) is found. When a tag is found a *loop function* is created. The following s-expressions, until a (GO *TAG*), are analysed and inserted into the loop function. The (GO *TAG*) is transformed into the recursive function call (*TAG*).

7.3 An Example of Code Transformation

The following example shows some of the more complicated aspects of the transformation process in more detail. I shall describe the transformation of the code shown below:

```
(defun example (x)
  (let ((lisp-type 'atom) (object-type 'unknown) (length-x 0)
        (result nil))
    (and (listp x) (setq lisp-type 'list)
         (cond ((and (equal (length x) 2) (numberp (car x))
                     (numberp (cadr x)))
                (setq object-type 'complex-number)
                (setq length-x 2)
                (setq result (list (* (car x) 2) (* (cadr x) 2))))
               (t (setq object-type 'list)
                  (setq result (palindrome x))
                  (setq length-x (length x)))))
    (list result lisp-type object-type length-x)))

(defun palindrome (x)
  (append x (reverse x)))
```

The above code can be thought of as part of a larger program that 'doubles' objects. The type of doubling carried out depends on the type of object input - this is explained in the next paragraph. The program returns four parts; the 'doubled' object, the Lisp representation of the object, the type of the object and the length of the object. This part of the code deals with complex numbers and ordinary lists. These two types of object are

both represented by Lisp lists.

If X is a list then the LISP-TYPE is set to LIST, otherwise LISP-TYPE is set to ATOM. If X is a list then it is either an ordinary list or a complex number. A complex number has two parts, each part being an integer. If X is a complex number then 'doubling' involves multiplying both parts by 2. If X is a list then 'doubling' involves making a palindrome out of the list.

The analyser proceeds in a depth first manner that is similar to that of the evaluator. When an s-expression is analysed the arguments are analysed first, the function is then analysed and the resulting representation of the whole s-expression returned. The analyser takes an object as input. The input slot of this object will not yet have been analysed and will contain raw Lisp code. The analyser module contains several general purpose submodules. These can take any type of object and perform appropriate tasks. The important submodules are enumerated below:

1. Analyse object - this submodule analyses the one object given as input. This object's input slot is initially raw Lisp code. The object is returned with the input slot set to the surface plan representation of the Lisp code.

2. Toplevel input analyser - this submodule analyses a single input of an object given. It assumes that the object does not occur within a function definition.

3. Function body input analyser - this submodule analyses a single input of the object given. It assumes that the object occurs within a function definition.

The analyser can work in one of two modes *normal* and *within a fork*. This is used when variables are encountered.

Objects are passed around between the submodules. When an object is passed to a submodule it becomes the *active* object.

If a student typed in the form (EXAMPLE 30) the following steps would occur. Firstly, a TOPLEVEL object would be created with input slot set to (EXAMPLE 30). This object

is passed to the analyser and then to the *toplevel input analyser*. The *toplevel input analyser* uses the input slot of the active object, which is raw Lisp code, to create a new object. The input analyser finds that the raw Lisp code is the user-defined function call (EXAMPLE 30). A FUNCTION APPLICATION and a NUMBER object are created. The definition of EXAMPLE is then analysed as follows.

A DEFUN object is created. The name slot is set to EXAMPLE. The parameter slot is set to X. The input slot of the object is set to the body of the function. The DEFUN object is sent to the *function body input analyser*. This submodule creates a LET object and sets the input slot of the object to the code within the LET. This object is passed to the *analyse object* submodule. This submodule creates temporary slots in a hash table for the local variables LISP-TYPE, OBJECT-TYPE, LENGTH-X and RESULT. These slots contain the variable name and the internal representation of the value of the variable. LISP-TYPE and OBJECT-TYPE have QUOTE objects as their values. LENGTH-X has a NUMBER object as its value and RESULT has the NIL object as its value. The analyser module then passes, in turn, the last two s-expressions within the LET to the *function body input analyser* in turn.

The first s-expression passed is (AND (LISTP X) ... An AND object is created with the input slot set to the rest of the AND expression. This object is passed to the *analyse object* submodule. The analyser is set to work in *within a fork* mode, each of the arguments to AND are passed, in turn, to this submodule.

The s-expression (LISTP X) is transformed into a LISTP object. The local variable X is transformed into a LOCAL VARIABLE object. A PRED object (call this PREDA) is created. The test slot of PREDA is set to the LISTP object.

The s-expression (SETQ LISP-TYPE 'LIST) is analysed in a special way because the current mode is *within a fork*. If the analyser were working in *normal* mode a SETQ would result in the representation of the old value of the variable being overwritten with a representation of the new value (the representation of the values of variables are stored in a hash table). Because the analyser is currently working in *within a fork* mode a QUOTE object with input slot set to LIST is added to the slot for LISP-TYPE in the hash table. The slot for LISP-TYPE now has the two QUOTE objects, one with its input slot set to LIST the other with its input slot set to ATOM. A

PRED object (call this PREDB) is created. The test slot of PREDB is set to the QUOTE object.

The third argument to AND is analysed next. The *function body input analyser* creates a COND object and passes this to the *analyse object* submodule. This passes each of the clauses of the COND in turn to the *function body input analyser.*

A PRED object (call this PRED1) is created. The test part of the first clause is analysed first i.e (AND (EQUAL (LENGTH X) 2) ... The *function body input analyser* creates an AND object with the input slot set to the arguments to AND. This is passed to the *analyse object* submodule. This passes the first argument (EQUAL (LENGTH X) 2) to the *function body input analyser.* This s-expression is transformed into EQUAL, LENGTH, LOCAL VARIABLE and NUMBER objects. A PRED object (call this PRED2) is created and its test slot set to the network of objects that represent the s-expression (EQUAL (LENGTH X) 2). In a similar fashion the next argument to AND, (NUMBERP (CAR X)) is analysed and a PRED object (call this PRED3) is created with its test slot set to the representation of this s-expression. The last argument to AND, (NUMBERP (CADR X)) is analysed. No PRED object is created for the last argument as the controlflow cannot split (i.e. is always the same) on the last argument to an AND. The true output port of PRED2 is set to PRED3 and the true output port of PRED3 is set to the NUMBERP object in the last argument to the AND. A JOIN is created for PRED2 (call this JOIN2) and for PRED3 (call this JOIN3). The false output port of PRED2 is set to the false input port of JOIN2 and the false output port of PRED3 is set to the false input port of JOIN3. The output port of the representation of the last argument to AND is set to true input port of JOIN3. The output port of JOIN3 is set to the true input port of JOIN2. The AND object created is discarded and the PRED2 is returned. Now the analyser 'pops up' to the first clause of the COND.

The result part of the first clause is analysed next. As mentioned earlier because the analyser is working in *within a fork* mode SETQs are analysed in a special way. A QUOTE object (call this QUOTE1) is pushed into the hash table slot of OBJECT-TYPE. A NUMBER object is created and pushed into the hash table slot of the variable LENGTH-X. A network of objects representing the s-expression (LIST (* (CAR X) 2) (* (CADR X) 2))) is pushed into the hash table slot of RESULT. The test slot of PRED1 is set to PRED2. The true output port of PRED1 is set to QUOTE1. The set-variables slot

of PRED1 is set to (OBJECT-TYPE LENGTH-X RESULT).

The second clause of the COND is analysed next. A PRED object (call this PRED4) is created with its test slot set to a T object. Representations for the three variable values 'LIST, (PALINDROME X) and (LENGTH X) are pushed into the appropriate hash table slots. Before the s-expression (PALINDROME X) is analysed the definition of the function PALINDROME is analysed and stored in a hash table. The set-variables slot of PRED4 is set to (OBJECT-TYPE RESULT LENGTH-X). The analyser now 'pops up' to the analysis of the whole COND.

The false output port of PRED1 is set to PRED4. JOIN-VALUE objects are created for the variables in the set-variables slot of PRED1 and PRED4, and pushed into the appropriate slots in the hash table. Each JOIN-VALUE object contains the representation of the variable and the test slots that needs to be true in order for the variable to have the particular value. The default slot of each JOIN-VALUE is to the surface plan representation of the value each variable had before the fork was entered. In the JOIN-VALUE representing LISP-TYPE, the default value is set to the QUOTE object with its input slot set to ATOM. Similarly, the JOIN-VALUEs representing OBJECT-TYPE, LENGTH-X and RESULT have default values is set to the QUOTE object with its input slot set to UNKNOWN, a NUMBER object and a NIL object respectively. JOINs are then created for PRED1 (call this JOIN1) and for PRED4 (call this JOIN4). The output port of the LIST object in the s-expression (LIST (* (CAR X ... is set to the true input port of JOIN1. The output port of the LENGTH object in the s-expression (LENGTH X) is set to the true input port of JOIN4. The false output port of PRED4 is set to the false input port of JOIN4. The output port of JOIN4 is set to the false input port of JOIN1. The COND object is now discarded and PRED1 is returned. The analyser now 'pops up' to the outer AND.

The true output port of PREDA is set to PREDB. The true output port of PREDB is set to PRED1. Two JOIN objects are created (JOINA and JOINB). The false output port of PREDA is set to the false input port of JOINA. The false output port of PREDB is set to the false input port of JOINB. The output port of JOIN1 is set to the true input port of JOINB. The output port of JOINB is set to the true input port of JOINA. The next step involves the variables set within the COND (inner-set variables). A JOIN-VALUE

object is created for the variable LISP-TYPE. The LISTP object for the s-expression (LISTP X) is inserted into the preds slot of the object. The test slots of PREDA and PREDB are added to the JOIN-VALUE objects of the variables OBJECT-TYPE, RESULT and LENGTH-X. The AND object is now discarded and PREDA is returned. The analyser now 'pops up' to the analysis of the LET.

The analyser is now back in *normal* mode. The third argument to LET is now analysed. A LIST object is created. The input slot of this object is set to the objects stored in the hash table under the slots of RESULT, LISP-TYPE, OBJECT-TYPE and LENGTH-X. The LET object is now discarded and PREDA and the LIST object are returned. The analyser now 'pops up' to the analysis of the DEFUN.

The input slot of the DEFUN object is set to the LIST object. PREDA is stored in the inside slot of the DEFUN object. The DEFUN object is stored in a hash table under EXAMPLE.

Figure 7-10 shows the surface plan representationof the input slot of the DEFUN. This is a representation of the last s-expression in the function EXAMPLE, (LIST RESULT LISP-TYPE OBJECT-TYPE LENGTH-X).

Figure 7-10a shows the first two arguments to LIST, RESULT and LISP-TYPE. The three join values on the left of figure 7-10a represent the value of RESULT. As mentioned in 7.2.4 join values have a default slot. This slot is set if a variable is assigned a value before being assigned a value inside a fork. The default slot is set to a *nil* object as RESULT is assigned the value NIL in the initialisation part of the LET. The top left join value represents RESULT having the value (LIST (* (CAR X) 2) (* (CADR X) 2))). RESULT is assigned this value if the three inputs to the preds slot are true. Pred2 (shown in figure 7-10c) represents the test part of the first clause in the COND, (AND (EQUAL (LENGTH X) 2) (NUMBERP (CAR X)) (NUMBERP (CADR X))). The *quote* and *list* objects represent the second s-expression given to the first AND, (SETQ LISP-TYPE 'LIST). The *listp* object represent the first s-expression given to the first AND, (LISTP X). The rightmost join value object represents the possible values that LISP-TYPE might be assigned.

Figure 7-10b represents the possible values the second two arguments to LIST might

have. The three leftmost join value objects represent the possible values of OBJECT-TYPE. The three rightmost join value objects represent the possible values of LENGTH-X.

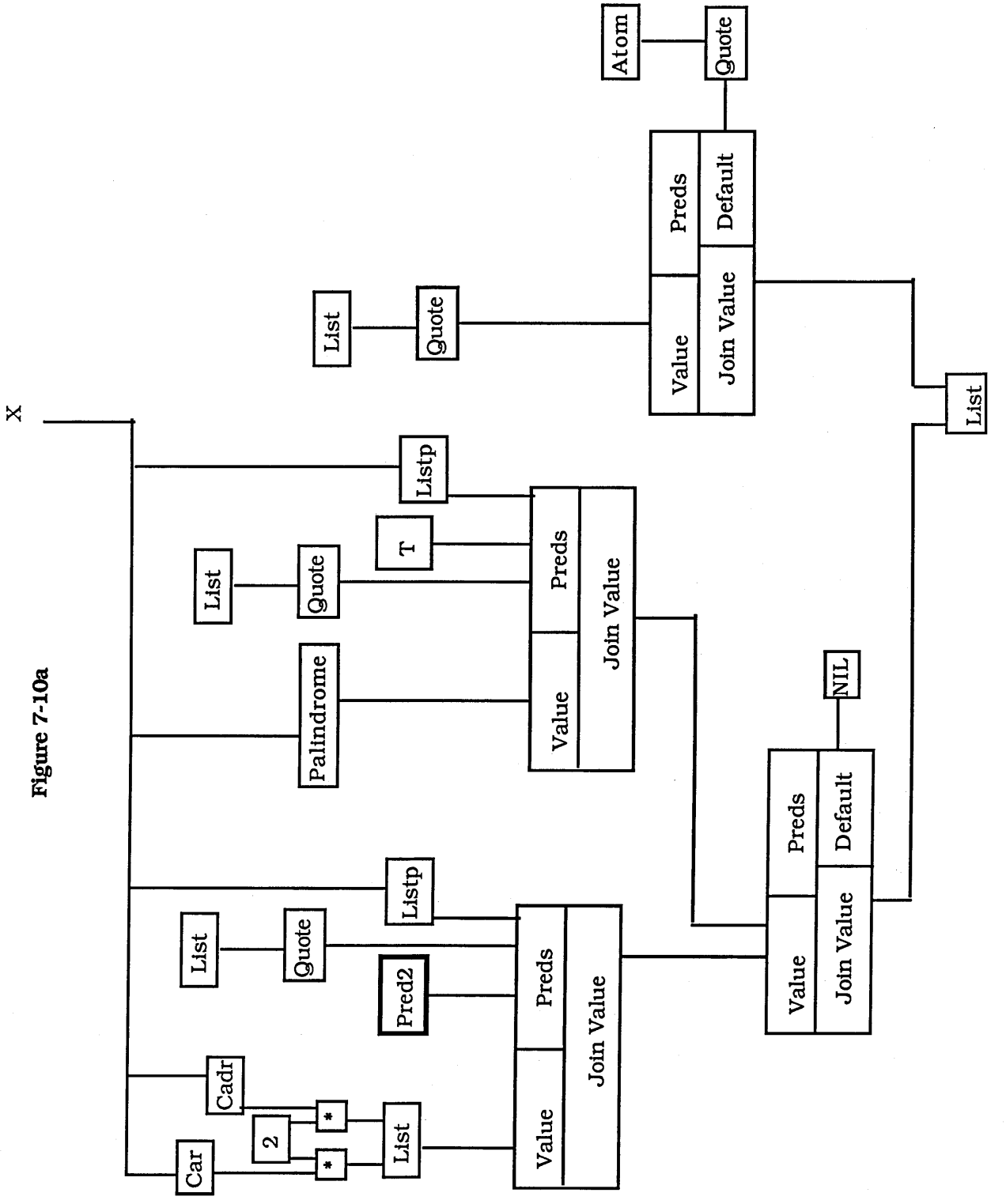

X
Listp
T
Preds
Value
Join Value
List
Quote
Palindrome
Atom
Quote
Preds
Default
Value
Join Value
List
Quote
Listp
Pred2
Preds
Value
Join Value
List
Quote
Cadr
2
*
*
Car
NIL
Preds
Default
Value
Join Value
List

Figure 7-10a

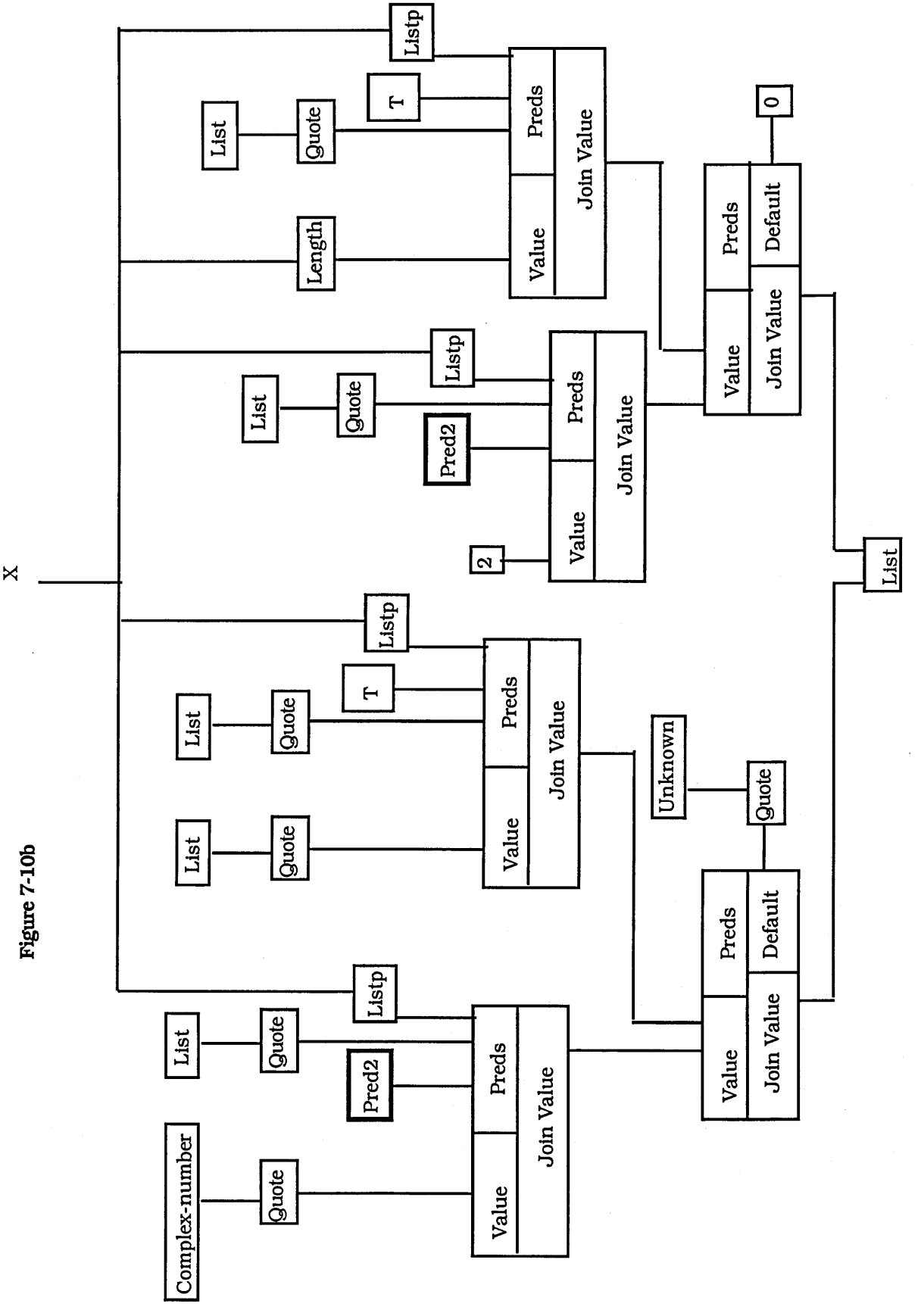
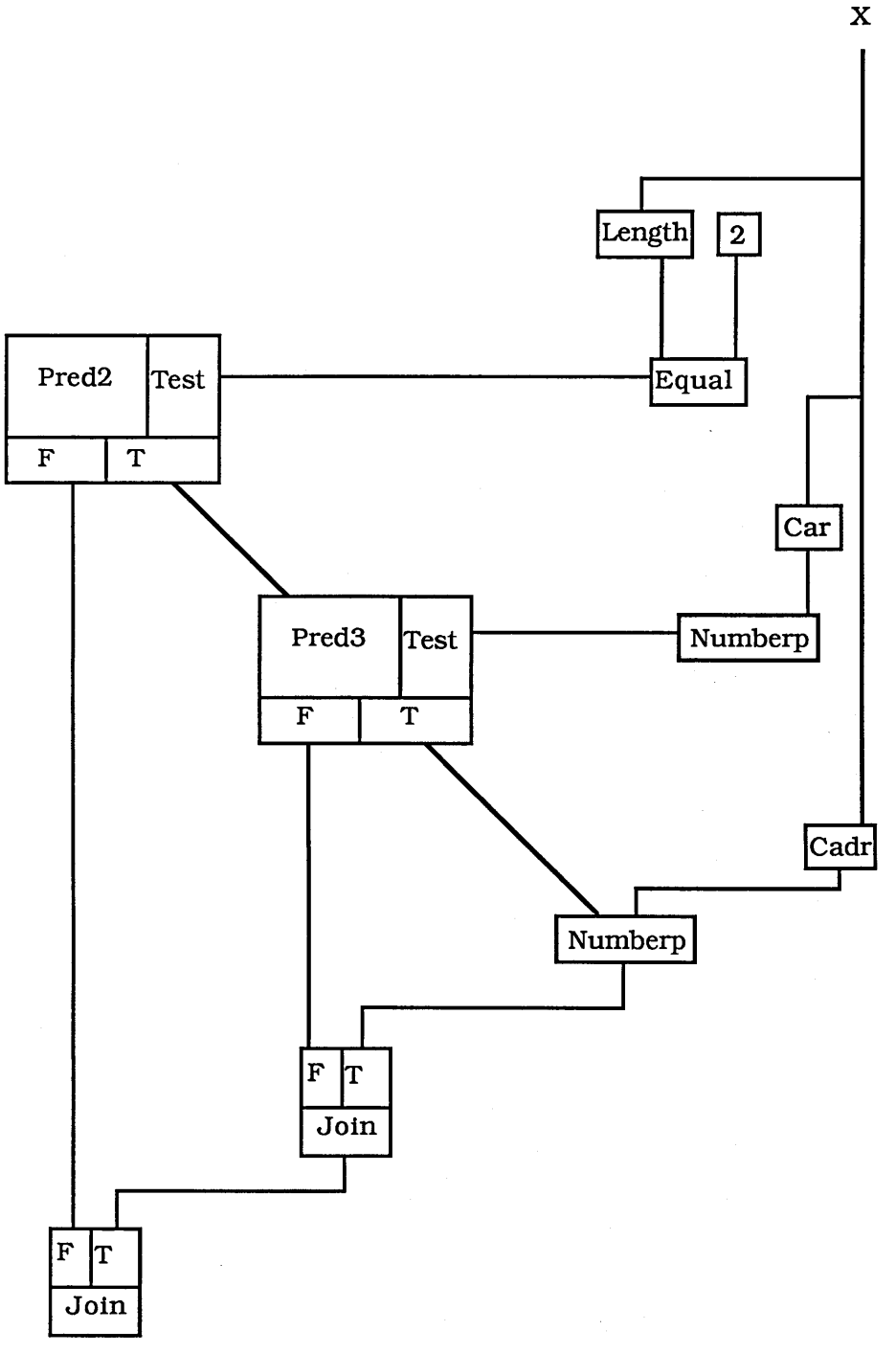

X
Listp
T
List
Quote
Preds
Value
Join Value
Length
0
Default
Listp
List
Quote
Pred2
Preds
Value
Join Value
2
List
Listp
T
List
Quote
List
Quote
Preds
Value
Join Value
Unknown
Quote
Listp
List
Quote
Pred2
Complex-number
Quote
Preds
Value
Join Value
Default

Figure 7-10b

Figure 7-10c



7.4 Current Limit of Analysis

As mentioned earlier the only loop construct that ITSY can currently deal with is the PROG. Extra code to transform the other constructs into the equivalent PROGs is needed. In most cases the addition of a new Common Lisp function would involve

adding a new type of object in ITSY's Lisp Object Hierarchy. Special forms would also require the addition of code in order to be parsed. The functions that would create the most difficulty would be the destructive functions. This is not too great a problem as these functions are generally not used by novices, or are used when the novice has had a fair amount of exposure to Lisp. A list of all the functions that ITSY can currently analyse is given in appendix J.

8. MATCHING ERROR CLICHES AGAINST THE TRANSFORMED CODE

This chapter describes in detail how ITSY matches an error cliche against the transformed code. Each error cliche has the following four parts:

a) Error Cliche Name - the name of the error cliche,

b) Surface Code Segment - the 'type' of object that the error cliche can match against,

c) Criteria - criteria that need to be satisfied in order for the error cliche to match,

d) Other Checks - tests that prevent false alarms.

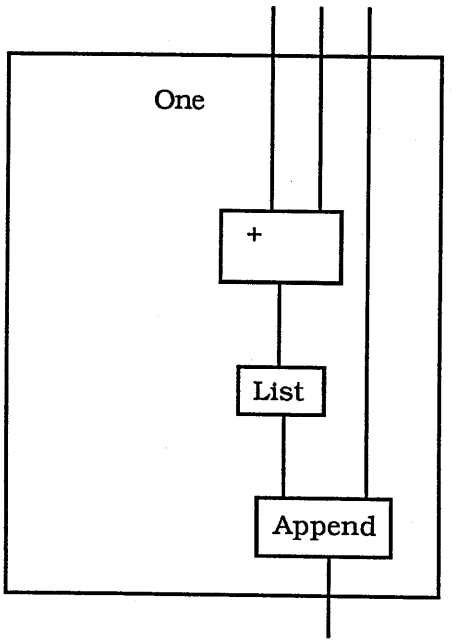
The matching process is carried out by two distinct modules. The *plan diagram traversing module* and the *error cliche matching module.* The plan diagram traversing module traverses the network of objects created by the code transforming module (described in chapter 7). The error cliche matching module is activated each time the plan diagram traversing module comes to a new node. The error cliches are 'active' - that is they actively attempt to match themselves against a segment of the network of objects. Error cliches are in fact implemented as messages, the Error Cliche Name corresponds to the name of the message and the Surface Code Segment corresponds to the class of objects the message can be sent to. The error cliche matching module 'fires' each error cliche in turn. The error cliche then actively tries to match itself against the current node in the network. This process stops when either an error cliche has matched itself against part of the network or all of the nodes have been examined.

In the Programmer's Apprentice project [Waters, 1985] raw code is first translated into surface plans then segments of the surface plans are matched against cliches producing plan diagrams. ITSY uses a variant of this. Student's code is first translated into surface code as in the Programmer's Apprentice. Then, instead of trying to turn the code into a plan diagram via cliches, ITSY tries to match *error* cliches against the code. If ITSY succeeds in doing this then a tutorial is given.

8.1 Traversing the Transformed Code

The output of the code analyser is a network of plan diagram segments represented by objects. This module "walks" through the network. I will show this process using an example. Consider the following surface plan representation (this is taken from figure 7-2):

Figure 8-1



The + is matched against the error cliches, then LIST is matched, then APPEND.

This method of matching means that ITSY will not always find the same error that the evaluator would. In the following code:

```
(append (list (car 1)) 2 3)
```

ITSY would find the error '2 is the wrong type of argument for APPEND' whereas the Lisp evaluator would report '1 was the wrong type of argument for CAR'. One might think that a simple to cure for this would be to proceed in a depth first manner. However if the traverser were altered to proceed in a depth first manner then ITSY would have

problems with the following code:

```
(appen (car 1) 2 3)
```

ITSY would report '1 was the wrong type of argument for CAR' whereas the Lisp evaluator would report 'APPEN is not a defined function'. The real cure for this is to divide the error cliches into two halves. Each node in the network would be visited twice. The traverser would visit a node, 'fire' half of the error cliches then visit the nodes 'below' the current node in a depth first manner. After visiting all of the nodes 'below' the current node the traverser would then fire the other half of the error cliches. The first half would contain error cliches that match against errors concerning the actual function being called (such as the Bracket Around a Variable error cliche), the other half would contain error cliches that match against the errors concerning the arguments to a function (such as Argument of the Wrong Type error cliche). This would have to be implemented in future versions of ITSY.

When possible ITSY gives a tutorial on a single error. There are two reasons for this. Firstly, the student would be confused if s/he were to be tutored about several different topics at once. Secondly, there is a chance that any errors yet to be discovered may have been caused by the student having the same misconception. The student is given a second chance to fix the other errors before receiving a tutorial about them.

The surface plan network is traversed until either an error cliche is found to match a segment of code, or all the code has been traversed. Each different type of plan diagram segment has its own built-in code traverser. As the network is traversed each plan diagram segment's traverser becomes active, once all parts of the segment have been inspected the traverser passes activity to another plan diagram segment's traverser. The different type of plan diagram segment traversers are described in turn below.

Two steps have been taken to make ITSY's messages coincide with the Lisp error messages if there is more than one error in the student's code. Firstly, the error cliches have been ordered. This ordering is based on the order in which the Lisp evaluator evaluates Lisp forms. The *wrong number of arguments* error cliche is before the *wrong type argument* error cliche, so the in the following example:

```
(car 1 2)
```

the *wrong number of arguments* error cliche matches first. Some of the error cliches have been artificially 'raised'. The error cliche *bracket around a variable* is one such error cliche to have been raised. This error cliche matches one level above it normally would. By level I mean what is commonly called 'list depth', so in the list:

```
((((a b) c) d) e f (((g))))
```

e and f are at level 1, d is at level 2, c is at level 3 and a, b and g are at level 4.

The following example will help explain, consider:

```
(+ (a) 'john)
```

(A is a variable). If the error cliche *bracket around a variable* were to match at the level of the s-expression (A), the error cliche *wrong type argument* (which matches at the level (+ (A) 'JOHN) would match first. This is because the + segment is checked before the arguments. The error cliche *bracket around a variable* has been 'raised' so that it matches at the level of (+ (A) 'JOHN).

8.1.1. Common Lisp Functions

This type of plan diagram segment is traversed as follows. First the object itself is matched against the current error cliches, then each of the arguments are traversed in turn. The plan diagram segment linked to the control-out slot is then traversed.

8.1.2. User Defined Function Application

First, the object itself is matched against the current error cliches. Second, the arguments are traversed in turn. The plan diagram representation of the function is retrieved from the user definition hash table. The input ports of the definition are connected to the appropriate ports inside the function application. The definition of the function is then traversed. In order to prevent endless cycling function definitions are only allowed to be analysed a certain number of times.

8.1.3. Function Definitions

User defined functions are traversed in two different ways depending on the context in which the definition was encountered.

Normally only the function name and the parameter list are matched as these are the only two sources of Lisp error in a function definition. That is to say, there are only two ways an error can occur whilst *defining* a function. There can be an error in the function name (eg. non-symbolic) or there can be an error in the parameters (eg. they contain T or NIL or the parameters are not a list). The body of the function cannot generate an error during the definition of a function. One could argue that if the function definition were not closed (ie. no right bracket to match against the first left bracket), this would result in a read error, but then the function would not have a body.

The other way a function definition can be analysed is if an application of the function is found. Once the input ports of the function have been connected to the ports of the application, the body of the function definition is traversed.

8.1.4 Forks

First the test part of the fork is traversed. If an error is found then the process halts. The true and false control outputs are then traversed concurrently until the pred's corresponding join is met. If an error is found in both the true and false paths then both errors are returned. At present ITSY cannot tell which of the two paths generated the Lisp error.

8.1.5 Loops

The code analyser transforms loops to an application of a loop function. The code traverser treats loops in the same way as a normal function application. When an application of a loop function is met, the body of the loop function is retrieved from the loop function hash table. The loop function is traversed once only. The recursive call within the loop is not re-analysed.

8.2 Returning Information About the Error

Once an error cliche has been matched against a surface plan segment an object is returned. The object contains 4 pieces of information about the error:

a) the error cliche that matched against the surface plan segment,

b) the surface plan segment that contains the erroneous piece of code,

c) The number of errors,

d) whether the error occurred inside code at toplevel, or in a file. The error may be contained in the actual code typed to the Lisp toplevel, or in the body of a function (loaded from a file) called from toplevel.

8.3 Matching a Code Segment Against an Error Cliche

As each part of the network is traversed an attempt is made to match each of the active error cliches in turn. The members and order of the active set is determined by the type of error signalled. Common Lisp errors as implemented on the 3600 Symbolics series have a type. ITSY uses this type to select the active set. For example, an error of the type 'undefined function' excludes a 'wrong type' error cliche.

When ITSY is trying to find an error cliche to match against a segment two rules are used:

a) Try to match the most complex error cliche first,

b) Try to match as high up as possible, that is at the highest level.

Matching has to proceed in a certain order as some error cliches subsume others. This can best be explained using an example. There are error cliches *Arguments in the Wrong Form* and *Wrong Number of Arguments Given to a Function.* The first error cliche matches against sections of code such as:

```
(expt (2 3))
```

where the arguments to the function EXPT have been presented in the wrong form. The *Wrong Number of Arguments Given to a Function*error cliche would match against this section of code because the function EXPT has been given the wrong number of arguments, but *Arguments in the Wrong Form* is the error cliche that applies in this case.

8.4 An Example of Matching

In order to make clear some of the concepts discussed earlier we shall describe in detail a match with the error cliche *Arguments in the Wrong Form.* Consider the following code:

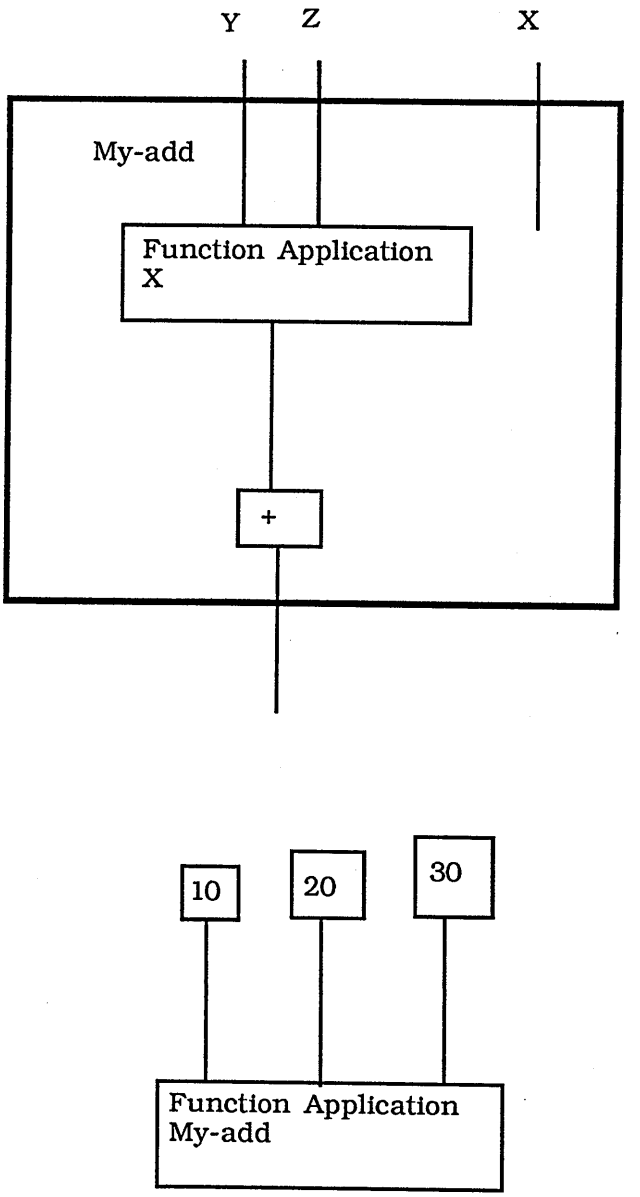
```
(defun my-add (x y z)
  (+ (x y z)))
```

When the student types the s-expression

```
(my-add 10 20 30)
```

at toplevel, the following surface plan diagram is generated:

Figure 8-2



The error cliche finder first looks at the function application to make sure it is a defined function then the arguments are traversed. None of the error cliches match up against the arguments or the function application so the function definition is traversed. During this process the function + is traversed. At this point the error cliche *Arguments in theWrong Form* matches against the surface plan segment for +. The function + satisfies the criteria for the error cliche *Arguments in the Wrong Form1* with the surface code segment *CL Function* that is: the function + has only been given one argument in the definition of MY-ADD and the function does not only take one

argument. The argument to the function + satisfies the subcliche *Arguments in the Wrong Form Subcliche* that is: the function '10' is not defined and the other checks are satisfied.

8.5 The Error Cliches

Each error cliche is represented as a collection of rules. For each type of surface plan segment there is a rule that determines whether or not that particular segment could match the error cliche. A rule has four parts:

a) Error Cliche Name - the name of the error cliche,

b) Surface Code Segment - the type of surface plan segment that the error cliche can match against. Note that the error cliche can also match against all children of the type of surface plan segment. For example an error cliche with Surface Code Segment *Function Definition* could also match against surface plan segments of type *Defun* and *Loop Defun.*

c) Criteria - a set of criteria that need to be satisfied,

d) Other Checks - a set of checks to make sure that this is not a false alarm. This is needed because sometimes the student can make several errors at once. For example, one of the manifestations of the *Arguments in the Wrong Form* error cliche traps errors where the student gives a list argument as separate quoted atoms:

```
(cons 'a '(b c d))
```

is written as:

```
(cons 'a 'b 'c 'd)
```

but the student could have meant

```
(cons 'a (list b c d))
```

if B, C, and D are variables. This manifestation of the error cliche checks quoted atoms to make sure that none are bound variables so as to distinguish between these two cases.

Some error cliches use *sub-cliches*. These have the same format as error cliches but are not error cliches in their own right. These are used when an error cliche extends over several surface plan segments. Sub-cliches can refer to the error cliche (or any part of the error cliche) that uses them by using the term *super cliche*.

The error cliches are enumerated below. Each error cliche has been numbered, some of the error cliches cover more than one Surface Code Segment (ie. an error cliche covers all the items with the same Error Cliche Name).

1. Error Cliche Name: *Bracket Around a Variable*

   Surface Code Segment: Function Application

   Criteria: Function is undefined.
   The 'name' of the function contains a bound symbol ('name' is explained below).

   Other Checks: The 'name' of the function is not itself a function.

An example of this error cliche is:

```
(defun my-first (x)
  (car (x)))
```

It may not be clear why we need the extra check to verify that the name of the function is not a function. This is because by 'name' we mean the first atom in the actual name of the function. For example if we had

```
(car (((x))))
```

the name of the apparent function (because of the bracketing error) would be X. This extra check prevents this error cliche matching against sections of code as:

```
(defun construct (element list)
  ((list element list)))
```

Where the wrongly bracketed item LIST is actually a function (this will match against the *Extra Bracket Around a Function* error cliche).

2. Error Cliche Name: *No Brackets Around a Function Call*

   Surface Code Segment: Symbols

   Criteria: The symbol is unbound.
   The symbol is the name of a function.

   Other Checks: None.

This error cliche matches against sections of code where the student has not placed a bracket before a function call as in:

```
(defun my-first (x)
  car x)
```

3. Error Cliche Name: *Extra Bracket Around a Function Call*

   Surface Code Segment: Function Application

   Criteria: The function is undefined.
   The 'name' of the function is a list.
   The first atom in the 'name' of the function is the name of a defined function.

Other Checks: None.

The following two error cliches (arguments in the wrong form1 and arguments in the wrong form2) correspond to a student not knowing how to give a function multiple arguments. Each error cliche has two different forms. An example of each of the four forms is given below:

```
(+ (1 2 3))
```

```
(+ '(1 2 3))
```

```
(cons 'a 'b 'c 'd)
```

```
(cons a b c d)
```

Two error cliches are needed below because function application is divided into two in the hierarchy of Lisp objects (see diagram 8.1).

4. Error Cliche Name: *Arguments in the Wrong Form1*

Surface Code Segment: CL Function

Criteria: The function has been given one argument which satisfies the subcliche *Arguments in the Wrong Form Subcliche.*
The function does not take just one argument.

Other Checks: None.

An example of the above is:

```
(* '(1 2 3))
```

Error Cliche Name: *Arguments in the Wrong Form1*

Surface Code Segment: Function Application

Criteria: The function is of type defined or recursive.
The function has been given one argument which satisfies the subcliche *Arguments in the Wrong Form Subcliche.*
The function does not take just one argument.

Other Checks: None.

An example of the above would be:

```
(roots (3 4 5))
```

where ROOTS is defined as:

```
(defun roots (x y z)
  (/ (- y (sqrt (- (expt y 2) (* 4 x z)))) (* x 2)))
```

Sub Cliche Name: *Arguments in the Wrong Form Subcliche*

Surface Code Segment: Function Application

Criteria: The function is not defined.

Other Checks: None of the arguments to the function are functions.
The type of the function name and the function's arguments satisfy the type constraints of the super cliche's function.
The number of arguments to the function is one less than the number of arguments the super cliche's function requires.

An example of where this would be used is:

```
(let ((x 1) (y 2) (z 3))
  (+ (x y z)))
```

Sub Cliche Name: *Arguments in the Wrong Form Subcliche*

Surface Code Segment: Quote

Criteria: The quoted object is a list.
The length of the list is the same as the number of arguments the function takes.

Other Checks: The elements of the list are the right type for the function.
The list does not contain the names of any functions.

An example of where this would be used is:

```
(roots '(1 2 3))
```

5. Error Cliche Name: *Arguments in the Wrong Form2*

Surface Code Segment: User Function Application

Criteria: The function is defined.
The function has been given too many arguments.
Each argument is either a quoted or an unbound atom.
The function can take arguments either of type list or of any type.

Other Checks: None of the arguments are functions.

None of the quoted atoms are variables.

An example of where this would be used is:

```
(pal 'a 'b 'c 'd)
```

Where the function pal is defined as:

```
(defun pal (l)
  (append l (reverse l)))
```

Error Cliche Name: *Arguments in the Wrong Form2*

Surface Code Segment: CL Function

Criteria: The function has been given too many arguments.
Each argument is either a quoted or unbound atom.
The function can take arguments either of type list or of any type.

Other Checks: None of the arguments are functions.
None of the quoted atoms are variables.

This error cliche would match against:

```
(cons 'x 'y 'z '1 '2 '3)
```

6. Error Cliche Name: *Not Quoting a List*

Surface Code Segment: Lisp Object

Criteria: The object expects either inputs of type list or inputs of any type.

One of the inputs satisfies the sub-cliche *Not Quoting a List Subcliche.*

Other Checks: None.

An example of the where the above would match is:

```
(list (a b c))
```

The following error cliches have been defined on the surface code segment one arg, two arg etc. rather than on CL Function to make the criteria and other checks easier.

Error Cliche Name: *Not Quoting a List*

Surface Code Segment: One Arg

Criteria: The first input satisfies the sub-cliche *Not Quoting a List Subcliche.*

Other Checks: The first input satisfies the type constraints of the function.

An example of the above is:

```
(car (x y z))
```

Error Cliche Name: *Not Quoting a List*

Surface Code Segment: Two Args

Criteria: The first or second input satisfies the subcliche *Not Quoting a List Subcliche.*

Other Checks: The same input satisfies the type constraints of the function.

An example of the above would be :

```
(cons 'a (b c d))
```

Error Cliche Name: *Not Quoting a List*

Surface Code Segment: Three Args

Criteria: Either the first, second or third input satisfies the sub-cliche *Not Quoting a List Subcliche.*

Other Checks: The same input satisfies the type constraints of the function.

An example of the above would be:

```
(subst 'a 'b (a b c))
```

Sub Cliche Name: *Not Quoting a List Subcliche*

Surface Code Segment: User Function Application

Criteria: The function is undefined.

Other Checks: None of the inputs to the function are variables or functions.

7. Error Cliche Name: *Not Quoting an Atom*

Surface Code Segment: Lisp Object

Criteria: One of the inputs satisfies the subcliche *Not Quoting an Atom Subcliche.*

The object expects input of atom type or input of any type.

Other Checks: None.

This would match against code such as:

```
(list a)
```

where the variable A is unbound.

Error Cliche Name: *Not Quoting an Atom*

Surface Code Segment: One Arg

Criteria: One of the inputs satisfies the subcliche *Not Quoting an Atom Subcliche.*
The object expects input of atom type or input of any type.

Other Checks: None.

An example of this would be:

```
(atom a)
```

where the variable A is unbound.

Error Cliche Name: *Not Quoting an Atom*

Surface Code Segment: Two Args

Criteria: One of the inputs satisfies the subcliche *Not Quoting*

an Atom Subcliche.
The object expects input of atom type or input of any type.

Other Checks: None.

An example of this would be:

```
(cons a '(b c))
```

where the variable A is unbound.

Error Cliche Name: *Not Quoting an Atom*

Surface Code Segment: Three Args

Criteria: One of the inputs satisfies the subcliche *Not Quoting an Atom Subcliche.*
The object expects input of atom type or input of any type.

Other Checks: None.

An example of this would be

```
(subst a b '(a b c))
```

the variables A and B are unbound.

Sub Cliche Name: Not Quoting an Atom Subcliche

Surface Code Segment: Symbols

Criteria: The symbol is unbound.

Other Checks: None.

8. Error Cliche Name: *Quoting a Variable*

Surface Code Segment: Quote

Criteria: The input to quote is an bound atom.
The input is not T or NIL.

Other Checks: The value of the atom is of the right type for the calling function.

An example of the code the above cliche would match against is:

```
(let ((a '(x y z))
  (car 'a))
```

9. Error Cliche Name: *Quoting a Function Call*

Surface Code Segment: Quote

Criteria: The input to quote is a list in which the first atom is the name of a defined function.

Other Checks: None.

By first atom we mean the atom we would reach first if we were to move depth first through the list. This means that this cliche will match against Lisp code such as:

```
(car '(((cons a (b c d)))))
```

10. Error Cliche Name: *Wrong Number of Arguments to a Function Call*

Surface Code Segment: CL Function

Criteria: The wrong number of arguments have been given to the function.

Other Checks: None.

An example of where the above would match is:

```
(expt 2 3 4)
```

Error Cliche Name: *Wrong Number of Arguments to a Function Call*

Surface Code Segment: Function Application

Criteria: The wrong number of arguments have been given to the function.

Other Checks: The function is defined.

An example of where the above would match is:

```
(my-add 1 3 2 4 5)
```

where MY-ADD is defined as:

```
(defun my-add (x y)
  (+ x y))
```

11. Error Cliche Name: *Wrong Type of Argument Given to a Function Call*

Surface Code Segment: Any-arg

Criteria: One of the arguments satisfies the subcliche *Wrong Type Argument Subcliche.*

Other Checks: None.

An example is:

```
(append 1 2)
```

Error Cliche Name: *Wrong Type of Argument Given to a Function Call*

Surface Code Segment: One-arg

Criteria: The first argument satisfies the subcliche *Wrong Type Argument Subcliche.*

Other Checks: None.

An example is:

```
(car 4)
```

Error Cliche Name: *Wrong Type of Argument Given to a Function Call*

Surface Code Segment: Two-args

Criteria: Either the first or the second argument satisfies the subcliche *Wrong Type Argument Subcliche.*

Other Checks: None.

An example is:

```
(expt '(a b c) '(d e f))
```

Error Cliche Name:	*Wrong Type of Argument Given to a Function Call*
Surface Code Segment:	Three-args
Criteria:	Either the first, second or third argument satisfies the subcliche *Wrong Type Argument Subcliche.*
Other Checks:	None.

An example is:

```
(subst 1 2 3)
```

Sub Cliche Name:	*Wrong Type Argument Subcliche*
Surface Code Segment:	Lisp Object
Criteria:	The type of the expected output of the object does not match the input expected by the super cliches object.
Other Checks:	None.

Sub Cliche Name:	*Wrong Type Argument Subcliche*
Surface Code Segment:	Car
Criteria:	The type of the CAR of the object's input does not

match the input expected by the super cliches object.

Other Checks: None.

The sub cliche above extends the *Wrong Type Argument* error cliche to trap errors where the CAR of a flat list (that is a list of depth 1 eg. (a b c d)) is given to a function that expects a list, as in:

```
(defun my-fun (x y)
  (append (car x) y))

(my-fun '(3 4 5) '(4 5 6))
```

the above was seen frequently in the experiment (see part III). This error cliche could be extended further by including a *flat list function.* By flat list function we mean any function that does not increase the depth of the list given. APPEND, REVERSE and CDR are examples of flat list functions. The error cliche would then match against errors where a flat list is manipulated by flat list functions before being passed to the CAR function then to a function that expects a list. So for example if MY-FUN were:

```
(defun my-fun (x y)
  (append (car (reverse (cdr x)) y))
```

would match against the error cliche but:

```
(defun my-fun (x y)
   (append (car (list (cdr x)) y))
```

would not.

12. Error Cliche Name: *Wrong Scope*

Surface Code Segment: Symbol

Criteria:	The symbol is unbound. The name of the symbol is the same as one of the parameters of the function that called this function.
Other Checks:	The name of the symbol is not the same as any of the parameters of the function the unbound symbol appears in.

An example of where the above error cliche would match is:

```
(defun wrong-scope1 (a b)
  (append (wrong-scope2) (wrong-scope2))

(defun wrong-scope2 ()
  (list a b))
```

9. PRESENTING THE TUTORIAL

This module of ITSY has two sub-modules: one to highlight the relevant section of code and the other to create the frames that explain both the source of and the concept behind the error.

9.1 Highlighting the Code

This sub-module is given the matching surface plan segment. The segment contains several pieces of information which are used in highlighting the code, in particular:

a) the Lisp code that the segment represents,

b) the function (if any) that the code is part of.

The highlighting process takes 6 steps.

1. The function object containing the segment is retrieved from the table. Each function object contains a list of all the sub-objects and the segments of code they represent. This is used in the next step.

2. The number of preceding surface plan diagram segments that represent identical sections of code is noted. This is needed because the student's code in the editor buffer is stored as a string. ITSY will search for a particular substring and the code may contain several identical substrings.

3. The file that contains the function is loaded into the editor. If the file is already loaded then the file's buffer is made the current buffer (that is, displayed in the editor window).

4. The function is then moved to the top of the editor window.

5. The third step is to find the position of the student's code inside the editor buffer. ITSY uses a builtin editor tool which will search for a given string. This step is complicated by

the fact that inside the editor the student's code is represented by a string, while ITSY analyses the internal representation of the code which is a list. This means that ITSY's version of the student's code will not have the white space characters. So, for example, ITSY would have the string

"((cons (car x) (cddr x)))"

but the student may have

"((cons
(car x)
(cddr x)))"

inside the editor buffer. The highlighting module overcomes this difficulty by breaking up the string into smaller units and searching for these ignoring the whitespace inbetween. The units consist of the elements of the string that cannot be broken up, for example, variable and function names.

6. Once the position of the student's code is found existing editor functions are used to highlight the code.

If an error occurs at Lisp toplevel the code is not highlighted though of course it remains visible on the screen. It was decided that it was not necessary to do this because code typed at toplevel is usually only one line long.

9.2 Explanation of Errors and Concepts

This submodule receives the following information:

a) the name of the matching error cliche,

b) the object representing the matching surface plan segment,

c) some information specific to the error cliche, such as some extra code,

d) whether the matching code was inside a COND or not,

e) whether the matching code occurred at toplevel or not.

If the matching code appears at toplevel it is usually not highlighted. The exception to this is if the matching code appears within the body of a function defined at toplevel. The function may have been defined some time ago and may not be visible on the screen. In order for the student to see exactly what segment of code the messages are referring to, ITSY prints the function definition into an editor buffer and then highlights the matching code.

9.2.1 The Explanation Frames

Once the code has been highlighted ITSY confirms the diagnosis of the error and explains the error/concept using an *explanation*. An *explanation* consists of a set of five *explanation frames*. Each one of the five frames has two parts: a message and a menu. The menus enable movement around the composite structure. Each message contains slots for context specific details like function and variable names and segments of buggy code.

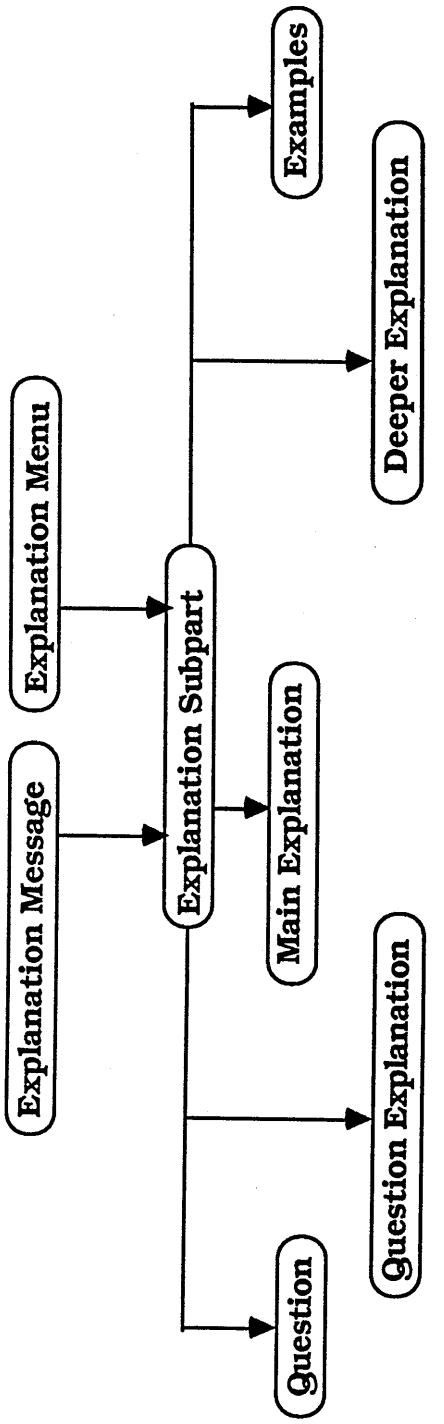


Figure 9-1

These five frames are:

1. A question to confirm ITSY's diagnosis.

2. A rewording of the question. It is not possible to guarantee that a student will understand the question in its initial form. It is vitally important that students understand the question, because if students answer the question incorrectly, then either they will receive a tutorial on a different error to the one that just occurred, or they will receive no tutorial when they need one. Great care has to be taken with any explanation - subjects were misled by the Lisp interpreter's error messages during the pilot study (see chapter 5).

3. This is the first explanation that the students see. From here they can move to 4, 5 and 6 (in any order). This frame is an explanation of the error in terms that a Lisp novice can understand. In the pilot study (Chapter 5) subjects did not understand error messages such as "pdl - overflow". This frame can be viewed as giving an 'English' version of the error message.

4. This is an explanation of the error in terms of the evaluator. There is evidence (from the pilot study (see chapter 5)) that novices do not understand the evaluator. It is hoped that this explanation will help explain how the evaluator works.

5. A set of labelled concrete examples illustrating the correct way to apply the concept. Kahney and Eisenstadt [1982] have shown that analogical mapping between examples and new problems is not easy. In this frame, the mappings are explicitly shown.

The explanation frames were designed using the following principles:

a) Each explanation part has the same structure - this reduces the amount that the student has to learn and limits confusion.

b) The student is able to choose exactly how much of the tutorial s/he wants to see.

c) The frame must obscure the student's code as little as possible.

9.2.2 The Message Controller

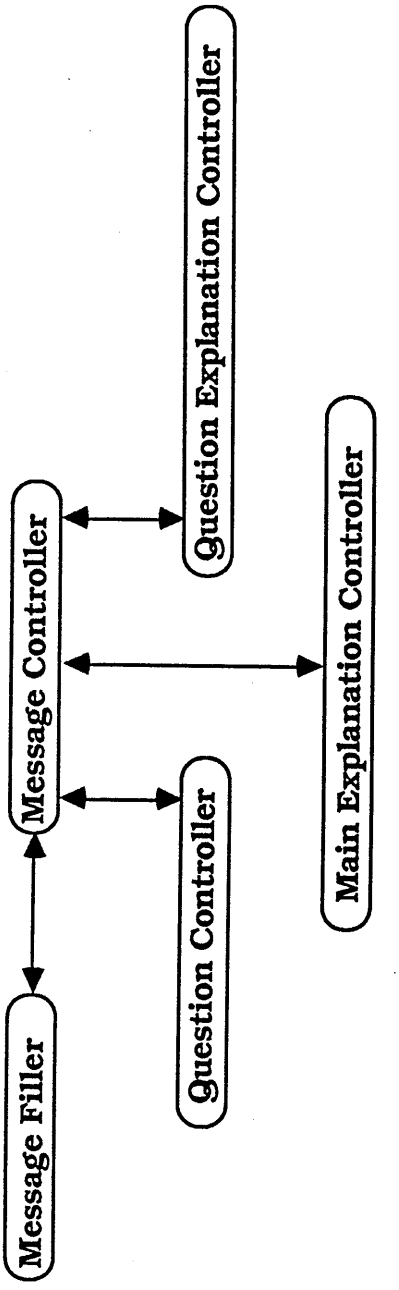


Figure 9-2

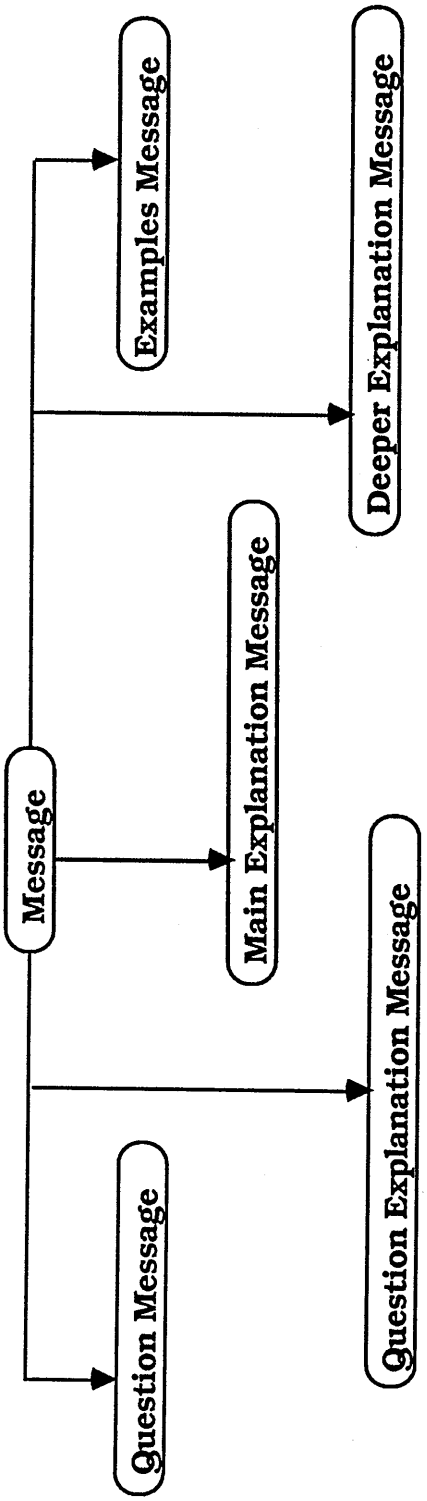


Figure 9-3 Message Hierarchy

The overall structure of the frames and messages can be seen in diagrams 9-1 and 9-3. As described earlier an explanation is made of five parts (diagram 9-1). Each part is made up of two subparts, a message window and a menu (diagram 9-1). The actual messages are stored in message objects, these have a similar overall structure to the frames and are described in more detail in section 9.2.3.

The overall control structure of this module can be seen in diagram 9-2. The message controller is responsible for the overall control of this module. The message filler fills in the slots in each of the messages and then recomputes the size and position of each of the five frames, so they are large enough to display the message string, then prints the five messages in each of the five message windows. The frame controllers are responsible for recomputing the items in a frame's menu and then displaying the frame.

The message controller first passes control to the message filler, this is described further in section 9.2.3. Once the message filler has filled and positioned the five frames, control passes back to the message controller. The message controller then interacts with each of the frame controllers. The message controller will instruct one of the frame controllers to display itself. The message controller passes a list of the frames already displayed on the screen. The frame controller uses this to alter the items in the menu. When an item is selected from a frame's menu the frame controller passes back the name of the item selected. The message controller uses this to determine which frames to activate and which to remove.

Each menu has an item that always appears. This is the *explain a term* item. This option gives the student an explanation of any term used in the explanation of an error. Choosing this item brings up a menu of all the terms used in tutorials. Each of the term explanations are stored in a hash table, under the name of the term. When a term is chosen the explanation is retrieved from the table and displayed in a window. The window is removed when any key is pressed.

Each menu except for the two question menus have a *cancel* item. Choosing this ends the tutorial returning the student to the Lisp toplevel, and all the frames currently displayed are removed. This ensures that the student cannot get stuck.

The other items in the menu are determined by the frame's controller. The frame

controllers for the question, question explanation and main explanation place the frame on the screen and position the mouse within the menu. The frame controllers for the deeper explanation and example frames, in addition to placing the frames on the screen, also add to the menu any of the two frames that are not yet displayed. To enable the frame controller to do this the message controller passes a list of all the frames currently displayed on the screen. The frame controller passes back to the messages controller the item chosen from the frame's menu.

9.2.3 The Explanation Text

This module is responsible for printing the messages in the message windows of the five frames. The message slots are stored in a hierarchy of objects, these objects are shown in figure 9-3. Each message object contains five sets of strings. Each string contains holes that are filled in with information specific to the error. Each error cliche has at least one message object. The hierarchy exists for two reasons. Firstly, some of the error cliches need the same messages in some of their frames. Secondly, the same error cliche needs slightly different messages in certain conditions.

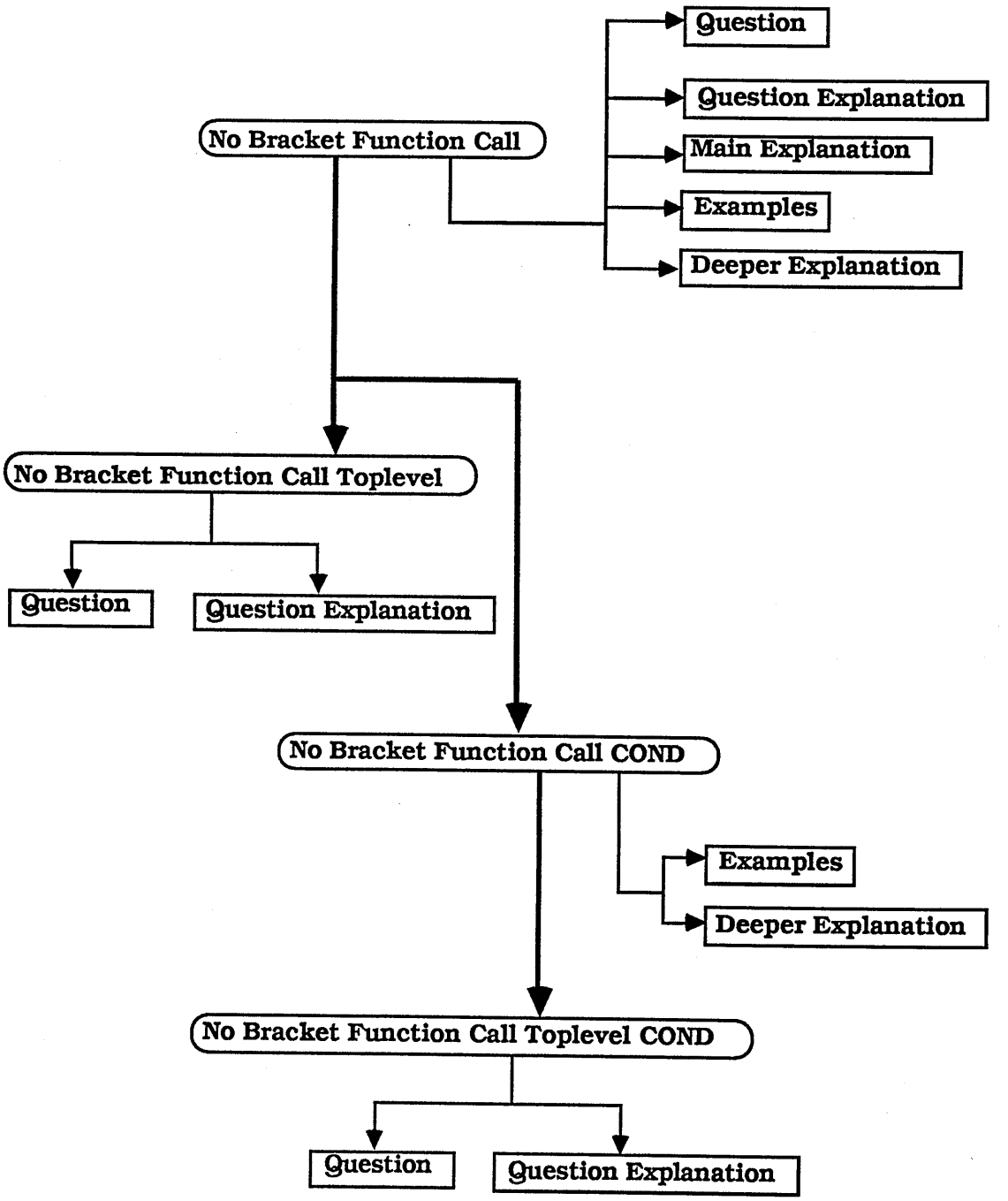
The first reason can best be explained using an example. Some errors are treated differently if they occur inside a COND. There are two error cliches corresponding to the error of not placing a bracket around a function call. The first, called *no bracket function call,* matches against sections of code where a function call does not occur within a set of brackets, and where this happens outside a COND. The second, called *no bracket function call in COND,* matches against the same sections of code when they occur inside a COND. The Question, Question Explanation and Main Explanation frames of these two error cliches are the same but the Deeper Explanation and Examples frames are different. The reason for this is that the Question and the Question Explanation are used to confirm ITSY's diagnosis; this is the same for both error cliches. Similarly the Main Explanation, which gives a version of the error message that students can understand, is the same. The Examples frame for the error cliche *no bracket function call* contains examples showing the application of a function outside a COND. The Examples frame for *no bracket function call in COND* shows the application of functions within the test and the action part of a COND clause.

The second reason for having a hierarchy is that in different circumstances the same

error cliche will lead to slightly different messages. These differences are mainly cosmetic in nature enabling the messages to read better. Each error cliche has at least two Question and Question Explanation frames. This is because these two frames refer to the highlighted line in the editor screen whenever the error occurs in code that is within a file, the line of text that refers to the highlighted line is left out if the error occurs at toplevel.

The hierarchy for the two error cliches *no bracket function call* and *no bracket function call COND* is shown in figure 9-4. The thicker lines show the inheritance structure and the thinner lines show the text for the message frames linked to the tutorial object.

Figure 9-4



There is a message filler for each different type of message object. A message object is first processed by its own message filler then passed up to each of its ancestors' message fillers in turn. This means that the holes in each of the five message strings are filled in 'bottom up'. So the message object for *No Bracket Function Call COND* at toplevel would

first have the Question and Question Explanation message holes filled in then the Examples and Deeper Explanation message holes would be filled. A hole is only filled once, so that if it is already filled the message filler ignores the hole. After the holes of a message have been filled the size of the corresponding explanation frame is changed and the message is printed in the frame's window. Once all five messages have been filled in and printed onto the five frames' message windows, the five frames are positioned so as not to overlap and the Question frame is displayed.

9.2.4 An Example of an Explanation Being Displayed

The example below shows how the messages for the explanation of the error cliche *No Bracket Around a Function Call in a COND.* The Question, Question Explanation and Main Explanation messages are taken from the explanation of the error cliche *No Bracket Around a Function Call.* The Deeper Explanation and Examples frames are filled in from the explanation of the error cliche *No Bracket Around a Function Call in a COND.*

The holes are represented by a tilde followed by the character 'a'. This is similar to the control strings used by the Common Lisp FORMAT function. Other special characters (such as newline characters and characters for highlighting segments of text) have been left out for readability.

Messages Taken From No Bracket Function Call

Question

" In your function ~a does ~a in
the highlighted line refer to the
function ~a rather than the variable ~a?"

Question Explanation

" Did you want to call the function ~a in the
highlighted line rather than get the value
of the variable ~a?"

Main Explanation

" The interpreter thinks that you want the
value of the variable ~a rather than call
the function ~a."

Messages Taken From No Bracket Function Call in COND

Deeper Explanation

" The syntax in COND is slightly different. There is usually
a double opening bracket after the word COND itself because
what follows in that position should be a list containing
both a test and an action. The test may consist of a function
call (usually a predicate such as ATOM or NULL), and it is
this function call which causes the extra opening bracket:

(COND ((NULL L) <action>)
....)

or the test may simply use the value of a variable, as in

(COND (VAR <action>)
...)

where VAR has either a NIL vale or a non-NIL value and there
is only one bracket after the word COND.
Naturally, it is possible to make a similar mistake by
putting too many brackets around the <action>."

Examples

"(COND ((NULL L) NIL)
first clause
((ATOM L) L)
second clause
(T (CDR L)))
third clause

```
(COND     ((NULL L)    NIL))
          test      action
          ((ATOM L)     L))
          test      action
          (T          (CDR L)))
          test      action"
```

This message object is passed the name of the function the error occurred in and the name of the function which ITSY suspects the student meant to call. If the error occurred inside a function named EXAMPLE-FUNCTION and the student should have called the function LIST then the two Confirmation frames would appear as in figure 9-5 and the three other frames would appear as figure 9-6.

Figure 9-5

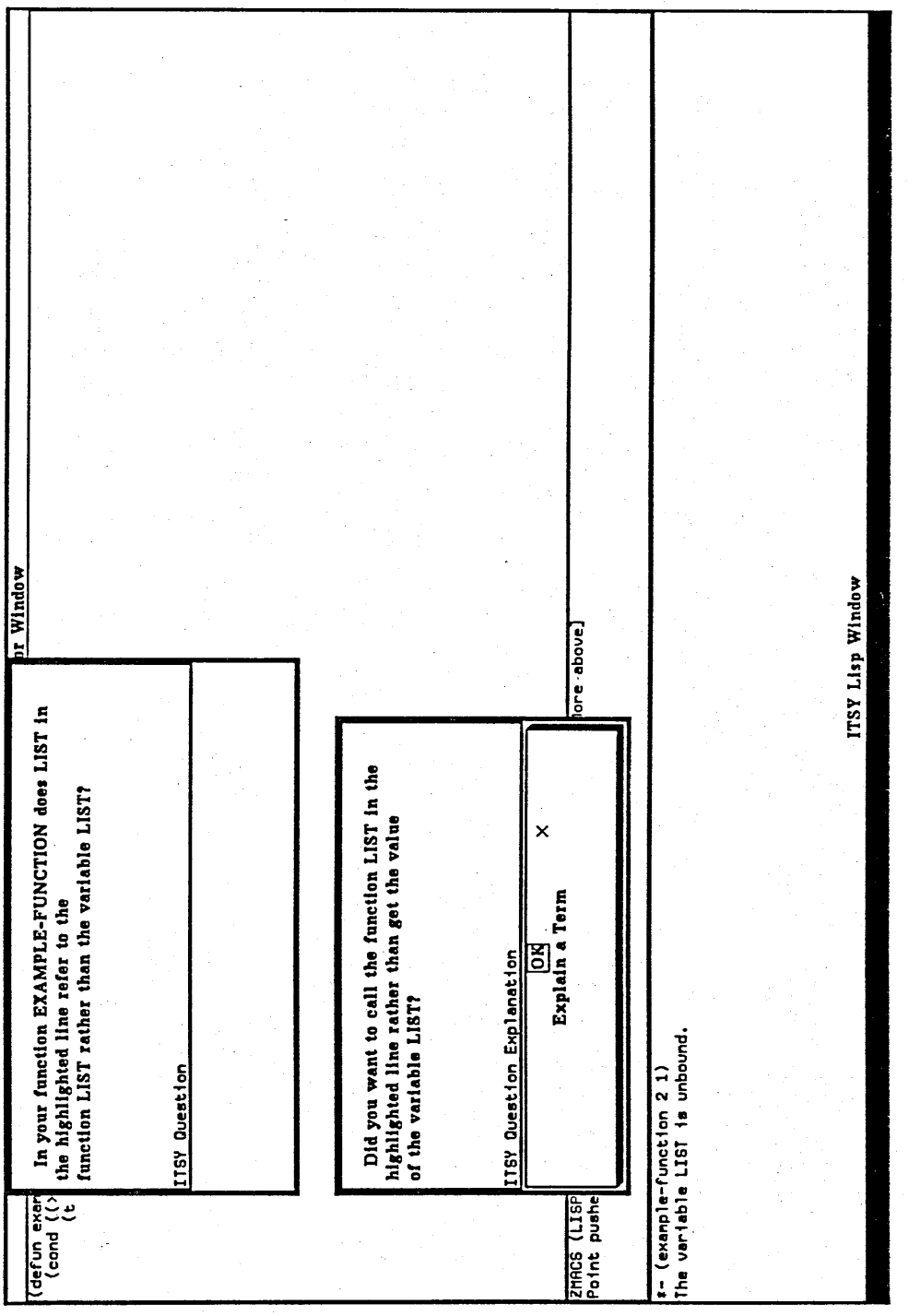
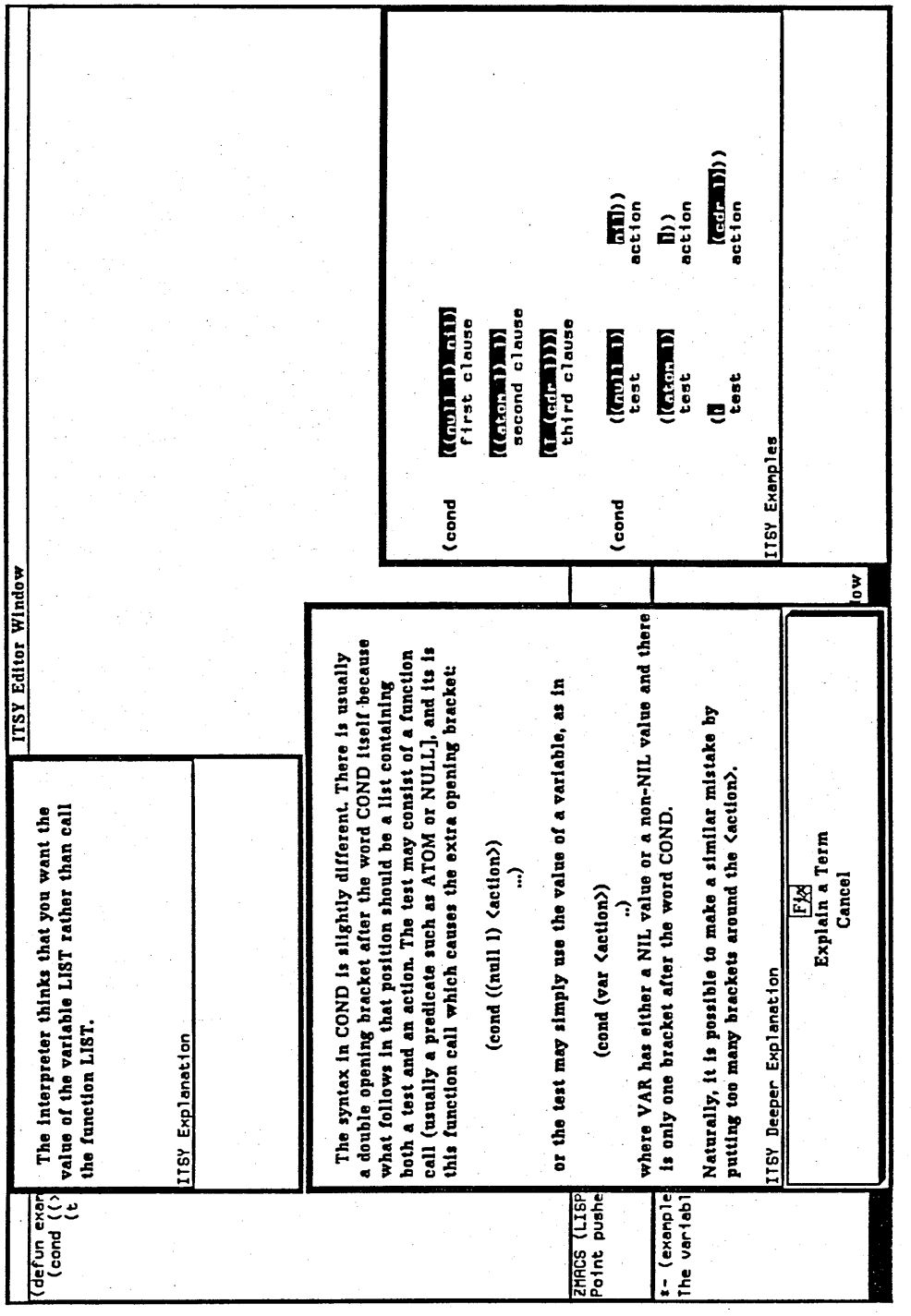

In your function EXAMPLE-FUNCTION does LIST in the highlighted line refer to the function LIST rather than the variable LIST?
ITSY Question
Did you want to call the function LIST in the highlighted line rather than get the value of the variable LIST?
ITSY Question Explanation
OK
Explain a Term
ITSY Lisp Window
The variable LIST is unbound.

Figure 9-6


ITSY Editor Window
The interpreter thinks that you want the value of the variable LIST rather than call the function LIST.
ITSY Explanation
The syntax in COND is slightly different. There is usually a double opening bracket after the word COND itself because what follows in that position should be a list containing both a test and an action. The test may consist of a function call (usually a predicate such as ATOM or NULL), and its is this function call which causes the extra opening bracket:
(cond ((null l) <action>)
...)
or the test may simply use the value of a variable, as in
(cond (var <action>)
...)
where VAR has either a NIL value or a non-NIL value and there is only one bracket after the word COND.
Naturally, it is possible to make a similar mistake by putting too many brackets around the <action>.
ITSY Deeper Explanation
Explain a Term
Cancel
(cond
first clause
second clause
third clause
test
action
ITSY Examples

9.3 Conclusions

The explanation philosophy used in the design of this module was developed from the pilot study. Subjects in this study had great difficulty in understanding *any* of the messages that the Lisp system displayed. This is one of the reasons for the existence of the reworded question frame. One of the problems when asking students questions about their code when it contains a bug is that this is when they will have most difficulty in understanding the question. If students were easily able understand the questions, they would not produce buggy code. It is important that students understand the questions posed by ITSY, otherwise they may miss a tutorial that they need. Subjects misunderstanding messages caused errors in the pilot study. For example, when the error message for a file not loading because it did not contain enough closing brackets appeared the subjects did not even know an error had occurred and that the new versions of their functions had not been loaded. The subjects then attributed any errors caused by the existing functions to the new versions of the functions. This error message is:

(read-eof #file-in |RS<R.>|)

The rest of the frames are designed to explain, at different levels, the reason why a section of code caused an error. The main explanation frame is really what the Lisp error message should be. The Lisp error message is given is usually at a lower level than the message given by ITSY. This is best illustrated by the following buggy sections of code. Below each section is the Lisp error message given followed by the main explanation message.

```
(append (a b c) (d e f))

Lisp Message:  A undefined function

ITSY Message:  The interpreter thinks that you want to call the
               function A instead of giving the list (A B C) as an
               argument
```

```
(let ((a '(x y z)) (b '(1 2 3))
  (append (a) (b)))

Lisp Message:  A undefined function

ITSY Message:  The interpreter thinks that this A is a function
```

There is evidence from the pilot study that the subjects did not understand how the Lisp evaluator worked. One of the pieces of evidence was that the subjects have no rule to determine whether an s-expression should be quoted or not. They would often add and subtract quotes on a random basis. The deep explanation frame was designed to try and give students an understanding of the evaluator by explaining, in terms of the evaluator, why the error occurred.

The examples frame was designed to give the students a concrete example of 'how to apply the concept correctly'. This balances the deep explanation frame which describes the reason behind the error in abstract terms.

10. THE STUDENT MODEL

10.1 Introduction

Student models are used to determine how student input should be interpreted. In ITSY the task of the student model is to determine whether the student should receive a tutorial once the cause of an error has been found. Because of this the student model is closely tied to the error cliches.

The student model consists of a graph. Each node in the graph represents a LISP concept associated with each of the error cliches in the library, that is to say there is a node in the graph for each error cliche. An error cliche matches against a segment of code that contains an error. Behind this error is a Lisp concept that the student did not understand, otherwise the student would not have made the error. There is a node in the student model that represents how well the student understands this concept. For instance, a type of error that students make is to try and use a lambda variable declared in the argument list of a function. A student could type the following:

```
(defun foo (a b c)
  (setq a 1)
  (setq b 2)
  (setq c 3))

(defun bar ()
  (plus b c a)) ; a, b, and c are NOT bound
                ; here
```

Associated with this type of error is the error cliche *Wrong Scope.* The student model will have a node to associated with this error cliche.

Each node has four states:

1. Concept has not yet been encountered.

2. Concept has been seen but not learnt.

3. Concept has been partially learnt.

4. Concept has been fully learnt.

All of the nodes are initially in state one. When a student sees a concept for the first time, the corresponding node changes state. If the student successfully applies the concept, the node moves to state 3, otherwise the node moves to state 2. For example, if the first s-expression a student typed in was:

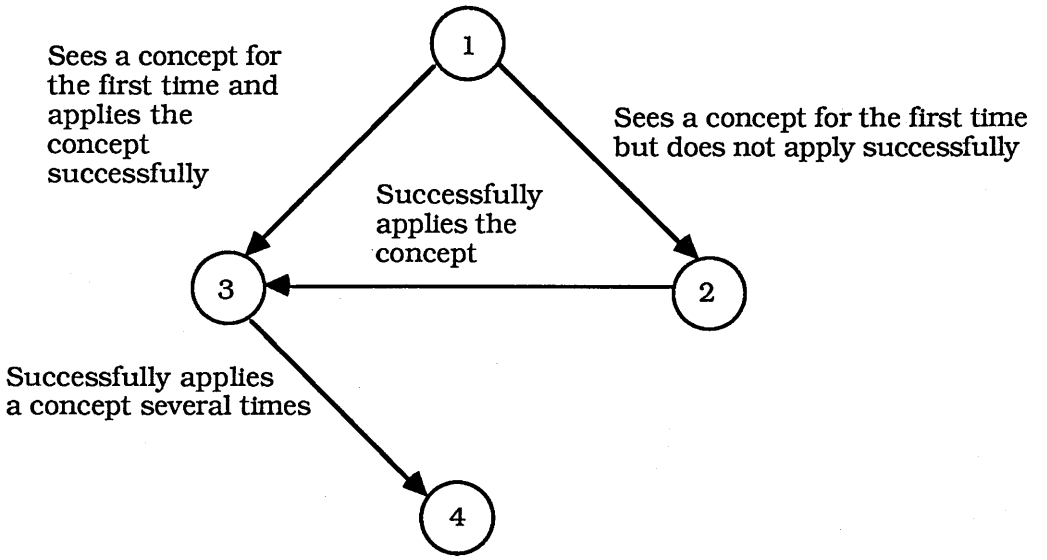
```
(car 1)
```

The node for the error cliche *Argument of the Wrong Type* would move from state 1 to state 2, but if a student had typed in:

```
(car '(a b c))
```

The node would have moved from state 1 to state 3. The transition paths are described in figure 10-1 below.

Figure 10-1



10.2 Representation

The nodes are represented by objects. Each object holds the current state of the node. The objects are stored in a hash table under the name of the error cliche. Each node has its own upgrade and downgrade handlers which alter the state of the node. The node's new state depends on the original state and what just happened.

10.3 Updating the Model

Updating the model is carried out in a similar way to matching an error cliche against a student's code. The transformed code, that is the student's code in surface plan form, (see Chapter 7) is traversed in a similar way to the code traverser used by the error cliche finder. There are two main differences. Firstly, *student model cliches* instead of error cliches are matched against the transformed code. Secondly, the whole graph of objects is traversed rather than stopping when a student model cliche matches.

The four states (see 10.1) are represented by the integers 1 - 4. The link between states 3 and 4 is achieved by adding an increment (less than one) to the current value. The current state is then the current value rounded down. The smaller the increment the

more times a student has to successfully apply the concept. Currently the value is 0.1 i.e. it takes 10 goes to advance from 3 to 4.

10.3.1 Student Model Cliches

Student model cliches are derived from error cliches. An error cliche matches a segment of code that contains a certain type of error. The type of error corresponds to a Lisp concept that the student does not understand. It is this concept that links a student model cliche to an error cliche. A student model cliche matches against any segments of code that show that the student understands this concept. Consider the error cliche *No Bracket Function Call*:

Error Cliche Name:	*No Brackets Around a Function Call*
Surface Code Segment:	Symbols
Criteria:	The symbol is unbound. The symbol is the name of a function.
Other Checks:	None.

the associated student model cliche is:

Student Model Cliche Name:	*No Brackets Around a Function Call*
Surface Code Segment:	CL Function
Criteria:	None.
Other Checks:	None.

You might think that the student model cliches could be just 'correct code' cliches (as they are used in the Programmer's Apprentice [Waters, 1985]), but they are not because they are the correct code of which the error cliche is an erroneous form. In other words they apply to places where the student *could* have made an error, but did not.

Below is a description of a *Null Student Model Cliche.* These student model cliches are an artifact of the hierarchy of Lisp Objects described in 7.1.2. Each student model cliche can not only match against objects the same type as specified in the Surface Code Segment but all the children of this type. Null Student Model Cliches are used to prevent a child of a type of object inheriting the student model cliche.

A Null Student Model Cliche does not have criteria or other checks slots. This describes exceptions to student model cliche rules. This means that the surface code segment which is called quote cannot match against student model cliche *No Brackets Around a Function Call.* Unless specified it can be assumed that every Lisp object has a Null Student Model Cliche. The equivalent of these (Null Error Cliches) also exist, but they were not shown in Chapter 8 in order to make the error cliches readable.

Null Student Model Cliche Name:	No Brackets Around a Function Call
Surface Code Segment:	Quote

Objects of type *Quote* inherit from the type *CL Function.* If it were not for this Null Student Model Cliche every time a student quoted an a Lisp s-expression the node corresponding to the error cliche *No Brackets Around a Function Call* would change state. This node should only change state when a student correctly applies a function.

Student Model Cliche Name:	No Brackets Around a Function Call
Surface Code Segment:	Function Application
Criteria:	The type of function application is normal or recursive
Other Checks:	None

The complete set of student model cliches are in appendix K.

10.3.2 Action Taken on Different Values of the Student Model

Whenever ITSY traps a student error the student model node associated with the error cliche is checked. The value of the node determines ITSY's next action. It was decided that the student should always have access to a tutorial, in case the model was inaccurate. The action carried out under each value is described below.

1. Concept has not yet been encountered.

The tutorial is given.

2. Concept has been seen but not learnt.

Extra help is given. This is not actually implemented yet. The extra help would be provided in the form of extra frames.

3. Concept has been partially learnt.

The student is asked if s/he wants a tutorial or not.

4. Concept has been fully learnt.

No tutorial is given, but a *present tutorial* option is added to the Lisp menu and the student notified how to obtain a tutorial. The *present tutorial* option would remain until the next student input to the Lisp toplevel window.

PART III

This section describes two studies. These studies were carried out for two reasons. Firstly, it was decided that novices did not generate really interesting errors in their first 20 hours of Lisp. Secondly, it would provide a way of evaluating ITSY and point to areas of weakness or for future research.

Six subjects were used in the studies. Each subject completed approximately 30 hours learning Lisp[1]. As in the first study the subjects sat at a terminal reading and completing exercises from Winston and Horn's "Lisp" [Winston & Horn, 1984].

Two subjects were used in the first study. They were used to "iron out the bugs" in the implementation. Because ITSY was buggy at this stage only the subject's errors are presented. The first two subjects had relatively little experience of computer programming.

The remaining four subjects had more programming experience. Three of the subjects had at least two years' experience in assembler programming. The fourth was an adviser for an academic computing service. The subjects' errors and a summary of ITSY's responses are presented.

[1] One subject only completed 23 hours

11. STUDY II: A PRELIMINARY EVALUATION OF ITSY

11.1 Objectives

As discussed above the objective of this preliminary study was to "iron out the bugs" in ITSY and check that there was nothing fatally wrong with either the design of the study or the implementation of ITSY. In fact several changes were made to ITSY after this preliminary evaluation.

11.2 Methods

Both the subjects used were non-programmers. As in the first study the subjects sat at a terminal reading from Winston and Horn's book "Lisp" (second edition) and attempting the exercises. If they had any problems I would help them using the keyboard. This ensured that all the interactions were recorded.

As in the first study, when the subjects arrived they were given a short tutorial. This tutorial covered the editor, the Lisp toplevel and the message frames. This tutorial involved going through the handouts shown in appendix D. The handouts give a summary of the operations available in the editor and at Lisp toplevel, and an overview of the type of explanation each message frame gives.

The interactions were recorded using a modified dribble function. In addition to recording input and output in the Lisp window extra information such as selections from menus and the amount of time spent looking at each message frame was also recorded.

11.3 Method Of Analysis

As in the first study errors have been classified according to the cause of the error. Each time a bug caused an error it was counted, so if the student typed in the same erroneous form over and over each evaluation was counted as a separate error.

Every application of an algorithmically incorrect function has been counted as an incorrect algorithm error, regardless of whether the actual result was correct or not. So, for example

```
(defun my-max (x y)
  (cond ((> x y) x)
        ((< x y) y)))
```

would be counted as an error every time it was applied because the function does not cater for the case when X and Y are equal.

11.4 Results

As in the first study all of the categories containing more than 1.5% of the errors are presented here (the individual totals are contained in appendices E and F).

11.4.1 Problems Caused By The Environment

No environmental errors due to the computing environment were recorded. The only times subjects became stuck were when ITSY crashed and had to be restarted. Subjects were conservative in the use of the available tools such as the movement and editing keys in the editor and the Lisp toplevel. More details were recorded in the next study.

11.4.2 Algorithmic Errors

1. Incorrect algorithm. As ITSY currently does not trap this class of errors they have been lumped together.

Percentage of the total number of errors: 30

11.4.3 Problems With The Language

1. Wrong number of arguments given to a function, because the arguments are in the wrong form. An example of this is:

```
(expt (3 4))
```

Percentage of the total number of errors: 2

2. Incorrectly putting a pair of brackets around an atom.

Percentage of the total number of errors: 4

3. Calling an undefined function.

Percentage of the total number of errors: 3

4. Incorrectly putting an extra set of brackets around a function call.

Percentage of the total number of errors: 4

5. Not putting brackets around a function call.

Percentage of the total number of errors: 5

As in the first study categories 4 and 5 does not include errors that occurred in a COND form, these have been separated out and are given below.

6. Extra set of brackets around a function call inside a COND

Percentage of the total number of errors: 5

7. Not putting brackets around a function call inside a COND.

Percentage of the total number of errors: 4

8. Not closing the test part of a clause. An example of this is:

```
(defun check-temperature (temp)
  (cond ((> temp 100.00 'ridiculously-hot)
         (< temp 0.00 'ridiculously-cold)
         (< 0.00 temp 100.00 'ok))))
```

Percentage of the total number of errors: 3

9. Not quoting an object that should be quoted.

Percentage of the total number of errors: 8

10. Spurious character in a file. This meant that the student had, either by accidentally typing or by not completely deleting some text, included a spurious character in their file. This leads to an error when the file is loaded.

Percentage of the total number of errors: 2

11. Text spelling error.

Percentage of the total number of errors: 4

12. Trying to give a value to a parameter globally. When subjects found that a parameter to a function was giving an error, for any reason, they sometimes thought this was due to the parameter not having a value. The subjects tried to pass a value to the parameter by SETQing the parameter globally.

Percentage of the total number of errors: 2

13. Unbound variable because of the variable was not declared in the parameter list. Subjects would sometimes not declare a variable in a parameter list. This could have been because they did not understand the scoping rules of Common Lisp [Steele, 1984]. Subjects would sometimes write code that would only work in a dynamically scoped Lisp such as MacLisp [Pitman, 1983], for example:

```
(defun s-fun (x y)
  (+ y (double)))

(defun double ()
  (setq x (* x 2)))
```

x is unbound in the function double - this would not be the case in MacLisp. It should have been declared in the parameter list , as in:

```
(defun s-fun (x y)
  (+ y (double x)))

  (defun double (x)
  (setq x (* x 2)))
```

Percentage of the total number of errors: 2

14. Wrong number of arguments given to a function

Percentage of the total number of errors: 6

15. Wrong type argument of argument given to a function

Percentage of the total number of errors: 3

The above does not include non-list arguments given to CONS, CAR, CDR and APPEND these are given below.

16. Non-lists to one of CONS, CAR, CDR and APPEND

Percentage of the total number of errors: 3

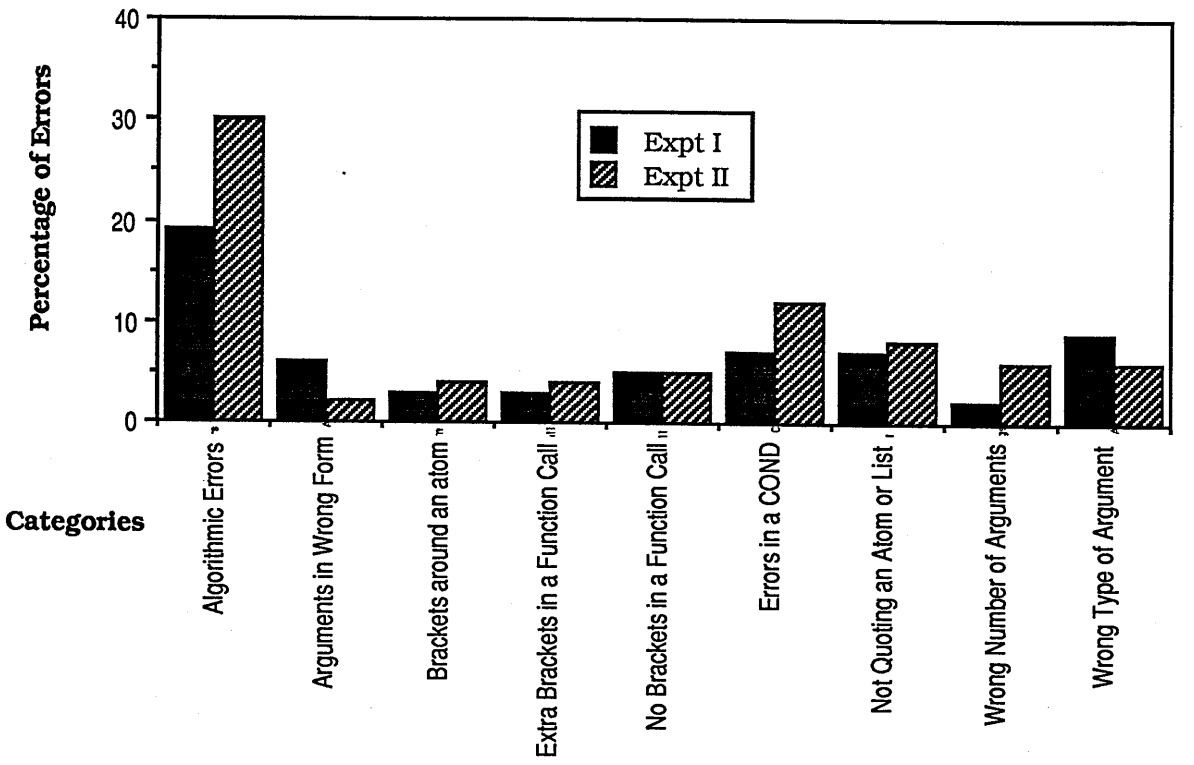
11.5 Conclusions

89% of the errors fell into sixteen categories above.

11.5.1 Comparisons with Study I

The following graph shows a comparison of nine of the different categories.

Figure 11-1 Comparison Graph Between Study I and Study II



The greatest difference between the two studies occurs in the Algorithmic Error category. The percentage of algorithmic errors has grown from 19 to 30. This is for three reasons. Firstly, the subjects were not experienced programmers. Secondly, the subjects spent more time learning Lisp and were writing more complex programs. This gives greater scope for making algorithmic errors. Thirdly, even though ITSY was buggy at this point, it was able to help the subjects with some of their problems with the language.

Subject	Total Lines Input	Number of Errors	Percentage of Errors / Line Input
C	526	297	56
A3	193	75	39
Total	719	327	45

Above is a table showing the percentage of errors per line of input (these are given for Study I in section 5.4). The average percentage of errors per line of input has increased from 13 to 45. This could be due to one of two reasons:

a) the subjects in the Study I were experienced COBOL programmers whereas the subjects in Study II had practically no programming experience,

b) the increase in length of the study, which meant that the subjects were tackling harder exercises. The harder exercises (especially those covering recursion) meant that the subjects attempts were more likely to contain bugs.

11.5.2 Changes to ITSY

Apart from fixing many small bugs the following changes were made to ITSY as a result of this study.

Adding an extra explanation frame. During this study it became clear that subjects did not always understand the message frames enough to correct the error. This following sequence is taken from the dribble files of subject C.

The subject is attempting Problem 3-1 [Winston & Horn, 1984 p. 43]:

> Some people are annoyed by the names of the critical primitives CAR, CDR, and CONS. Define new procedures OUR-FIRST, OUR-REST, and CONSTRUCT that do the same things. ...

and had defined the function OUR-FIRST as follows:

```
(defun our-first (items) (car (items)))
```

the subject typed in the s-expression

```
*- (our-first '(one two three))
(Error (OUR-FIRST (QUOTE (ONE TWO THREE))) zetalisp-
system:undefined-function #<undefined-function-trap 12273564>) The
function ITEM is undefined.
```

The question frame was exposed with the message:

```
In your function OUR-FIRST does ITEM in the highlighted
line refer to the variable ITEM in the parameter list?
```

The subject selected 'yes' from the menu and the main explanation frame was exposed with the message:

```
The interpreter thinks that this ITEM is a function
```

The subject selected 'examples' from the menu and the example frame was exposed with the message:

```
(+              3 4)
function        arguments
(append         '(a b c) '(d e f))
function              arguments
```

The subject selected 'deeper explanation' from the menu and the deeper explanation frame was exposed with the message:

```
The first object after an opening bracket is usually a
```

```
function, the remaining objects are the arguments to the
function.  The interpreter is following this rule.
```

The subject then typed:

```
*- (items one two three)
```

It is clear from the above that the subject does not understand the message frames. An extra frame was added to the explanation messages - a Fix frame. This shows the student, in general terms, how to fix the error. The Fix message for the above error is:

```
For ITEM to be regarded as a variable you need to remove
the pair of brackets which immediately surround it. Thus
if you wanted the function FOO to have the argument
ITEM, it would be wrong to have (FOO (ITEM)) but correct
to have (FOO ITEM).
```

To show how the Fix frame fits in with the other five the messages. The five messages presented in section 9.2.4 are given below with the Fix message included.

Question

" In your function ~a does ~a in
the highlighted line refer to the
function ~a rather than the variable ~a?"

Question Explanation

" Did you want to call the function ~a in the
highlighted line rather than get the value
of the variable ~a?"

Main Explanation

" The interpreter thinks that you want the

value of the variable ~a rather than call
the function ~a."

Deeper Explanation

" The syntax in COND is slightly different. There is usually a double opening bracket after the word COND itself because what follows in that position should be a list containing both a test and an action. The test may consist of a function call (usually a predicate such as ATOM or NULL), and it is this function call which causes the extra opening bracket:

(COND ((NULL L) <action>)
....)

or the test may simply use the value of a variable, as in

(COND (VAR <action>)
...)

where VAR has either a NIL vale or a non-NIL value and there is only one bracket after the word COND.
Naturally, it is possible to make a similar mistake by putting too many brackets around the <action>."

Examples

"(COND ((NULL L) NIL)
first clause
((ATOM L) L)
second clause
(T (CDR L)))
third clause

(COND ((NULL L) NIL))
test action
((ATOM L) L))
test action
(T (CDR L)))
test action"

Fix

" In order to call a function FOO with arguments X Y
in the test part of a COND (COND (FOO X Y ..
would be wrong but (COND ((FOO X Y) ... would
be correct. In order to call a function FOO with
arguments X Y in the action part of a COND
[with test (NULL L)]
(COND ((NULL L) FOO X Y) and
(COND ((NULL L)) (FOO X Y) .. would be wrong
but (COND ((NULL l) (FOO X Y)) .. would be
right."

It was found that the subjects often left exercises when their solutions still contained algorithmic errors. This was because they believed them to be correct. In order to prevent this a new tool was added to ITSY - *test a function.* Test a function enables a student to try out one of their solutions on a set of prestored examples. ITSY tells the student whether the solution is correct, incorrect or leads to an error. The implementation of this tool is described below.

Stored in a hash table is an entry for each function that the students have to define. Each entry contains a number of *test input / output pairs* and a *test function.* The student's function is applied to a *test input.* One of three things happens when the student's function is applied:

a) The function application gives an error - in this case ITSY tells the student that the function causes an error,

b) The function does not give an error but the output does not match the stored *test output* (when compared using the test function) - in this case ITSY tells the student that the function is not correct,

c) The function does not give an error and the output matches the stored *test output* - in this case ITSY tells the student that the function is probably correct,

The *test function* is necessary because some of the examples do not specify the output exactly. For example, some of the exercises involve the manipulation of sets. In these cases the student's output has to be a certain set (ie. a list where the order of the elements is not important). As an example, here are the test input/output pairs and the test function exercise 4-13 [Winston & Horn, 1984 p 73]

> Define OUR-UNION. The union of two sets is a set containing all the elements that are in either of the two sets. ...

```
Input:          (a b c) (a x y z c)

Output:         (a b c x y z)


Input:          (b c) (a x y z c)

Output:         (a b c x y z)


Input:          (a b c) (d e f a)

Output:         (a b c d e f)


Input:          (a b c) (b x y z c)

Output:         (a b c x y z)

Test Function:  same-setp
```

where same-setp is defined as:

```
(defun same-setp (set1 set2)
  (and (subsetp set1 set2) (subsetp set2 set1)))
```

The *test a function* tool can be seen in section 3.5.

The biggest change to ITSY as a result of this study was the decision to turn the student model off. This decision was taken for two related reasons. Firstly, the model needed "tuning" because the nodes were being bumped up too early, often students would be in the *concept fully learnt* state too early. Secondly, the students were less likely to look at a tutorial if the initial Question frame did not appear, that is if the tutorial were merely an option on the Lisp menu. A possible solution to this problem is described in chapter 13.

This did not affect the subjects as displaying the question takes a relatively small amount of time compared to finding the error and subjects can reply *no* to a Question Frame so cancelling a tutorial.

Extra commands to change the shape of the frames (to have a medium or large editor pane) were added because of comments made by these two subjects. These are described, in detail, in Chapter 7.

12. STUDY III: AN EVALUATION OF ITSY

12.1 Objectives

The objective of this study was to both evaluate ITSY and collect more complicated error cliches.

12.2 Methods

Four subjects were used. Three of the subjects had at least two years' assembler programming experience. The other is a member of an academic computing advisory service and has had over two years experience programming in FORTRAN. The methods used for this study were the same as the second study.

12.3 Method Of Analysis

As in the first study errors have been classified according to the cause of the error. Each error has been classified in one of nine ways, each corresponding to one of nine ways errors were treated by ITSY and the subjects. If ITSY trapped the error the subjects either examined or ignored the tutorial frames. The subjects then either fixed the error, failed to fix the error or left the error tackling a new problem. The nine categories are:

1. The student used ITSY and fixed the error.

2. The student used ITSY and tried but failed to fix the error.

3. The student used ITSY and left the error without trying to fix it.

4. The student did not use ITSY and fixed the error.

5. The student did not use ITSY and failed to fix the error.

6. The student did not use ITSY and left the error.

7. ITSY did not find the cause of the error and the student fixed the error.

8. ITSY did not find the cause of the the error and the student failed to fix the error.

9. ITSY did not find the cause of the error and the student left the error.

12.4 Results

12.4.1 Errors

The results are presented in two graphs. The two graphs are summaries of the table in Appendix F, this table gives the total results for the four subjects. The subjects' individual results are given in Appendix G. The two graphs show all the error categories except the category *incorrect algorithm* which is a singularity. 147 algorithmic errors were made. None of these were caught, 16 were fixed by the subjects, 130 were not fixed and 1 was left.

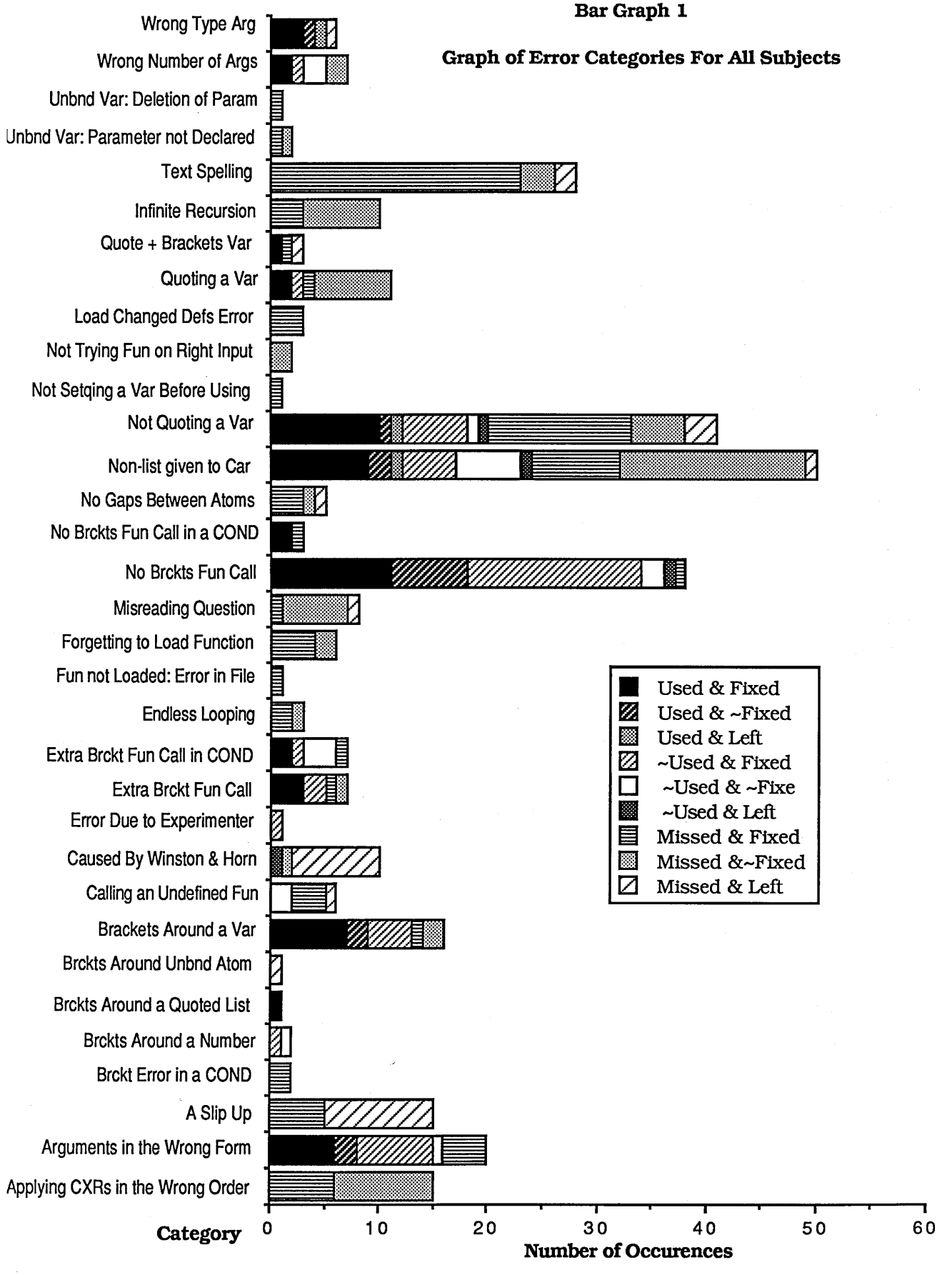
The first bar graph shows the results for each category. The total length of a bar gives the total number of errors in the category. The nine different shadings used in the bars correspond to the nine different categories. The second bar graph (spread over 4 pages) gives the same information but the nine categories have been split up rather than placed on top of one another.

From the first graph it can be seen that the commonest errors fell into the *text spelling, not quoting, non-lists given one of CONS CAR and CDR, no brackets around a function call, brackets around a variable, slip ups* and *giving a function arguments in the wrong form* error categories.

From the second graph it can be seen that when an error was trapped by ITSY the commonest sequence of events was for students to use ITSY and then fix the error. There are two exceptions to this *arguments in the wrong form* and *no brackets around a function call.* For these two categories the commonest occurrence was for the student to not use ITSY and fix the error.

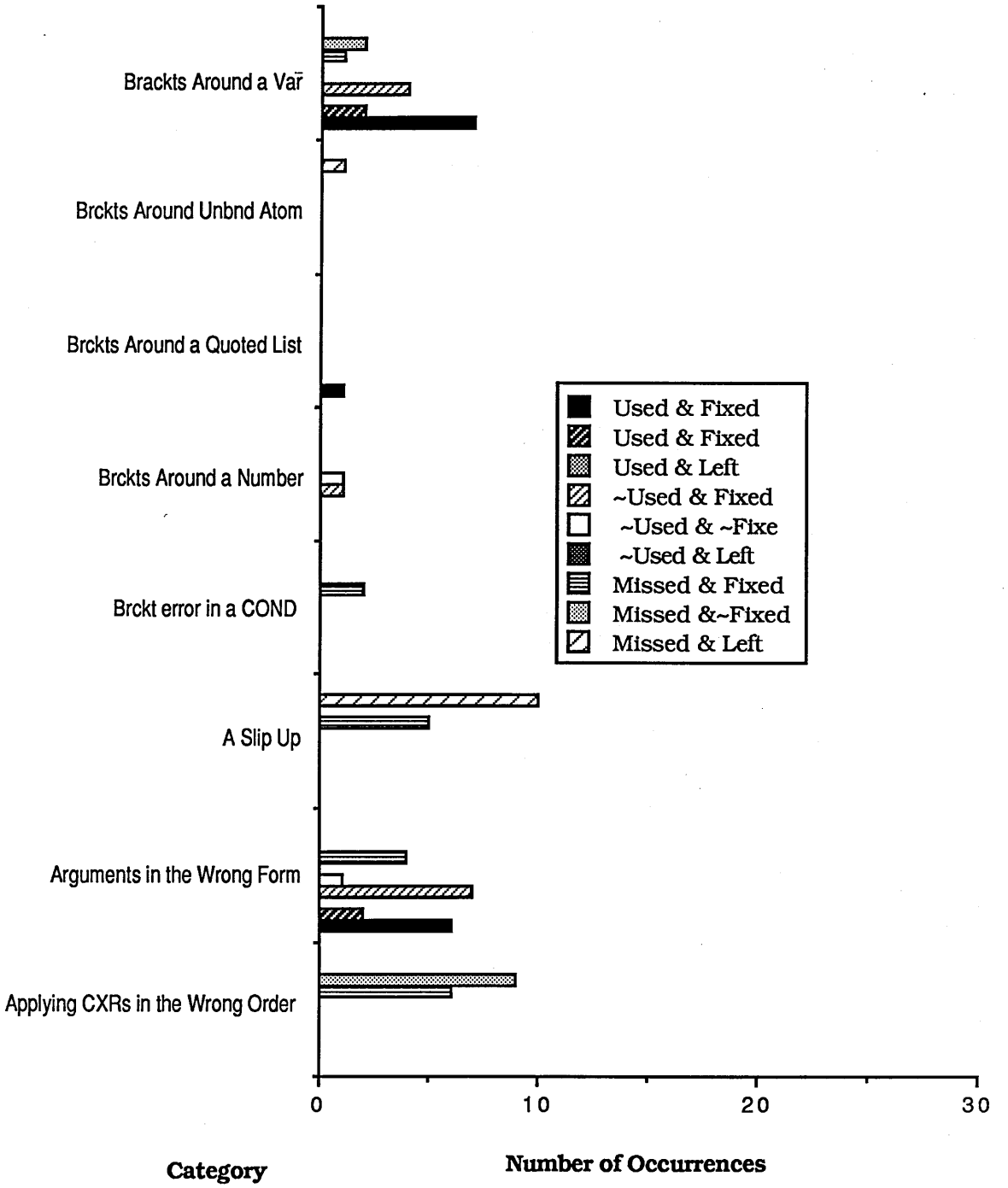
Bar Graph 1

Graph of Error Categories For All Subjects



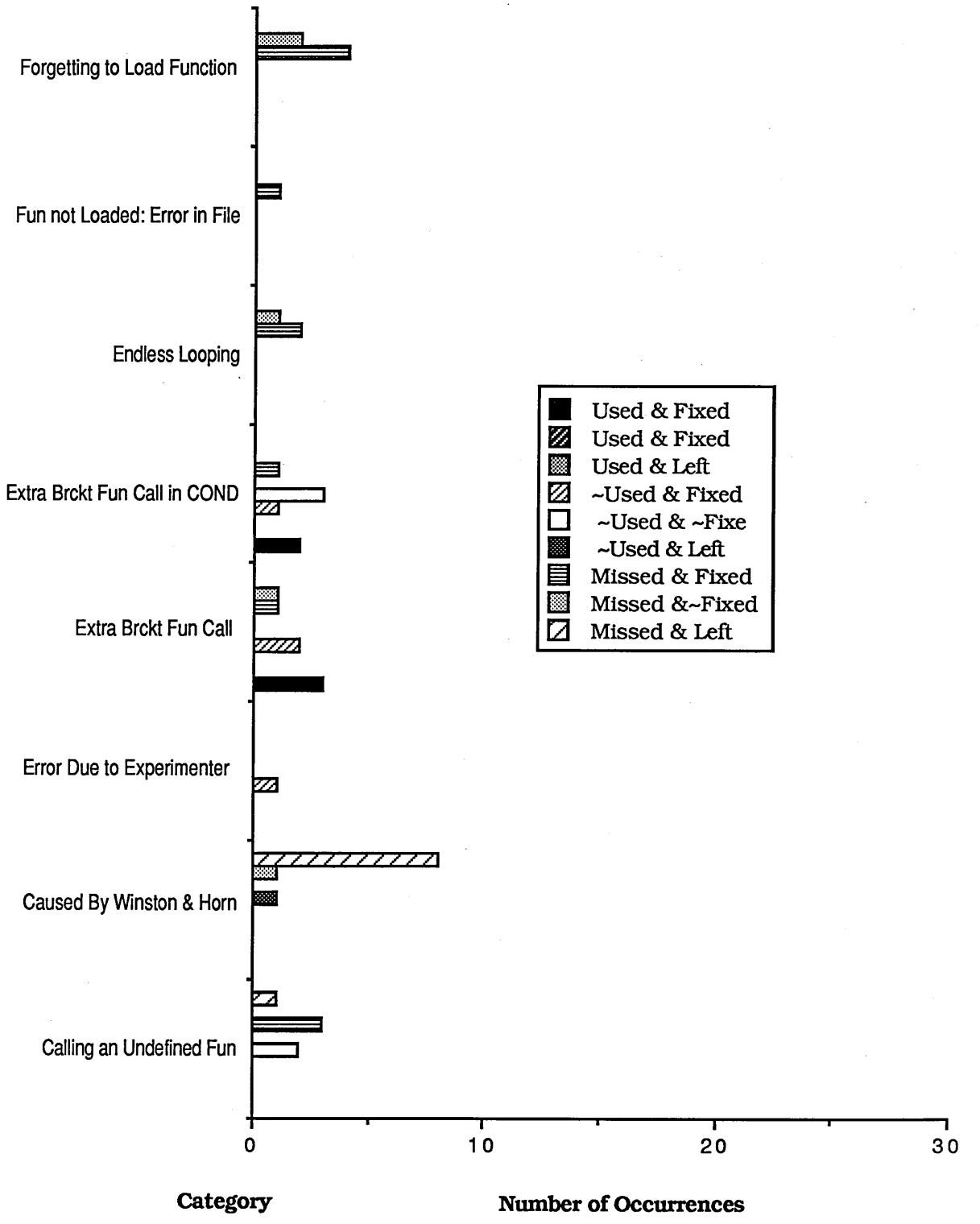
Bar Graph 2a

Graph of Error Categories For All Subjects



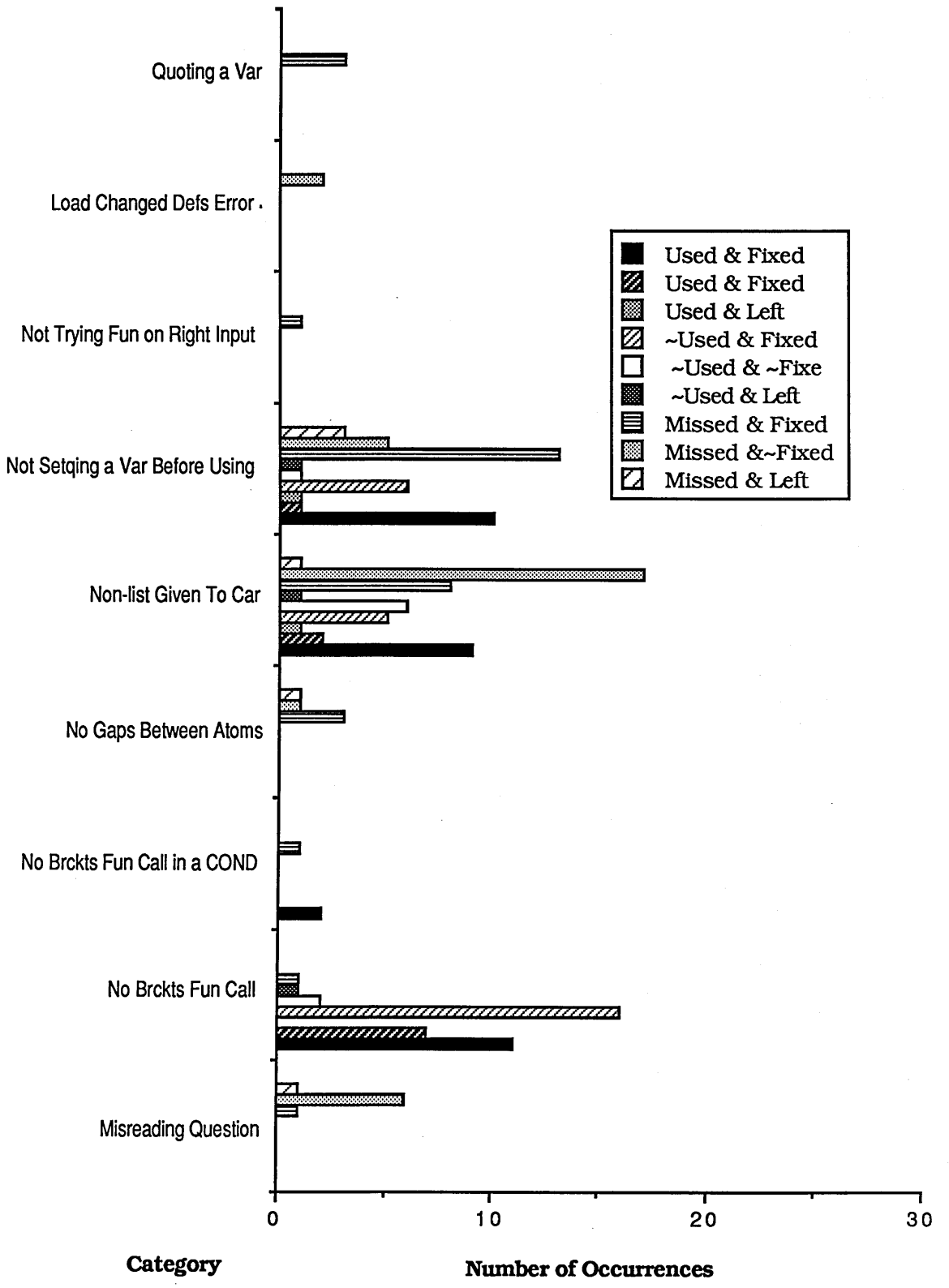
Bar Graph 2b

Graph of Error Categories For All Subjects



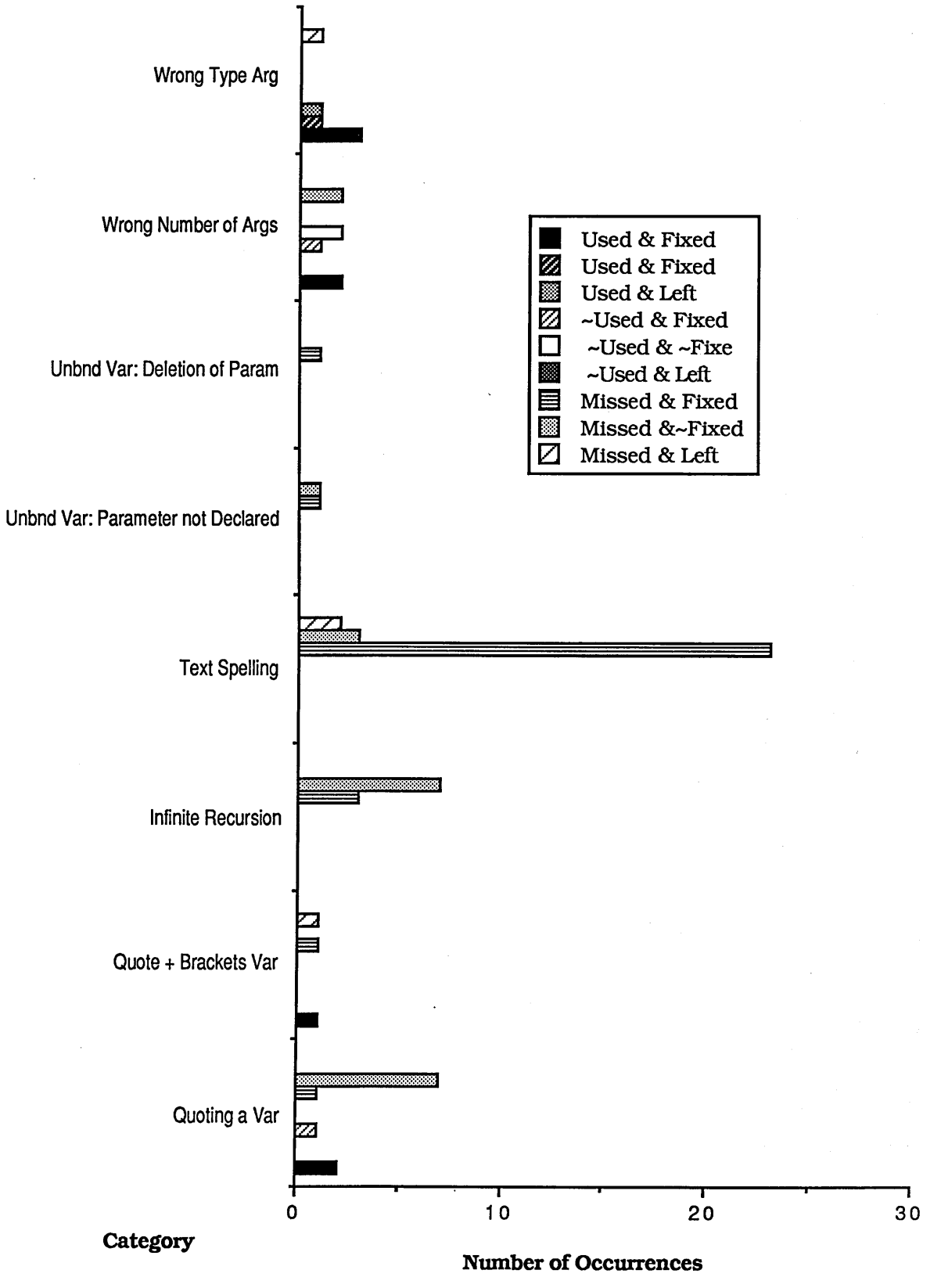
Bar Graph 2c

Graph of Error Categories For All Subjects



Bar Graph 2d

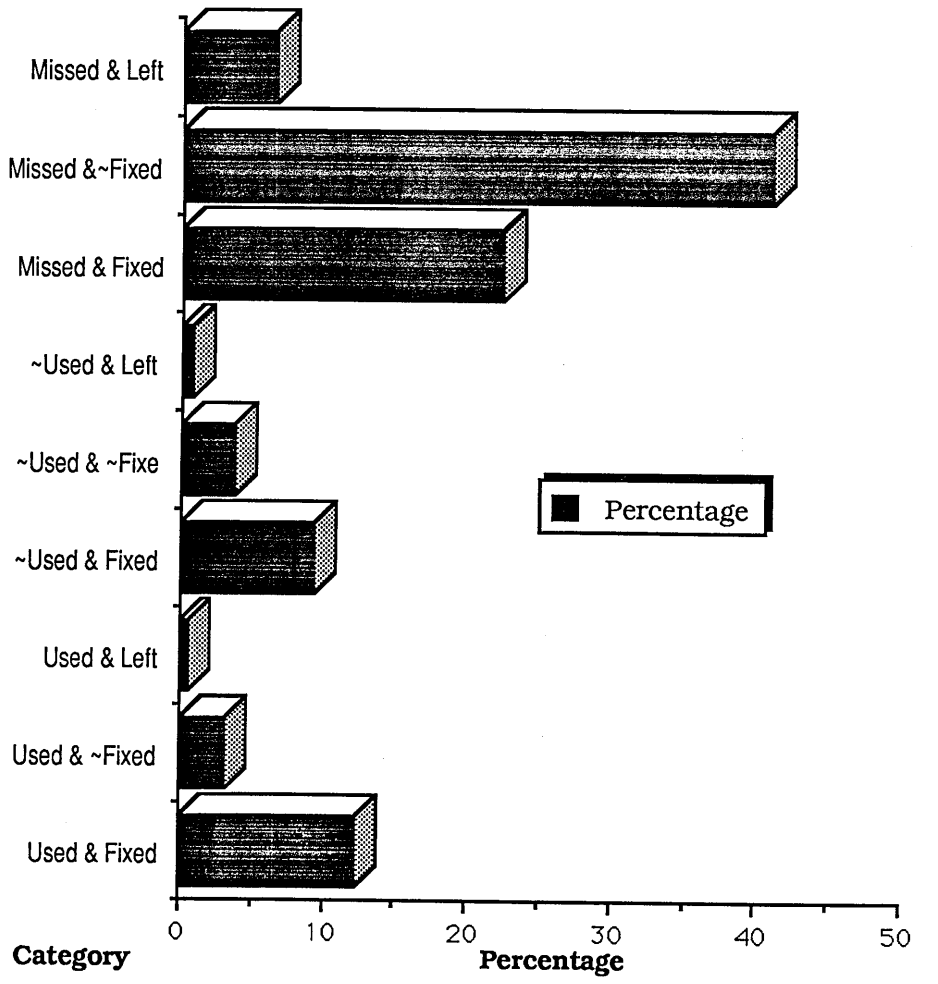
Graph of Error Categories For All Subjects



ITSY trapped 30% of all the errors. Of the errors ITSY had been 'designed' to trap ITSY managed to trap 48%. Of the errors not trapped 42% were in the *not quoting* category and 34% were in the *non-lists given to one of CAR CDR and CONS and APPEND* and 13% were in the *quoting a variable. Not quoting* errors were missed because ITSY only counts a non-quoting errors if there is a function taking the non-quoted object as an argument and if the object is of the right type for the function. Because of this, unquoted lists or atoms typed by themselves at the Lisp Toplevel were not trapped. The *non-list* errors were missed because of the simplicity of the error cliche - errors where a non-list argument to a function was computed rather than just given were not trapped. *Quoting a variable* errors were missed because the variables were misspelt.

Bar Graph 3

Totals Only



The third graph shows the total percentages for each of the nine different types of actions

that can occur.

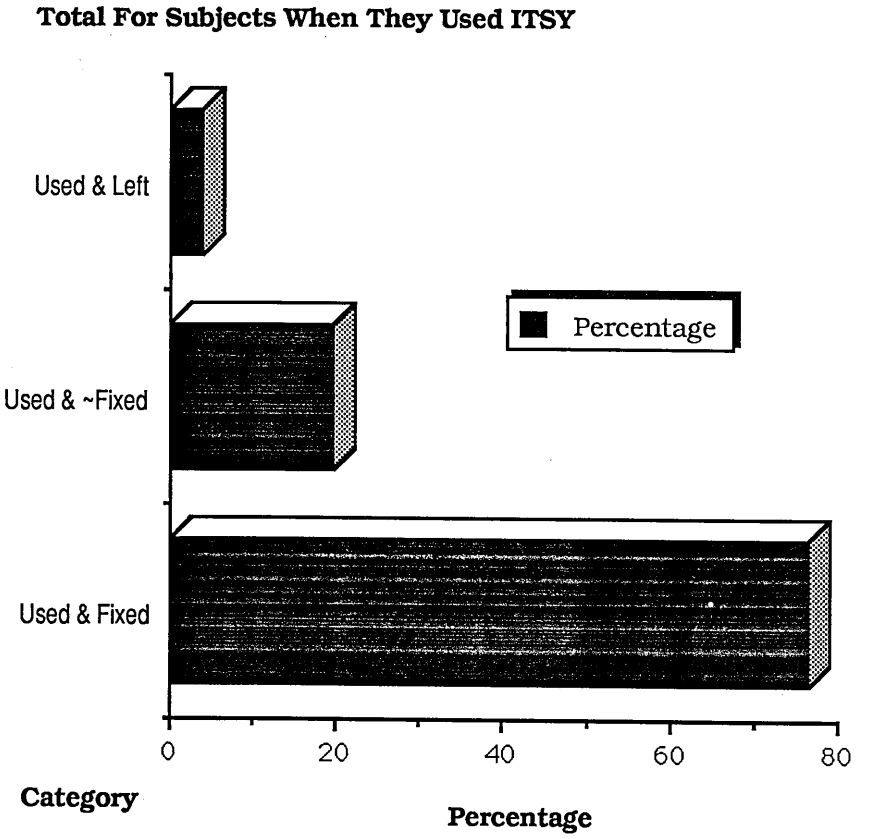
Below is a table showing the total number of errors and the percentage of errors per line of input.

Name	Total Number of Lines Input	Number of Errors	Percentage of Errors / Line Input
S	588	231	39
C2	331	105	32
L	252	85	34
S2	191	58	30
Total	1362	479	35

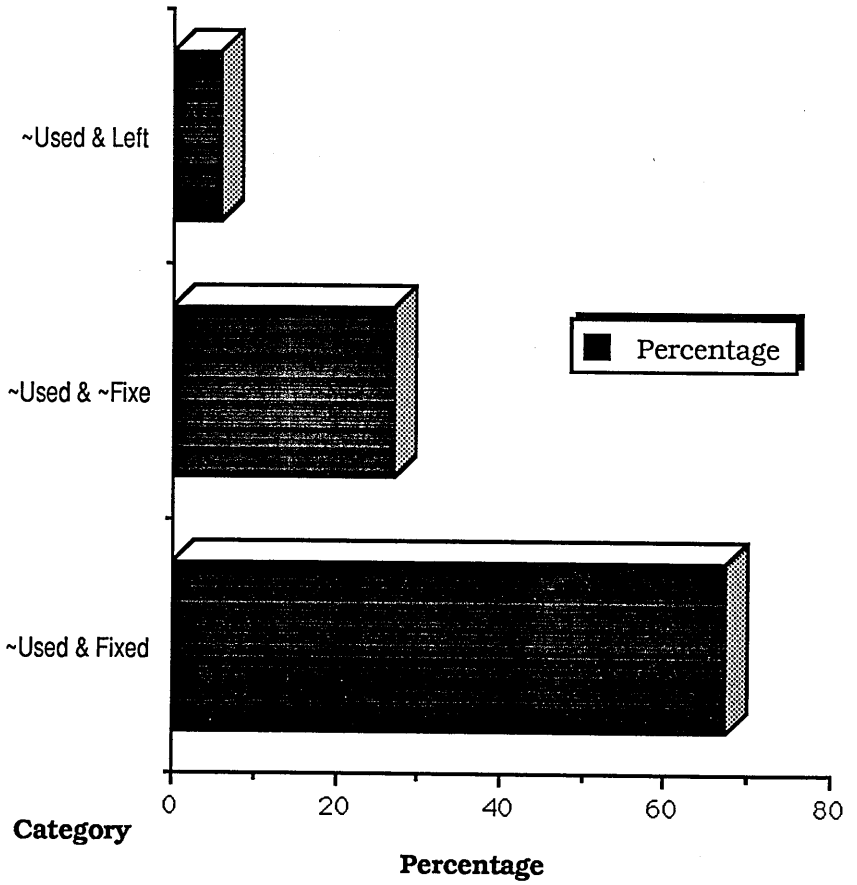
The average percentage of errors per line of input has increased from 13% in study I to 35% in this study, although this is less than the 45% for study II. The decrease from 45 in study II to 35 in this study could due to the fact that the subjects in this study were experienced programmers. Because relatively few subjects were used it is not possible to tell how much ITSY helped the students. The reasons for the increase in studies II and III when compared to study I were stated in 11.5.1.

12.4.2 Messages

Bar Graph 4



Bar Graph 5



The success of the messages can be seen from the fourth and fifth graphs. The two graphs show the percentage of errors fixed when trapped by ITSY. The fourth graph shows the percentage when the students used ITSY and the fifth when they chose not to. The errors fixed when not trapped by ITSY have not been included as these may be harder than those trapped. There is a 9% difference between the percentage of errors fixed when the students used ITSY and when they did not.

Below is a table containing the percentage of time that was spent looking at each part of the tutorial frames (more detailed information is given in appendix J).

Frame	Percentage of Total Time
Question	38
Question Explanation	2
Main Explanation	25
Fix	20
Examples	9
Deeper Explanation	7

The question explanation frame was only used 2% of the time. This shows that the subjects understood the question. The main explanation and the fix frames were far more popular than the examples or deeper explanation frames. The reason for the unpopularity of the deeper explanation frame could be that subjects were reluctant to read more than a short paragraph of text. The deeper explanation frames contained the longest messages. A possible reason for the unpopularity of the example frame could be that the subjects had trouble mapping from the examples given to their code.

12.5 Extra Errors

Some of the errors have been separated out for two reasons. The *computer environment* errors have been separated out because dribble files do not provide a reliable way to collect these in the complex environment found on the Symbolics 3600 family. Some of these errors will be missed because they will not be recorded. The number of computer environment errors are considerably lower than they were in study I. Four of these errors were found in the dribble files. One was due to a subject choosing the Add Comments item from the Lisp menu and without realising it. The other three were due to subjects trying to bring up a menu using the mouse. At certain times (such as when

incremental garbage collection was taking place) ITSY's response time grew. In three cases when a menu did not appear instantaneously the subject kept pressing the mouse repeatedly. This caused a number of menus to appear which confused the subject.

Another set of errors have been separated out because they appeared when a subject was using the Test a Function tool (see section 11.5.2). This tool was originally designed for subjects to test a particular function if they thought it was correct. Two of the subjects decided that it was quicker to try out their functions using Test a Function first and then only to resort to using the Lisp toplevel if they could not fix the error after some time. These subjects were using this as a way of switching ITSY off. This increased the response time if their function contained an error because no analysis would take place. This happened towards the end of the study.

Both set of errors are presented below:

Category	Number of Occurrences
A Slip Up	1
Brackets Around a Quoted Variable	2
Brackets Around a Variable	6
Brackets Around a Variable in a COND	2
Calling a Function that Doesn't Exist	3
Error due to Experimenter	1
Extra Set of Brackets Around a Function Call	3
Extra Set of Brackets Around a Function Call in COND	2
Function Not Loaded Because of Another Error	4

Incorrect Algorithm	79
Misreading the Question	6
No Brackets Around a Function Call	3
No Brackets Around a Function Call in a COND	1
No Gaps Between Atoms	1
Non-lists Given to One of CONS, CAR, CDR and APPEND	20
Quoting a Function	1
Quoting a Variable	9
Stack Overflow Due to Infinite Recursion	2
Text Spelling Error	11
Wrong Number of Arguments Given to a Function	12

12.5 New Error Cliches

Two new error cliches were found in this study - they are described in Chapter 8. They are the *Wrong Scope* error cliche and the sub cliche *Wrong Type Argument Subcliche.* The *Wrong Scope* error cliche matches against segments of code such as:

```
(defun wrong-scope1 (a b)
  (append (wrong-scope2) (wrong-scope2))

(defun wrong-scope2 ()
  (list a b))
```

The student believes that the variables a and b are bound.

The *Wrong Type Argument Subcliche* on the surface code segment Car extends the *Wrong Type Argument* error cliche. One of the commonest cases of this error cliche found in this study was:

```
(append (car x) ...
```

where X was a flat list (eg. '(a b c)). This subcliche checks to see if the CAR of the input matches the expected type of the super cliche's object in this case APPEND.

12.7 Conclusions

The evaluation methodology presented is general and can be applied to any system which claims to help its users in some way. One of the evaluation methods normally used is to take have two sets of subjects. One set uses the system, the other uses the 'bare machine'. The two sets are compared on pre and post test scores. The problem with this is that a relatively large number of subjects are needed (in order to gain statistical significance) and half are 'wasted' in that they do use the system and cannot contribute (directly) to the data concerning the system. This problem is especially great for systems such as ITSY which aim to help novice programmers - each subject needs to use the system for thirty hours.

If the methodology described in this chapter is used fewer subjects are needed and *all* the subjects contribute to the system.

Below is a copy of the first bar graph presented with fewer (more gross) error categories. Each of the new categories was formed from several of the categories shown in the first bar graph. The method of grossing for each category was as follows:

1. Other Errors - all the errors that did not fit into one of the other categories.

2. Endless Loops/Recursion - all the errors that either caused endless looping or infinite recursion.

3. Spelling & Slips - all the errors caused by spelling mistakes or by simple slip ups.

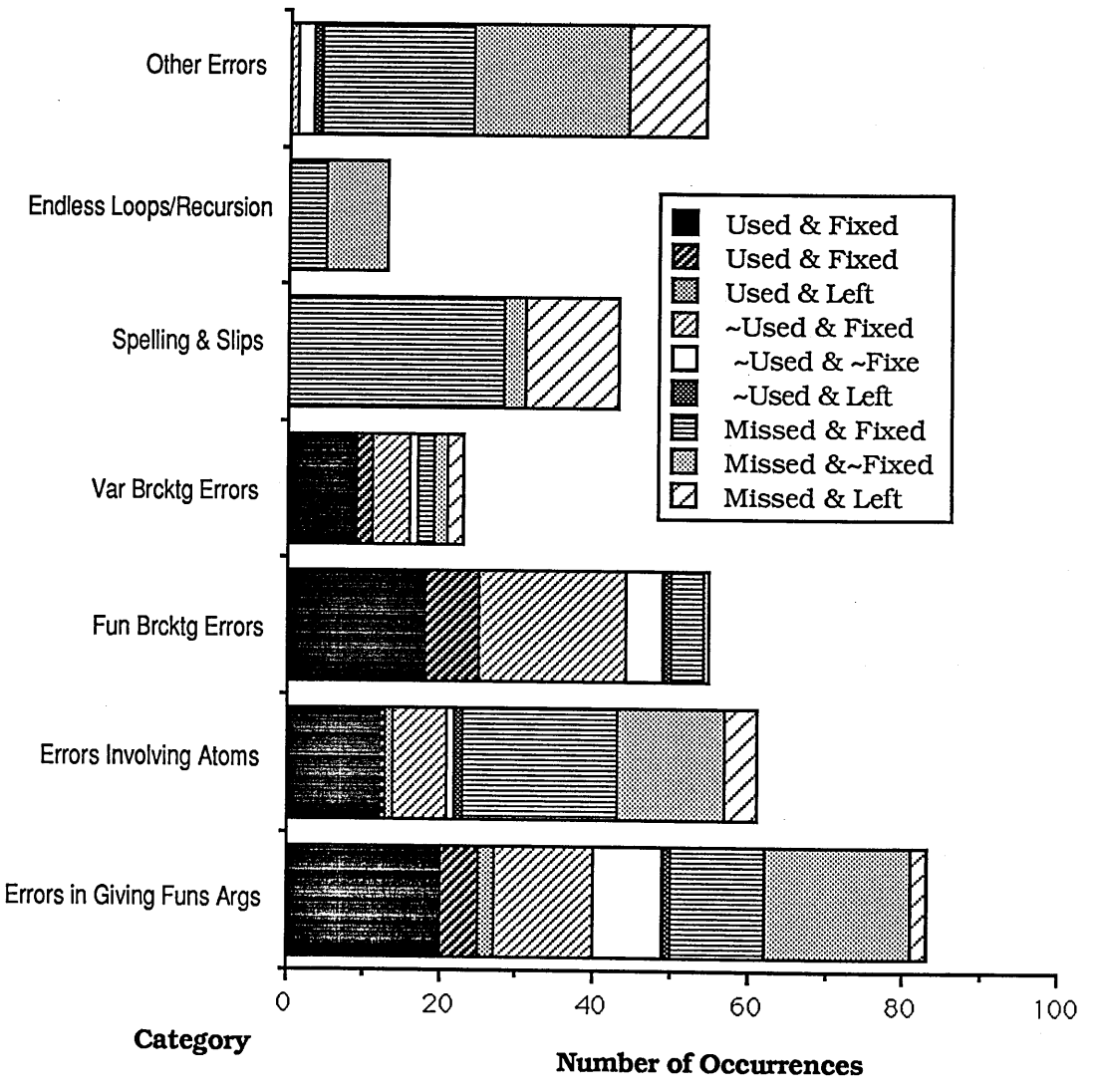
4. Var Brcktg Errors - all the errors caused by placing brackets around variables.

5. Fun Brcktg Errors - all the errors caused by placing the incorrect number of brackets around the application of functions.

6. Errors Involving Atoms - all the errors involving atoms such as *not quoting.*

7. Errors in Giving Funs Args - all the errors caused by giving arguments to functions incorrectly.

Bar Graph 6
Gross Totals For All Subjects



The results of the study outlined in this chapter could be used to improve ITSY. The study could then be repeated and ITSY improved further still. As this iteration continued two things would happen. Firstly, the bars would get shorter. Secondly, the solid black sections of the bar graph shown above would cover a greater and greater proportion of each bar. This would be because ITSY would be providing more help.

13. CONCLUSIONS AND FUTURE DIRECTIONS

This thesis covers two main areas of investigation. These areas correspond to the first two parts of this document.

Firstly, the first part provides a detailed description of the types of errors that professional programmers make when learning Lisp using a 'traditional' (i.e. glass teletype) Lisp environment. One quarter of the errors found were caused by the environment. The Lisp environment used in this study was designed for expert Lisp programmers. Unfortunately, some of the tools designed to improve programmer productivity hampered the subjects. As the subjects were very reluctant to use any of the tools provided by the environment, the subjects would have made fewer mistakes and progressed further if tools had been 'turned off'.

Nearly half the errors found in the study were context independent Lisp errors. These were errors were caused by the subjects using *incorrect Lisp forms*. Incorrect Lisp forms are segments of Lisp code that do not follow the discourse rules of Lisp. An example of an incorrect Lisp form would be (CAR 1). It is these incorrect Lisp forms that the error cliches were designed to match against. The main reason for subjects making these errors was the fact that they did not understand the Lisp evaluator. Often subjects would add and subtract quotes in a seemingly random fashion until their function worked. From these results, there is a case for teaching novices about the evaluator first.

Secondly, the concept of a programming cliche has been inverted and used as a basis for a system designed to help overcome the difficulties described in the first part of the thesis. The help given to students is based on the bugs they make. This is different from systems that view the student as a subset of an expert eg. WEST [Burton & Brown, 1976]. In systems such as WEST the student is measured in terms of an expert. When a student makes a mistake the student is assumed to have an 'expert concept' missing. The evidence from the first study points to the fact that students share common incorrect concepts. When these concepts are applied the same incorrect Lisp forms are produced. The third study showed that it is possible to trap these incorrect Lisp forms and explain to students the concepts that they have misunderstood.

One of the aims of ITSY is to give students enough help so that they can use bare system unaided. This is why the error messages generated by the Lisp system have been left in and why ITSY coaches students on tools. ITSY sits in the background until a student has written a program, only giving help when the student tries the program out. The reason for this is get the student used to the normal cycle of developing Lisp software. One of the problems with this however is that ITSY cannot offer any help if the student is completely stuck, for example, if the student has trouble developing an algorithm or specification.

This approach can be used in the design of computing systems built to help novices in certain domains. The constraint on the domain is that students' answers are complex enough to contain patterns of errors (so one word answers would not suffice). This would include domains where students are learning procedural skills - such as arithmetic, algebra or mechanics.

At present ITSY uses the relatively low-level surface plan representation of code. The reason for this is that novice Lisp programmers make syntax errors. PROUST also uses a low-level representation for PASCAL code in order to catch low-level errors, as Johnson says [Johnson, 1985 p 248]:

> This low-level representation means that an abundance of transformation rules are required in order to understand code written by more advanced students, who rearrange their code at will.

Johnson suggests a remedy for this:

> One could then construct two PROUSTs: a PROUST1 which specializes in low-level bugs, but is confined to fairly small programs, and a PROUST2 which specializes in high-level bugs, and which works on larger programs.

If the code is abstracted too far the low level errors will be lost.

ITSY has no such problem. The power of PLAN representation is that they allow higher level PLANs to be built up from low level PLANs. Figure 13-1 shows how ITSY could use the full power of PLAN representation to detect bugs from both complete Lisp novices and more advanced students.

Figure 13-1

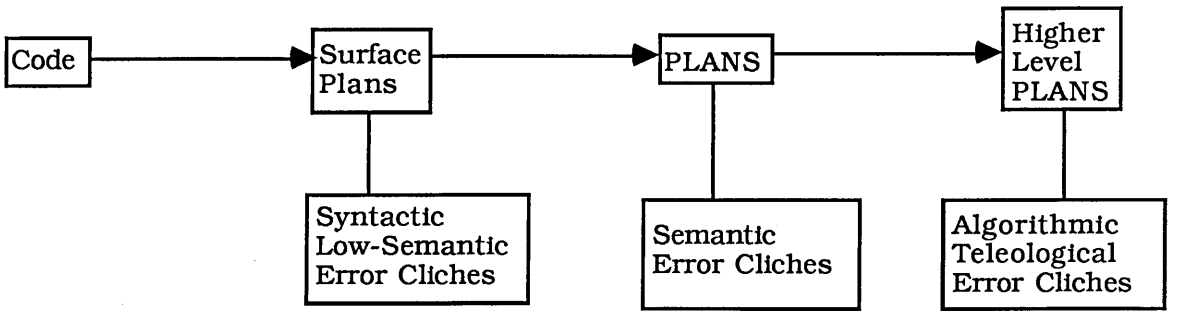


Figure 13-1 shows how ITSY could detect the algorithmic errors. In study III (Chapter 3) 32%[1] of the errors were algorithmic, by far the largest category. In order to trap these errors ITSY would need knowledge of the exercise that the student was attempting. This extra knowledge would be represented in two ways:

1. Context sensitive error cliches.

2. The solution to the exercise in PLAN form. This solution would represented as abstractly as possible, so it would be able to match against all the possible different implementations of the solution.

If the input typed by the student does not produce a Lisp error these two extra chunks of knowledge would be used to find errors in the student's program.

Context sensitive error cliches are error cliches specific to a particular exercise; they are similar to the buggy algorithms in TALUS [Murray, 1986]. The use of context sensitive error cliches can best be explained by the use of an example. Consider exercises 3-2 and 3-3 in Winston and Horn:

"3-2:Define ROTATE-L, a function that takes a list as its first argument and returns a new list in which the former first element becomes the last. The following illustrates:

(ROTATE-L '(A B C))
(B C A)

[1] This includes the 'misreading the question' category

(ROTATE-L (ROTATE-L '(A B C))
(C A B)

3-3:Define ROTATE-R. It is like ROTATE-L except that it is to rotate in the other direction."

One of the subjects second attempt was:

```
(defun rotate-r (exp-1)
  (append (rest (rest exp-1))
          (list (first exp-1) (first (rest exp-1)))))
```

The above code is valid but, the subject thought that the function ROTATE-R would have to deal exclusively with lists containing only three elements. It is only possible to trap these type of errors with knowledge about the actual exercise being attempted. In this exercise, ITSY would have a cliche that matched against code to make a list of three elements.

Whereas the context sensitive error cliches would be used to detect major differences between the student's attempt and the correct solution the PLAN form of the solution would be used to detect smaller variations, for example swapping APPEND for LIST.

The only coaching currently provided by ITSY is on the editor. Other possibilities for coaching in a future implementation of ITSY would be:

a) Coaching on the Lisp stepper. This could occur one of the more complex error cliches (as yet not implemented) were found to match the student's input.

b) Coaching on the Lisp tracer. This would also occur when a more complex error cliche matched the student's code.

c) Coaching on the Inspector. The inspector is a graphical tool for examining Lisp objects

such as DEFSTRUCTS or HASHTABLES. This would occur the first time a student created a complicated data structure such as a DEFSTRUCT.

At present the nodes in the student model are 'bumped up' too quickly. The reason for this is that the transition function between the nodes were determined arbitrarily. By transition function we mean the function that determines when a node changes state. In the case of ITSY's student model the transition function simply counts the number of times an event (such as the student applying a function correctly) occurs. We suspect that determining the transition function is not a trivial problem for two reasons. Firstly, each different concept will need a different set of transition functions, as concept are not equally easy to learn. Secondly, each student will need a different set of transition functions and this will change as the student progresses.

The first step in determining the transition functions would be to re-analyse the raw data collected in the studies. This re-analysis would include counting the number of times a student used a particular concept correctly before the student seemed to fully understand the concept.

Another possible extension to ITSY would be the addition of a *student history*. This would provide ITSY with *focus*. The need for a history can be seen from the following example taken form Study II (Chapter 11). The student has defined the function my-first:

```
(defun our-first (items)
  (car (items)))
```

the following interaction then took place at Lisp toplevel

```
*- (our-first (one two three))
;; The function ONE is undefined

*- (our-first '(one two three))
;; The function ITEMS is undefined

*- (items one two three)
```

```
;; The function ITEMS is undefined

*- (items '(one two three))
;; The function ITEMS is undefined

*- (items 'one 'two 'three)
;; The function ITEMS is undefined

*- (items '(one two three))
;; The function ITEMS is undefined

*- (items 'one 'two 'three)
;; The function ITEMS is undefined

*- (items '(one) '(two) '(three))
;; The function ITEMS is undefined

*- (subst '(one two three) 'items (our-first))
;; The function our-first was given too few arguments

*- (subst '(one two three) 'items '(our-first))
(OUR-FIRST)

*- (subst '(one two three) 'items 'our-first)
OUR-FIRST
```

The subject then left this exercise.

This would use a push down stack of previous *events*. An event would be the analysis of the student's input (including the analysis of any functions referred to) and any error cliches that matched against the input. The stack would be *weighted* so that the more distant in time an event occurred the less influence it would have on the current input. ITSY would then use the stack to focus its search for an error. The stack would be used when ITSY was unable to match an error cliche against the student's input. In the above example ITSY would be unable to match an error cliche against the input:

```
(items '(one two three))
```

But on the stack would be information on the line

```
(our-first '(one two three))
```

This would contain the following:

a) The error cliche *Brackets Around a Variable* matched against this input,

b) The name of the variable surrounded by brackets was ITEM.

Using this information ITSY would determine that the student was trying to give a value to the variable ITEM.

ITSY's approach could be extended to experts in the form of an advanced Program Debugging system. Since PLANs are language independent the debugger could cope with any programming language that had an analyser (analysers exist for Lisp, Fortran and PL1 [Waters, 1985]). Because we cannot restrict expert programmers to work in a restricted context or burden the programmer with the need to supply specification, the system would only be able to detect a certain *class* of errors. This *class* would include errors such as: unreachable statements, endless recursion and non-terminating loops. It would not be possible to detect deep semantic or conceptual errors. Such errors require knowledge about the actual task being attempted.

In order to increase efficiency not all the error cliches would be active for any error. The set of error cliches activated would depend on the type of error signalled this would tie the debugger closely to the *normal* program debugger. For example, the error *stack overflow* would trigger the *endless recursion* error cliche, but not the *incorrect loop initialisation* error cliche.

Expert error cliches for Lisp could include:

a) placing mixins to Flavors [Weinreb & Moon, 1982] in the wrong order,

b) surgically changing a list (using one of RPLACA, RPLACD, NCONC etc.) that will be used later,

c) using a THROW when not inside a CATCH.

Expert error cliches could also include 'bad style cliches' such as using inefficient code.

APPENDICES

REFERENCES

Adam, A. & Laurent, J. P. Automatic Diagnostics of Semantic Errors. Proceedings of the AISB-80 Conference on Artificial Intelligence, July 1980.

Anderson, J. R. & Reiser, J. B. The LISP Tutor. BYTE: The Small Systems Journal. Vol. 10. No. 4. pp 159-175. April 1985.

Anderson, J. R., Pirolli, P. & Farrell, R. Learning to Program Recursive Functions. To Appear in The Nature of Expertise. Chi, M., Glaser, R. & Farr, M. (eds). Hillsdale New Jersey: Erlbaum, 1984.

Barr, A., Beard, M., & Atkinson, R. C. The Computer as a Tutorial Laboratory: The Stanford BIP Project. International Journal of Man-Machine Studies, Vol. 8, pp 567-596, 1976.

Boies, S. J. & Gould, J. D. Syntactic Errors in Computer Programming. Human Factors Vol. 16, 253-257, 1974.

Bonar, J. & Soloway, E. Pre-Programming Knowledge: A Major Source of Misconceptions in Novice Programmers. Human-Computer Interaction, Fall 1985.

Brotsky, D. Program Understanding through Cliche Recognition. MIT Artificial Intelligence Laboratory. Working Paper 224 December 1981.

Brown, J. S. & Burton, R. R. A Paradigmatic Example of an Artificially Intelligent Instructional System. Proceedings of the First International Conference on Applied General Systems Research: Recent Developments and Trends, Binghampton, New York, August 1977.

Burton, R. R. & Brown, J. S. An Investigation of Computer Coaching for Informal Learning Activities. In Intelligent Tutoring Systems (eds) Sleeman, D. & Brown, J. S. August 1978.

Cerri, S. A., Fabbrizzi, M. & Marsili, G. The Rather Intelligent Little Lisper. AISBQ Vol 50, pp 21-24 Spring/Summer 1984.

Clancey, W. J. Tutoring rules for guiding a case method dialogue. International Journal of Man-Machine Studies. Vol. 11, pp 25-49, 1979.

di Sessa, A. A. A Principled Design for an Integrated Computational Environment. MIT Laboratory of Computer Science. July 1982

du Boulay, J. B. H. LOGO learning by School Teachers. Edinburgh: Doctoral dissertation, Department of Artificial Inteligence, University of Edinburgh, 1979.

du Boulay, B., O'Shea, T. & Monk, J. The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices. International Journal of Man-Machine Studies Vol.14, pp. 237-249, 1981.

Eisenstadt, M. A User-Friendly Software Environment for the Novice Programmer. Communications of the ACM Vol. 26 No. 12, pp 1058-1064, December, 1983.

Eisenstadt, M. & Laubsch, J. Towards an Automated Debugging Assistant for Novice Programmers. Proceedings of the AISB-80 Conference on Artificial Intelligence, July, 1980.

Eisenstadt, M., Laubsch, J. H. & Kahney J. H. Creating Pleasant Programming Environments for Cognitive Science Students. Proceedings of the Third Annual Conference of the Cognitive Science Society. August, 1981.

Eisenstadt, M. & Lewis, M. Errors in an Interactive Programming Environment: Causes and Cures. Human Cognition Research Laboratory. Milton Keynes, MK7 6AA England Tech. Rep. 4 (2nd Ed.) September, 1985.

Elsom-Cook, M. T. Design Considerations of an Intelligent Tutoring System for Programming Languages. Warwick: Doctoral dissertation, Department of Psychology, University of Warwick. October 1984.

Goldstein I. P. Summary of MYCROFT: A System for Understanding Simple Picture Programs. Artificial Intelligence. Vol. 6 pp. 249-288, 1975.

Goldstein, I. P. & Papert, S. Artificial Intelligence, Language, and the Study of Knowledge. Cognitive Science, Volume 1, Number 1, 1977.

Hasemer, T. A Very Friendly Software Environment for SOLO in New Horizons in Educational Computing, (ed) Yazdani M.Ellis Horwood, London. pp 84-100 1983.

Hasemer, T. An Empirically-Based Debugging System for Novice Programmers. Human Cognition Research Laboratory. The Open University, Milton Keynes, England. Technical Report. No. 6, November 1983.

Hasemer, T. A Beginner's Guide to Lisp. Addison-Wesley 1984.

Johnson, W. L, Draper & S. Soloway, E.Classifying Bugs is a Tricky Business. Proceedings of the Seventh Annual NASA/Godard Workshop on Software Engineering, Baltimore, 1982.

Johnson, W. L., Draper, S. & Soloway, E. An Effective Bug Classification Scheme Must Take the Programmer into Account. Proceedings of The Workshop on High-Level Debugging, Palo Alto, 1983.

Johnson, L. W. & Soloway, E. PROUST: An Automatic Debugger for Pascal Programs. BYTE The Small Systems Journal. Vol.10. No. 4. pp 179-190 April 1985

Kahney, H. & Eisenstadt, M. Programmers' Mental Models of their Programming Tasks: The Interaction of Real World Knowledge and Programming Knowledge. Proceedings of the Fourth Annual Conference of the Cognitive Science Society. Ann Arbor, Michigan, 1982.

Laubsch, J. & Eisenstadt, M. Domain Specific Debugging Aids for Novice Programmers. Proceedings of the Seventh International Joint Conference on Artificial Intelligence, Vancouver, B.C. Canada, August 1981.

Laubsch, J. & Eisenstadt, M. Using Temporal Abstraction to Understand Recursive Programs Involving Side Effects. Proceedings of the National Conference on Artificial Intelligence August 1982.

Lewis, M. Improving Solo's User Interface: An Empirical Study of User Behaviour and Proposals for Cost-Effective Enhancements to Solo. Computer Assisted Learning Research Group The Open University, Milton Keynes, England. Technical Report No 7. April 1980.

Lieberman, H. Steps Toward Better Debugging Tools for Lisp. Proceedings ACM Symposium on Functional Programming, 1984.

Lukey, F. J. Understanding and Debugging Programs. International Journal of Man-Machine Studies Vol. 12, pp. 189-202, 1980.

Lutz, R. Towards an Intelligent Debugging System for Pascal Programs. Human Cognition Research Laboratory, The Open University, Milton Keynes, England. Technical Report No. 8, April 1984.

McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. & Levin, M. I. LISP 1.5 Programmer's Manual, The MIT Press, Cambridge, Massachusetts 1962.

Murray, W. R. Automatic Program Debugging for Intelligent Tutoring Systems. Texas: Doctoral dissertation, Artificial Intelligence Laboratory, The University of Texas at Austin. June 1986.

Norman, D. A., Design Principles for Human-Computer Interfaces. Proceedings of the CHI 1983 Conference on Human Factors in Computer Systems. Boston, December 1983.

O'Shea, T. & Self, J. Learning and Teaching with Computers. Harvester Press. 1983.

Pitman, K. M. The Revised MacLISP Manual. MIT Laboratory for Computer Science Cambridge, Massachussets (MIT/LCR/TR 295), May 1983.

Rajan, T. M., APT: The Design of Animated Tracing Tools for Novice Programmers. Human Cognition Research Laboratory, The Open University, Milton Keynes, England, Technical Report No. 15. March 1985.

Rich, C., Shrobe, H. E., Waters, R. C., Sussman, G. J. & Hewitt, C. E. Programming Viewed as an Engineering Activity. MIT Artificial Intelligence Laboratory, A.I. Memo 459, January 1978.

Rich, C. Inspection Methods in Programming. MIT Artificial Intelligence Laboratory, Report No. AI-TR-604. June 1981.

Ruth, G. R. Intelligent Program Analysis. Artificial Intelligence, Vol. 7, pp. 65-851976.

Shapiro, D. G. Sniffer: A System that Understands Bugs. MIT Artificial Intelligence Laboratory. A.I. Memo No. 638 June, 1981.

Shortliffe, E. H. Computer Based Medical Consultations: MYCIN. New Yok: American Elsevier, 1976.

Shrobe, H. E., Waters, R. C. and Sussman G. J. A Hypothetical Monologue Illustrating the Knowledge Underlying Program Analysis. MIT Artificial Intelligence Laboratory, A.I. Memo 507, January 1979.

Sleeman, D. H. & Smith, M. J. Modelling Student's Problem Solving. Artificial Intelligence Vol. pp 16 171-188, 1981.

Spohrer, J. G. & Soloway, E. Analysing the High Frequency Bugs in Novice Programs. Empirical Studies of Programmers, Soloway, E. & Iyengar, S. (eds). Ablex Publishing Corporation Norwood, New Jersy. pp 230-251. 1986

Stallman, R. Emacs the Extensible, Customisable, Self-Documenting Display Editor. Proceedings of ACM SIGPLAN-SIGOA Symposium. Text Manipulation. ACM SIGPLAN Notices vol. 16, no. 6, June 1981.

Steele, G. L. Common Lisp: The Language. Digital Press, 1984.

Touretzky, D. S. A Gentle Introduction to Symbolic Computation. Harper and Row, 1984.

Waters, R. C. Automatic Analysis of the Logical Structure of Programs. Technical Report No. TR-492, December 1978.

Waters, R. C. A Method for Analysing Loop Programs. IEEE Transactions on Software Engineering, Vol. SE-5 No. 3 pp 237-247, May 1979.

Waters, R. C. The Programmer's Apprentice: Knowledge Based Program Editing. IEEE Transactions on Software Engineering, Vol SL-8 No.1, January 1982.

Waters, R. C. KBEmacs: A Step Toward the Programmer's Apprentice. MIT Artificial Intelligence Laboratory, Technical Report 753, May 1985.

Weinreb, D. & Moon, D. Lisp Machine Manual, 1981.

Wertz, H. Stereotyped Program Debugging: an aid for novice programmers. International Journal of Man-Machine Studies. Vol. 16, pp. 379-392, 1982.

Wescourt, K. T., Beard, M., Gould, L. & Barr, A. Knowledge Based CAI: CINS for Individualised Curriculum Sequencing. Stanford University, Stanford California, Technical Report No. 290 Inst. for Mathematical Studies in the Social Sciences, 1977.

Wilensky, R. Lispcraft. W. W. Norton and Co. London, 1984.

Winston, P. H. & Horn, B. K. P. Lisp. Addison-Wesley 1981.

Winston, P. H. & Horn, B. K. P. Lisp, 2nd Edition. Addison-Wesley 1984

Zelinka, L. M. Automated Program Recognition MsC Thesis MIT Electrical Engineering and Computer Science, June 1986.

APPENDIX A

INSTRUCTIONS FOR STUDY I

This appendix contains the two sheets that I handed out to the subjects, at the start of the experiment. The first sheet gives a brief introduction, telling the subjects how to log on and how to attempt the exercises. The second sheet gives a summary of the editor and LISP top-level commands.

LISP SESSION

1. Press return until you are prompted by

ENTER CLASS

then type 5 return. You will be prompted by

Enter Command or Course Code:

type in LOG R.LISPCLASS.N where n is your number (0..8). Then press the ESC button, you will then be prompted by (PASSWORD) the password is your number i.e. n.

2. When you log in you will automatically be put into the LISP toplevel. After a few minutes the word NIL will appear on the screen.

3. Start on chapter 2 in Winston & Horn. The exercises in chapter 2 can be typed into the LISP top level. First write the answer on the sheet provided, then type in your answer.
If your first attempt is wrong try typing in another, feel free to write any comments you feel relevant on the top level as well, you can write comments by typing a semi-colon at the start of the line. If you make an error in the LISP top level LISP will type something like

ERROR
Debug option (type ? for help):

if you get this type ^g (control g).

4. When you reach chapter 3 then you will need to write your function definitions into a file and then load them. Enter emacs by typing ^e. Once you have entered emacs you need to create a file. Type ^x ^f, the editor will the prompt you with Find File (Default RS:<R.LISPCLASS>GAZONK.DEL.0): then type in the name of the file you want to create. The file name should end with .lsp (eg lispfile.lsp). If the file already exists you will enter this file and the contents will appear on the screen. You can now type in the definitions.

5. Once you have typed in all the definitions you can save the file contents by typing ^x ^s. Quit emacs by typing ^c. You will now be back in the LISP top level. Load your file by typing ^l then the file name inside two vertical bars (eg. |lispfile.lsp|)

6. When you enter emacs for the second time you will be taken back to the last file that you were editing, you can then type in the extra definitions then re-load the file.

7. Whenever you are stuck ring me on 3701 and I will then advise you via your terminal.

8. Once you have finished type "(stop)" to leave the LISP top level.

Summary of Commands

^e	Enter Emacs
^c	Quit Emacs

Emacs Commands

↑	Up one line
↓	Down one line
←	Left one character
→	Right one character
Keypad 4	Left one word
Keypad 6	Right one word
^x ^f	Find a file (must finish with .lsp)
^x ^s	Write a file

Linefeed or Return then Tab automatically indents the text.

enter	Load just one function.

Lisp Top Level Commands

^g	Stop a program running
^l	Load a file (l__l)
^f	Look at a function
(step t)	Turn tracer on Space Bar Return to step through
(step nil)	Turn tracer off
;	Anything typed after a semi-colon is ignored by the interpreter.

If you get a error in the LISP top level LISP will type something like

ERROR
Debug option (type ? for help):

if you get this type a control g (G).

APPENDIX B

Raw Data

This appendix contains the raw data collected from the study. Each column represents a particular subject. For example, in the 2 lines

"The textual environment

6 4 1 3 3 0 0 0 0"

should be read as follows:

The first subject made 6 errors involving the textual environment. The second subject made 4 such errors, the third subject made 1 such error, and so on.

Subject

K	J	A	B	D	E	J2	B2	P

Problems Caused by the Environment

1. The textual environment.

6	4	1	3	3	0	0	0	0

2. The computing environment

1	9	7	4	10	1	2	18	4

Algorithmic Errors

1. Not realising that a solution is incorrect.

0	10	0	0	0	0	0	8	0

2. Using the wrong function, which is not one of APPEND, CONS and LIST.

0	1	1	3	0	0	2	0	0

3. Using the wrong function out of APPEND, CONS and LIST.

0	4	7	0	0	0	0	4	0

4. Using the wrong combination of CAR's and CDR's.

5	5	4	10	5	7	1	0	0

5. Errors with recursion.

0	11	3	0	0	0	0	2	2

6. Other Algorithmic errors.

0	18	9	0	0	0	0	5	3

Subject

K	J	A	B	D	E	J2	B2	P

Errors in Lisp

1. Simple Errors.

8	25	7	2	2	1	5	11	5

2. Forgetting to load a function.

0	1	3	1	0	0	4	1	0

3. Unbound atoms.

0	1	3	0	1	0	0	0	0

4. Putting brackets around an atom

2	3	15	0	1	0	0	2	1

5. Stuck at top-level, because there are not enough closing brackets

2	0	1	1	2	6	1	1	1

6. Putting an extra set of brackets around a function call.

0	0	14	0	0	0	3	0	0

7. Not putting brackets around a function call.

2	0	14	6	5	0	0	1	1

8. Wrong number of arguments given to a function.

0	0	4	2	2	0	0	1	1

9. Wrong number of arguments given to a function, because the arguments are in the wrong form.

0	5	1	4	6	0	0	4	4

10. Arguments given are of the wrong type.

1	15	16	0	12	0	4	7	1

11. Errors concerning the special form DEFUN.

3	2	0	0	0	0	2	0	0

12. Errors in the test part of a clause in a COND special form.

0	0	4	0	0	0	1	3	0

Subject

K	J	A	B	D	E	J2	B2	P
13. Errors in the result part of a clause in a COND special form.								
0	16	20	0	0	0	0	0	0
14. Quoting an object that should not be.								
0	7	0	0	0	0	0	1	2
15. Not quoting an object that should be.								
1	4	16	5	6	1	1	3	9
16. A file not loading, because there are not enough closing brackets.								
0	0	20	0	0	0	0	0	7
17. Other Errors.								
0	9	3	0	1	20	0	1	4

The large variations between the number of errors made by the subjects is due to two reasons. Firstly, some subjects made fewer errors per line of input than others. Secondly, some of the subjects attended more sessions than others.

APPENDIX C

Dribble Files

This contains two of the dribble files from the study. My comments are preceded by two semi-colons, the subjects' by one. Extra comments added "after the event" are in this font.

```
|Dribbling.|

(defun stop ()
  (undribble)
  (quit))
STOP

(setq W nil)
NIL
```

The function COMPLEXP is exercise 3-10 on page 42 of "Lisp" [Winston & Horn, 1981]. The problem statement is:

"Problem 3-10: Define COMPLEXP, a predicate that takes three arguments, A, B, and C, and returns T if b squared - 4ac is less than zero."

```
[LEDIT Created.]
[Reading from LEDIT...]
(defun complexp (a b c)
  (lessp (difference (expt b 2)
         (times 4 a c)) 0))
COMPLEXP
```

The function is "zapped" from the file

```
[LEDIT Completed.][LEDIT Continued.]
[Reading from LEDIT...]
(defun complexp (a b c)
  (lessp (difference (expt b 2) (times 4 a c))))
COMPLEXP
```

The subject is going to load his file brin.lsp

```
[LEDIT Completed.]file: |brin.lsp|
(defun first (exp-1) (car exp-1))
(defun rest (exp-1) (cdr exp-1))
(defun insert (new exp-1) (cons new exp-1))
(defun rotate-l (exp-1) (append (rest exp-1) (cons (first exp-1)
nil)))
(defun rotate-r (exp-1) (append (cons (first (reverse exp-1)) nil)
  (reverse (rest (reverse exp-1)))))
(defun palindromize (exp-1) (append exp-1 (reverse exp-1)))
(defun f-to-c (f) (difference (quotient (plus f 40) 1.8) 40))
(defun c-to-f (c) (difference (times (plus c 40) 1.8) 40))
(defun roots (a b c) (list (quotient (plus (minus b) (sqrt
(difference (expt b 2) (times 4 a c)))) (times 2 a)) (quotient
(difference (minus b) (sqrt (difference (expt b 2) (times 4 a c))))
(times 2 a))))
(defun evenp (num) (zerop (remainder num 2)))
(defun palindromep (list1) (equal list1 (reverse list1)))
(defun rightp (elta eltb eltc)
  (equal (expt elta 2) (plus (expt eltb 2) (expt eltc 2))))
(defun complexp (a b c) (lessp (difference (expt b 2) (times 4 a
c)) 0))
```

```
[LLOAD of file RS:<R.LISPCLASS.7>BRIN.LSP.25 completed.]
QUIT*
```

The subject uses a top level tool to look at the definition of a loaded function.

```
function: complexp

(DEFUN COMPLEXP (A B C) (LESSP (DIFFERENCE (EXPT B 2) (TIMES 4 A
C)) 0))

QUIT*
```

The function NILCAR is exercise 3-11 on page 45 of "Lisp". The problem statement is:

"Problem 3-11: In some LISP's, trying to take the CAR or CDR of NIL causes an error. Define NILCAR and NILCDR in terms of CAR and CDR such that they work like CAR and CDR, but return NIL if given NIL as their argument no matter what CAR and CDR do."

```
[LEDIT Continued.]
[Reading from LEDIT...]
(defun nilcar (exp-1)
  (cond ((null exp-1) nil)
        (nil (first exp-1))))
NILCAR

[LEDIT Completed.]function: nilcar

(DEFUN NILCAR (EXP-1) (COND ((NULL EXP-1) NIL) (NIL (FIRST EXP-
1))))

QUIT*

; i dont really understand how cond works, but here goes.
(setq lista '(a b c d e f))(A B C D E F)

(nilcar lista)NIL

(nilcar nil)NIL

[LEDIT Continued.]
[Reading from LEDIT...]
(defun nilcar (exp-1)
  (cond ((null exp-1) nil)
        (t (first exp-1))))
NILCAR

[LEDIT Completed.]function: nilcar

(DEFUN NILCAR (EXP-1) (COND ((NULL EXP-1) NIL) (T (FIRST EXP-1))))

QUIT*

; ok lets try it this way
(nilcar lista)A

(nilcar ())NIL

(nilcar nil)NIL

(nilcar '(1234 asdf 5678))1234
```

```
; i think i u
; sorry.... i think i'm beginning to understand
[LEDIT Continued.]
[Reading from LEDIT...](defun nilcdr (exp-1)
  (cond ((null exp-1) nil)
        (t (rest exp-1))))
NILCDR
[LEDIT Completed.]function: nilcdr

(DEFUN NILCDR (EXP-1) (COND ((NULL EXP-1) NIL) (T (REST EXP-1))))

QUIT*

(cdr nil)NIL

(nilcdr nil)NIL

(nilcdr lista)(B C D E F)

(nilcdr '(1234 asdf 5678))(ASDF 5678)

function: nilcdr

(DEFUN NILCDR (EXP-1) (COND ((NULL EXP-1) NIL) (T (REST EXP-1))))

QUIT*
```

The function CHECK-TEMPERATURE is exercise 3-12 on page 45 of "Lisp". The problem statement is:

"Problem 3-12: Some people prefer the Fahrenheit scale to the Celsius scale, because they find it aesthetically pleasing that 0 degrees and 100 degrees are pinned to temperatures that bracket the temperature spectrum of temperate climates, 0 degrees being ridiculously cold and 100 degrees being ridiculously hot. Define CHECK-TEMPERATURE, a function that is to take one argument, such that it returns RIDICULOUSLY-HOT if the argument is greater than 100, RIDICULOUSLY-COLD if the argument is less than 0, and OK otherwise."

```
[LEDIT Continued.]
[Reading from LEDIT...]
(defun check-temperature (temp)
  (cond ((greaterp temp 100) '(ridiculously-hot))
        ((lessp temp 0) '(ridiculously-cold))
        (t '(ok))))
CHECK-TEMPERATURE

[LEDIT Completed.]
; problem 3.12

function: check-temperature


(DEFUN CHECK-TEMPERATURE (TEMP)
  (COND ((GREATERP TEMP 100) '(RIDICULOUSLY-HOT))
        ((LESSP TEMP 0) '(RIDICULOUSLY-COLD))
        (T '(OK))))

QUIT*

(check-temperature 102)(RIDICULOUSLY-HOT)
```

```
(check-temperature 25)(OK)

(check-temperature -32)(RIDICULOUSLY-COLD)
(check-temperature 0)(OK)

; it would be better i s'pose if the returns were'nt lists ?
[LEDIT Continued.]
[Reading from LEDIT...]
(defun check-temperature (temp)
  (cond ((greaterp temp 100) (car '(ridiculously-hot)))
        ((lessp temp 0) (car '(ridiculously-cold)))
        (t (car '(ok)))))
CHECK-TEMPERATURE

[LEDIT Completed.]function: check-temperature


(DEFUN CHECK-TEMPERATURE (TEMP)
  (COND ((GREATERP TEMP 100) (CAR '(RIDICULOUSLY-HOT)))
        ((LESSP TEMP 0) (CAR '(RIDICULOUSLY-COLD)))
        (T (CAR '(OK)))))

QUIT*

(check-temperature 89)OK

(check-temperature 190)RIDICULOUSLY-HOT

(check-temperature -1500000000)RIDICULOUSLY-COLD
```

The function CIRCLE is exercise 3-13 on page 47 of "Lisp". The problem statement is:

"Problem 3-13: Define CIRCLE such that it returns a list of the circumference and area of a circle whose radius is given. Assume PI is to be a free variable with the appropriate value."

```
; problem 3.13
[LEDIT Continued.]
[Reading from LEDIT...]
(defun circle (radius-1)
  (list (times 2 pi radius-1)
        (times pi radius-1 radius-1)))
CIRCLE
[LEDIT Completed.]function: circle

(DEFUN CIRCLE (RADIUS-1)
  (LIST (TIMES 2 PI RADIUS-1) (TIMES PI RADIUS-1 RADIUS-1)))

QUIT*

(setq pi 3.142)3.142

(circle 1)(6.284 3.142)

(circle 5)(31.4199998 78.549999)

(circle pi)(19.7443278 31.018339)

; its just occured to me that i went around the houses a bit in
3.12
[LEDIT Continued.]
```

```
[Reading from LEDIT...]
(defun check-temperature (temp)
  (cond ((greaterp temp 100)  'ridiculously-hot)
        ((lessp temp 0) 'ridiculously-cold)
        (t  'ok)))
CHECK-TEMPERATURE

[LEDIT Completed.]function: check-temperature

(DEFUN CHECK-TEMPERATURE (TEMP)
  (COND ((GREATERP TEMP 100) 'RIDICULOUSLY-HOT)
        ((LESSP TEMP 0) 'RIDICULOUSLY-COLD)
        (T 'OK)))
QUIT*

(check-temperature 13)OK

(check-temperature 104)RIDICULOUSLY-HOT

; thats better

function: member
MEMBER compiled.

QUIT*

(stop)
```

The next dribble file shows how on-line advice can help a student.

```
|Dribbling.|

(defun stop ()
  (undribble)
  (quit))
STOP

(setq W nil)
NIL

(

(atom (cadr '((b c) (d e))))
)
;NIL UNDEFINED FUNCTION OBJECT
QUIT*
(atom (cadr '((b c) (d e))))NIL

(atom d)
;D UNBOUND VARIABLE
QUIT*
(cadr ((b c) (d e)))
;B UNDEFINED FUNCTION OBJECT
QUIT*
(cadr '((b c) (d e)))(D E)


(cdr '((b c) (d e)))((D E))

(cons 'a ())(A)
```

The functions MYSTERY, STRANGE and SQUASH are exercises 4-1, 4-2 and 4-3 on page 57 of "Lisp". The problem statements are:

"Problem 4-1: Describe the evident purpose of the following function:

```
(DEFUN MYSTERY (S)
 (COND ((NULL S) 1)
    ((ATOM S) 0)
     (T (MAX (ADD1 (MYSTERY (CAR S)))
         (MYSTERY (CDR S))))))
```

Problem 4-2: Describe the evident purpose of the following function:

```
(DEFUN STRANGE (L)
 (COND ((NULL L) NIL)
    ((ATOM L) L)
    (T (CONS (STRANGE (CAR L))
         (STRANGE (CDR L))))))
```

Problem 4-3: Define SQUASH, a function that takes an s-expression as its argument and returns a nonnested list of all atoms found in the s-expression. Here is an example:

```
(SQUASH '(A (A (A (A B))) (((A B) B) B) B))
 (A A A A B A B B B B)
```

```
[LEDIT Created.]
[Reading from LEDIT...](defun squash (lista)
  (cond ((null lista) nil)
        ((atom lista) lista))
        (t (cons (squash (car lista))
                 ())))
SQUASH

[LEDIT Completed.]
(setq l '(a (b c) (d e)))(A (B C) (D E)))

(squash l)
;T UNDEFINED FUNCTION OBJECT

|brin.lsp|(defun first (exp-1) (car exp-1))
(defun rest (exp-1) (cdr exp-1))
(defun insert (new exp-1) (cons new exp-1))
(defun rotate-l (exp-1) (append (rest exp-1) (cons (first exp-1)
nil)))
(defun rotate-r (exp-1) (append (cons (first (reverse exp-1)) nil)
(reverse (rest (reverse exp-1)))))
(defun palindromize (exp-1) (append exp-1 (reverse exp-1)))
(defun f-to-c (f) (difference (quotient (plus f 40) 1.8) 40))
(defun c-to-f (c) (difference (times (plus c 40) 1.8) 40))
(defun roots (a b c) (list (quotient (plus (minus b)(sqrt
(difference (expt b 2) (times 4 a c)))) (times 2 a)) (quotient
(difference (minus b) (sqrt (difference (expt b 2) (times 4 a
c))))(times 2 a))))
(defun evenp (num) (zerop (remainder num 2)))
(defun palindromep (list1) (equal list1 (reverse list1)))
(defun rightp (elta eltb eltc)
  (equal (expt elta 2) (plus (expt eltb 2) (expt eltc 2))))
(defun complexp (a b c)
  (lessp (difference (expt b 2) (times 4 a c)) 0))
(defun nilcar (exp-1)
  (cond ((null exp-1) nil)
        (t (first exp-1))))
(defun nilcdr (exp-1)
  (cond ((null exp-1) nil)
        (t (rest exp-1))))
(defun check-temperature (temp)
  (cond ((greaterp temp 100)  'ridiculously-hot)
        ((lessp temp 0) 'ridiculously-cold)
        (t  'ok)))
(defun circle (radius-1)
  (list (times 2 pi radius-1)
  (times pi radius-1 radius-1)))
(defun mystery (s)
  (cond ((null s) 1)
        ((atom s) 0)
        (t (max (add1 (mystery (car s)))
                (mystery (cdr s))))))
(defun strange (l)
  (cond ((null l) nil)
        ((atom l) l)
        (t (cons (strange (car l))
                 (strange (cdr l))))))
(defun squash (lista)
  (cond ((null lista) nil)
        ((atom lista) lista))
        (t (cons (squash (car lista))
                 ())))
```

```
QUIT*

(strange l)(A (B C) (D E))

l
(A (B C) (D E))

(atom l)NIL

(atom (car l))T

(atom (cdr l))NIL

(atom (cadr l))NIL

(atom (caadr l))T

(caadr l)B

(setq l (cdr l))((B C) (D E))

l
((B C) (D E))

(setq l (cdr l))((D E))

(setq l (cdr l))NIL

(setq l '((d e)))((D E))

(car l)(D E)

(caar l)D

(cdr l)NIL

[LEDIT Continued.]
[LEDIT Completed.];;hello brian is the problem to do with squash?
; most definitely
;;what exactly?
; i think the solution should be the same as starnge in 4-2, but
; it obviously isnt. i just cant see a way of doing it
function: squash

(DEFUN SQUASH (LISTA)
  (COND ((NULL LISTA) NIL) ((ATOM LISTA) LISTA))
  (T (CONS (SQUASH (CAR LISTA)) NIL)))

QUIT*

; my edit file contains an old attempt
;;what you need is a way of stripping brackets from a list, the
;;way to do this is use append eg. append (1 2 3) (4 5 6)
;;gives (1 2 3 4 5 6)
;;does that help at all?
; possibly. i tried using append in an earlier attempt b
ut

;UT UNBOUND VARIABLE
*

; ut i got an error, so i guess i was'nt using it properly.
; i'll give it another try.
```

```
;;the most likely reason for your error is that both arguments to
append
;;must be lists, append can't strip a bracket off of an atom, so
you'll
;;need to cope with that if lista is an atom, I'll leave you to
think
;;about that but ring again in another ten minutes if your still
stuck
; ok thanx
(setq a 'z)Z

(list a)(Z)

(list 'a)(A)

[LEDIT Continued.]
[Reading from LEDIT...](defun squash (lista)
  (cond ((null lista) nil)
        ((atom lista) (list lista))
        (t (append (squash (car lista))
                   (squash (cdr lista))))))
SQUASH

[LEDIT Completed.]function: squash


(DEFUN SQUASH (LISTA)
  (COND ((NULL LISTA) NIL)
        ((ATOM LISTA) (LIST LISTA))
        (T (APPEND (SQUASH (CAR LISTA)) (SQUASH (CDR LISTA))))))

QUIT*

l
((D E))

s

;S UNBOUND VARIABLE
*
(setq l '(a (b c) (d e)))(A (B C) (D E))

(squash l)(A B C D E)

; whhoooppeeeee. to think i was so close about 5 hours ago !!
(squash '(a (a (a (a b))) (((a b) b) b)))(A A A A B A B B B)

(stop)
```

APPENDIX D

INSTRUCTIONS FOR STUDY II

ITSY

When you make an error, the first thing that will happen is that the 'real' error message (the error message tht would have appeared if you were not using ITSY) will appear, the second is that ITSY will provide a short tutorial using six frames. Each frame contains a message and a menu. The six frames are as follows:

1. The 'Question Frame'. The aim of this frame is to check that ITSY has the right diagnosis fo the error and you want a tutorial. The menus consists of the following choisces:

Yes:	Click on this if you want the tutorial
No:	This will take you back to Lisp
Explain Question:	This will provide an explanation of the question

2. The 'Explain Question Frame'. This will provide an explanation of the question if have trouble understanding it. The menu consists of the following choices:

OK:	Click on this to go back to the question.

3. The 'Main Explanation Frame'. This shoul dgive a short explanation of the error. The menu consists of the following choices:

Examples:	Click on this to go to the example menu
Deeper Explanation:	Click on this to go to the deeper explanation menu
Cancel:	Click on this to go back to Lisp.

4. The 'Deeper Explanation Frame'. This will give a longer explanation of th error. The menu consists of the following choices:

Examples:	Click on this to go to the example menu
Cancel:	Click on this to go back to Lisp.

5. The 'Examples Frame'. This will give some examples. The menu consists of the following choices:

Cancel:	Click on this to go back to Lisp.

6. The 'Fix Frame'. This will give a possible 'fix' for the error.

If you are not sure what the first 'question frame' means, then click on 'yes'. If you click on 'no' you will not get the tutorial. As you progress ITSY will ask you if you want a tutorial before presenting it, and later still ITSY will put a 'Present Tutorial' option on the ITSY Lisp menu which you can select if you want the tutorial.

Once you start defining your own comments, you can select the 'Add Comments' option on the ITSY Lisp menu. Once you'v selected this you can

type your comments into the Lisp Window. To end the comments type end at the beginning of a line.

You can describe any Common Lisp function by selecting 'Describe Function' on the ITSY Lisp menu. ITSY will then print a description of the function.

ITSY Lisp Top Level

When you first use ITSY the screen will have two parts, the lisp top level and a status line. The status line is at the very bottom of the screen. The status line will give information such as whether ITSY is waiting for input or evaluating an s-expression or trying to find an error. If you click on the middle or right mouse button a menu will appear. If you click on 'describe a function' the type the name of a Common Lisp function a description of the function will appear.

Later the shape of the screen will change. An editor window will appear on top of the Lisp window. You can select either the editor window or the Lisp window by moving the mouse to the desired window and clicking on it with the Left mouse button.

In both the Lisp window and the editor window various actions can be carried out by usng control keys and the mouse; a summary of the editor commands is contained on another sheet.

The following is a summary of the key commoands available in the Lisp window.

control-f means hold the control key down while pressing the f key.

meta-f means hold the meta key down while pressing the f key

control-meta-f means hold both the control and meta key down while pressing the f key.

Key Command	Action
control-f	Move forward a character
control-b	Move backward a character
meta-f	Move forward a word
meta-b	Move backward a word
control-k	Kill the current line from the cursor position
control-c	Yank the previous input
meta-c	(use only after a control-c). Yank the input before the previous input.
control-a	Move to the beginning of the line
control-e	Move to the end of the line
control-d	Delete the next character
rubout	Delete the previous character
meta-d	Delete the next word

The Editor

The following is the correct sequence to use in order to write a function and load it into Lisp.

1. Select the editor window by moving the mouse over the window and clicking the left mouse button, or by selecting the editor option on the menu. (see menu)

2. Find a file. You will be prompted for a name. Type the name of your file (You could use your name as the filename) (see menu)

3. Type in the definition of your function.

4. Save the buffer - (see menu).

5. Load the file into Lisp (see menu).

If you make changes to the buffer once you have loaded the file, you can load the changed functions by selecting 'Load changed Functions' on the menu.

Editor Commands

The following is a summary of the key commands available in the Editor window.

control-f means hold the control key down while pressing the f key.

meta-f means hold the meta key down while pressing the f key.

control-meta-f means hold both the control and meta key down while pressing the f key.

Key Command	Action
control-f	Move forward a character
control-b	Move backward a character
meta-f	Move forward a word
meta-b	Move backward a word
control-meta-f	Move forward an s-expression
control-meta-b	Move backward an s-expression
control-n	Move down one line
control-p	Move up one line
control-k	Kill the current lien from the cursor

control-a	Move to the beginning of the line
control-e	Move to the end of the line
control-d	Delete the next character
rubout	Delete the previous character
meta-d	Delete the next word
meta-rubout	Delete the previous word
control-meta-d	Delete the next s-expression
control-meta-rubout	Delete the previous s-expression
control-v	Move down a screen
meta-v	Move up a screen
Mouse Left	Select the window under the mouse.
	It is possible to move the character cursor around once inside the editor using the mouse. Move the mouse to the desired position and then press the left button.

Mouse Middle Mouse Right
Bring up a menu

Menu Choices are

Lisp:	Go to the Lisp window
Find a file:	Create or find a file
Load a file:	Load an existing file into Lisp A menu will appear containing all of the files you have saved. Selecting one of these will load the file into the Lisp environment.
Save this Buffer:	Save the current Buffer
List all the Buffers:	List the current buffers. A list of buffers will appear at the top of the window. You can select one of these by clicking on it with the mouse
Load Changed Functions:	Load the Changed functions into Lisp.
meta-rubout	Delete the previous word

APPENDIX E

Total Number Errors for Study II

The first number given is the number of errors made in the particular category. The second number given is the percentage of the total number of errors made in this category.

Error

Apply CARs and CDRs in the wrong order

4 1.1

Arguments in the wrong form

9 2.4

A Slip Up

3 0.8

Bracket Error Outside a Clause

3 0.8

Bracket around a quoted list

1 0.3

Brackets around a number

1 0.3

Brackets around a variable

14 3.8

Brackets around a variable in COND

2 0.5

Brackets around an unbound atom

5 1.3

Calling a Function that doesn't exist

10 2.7

Caused by Winston and Horn

2 0.5

Error due experimenter

0

Extra Set of Brackets around a function call

15 4.0

Extra set of brackets around a function call inside a COND

18 4.8

Function not loaded because of another error

1 0.3

Forgetting to load a function

0

Incorrect algorithm

111 29.8

Misreading the question

0

Missing a Function Call

2 0.5

No brackets around a function call

19 5.1

No brackets around a function call in a COND

7 1.9

No brackets around a test function

8 2.2

No gaps between atoms

3 0.3

Non-lists to Cons car cdr append

12 3.2

Not Closing the Test part of a Clause

11 3.0

Not Quoting

31 8.3

Not Setting a Variable before using it

3 0.3

Not Trying a function on the right input

0

Problem with load changed functions

0

Quoting a Function

1 0.3

quoting a variable

4 1.1

Quoting and putting brackets around a variable

0

Slip with a Bracket

1 0.3

Spurious Character in File

8 2.2

Stack overflow due to infinite recursion

1 0.3

Text spelling error

13 3.5

Trying to give a value to a parameter globally

6 1.6

Unbound variable because of lexical Scoping

8 2.2

Unbound variable due to the deletion of a parameter

0

Unknown

4 1.1

Wrong Combination of Cars and Cdrs

0

Wrong number of arguments

21 5.6

Wrong type argument

10 2.7

Total number of errors 372

Number of times Advice given 10

APPENDIX F

RESULTS FOR STUDY III

Total Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Applying CARs and CDRs in the Wrong Order	0	0	0	0	0	0	6	9	0
Arguments in the Wrong Form	6	2	0	7	1	0	4	0	0
A Slip Up	0	0	0	0	0	0	5	0	10
Bracketing Error Inside a COND	0	0	0	0	0	0	2	0	0
Brackets Around a Number	0	0	0	1	1	0	0	0	0
Brackets Around a Quoted List	1	0	0	0	0	0	0	0	0
Brackets Around an Unbound Atom	0	0	0	0	0	0	0	0	1

Total Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Brackets Around a Variable	7	2	0	4	0	0	1	2	0
Calling an Undefined Function	0	0	0	0	2	0	3	0	1
Caused by Winston and Horn	0	0	0	0	0	1	0	1	8
Error Due to Experimenter	0	0	0	1	0	0	0	0	0
Extra set of Brackets Around a Function Call	3	0	0	2	0	1	1	0	0
Extra Set of Brackets Around a Function Call Inside a COND	2	0	0	1	3	0	1	0	0
Endless Looping	0	0	0	0	0	0	2	1	0
Function Not Loaded Because of Error	0	0	0	0	0	0	1	0	0

Total Analysis

Error	Used ITSY and fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Forgetting to Load a Function	0	0	0	0	0	0	4	2	0
Incorrect Algorithm	0	0	0	0	0	0	16	130	1
Misreading the Question	0	0	0	0	0	0	1	6	1
No Brackets Around a Function Call	11	7	0	16	2	1	1	0	0
No Brackets Around a Function Call Inside a COND	2	0	0	0	0	0	1	0	0
No gaps Between Atoms	0	0	0	0	0	0	3	1	1
Non-lists Given to CAR CDR CONS APPEND	9	2	1	5	6	1	8	17	1

Total Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Not Quoting	10	1	1	6	1	1	13	5	3
Not SETQing a Variable Before Using It	0	0	0	0	0	0	1	0	0
Not Trying a Function on the Right Input	0	0	0	0	0	0	0	2	0
Problem with Load Changed Functions	0	0	0	0	0	0	3	0	0
Quoting and Putting Brackets Around a Variable	1	0	0	0	0	0	1	0	1
Quoting a Variable	2	0	0	1	0	0	1	7	
Stack Overflow Due to Infinite Recursion	0	0	0	0	0	0	3	7	0
Text spelling error	0	0	0	0	0	0	23	3	2

Total Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Unbound Variable Because of Lexical Scoping	0	0	0	0	0	0	1	1	0
Unbound Variable Due to the Deletion of a Parameter	0	0	0	0	0	0	1	0	0
Wrong Number of Arguments	2	0	0	1	2	0	0	2	0
Wrong Type of argument	3	1	1	0	0	0	0	0	1
Total	59	15	3	45	18	4	107	197	31

APPENDIX G

INDIVIDUAL RESULTS FOR STUDY III

C2's Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Applying CARs and CDRs in the Wrong Order							1	3	
Arguments in the Wrong Form	1			2	1		3		
A Slip Up									2
Bracketing Error Inside a COND									
Brackets Around a Number					1				
Brackets Around a Quoted List									
Brackets Around an Unbound Atom									

C2's Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Brackets Around a Variable							1	2	
Calling an Undefined Function					2				
Caused by Winston and Horn									2
Error Due to Experimenter									
Extra set of Brackets Around a Function Call				1					
Extra Set of Brackets Around a Function Call Inside a COND							1		
Endless Looping							2	1	
Function Not Loaded Because of Error									

C2's Analysis

Error	Used ITSY and fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Forgetting to Load a Function								1	
Incorrect Algorithm							4	21	1
Misreading the Question							1	5	
No Brackets Around a Function Call	4	3		6	2				
No Brackets Around a Function Call Inside a COND	1						1		
No gaps Between Atoms							1	1	1
Non-lists Given to CAR CDR CONS APPEND	2			2	1		1		

C2's Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Not Quoting	1	1		1			3	1	2
Not SETQing a Variable Before Using It									
Not Trying a Function on the Right Input									
Problem with Load Changed Functions									
Quoting and Putting Brackets Around a Variable									
Quoting a Variable									
Stack Overflow Due to Infinite Recursion								4	
Text spelling error							3	2	

C2's Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Unbound Variable Because of Lexical Scoping									
Unbound Variable Due to the Deletion of a Parameter									
Wrong Number of Arguments									
Wrong Type of argument	1								
Total	10	4	0	12	7	0	22	41	8

L's Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Applying CARs and CDRs in the Wrong Order								5	
Arguments in the Wrong Form	2								
A Slip Up							2		5
Bracketing Error Inside a COND							1		
Brackets Around a Number									
Brackets Around a Quoted List									
Brackets Around an Unbound Atom									

L's Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Brackets Around a Variable	2								
Calling an Undefined Function									
Caused by Winston and Horn								1	2
Error Due to Experimenter									
Extra set of Brackets Around a Function Call	2								
Extra Set of Brackets Around a Function Call Inside a COND	1								
Endless Looping									
Function Not Loaded Because of Error							1		

L's Analysis

Error	Used ITSY and fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Forgetting to Load a Function									
Incorrect Algorithm							1	13	
Misreading the Question									
No Brackets Around a Function Call	6	3		2					
No Brackets Around a Function Call Inside a COND	1								
No gaps Between Atoms									
Non-lists Given to CAR CDR CONS APPEND	3	1	1				1	2	

L's Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Not Quoting	4			1			2	2	1
Not SETQing a Variable Before Using It							1		
Not Trying a Function on the Right Input									
Problem with Load Changed Functions									
Quoting and Putting Brackets Around a Variable	1								
Quoting a Variable	1								
Stack Overflow Due to Infinite Recursion							2	1	
Text spelling error							9	1	1

L's Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Unbound Variable Because of Lexical Scoping									
Unbound Variable Due to the Deletion of a Parameter									
Wrong Number of Arguments									
Wrong Type of argument									
Total	23	4	1	3	0	0	20	25	9

S's Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Applying CARs and CDRs in the Wrong Order							5		
Arguments in the Wrong Form	2	1		4					
A Slip Up									
Bracketing Error Inside a COND									
Brackets Around a Number									
Brackets Around a Quoted List									
Brackets Around an Unbound Atom									1

S's Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Brackets Around a Variable	4	1		4					
Calling an Undefined Function							2		1
Caused by Winston and Horn						1			2
Error Due to Experimenter				1					
Extra set of Brackets Around a Function Call	1						1	1	
Extra Set of Brackets Around a Function Call Inside a COND	1			1	3				
Endless Looping									
Function Not Loaded Because of Error									

S's Analysis

Error	Used ITSY and fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Forgetting to Load a Function							4		
Incorrect Algorithm							6	92	
Misreading the Question								1	1
No Brackets Around a Function Call	1			3		1			
No Brackets Around a Function Call Inside a COND									
No gaps Between Atoms							1		
Non-lists Given to CAR CDR CONS APPEND	4	1		3	5	1	6	15	1

S's Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Not Quoting	4		1	2	1	1	6		
Not SETQing a Variable Before Using It									
Not Trying a Function on the Right Input								2	
Problem with Load Changed Functions							2		
Quoting and Putting Brackets Around a Variable							1		
Quoting a Variable	1			1			1	7	
Stack Overflow Due to Infinite Recursion								2	
Text spelling error							7		

S's Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Unbound Variable Because of Lexical Scoping									
Unbound Variable Due to the Deletion of a Parameter							1		
Wrong Number of Arguments	2			1					
Wrong Type of argument	2	1	1						1
Total	22	4	2	20	9	4	43	120	7

S2's Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Applying CARs and CDRs in the Wrong Order								1	
Arguments in the Wrong Form	1	1		1			1		
A Slip Up							3		3
Bracketing Error Inside a COND							1		
Brackets Around a Number				1					
Brackets Around a Quoted List	1								
Brackets Around an Unbound Atom									

S2's Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Brackets Around a Variable	1	1							
Calling an Undefined Function							1		
Caused by Winston and Horn									2
Error Due to Experimenter									
Extra set of Brackets Around a Function Call				1					
Extra Set of Brackets Around a Function Call Inside a COND									
Endless Looping									
Function Not Loaded Because of Error									

S2's Analysis

Error	Used ITSY and fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Forgetting to Load a Function								1	
Incorrect Algorithm							5	4	
Misreading the Question									
No Brackets Around a Function Call		1		5			1		
No Brackets Around a Function Call Inside a COND									
No gaps Between Atoms							1		
Non-lists Given to CAR CDR CONS APPEND									

S2's Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Not Quoting	1			2			2	2	
Not SETQing a Variable Before Using It									
Not Trying a Function on the Right Input									
Problem with Load Changed Functions							1		
Quoting and Putting Brackets Around a Variable									1
Quoting a Variable									
Stack Overflow Due to Infinite Recursion							1		
Text spelling error							4		1

S2's Analysis

Error	Used ITSY and Fixed Error	Used ITSY and Didn't Fix Error	Used ITSY and Left Error	Didn't Use ITSY and Fixed Error	Didn't Use ITSY and Didn't Fix Error	Didn't Use ITSY and Left Error	Error Not Caught and Fixed	Error Not Caught and NOT Fixed	Error Not Caught and Left
Unbound Variable Because of Lexical Scoping							1	1	
Unbound Variable Due to the Deletion of a Parameter									
Wrong Number of Arguments					2			2	
Wrong Type of argument									
Total	4	3	0	10	2	0	21	11	7

APPENDIX H

DRIBBLE FILE FOR STUDY III

After the event comments are in this font.

```
"Dribble for LOUISE at It is Friday the nineteenth of September
1986; and the ti
me is: 15 16 41



"
"*- "(QUOTE (((((NIL)))))
(((((NIL)))))
"*- "(QUOTE (DDD))
(DDD)
"*- "(CAR (CDR (CDR (CDR (QUOTE (((A) (O) (P) (G)))))))
NIL
"*- "(CAR (CDR (CDR (CDR (QUOTE (((A) (O) (P) (G)))))))
NIL
"*- "(CAR (CDR (CDR (CAR (QUOTE (((A) (O) (P) (G)))))))
(P)
"*- "(CAR (CDR (CAR (CDR (CDR (QUOTE (A (O) ((P)) (((G)))))))))
NIL
"*- "(CAR (CAR (CDR (CDR (QUOTE (A (O) ((P)) (((G)))))))))
(P)
"*- "(CDR (CAR (QUOTE ((((A) O) P) G))))
(P)
"*- "(APPEND (QUOTE (A B C)) (QUOTE NIL))
(A B C)
"*- "(LIST (QUOTE (A B C)) (QUOTE NIL))
((A B C) NIL)
"*- "(CONS (QUOTE (A B C)) (QUOTE NIL))
((A B C))
"*- "(SETQ TOOLS (LIST (QUOTE HAMMER) (QUOTE SCREWDRIVER)))
(HAMMER SCREWDRIVER)
"*- "(CONS (QUOTE PLIERS) TOOLS)
(PLIERS HAMMER SCREWDRIVER)
"*- "TOOLS
(HAMMER SCREWDRIVER)
"*- "(SETQ TOOLS (CONS (QUOTE PLIERS) TOOLS))
(PLIERS HAMMER SCREWDRIVER)
"*- "TOOLS
```

```
(PLIERS HAMMER SCREWDRIVER)
"*- "(APPEND (QUOTE (SAW WRENCH)) TOOLS)
(SAW WRENCH PLIERS HAMMER SCREWDRIVER)
"*- "TOOLS
(PLIERS HAMMER SCREWDRIVER)
"*- "(SETQ TOOLS (APPEND (QUOTE (SAW WRENCH)) TOOLS))
(SAW WRENCH PLIERS HAMMER SCREWDRIVER)
"*- "TOOLS
(SAW WRENCH PLIERS HAMMER SCREWDRIVER)
"*- "(LENGTH (QUOTE (PLATO SOCRATES ARISTOTLE)))
3
"*- "(LENGTH (QUOTE ((PLATO) (SOCRATES) (ARISTOTLE))))
3
"*- "(REVERSE (QUOTE (PLATO SOCRATES ARISTOTLE)))
(ARISTOTLE SOCRATES PLATO)
"*- "(REVERSE (QUOTE ((PLATO) (SOCRATES) (ARISTOTLE))))
((ARISTOTLE) (SOCRATES) (PLATO))
"*- "(REVERSE (QUOTE ((PLATO SOCRATES ARISTOTLE))))
((PLATO SOCRATES ARISTOTLE))
"*- "(LENGTH (QUOTE ((CAR CHEVROLET) (DRINK COKE) (CEREAL
WHEATIES))))
3
"*- "(REVERSE (QUOTE ((CAR CHEVROLET) (DRINK COKE) (CERAL
WHEATIES))))
((CERAL WHEATIES) (DRINK COKE) (CAR CHEVROLET))
"*- "(APPEND (QUOTE ((CAR CHEVROLET) (DRINK COKE))))
((CAR CHEVROLET) (DRINK COKE))
"*- "(SUBST (QUOTE OUT) (QUOTE IN) (SHORT SKIRTS ARE IN))
```

The subject has made her first error. The following three lines are internal Lisp error handling stuff.

```
(ERROR (SUBST (QUOTE OUT) (QUOTE IN) (SHORT SKIRTS ARE IN))
 ZETALISP-SYSTEM:UNDE
FINED-FUNCTION #<UNDEFINED-FUNCTION-TRAP 16643677>)
```

This is the Lisp error message

```
The function SHORT is undefined.
```

This is internal stuff from ITSY.

```
"message: #<N-Q-LIST-EXP 24534630> type: N-Q-LIST orig-type: NIL
:extra-info NIL
 :fn-name (TOP-LEVEL SHORT SKIRTS ARE IN) code: NIL"
"message: #<N-Q-LIST-EXP 24534630> type: N-Q-LIST orig-type: NIL
:extra-info NIL
 :fn-name (SHORT SKIRTS ARE IN) code: NIL"
"Student know N-Q-LIST 2"
"message: #<N-Q-LIST-TOP-LEVEL-EXP 24535230> type: N-Q-LIST-TOP-
```

```
LEVEL orig-type:
 N-Q-LIST :extra-info NIL :fn-name NIL code: (SHORT SKIRTS ARE
IN)"
```

These are the error messages that could be displayed. The order is Question, Question Explanation, Main Explanation, Deeper Explanation, Examples and Fix.

```
"  Did you intend to do one of the following:
      ~%1. use the list (SHORT SKIRTS ARE IN) as an argument
rather than
      ~%call the function SHORT.
      ~%2. call the function SHORT"
"  Did you want to
      ~%1. give the literal value of the list (SHORT SKIRTS ARE
IN)
      ~%rather than have (SHORT SKIRTS ARE IN) evaluated or
      ~%2. call the function SHORT."
"  You wanted to
      ~%1. give the literal value of the list (SHORT SKIRTS ARE
IN)
      ~%rather than have (SHORT SKIRTS ARE IN) evaluated or
      ~%2. call the function SHORT.~%
      ~%If you wanted to carry out 1
      ~%the interpreter thinks that you want the
      ~%call the function SHORT with the arguments SKIRTS ARE IN
      ~%instead of giving the list (SHORT SKIRTS ARE IN) as an
argument.~%
      ~%If you wanted to carry out 2
      ~%the interpreter cannot call the function
      ~%SHORT because it is not defined. Maybe you
      ~%have misspelt the function name, or you
      ~%have forgotten to load the file containing
      ~%the function definition."
"  The correct way to give the literal value of list
      ~%as the argument to a function is to quote it.
      ~%The correct way to define a function is to use
      ~%defun see pages 39-43 of Winston and Horn."
"~%(append      ~$'(cons 'a '(b c))~&  ~$'(1 2 3 4)~&)~%~%
   two literal lists as arguments to append
   ~%~%~$'(a literal list)~&~%~%
            a literal list
      ~%~%~%If we want to call a function ~$foo~&
```

```
      ~%to add two numbers then we can define foo by
      ~%~%(defun foo (x y) (+ x y))"
" If you want to give the lists (X Y Z) and (1 2 3)
      ~%as arguments to APPEND, then
      ~%(APPEND (X Y Z) (1 2 3)) would be wrong, but
      ~%(APPEND '(X Y Z) '(1 2 3)) would be right."
```

The subject has decided to look at the Question Frame

```
"Expose Question 1 It is Friday the nineteenth of September 1986;
and the time i
s: 15 31 56"
"I'm Exp Menu 1 Choose YES at It is Friday the nineteenth of
September 1986; and
 the time is: 15 32 6"
```

The subject has selected the item Yes on the Question menu, and the Main Explanation frame is exposed.

```
"Expose Main Explanation 1 It is Friday the nineteenth of
September 1986; and th
e time is: 15 32 15"
```

The subject has decided to cancel the rest of the tutorial.

```
"I'm Exp Menu 3 Choose CANCEL at It is Friday the nineteenth of
September 1986;
and the time is: 15 33 29"
"Explanation Over It is Friday the nineteenth of September 1986;
and the time is
: 15 33 52"
NIL
"*- "(SUBST (QUOTE OUT) (QUOTE IN) (QUOTE (SHORT SKIRTS ARE IN)))
(SHORT SKIRTS ARE OUT)
"*- "(SUBST (QUOTE IN) (QUOTE OUT) (QUOTE (SHORT SKIRTS ARE IN)))
(SHORT SKIRTS ARE IN)
"*- "(LAST (QUOTE (SHORT SKIRTS ARE IN)))
(IN)
"*- "(SETQ METHOD1 (QUOTE +))
+
"*- "(SETQ METHOD2 (QUOTE -))
-
"*- "(SETQ METHOD METHOD1)
+
```

```
"*- "METHOD
+
"*- "(EVEL METHOD)
(ERROR (EVEL METHOD) ZETALISP-SYSTEM:UNDEFINED-FUNCTION
#<UNDEFINED-FUNCTION-TRA
P 16722125>) The function EVEL is undefined.
```

ITSY was unable to find this error.

```
"Unable to find any errors"
"looking at warn about possible cause It is Friday the nineteenth
of September 1
986; and the time is: 15 37 4"
"finished looking at about possible cause It is Friday the
nineteenth of Septemb
er 1986; and the time is: 15 38 0"
NIL
"*- "(EVAL METHOD)
(ERROR (EVAL METHOD) ZETALISP-SYSTEM:UNBOUND-VARIABLE #<UNBOUND-
SYMBOL-TRAP 1673
5233>) The variable + is unbound.
```

ITSY was unable to find this error.

```
"Unable to find any errors"
"looking at warn about possible cause It is Friday the nineteenth
of September 1
986; and the time is: 15 40 10"
"finished looking at about possible cause It is Friday the
nineteenth of Septemb
er 1986; and the time is: 15 40 16"
NIL
```

The subject has brought up the Editor Menu and selected the Go to Lisp item.

```
#"I'm Menu Ptype 1 Choose GO-TO-LISP at It is Friday the
nineteenth of September
 1986; and the time is: 15 41 41"
"lisp mouse click It is Friday the nineteenth of September 1986;
and the time is
: 15 41 52"
"I'm Ptype Dynamic Momentary Menu 1 Choose NIL at It is Friday
the nineteenth of
 September 1986; and the time is: 15 41 58"
"lisp mouse click It is Friday the nineteenth of September 1986;
and the time is
: 15 41 59"
```

The subject has brought up a menu and not selected any item (the subject accomplished this by moving the mouse away from the menu).

```
"I'm Ptype Dynamic Momentary Menu 1 Choose NIL at It is Friday
the nineteenth of
 September 1986; and the time is: 15 42 1"
"lisp mouse click It is Friday the nineteenth of September 1986;
and the time is
: 15 42 1"
"I'm Ptype Dynamic Momentary Menu 1 Choose NIL at It is Friday
the nineteenth of
 September 1986; and the time is: 15 42 10"
"lisp mouse click It is Friday the nineteenth of September 1986;
and the time is
: 15 42 10"
"I'm Ptype Dynamic Momentary Menu 1 Choose NIL at It is Friday
the nineteenth of
 September 1986; and the time is: 15 42 17"
"*- "(EVAL (EVAL (QUOTE (QUOTE METHOD))))
+
"lisp mouse click It is Friday the nineteenth of September 1986;
and the time is
: 15 45 1"
"I'm Ptype Dynamic Momentary Menu 1 Choose NIL at It is Friday
the nineteenth of
 September 1986; and the time is: 15 45 31"
"*- "(DEFUN OUR-FIRST (OURLIST) (CAR (QUOTE (OURLIST))))
OUR-FIRST
"*- "TOOLS
(SAW WRENCH PLIERS HAMMER SCREWDRIVER)
"*- "(OUR-FIRST TOOLS)
OURLIST
"*- "CAR
(ERROR CAR ZETALISP-SYSTEM:UNBOUND-VARIABLE #<UNBOUND-SYMBOL-TRAP
20600620>) The
 variable CAR is unbound.
```

ITSY has correctly trapped this error.

```
"message: #<NO-BKT-FN-EXP 24534570> type: NO-BKT-FN orig-type:
NIL :extra-info N
IL :fn-name (TOP-LEVEL . CAR) code: NIL"
"message: #<NO-BKT-FN-EXP 24534570> type: NO-BKT-FN orig-type:
NIL :extra-info N
IL :fn-name CAR code: NIL"
```

```
"Student know NO-BKT-FN 2"
"message: #<NO-BKT-FN-TOP-LEVEL-EXP 24535270> type: NO-BKT-FN-
TOP-LEVEL orig-typ
e: NO-BKT-FN :extra-info NIL :fn-name NIL code: CAR"
"  Does CAR refer to the function CAR rather
   ~%than the variable CAR?"
"  Did you want to call the function CAR
   ~%rather than get the value of the variable CAR?"
"  The interpreter thinks that you want the
   ~%value of the variable CAR rather than call
   ~%the function CAR."
"  The correct way to call a function is to
   ~%write a single opening bracket followed
   ~%first by the name of the function, then
   ~%its arguments and finally a closing
   ~%bracket. If you leave out the opening
   ~%and closing brackets the interpreter will
   ~%think that the function name is intended
   ~%as a variable name, and will try to
   ~%evaluate it."
"~%   (~$*~&          ~$6~&  ~$9~&)
   ~%   function    arguments
   ~%~%  (~$length~&       ~$'(a b c)~&)
   ~%   function    argument
   ~%~%   (~$subst~&       ~$cats~& ~$dogs~& ~$'(dogs drink
milk)~&
   ~%   function         arguments"
"  For CAR to be regarded as a function
   ~%you need to add the pair of brackets
   ~%to surround the function and the
   ~%arguments. If I wanted to call the
   ~%function FOO with arguments 1 and 2
   ~%FOO 1 2 would be wrong but (FOO 1 2)
   ~%would be right."
"Expose Question 1 It is Friday the nineteenth of September 1986;
and the time i
s: 16 2 10"
"I'm Exp Menu 1 Choose NO at It is Friday the nineteenth of
September 1986; and
the time is: 16 2 50"
"Explanation Over It is Friday the nineteenth of September 1986;
and the time is
: 16 2 58"
```

APPENDIX I

FRAME TIMES FOR STUDY III

S Message Times

Message	Time min	sec	Percentage of total time
Question	12	15	26
Explain Question	1	03	2
Main Explanation	9	33	20
Deeper Explanation	6	12	13
Fix	13	14	28
Example	5	39	12
Total	47	56	

C2 Message Times

Message	Time min	sec	Percentage of total time
Question	15	52	47
Explain Question	0	23	1
Main Explanation	13	04	39
Deeper Explanation	0	17	1
Fix	02	32	8
Example	1	25	4
Total	33	33	

L Message Times

Message	Time min	sec	Percentage of total time
Question	12	45	41
Explain Question	0	35	2
Main Explanation	5	52	19
Deeper Explanation	0	47	3
Fix	6	32	21
Example	4	18	14
Total	30	49	

S2 Total Message Times

Message	Time min	sec	Percentage of total time
Question	5	52	50
Explain Question	0	0	0
Main Explanation	2	42	23
Deeper Explanation	0	55	8
Fix	2	09	18
Example	0	10	1
Total	11	48	

Raw Times

Times	Q	EQ	M	F	Eg	De
C2	952	23	784	152	85	17
S	735	63	573	794	339	372
L	765	35	352	392	258	47
S2	352	0	162	129	10	55
TOTAL	2804	121	1871	1467	692	491
Total %	38	2	25	20	9	7

APPENDIX J

A List of all the Functions ITSY can Currently Analyse

ITSY can currently analyse the following 86 Common Lisp Functions:

+	*	/	max	min	expt
sqrt	-	float	truncate	rem	round
car	cdr	quote	caar	cadr	cdar
cddr	caaar	caadr	cadar	caddr	cdaar
cdadr	cddar	cdddr	caaaar	caaadr	caadar
caaddr	cadaar	cadadr	caddar	cadddr	cdaaar
cdaadr	cdadar	cdaddr	cddaar	cddadr	cdddar
cddddr	append	list	cons	length	reverse
subst	remove	last	eval	defun	atom
listp	equal	=	null	member	numberp
<	>	zerop	minusp	evenp	not
and	or	cond	abs	funcall	let
let*	setq	psetq	mapcar	oddp	apply
mapcan	do	return	do*	go	prog
prog1	progn				

APPENDIX K

THE STUDENT MODEL CLICHES

Although the student model cliches *No Brackets Around a Function Call* (Student Model Cliche 1) and*Extra Brackets Around a Function Call* (Student Model Cliche 3) have been enumerated separately, they have actually been merged in ITSY for efficiency reasons.

1. Student Model Cliche Name: *No Brackets Around a Function Call*

   Surface Code Segment: CL Function

   Criteria: None

   Other Checks: None

   Null Student Model Cliche Name: *No Brackets Around a Function Call*

   Surface Code Segment: Quote

   Student Model Cliche Name: *No Brackets Around a Function Call*

   Surface Code Segment: Function Application

   Criteria: The type of function application is normal or recursive

   Other Checks: None

2. Student Model Cliche Name: *Bracket Around a Variable*

   Surface Code Segment: Symbol

   Criteria: The symbol is bound and not the name of a function

   Other Checks: None

3. Student Model Cliche Name: *Extra Brackets Around a Function Call*

   Surface Code Segment: CL Function

   Criteria: None

   Other Checks: None

   Null Student Model Cliche Name: *Extra Brackets Around a Function Call*

   Surface Code Segment: Quote

   Student Model Cliche Name: *Extra Brackets Around a Function Call*

   Surface Code Segment: Function Application

   Criteria: The type of function application is normal or recursive

   Other Checks: None

4. Student Model Cliche Name: *Arguments in the Wrong Form1*

Surface Code Segment: Non Connective

Criteria: The function takes a set number of arguments
The function has been given the right number of arguments

Other Checks: None

Null Student Model Cliche Name: *Arguments in the Wrong Form1*

Surface Code Segment: Quote

5. Student Model Cliche Name: *Arguments in the Wrong Form2*

Surface Code Segment: Non Connective

Criteria: The function takes a set number of arguments
The function has been given the right number of arguments

Other Checks: None

Null Student Model Cliche Name: *Arguments in the Wrong Form2*

Surface Code Segment: Quote

6. Student Model Cliche Name: *Not Quoting a List*

Surface Code Segment: Quote

	Criteria:	The input to the segment is a list None of the elements of the list are functions
	Other Checks:	None
7.	Student Model Cliche Name:	*Not Quoting an Atom*
	Surface Code Segment:	Quote
	Criteria:	The input to the segment is an atom The atom is not a variable
	Other Checks:	None
8.	Student Model Cliche Name:	*Quoting a Variable*
	Surface Code Segment:	Symbol
	Criteria:	The symbol is bound
	Other Checks:	None
9.	Student Model Cliche Name:	*Quoting a Function Call*
	Surface Code Segment:	Function Application
	Criteria:	The function is of type normal or recursive
	Other Checks:	None

10. Student Model Cliche Name: *Quoting a Function Call*

Surface Code Segment: CL Function

Criteria: None

Other Checks: None

The student model cliche *Wrong Number of Arguments to a Function Call* on the surface code segment Non Connective relies on the fact that non-function non-connectives do not have a set number of arguments. This should really have been defined on the surface code segments Function Application and CL Function

11. Student Model Cliche Name: *Wrong Number of Arguments to a Function Call*

Surface Code Segment: Non Connective

Criteria: The function takes a set number of arguments
The function has been given the right number of arguments

Other Checks: None

Null Student Model Cliche Name: *Wrong Number of Arguments to a Function Call*

Surface Code Segment: Quote

12. Student Model Cliche Name: *Wrong Type of Argument Given to a*

Function Call

Surface Code Segment: Any-arg

Criteria: All of the arguments are of the right type

Other Checks: None

Student Model Cliche Name: *Wrong Type of Argument Given to a Function Call*

Surface Code Segment: One-arg

Criteria: The first argument is of the right type

Other Checks: None

Student Model Cliche Name: *Wrong Type of Argument Given to a Function Call*

Surface Code Segment: Two-args

Criteria: Both the first and the second argument are of the right type

Other Checks: None

Student Model Cliche Name: *Wrong Type of Argument Given to a Function Call*

Surface Code Segment: Three-args

Criteria: The first, second and third argument are of the right type

Other Checks: None

Null Student Model Cliche Name: *Wrong Type of Argument Given to a Function Call*

Surface Code Segment: Quote

13. Student Model Cliche Name: *Wrong Scope*

Surface Code Segment: Local Var

Criteria: None

Other Checks: None