

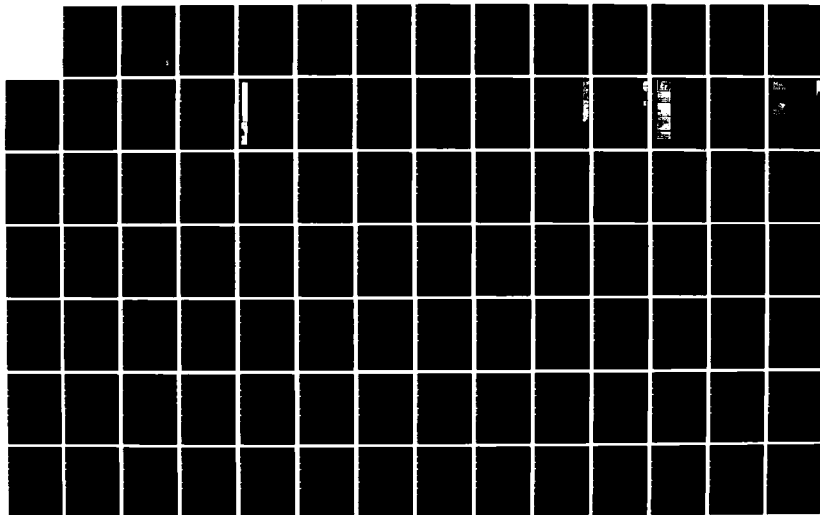
AD-A157 505

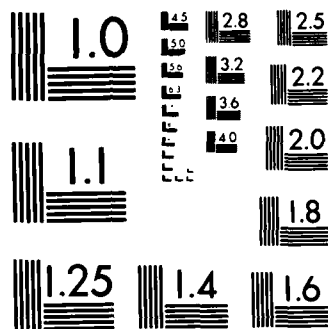
MICRO-PROUST(U) YALE UNIV NEW HAVEN CT DEPT OF COMPUTER 1/2
SCIENCE W L JOHNSON ET AL. JUN 85 YALEU/CSD/RR-402
N00014-82-K-0714

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NBS 1963-A

1 (2) 12

AD-A157 505



Micro-PROUST

W. Lewis Johnson and Elliot Soloway

YALEU/CSD/RR #402

June 1985

DTIC FILE COPY

This document has been approved
for public release and sale; its
distribution is unlimited.

DTIC
ELECTE
AUG 5 1985

S

A

85 07 01 023

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	<i>Not in file</i>
By	
Distribution/	
Availability Codes	
A1	



2

Micro-PROUST

W. Lewis Johnson and Elliot Soloway

YALEU/CSD/RR #402

June 1985

DTIC
ELECTE
AUG 5 1985
S A D

This document has been approved
for publication and sale, its
distribution is unlimited.

Micro-PROUST

W. Lewis Johnson and Elliot Soloway

YALEU/CSD/RR #402

June 1985

The research described in this paper was co-sponsored by the Personnel and Training Research Groups, Psychological Sciences Division, Office of Naval Research and the Army Research Institute for the Behavioral and Social Sciences, under Contract No. N00014-82-K-0714, Contract Authority Identification Number 154-492. Approved for public release; distribution unlimited. Reproduction in whole or part is permitted for any purpose of the United States Government.

Micro-PROUST

Designed by:
W. Lewis Johnson
Elliot Soloway
Department of Computer Science
Yale University
P.O. Box 2158
New Haven, Connecticut 06520

Programmed by:
Bret Wallach
Advanced Processing, San Diego, Calif.

The research described in this paper was co-sponsored by the Personnel and Training Research Groups, Psychological Sciences Division, Office of Naval Research and the Army Research Institute for the Behavioral and Social Sciences, under Contract No. N00014-82-K-0714, Contract Authority Identification Number 154-492. Approved for public release; distribution unlimited. Reproduction in whole or part is permitted for any purpose of the United States Government.

The authors gratefully acknowledge the support of Courseware, Inc., of San Diego, California, for providing the funds to develop Micro-PROUST. We would particularly like to thank Dr. Gregg Kearlsey of Courseware for his unflagging effort and support in seeing this project through to fruition.

424157 525

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER #402	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Micro-PROUST		5. TYPE OF REPORT & PERIOD COVERED Research Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) W. Lewis Johnson and Elliot Soloway		8. CONTRACT OR GRANT NUMBER(s) N00014-82-K-0714
9. PERFORMING ORGANIZATION NAME AND ADDRESS Yale University - Dept. of Computer Science 10 Hillhouse Avenue New Haven, CT 06520		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE June, 1985
		13. NUMBER OF PAGES 136
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research Information Systems Program Arlington, VA 22217		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Distribution of this report is unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) intelligent tutoring sytems; student modelling; automatic program debugging.		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) PROUST is a system that can identify, for a class of moderately complex introductory looping assignments, the non-syntactic bugs in novices' programs. PROUST is a 15,000 LISP program and runs on a VAX. Micro-PROUST is a program meant to capture the essence of PROUST. Micro-PROUST is a 1500 line LISP program and runs on an IBM PC (with 512K). In this document we present the inner workings of Micro-PROUST. Our intent is to enable those who so are inclined to see at a nuts and bolts level how a system like PROUST actually works.		

-- OFFICIAL DISTIRUBTION LIST --

Army		Private Sector	
Technical Director U S Army Research Institute for the Behavioral and Social Sciences 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. Michael Genesereth Department of Computer Science Stanford University Stanford, California 94305	1 copy
Mr. James Baker Army Research Institute 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. Dedre Gentner Bolt Beranek & Newman 10 Moulton Street Cambridge, Massachusetts 02138	1 copy
Dr. Beatrice J. Farr U S Army Research Institute 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. Robert Glaser Learning Research & Development Center University of Pittsburgh 3939 O'Hara Street Pittsburgh, Pennsylvania 15260	1 copy
Dr. Milton S. Katz Williams Technical Area U S Army Research Institute 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. Joseph Goguen SRI International 333 Ravenswood Avenue Menlo Park, California 94025	1 copy
Dr. Marshall Narva U S Army Research Institute for the Behavioral & Social Sciences 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. Bert Green Johns Hopkins University Department of Psychology Charles & 34th Street Baltimore, Maryland 21218	1 copy
Dr. Harold F. O'Neill, Jr. Director, Training Research Lab Army Research Institute 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. James G. Greeno LRDC University of Pittsburgh 3939 O'Hara Street Pittsburgh, Pennsylvania 15213	1 copy
Commander, US Army Research Institute for the Behavioral & Social Sciences Attn PERI-BR (Dr. Judith Orasanu) 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. Barbara Hayes-Roth Department of Computer Science Stanford University Stanford, California 95305	1 copy
Joseph Psotka, Ph.D. Attn PERI-IC Army Research Institute 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. Frederick Hayes-Roth Teknowledge 525 University Avenue Palo Alto, California 94301	1 copy
Dr. Robert Sasmor U S Army Research Institute for the Behavioral and Social Sciences 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Glena Greenwald, Ed Human Intelligence Newsletter P O Box 1163 Birmingham, Michigan 48012	1 copy
Dr. Robert Wisher Army Research Institute 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. Earl Hunt Department of Psychology University of Washington Seattle, Washington 98105	1 copy
		Dr. Marcel Just Department of Psychology Carnegie-Mellon University Pittsburgh, Pennsylvania 15213	1 copy

Air Force

U S Air Force Office of Scientific Research Life Sciences Directorate, NL Bolling Air Force Base Washington, DC 20332	1 copy	Dr David Kieras Department of Psychology University of Arizona Tucson, Arizona 85721	1 copy
Dr Earl A Alluisi HQ AFHRL (AFSC) Brooks AFB, Texas 78235	1 copy	Dr Walter Kintsch Department of Psychology University of Colorado Boulder, Colorado 80302	1 copy
Bryan Dallman AFHRL/LRT Lowry AFB, Colorado 80230	1 copy	Dr Stephen Kosslyn Department of Psychology The John Hopkins University Baltimore, Maryland 21218	1 copy
Dr Genevieve Haddad Program Manager Life Sciences Directorate AFOSR Bolling AFB, DC 20332	1 copy	Dr Pat Langley The Robotics Institute Carnegie-Mellon University Pittsburgh, Pennsylvania 15213	1 copy
Dr John Tangney AFOSR/NL Bolling AFB, DC 20332	1 copy	Dr Jill Larkin Department of Psychology Carnegie-Mellon University Pittsburgh, Pennsylvania 15213	1 copy
Dr Joseph Yasutake AFHRL/LRT Lowry AFB, Colorado 80230	1 copy		
Marine Corps		Dr Alan Lesgold Learning R&D Center University of Pittsburgh 3939 O'Hara Street Pittsburgh, Pennsylvania 15213	1 copy
H William Greenup Education Advisor (E031) Education Center, MCDEC Quantico, Virginia 22134	1 copy	Dr Jim Levin University of California at San Diego Laboratory for Comparative Human Cognition - D003A La Jolla, California 92093	1 copy
Special Assistant for Marine Corps Matters Code 100M Office of Naval Research 800 N Quincy Street Arlington, Virginia 22217	1 copy	Dr Michael Levine Department of Educational Psychology 210 Education Bldg University of Illinois Champaign, Illinois 61801	1 copy
Dr A L Slafkosky Scientific Advisor (Code RD-1) HQ, U S Marine Corps Washington, DC 20380	1 copy	Dr Marcia Linn University of California Director, Adolescent Reasoning Project Berkeley, California 94720	1 copy
Department of Defense		Dr Jay McClelland Department of Psychology MIT Cambridge, Massachusetts 02139	1 copy
Defense Technical Information Center Cameron Station, Bldg 5 Alexandria, Virginia 22314 Attn TC	12 copies	Dr James R Miller Computer Thought Corporation 1721 West Plano Highway Plano, Texas 75075	1 copy
Military Assistant for Training and Personnel Technology Office of the Under Secretary of Defense for Research & Engineering Room, 3D129, The Pentagon Washington, DC 20301	1 copy	Dr Mark Miller Computer Thought Corporation 1721 West Plano Highway Plano, Texas 75075	1 copy
Major Jack Thorpe DARPA 1400 Wilson Blvd Arlington, Virginia 22209	1 copy		

Navy		Dr. Tom Moran	1 copy
Robert Ahlers	1 copy	Xerox PARC	
Code N711		3333 Coyote Hill Road	
Human Factors Laboratory		Palo Alto, California 94304	
NAVTRAEQUIPCEN		Dr. Allen Munro	1 copy
Orlando, Florida 32813		Behavioral Technology Laboratories	
Code N711	1 copy	1845 Elena Avenue, Fourth Floor	
Attn: Arthur S. Blaives		Redondo Beach, California 90277	
Naval Training Equipment Center		Dr. Donald Norman	1 copy
Orlando, Florida 32813		Cognitive Science, C-015	
Liaison Scientist	1 copy	Univ. of California, San Diego	
Office of Naval Research		La Jolla, California 92093	
Branch Office, London		Dr. Jesse Orlansky	1 copy
Box 39		Institute for Defense Analyses	
FPO New York, New York 09510		1801 N. Beauregard Street	
Dr. Richard Cantone	1 copy	Alexandria, Virginia 22311	
Navy Research Laboratory		Professor Seymour Papert	1 copy
Code 7510		20C-109	
Washington, DC 20375		MIT	
Chief of Naval Education and Training	1 copy	Cambridge, Massachusetts 02139	
Liason Office		Dr. Nancy Pennington	1 copy
Air Force Human Resource Laboratory		University of Chicago	
Operations Training Division		Graduate School of Business	
WILLIAMS AFB, Arizona 85224		1101 E. 58th Street	
Dr. Stanley Collier	1 copy	Chicago, Illinois 60637	
Office of Naval Technology		Dr. Richard A. Pofist	1 copy
800 N. Quincy Street		Director, Special Projects	
Arlington, Virginia 22217		MECC	
CDR Mike Curran	1 copy	2354 Hidden Valley Lane	
Office of Naval Research		Stillwater, Minnesota 55082	
800 N. Quincy Street		Dr. Peter Polson	1 copy
Code 270		Department of Psychology	
Arlington, Virginia 22217		University of Colorado	
Dr. John Ford	1 copy	Boulder, Colorado 80309	
Navy Personnel R&D Center		Dr. Fred Reif	1 copy
San Diego, California 92152		Physics Department	
Dr. Jude Franklin	1 copy	University of California	
Code 7510		Berkeley, California 94720	
Navy Research Laboratory		Dr. Lauren Resnick	1 copy
Washington, DC 20375		LRDC	
Dr. Mike Gaynor	1 copy	University of Pittsburgh	
Navy Research Laboratory		3939 O'Hara Street	
Code 7510		Pittsburgh, Pennsylvania 15213	
Washington, DC 20375		Mary S. Riley	1 copy
Dr. Jim Hollan	1 copy	Program in Cognitive Science	
Code 14		Center for Human Information Processing	
Navy Personnel R&D Center		University of California, San Diego	
San Diego, California 92152		La Jolla, California 92093	
Dr. Ed Hutchins	1 copy	Dr. Andrew Rose	1 copy
Navy Personnel R&D Center		American Institutes for Research	
San Diego, California 92152		1055 Thomas Jefferson Street, NW	
		Washington, DC 20007	

Dr. Norman J. Kerr Chief of Naval Technical Training Naval Air Station Memphis (75) Millington, Tennessee 38054	1 copy	Dr. Ernst Z. Rothkopf Bell Laboratories Murray Hill, New Jersey 07974	1 copy
Dr. James Lester ONR Detachment 495 Summer Street Boston, Massachusetts 02210	1 copy	Dr. William B. Rouse Georgia Institute of Technology School of Industrial & Systems Engineering Atlanta, Georgia 30332	1 copy
Dr. William L. Maloy (02) Chief of Naval Education and Training Naval Air Station Pensacola, Florida 32508	1 copy	Dr. David Rumelhart Center for Human Information Processing University of California, San Diego La Jolla, California 92093	1 copy
Dr. Joe McLachlan Navy Personnel R&D Center San Diego, California 92152	1 copy	Dr. Michael J. Samet Perceptronics, Inc 6271 Varrel Avenue Woodland Hills, California 91364	1 copy
Dr. William Montague NPRDC Code 13 San Diego, California 92152	1 copy	Dr. Roger Schank Yale University Department of Computer Science P O Box 2158 New Haven, Connecticut 06520	1 copy
Library, Code P201L Navy Personnel R&D Center San Diego, California 92152	1 copy	Dr. Walter Schneider Psychology Department 603 E. Daniel Champaign, Illinois 61820	1 copy
Technical Director Navy Personnel R&D Center San Diego, California 92152	1 copy	Dr. Alan Schoenfeld Mathematics and Education The University of Rochester Rochester, New York 14627	1 copy
Commanding Officer Naval Research Laboratory Code 2627 Washington, DC 20390	6 copies	Mr. Colin Sheppard Applied Psychology Unit Admiralty Marine Technology Est Teddington, Middlesex United Kingdom	1 copy
Office of Naval Research Code 433 800 N. Quincy Street Arlington, Virginia 22217	1 copy	Dr. H. Wallace Sinaiko Program Director Manpower Research and Advisory Service Smithsonian Institution 801 North Pitt Street Alexandria, Virginia 22314	1 copy
Personnel & Training Research Group Code 442PT Office of Naval Research Arlington, Virginia 22217	6 copies	Dr. Edward E. Smith Bolt Beranek & Newman 50 Moulton Street Cambridge, Massachusetts 02138	1 copy
Office of the Chief of Naval Operations Research Development & Studies Branch OP 115 Washington, DC 20350	1 copy	Dr. Richard Snow School of Education Stanford University Stanford, California 94305	1 copy
LT Frank C. Petho, MSC, USN (Ph D) CNET (N-432) NAS Pensacola, Florida 32508	1 copy	Dr. Kathryn T. Spoehr Psychology Department Brown University Providence, Rhode Island 02912	1 copy
Dr. Gary Poock Operations Research Development Code 55PK Naval Postgraduate School Monterey, California 93940	1 copy		

Dr. Gil Ricard Code N711 NTEC Orlando, Florida 32813	1 copy	Dr. Robert Sternberg Department of Psychology Yale University Box 11A, Yale Station New Haven, Connecticut 06520	1 copy
Dr. Worth Scanland CNET (N-5) NAS, Pensacola, Florida 32508	1 copy	Dr. Albert Stevens Bolt Beranek & Newman 10 Moulton Street Cambridge, Massachusetts 02238	1 copy
Dr. Robert G. Smith Office of Chief of Naval Operations OP-987H Washington, DC 20350	1 copy	David E. Stone, Ph.D. Hazeltine Corporation 7680 Old Springhouse Road McLean, Virginia 22102	1 copy
Dr. Alfred F. Smode, Director Training Analysis & Evaluation Group Department of the Navy Orlando, Florida 32813	1 copy	Dr. Patrick Suppes Institute for Mathematical Studies in the Social Sciences Stanford University Stanford, California 94305	1 copy
Dr. Richard Sorensen Navy Personnel R&D Center San Diego, California 92152	1 copy	Dr. Kikumi Tatsuoka Computer Based Education Research Lab 252 Engineering Research Laboratory Urbana, Illinois 61801	1 copy
Dr. Frederick Steinheiser CNO - DP115 Navy Annex Arlington, Virginia 20370	1 copy	Dr. Maurice Tatsuoka 220 Education Bldg 1310 S. Sixth Street Champaign, Illinois 61820	1 copy
Roger Weissinger-Baylon Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	1 copy	Dr. Perry W. Thorndyke Perceptronics, Inc. 545 Middlefield Road, Suite 140 Menlo Park, California 94025	1 copy
Mr. John H. Wolfe Navy Personnel R&D Center San Diego, California 92152	1 copy	Dr. Douglas Towne University of So. California Behavioral Technology Labs 1845 S. Elena Avenue Redondo Beach, California 90277	1 copy
Dr. Wallace Wulfeck, III Navy Personnel R&D Center San Diego, California 92152	1 copy	Dr. Kurt Van Lehn Xerox PARC 3333 Coyote Hill Road Palo Alto, California 94304	1 copy
Private Sector		Dr. Keith T. Wescourt Perceptronics, Inc. 545 Middlefield Road, Suite 140 Menlo Park, California 94025	1 copy
Dr. John R. Anderson Department of Psychology Carnegie-Mellon University Pittsburgh, Pennsylvania 15213	1 copy	William B. Whitten Bell Laboratories 2D-610 Holmdel, New Jersey 07733	1 copy
Dr. John Annett Department of Psychology University of Warwick Coventry CV4 7AJ ENGLAND	1 copy	Dr. Mike Williams Xerox PARC 3333 Coyote Hill Road Palo Alto, California 94304	1 copy
Dr. Michael Atwood ITT - Programming 1000 Oronoque Lane Stratford, Connecticut 06497	1 copy		

Civilian Agencies

Dr. Patricia Baggett Department of Psychology University of Colorado Boulder, Colorado 80309	1 copy	Dr. Patricia A. Butler NIE-BRM Bldg. Stop #7 1200 19th Street NW Washington, DC 20208	1 copy
Ms. Carole A. Bagley Minnesota Educational Computing Consortium 2354 Hidden Valley Lane Stillwater, Minnesota 55082	1 copy	Dr. Susan Chipman Learning and Development National Institute of Education 1200 19th Street NW Washington, DC 20208	1 copy
Dr. Jonathan Baaron 80 Glenn Avenue Berwyn, Pennsylvania 19312	1 copy	Edward Esty Department of Education, OERI MS 40 1200 19th Street, NW Washington, DC 20208	1 copy
Mr. Avron Barr Department of Computer Science Stanford University Stanford, California 94305	1 copy	Edward J. Fuentes Department of Education 1200 19th Street, NW Washington, DC 20208	1 copy
Dr. John Black Yale University Box 11A, Yale Station New Haven, Connecticut 06520	1 copy	TAE, TAK National Institute of Education 1200 19th Street, NW Washington, DC 20208	1 copy
Dr. John S. Brown XEROX Palo Alto Research Center 3333 Coyote Road Palo Alto, California 94304	1 copy	Dr. John Mays National Institute of Education 1200 19th Street, NW Washington, DC 20208	1 copy
Dr. Bruce Buchanan Department of Computer Science Stanford University Stanford, California 94305	1 copy	Dr. Arthur Meled 724 Brown U. S. Dept. of Education Washington, DC 20208	1 copy
Dr. Jaime Carbonell Department of Psychology Carnegie-Mellon University Pittsburgh, Pennsylvania 15213	1 copy	Dr. Andrew R. Molnar Office of Scientific and Engineering Personnel and Education National Science Foundation Washington, DC 20550	1 copy
Dr. Pat Carpenter Department of Psychology Carnegie-Mellon University Pittsburgh, Pennsylvania 15213	1 copy	Everett Palmer Research Scientist Mail Stop 239-3 NASA Ames Research Center Moffett Field, California 94035	1 copy
Dr. William Chase Department of Psychology Carnegie-Mellon University Pittsburgh, Pennsylvania 15213	1 copy	Dr. Mary Stoddard C 10, Mail Stop B296 Los Alamos National Laboratories Los Alamos, New Mexico 87545	1 copy
Dr. Micheline Chi Learning R & D Center University of Pittsburgh 3939 O'Hara Street Pittsburgh, Pennsylvania 15213	1 copy	Chief, Psychological Research Branch U. S. Coast Guard (G-P-1/2/TP42) Washington, DC 20593	1 copy

Dr. William Clancey Department of Computer Science Stanford University Stanford, California 94306	1 copy	Dr. Frank Withrow U. S. Office of Education 400 Maryland Avenue SW Washington, DC 20202	1 copy
Dr. Allan M. Collins Bolt Beranek & Newman, Inc 50 Moulton Street Cambridge, Massachusetts 02138	1 copy	Dr. Joseph L. Young, Director Memory & Cognitive Processes National Science Foundation Washington, DC 20550	1 copy
ERIC Facility-Acquisitions 4833 Rugby Avenue Bethesda, Maryland 20014	1 copy		
Mr. Wallace Feurzeig Department of Educational Technology Bolt Beranek and Newman 10 Moulton Street Cambridge, Massachusetts 02238	1 copy		
Dr. Dexter Fletcher WICAT Research Institute 1875 S. State Street Orem, Utah 22333	1 copy		
Dr. John R. Frederiksen Bolt Beranek & Newman 50 Moulton Street Cambridge, Massachusetts 02138	1 copy		

Mac Inker

Re-ink any fabric ribbon **AUTOMATICALLY** for less than 5¢. Extremely simple operation with built-in electric motor. We have a **MAC-INKER** for any printer: cartridge/spool/harmonica/zip pack. Lubricant ink safe for dot matrix printheads. Multicolored inks, uninked cartridges available. Ask for brochure. Thousands of satisfied customers.

\$54.00 +

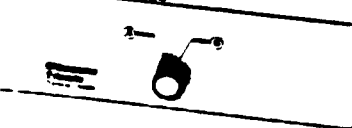


Mac Switch

Mac Switch lets you share your computer with any two peripherals (serial or parallel). Ideal for word processors—never type an address twice. Ask us for brochure with tips on how to share two peripherals (or two computers) with **MAC SWITCH**. Total satisfaction or full refund.

\$99.00

Please Peripherals



Order toll free 1-800-547-3303

Computer Friends

6415 SW Canyon Court
Suite #10
Portland, Oregon 97221
(503) 297-2321

PROUST

Ultimately, PROUST will be incorporated into a programming curriculum for students.

nosed or missed entirely. Bugs are counted as false alarms if they are either not present in the program or if they are present but misdiagnosed. Consequently, misdiagnosed bugs are counted both as false alarms and as not recognized, which inflates the total number of diagnosis errors.

When PROUST fails to understand a program completely, its ability to recognize bugs deteriorates: 17 percent of the programs were analyzed partially. In such cases PROUST deleted from its bug descriptions those bug analyses that were questionable, given that the program was only partially understood. The bug descriptions that remained were frequently wrong, but at least PROUST was able to warn the student to take the analysis with a grain of salt. The remaining 4 percent of the programs deviated from PROUST's expectations so drastically it could not analyze them at all. In these cases no bug report was generated.

We are not yet sufficiently satisfied with PROUST's accuracy to make it generally available to students. The false-alarm rate should be lower, and the fraction of programs that PROUST analyzes completely should be higher. When part of a program cannot be analyzed, PROUST should try to determine why that part of the program cannot be analyzed and try to account for the unanalyzed code. Once this is done we expect PROUST to succeed on 80 to 85 percent of the programs it analyzes. At that stage we will make it available to students on line.

CONCLUSION

PROUST is capable of high-quality analysis of bugs in novice programs.

It is almost at the level where it could be incorporated into a programming curriculum and provide significant benefits to students. Here we have given a simplified view of how PROUST finds bugs. The next step is to build an automated programming course around PROUST. Such a system would not only correct students' mistakes but would also suggest additional problems for the students to solve to give them practice where they need it. ■

AUTHORS' NOTE

This work was cosponsored by the Personnel and Training Research Groups, Psychological Sciences Division, Office of Naval Research, and the Army Research Institute for the Behavioral and Social Sciences, under Contract Number N00014-82-K-0714, Contract Authority Identification Number Nr 154-492.

Additional papers dealing with bug classification, automatic debugging, and the cognitive underpinnings of programming can be obtained by writing to the following address: Cognition and Programming Project, Department of Computer Science, Yale University, POB 2158 Yale Station, New Haven, CT 06520.

Special thanks to Greg Kearsley and Leszek Izdebski of Courseware Inc. and Bret Wallach of Advanced Processing for their efforts in developing Micro-PROUST.

REFERENCES

1. Fossdick, L. D. and L. J. Osterweil. "Data Flow Analysis in Software Reliability." *Computing Surveys* 8, vol. 3, 1976, pages 305-330.
2. Harandi, M. T. "Knowledge-Based Program Debugging: A Heuristic Model." *Proceedings of the 1983 SOFTFAIR*.
3. Wertz, H. "Stereotyped Program Debugging: An Aid for Novice Programmers." *International Journal of Man-Machine Studies* 16, 1982, pages 379-392.
4. Shortliffe, E. H. *Computer-Based Medical Consultations: MYCIN*. New York: American Elsevier Publishing Co., 1976.
5. Minsky, M. "A Framework for Representing Knowledge." *The Psychology of Computer Vision*, P. Winston, ed. New York: McGraw-Hill, 1975.
6. Johnson, W. L. "Intention-Based Diagnosis of Programming Errors." *Yale University Department of Computer Science*, 1984.

pr

AT spe
QuadS
Quadr
in IBM
innova
the pr
compu
transp
and w
1-2-3,
faster
before
interfa
But be
withou
than S
price o
and tu
virtual

ponent that failed to match the program must be an IF statement. The Error-Pattern slot has the value (IF . WHILE); this indicates that a WHILE statement was found when an IF statement was expected. These test conditions are both met in the WHILE-for-IF example, so the action part of the rule is activated. The action part of this rule consists of a Bug slot: the filler of this slot is a description of the bug associated with the plan difference. The bug in this case is a WHILE-for-IF confusion. PROUST's bug analyses of student programs consist of bug descriptions such as this. When PROUST presents its findings to the student, it takes each bug description and generates an English-language translation for it and, if appropriate, generates data illustrating the presence of the bug.

TEST RESULTS

PROUST has been tested on large numbers of beginners' programs. We assigned a class of novice programmers the Rainfall Problem (an elaboration of the Averaging Problem), which is shown in figure 8a.

We modified the Pascal compiler our students were using so that it would save copies of every syntactically correct program that they compiled. This allowed us to examine not only the final solution the students handed in, but also every intermediate version of their program. Since the first versions are likely to be the buggiest, this let us test PROUST under the most difficult conditions possible.

Figure 8b shows the results of running PROUST on the Rainfall Problem. There are 206 different attempted solutions to the Rainfall Problem in the test set. Of these, PROUST was able to derive a complete understanding of 79 percent of the programs, identifying 94 percent of the bugs, a percentage far higher than people are able to achieve. The chart also indicates that 6 percent of the bugs were not recognized and 55 were false alarms. Bugs are counted as not recognized if they are either misdiag-

(continued)

Listing 2: A correct implementation of the Bad Input Test plan.

```
WHILE Val <= 0 DO
  BEGIN
    WriteLn( 'Invalid data, please reenter' );
    Read( Val );
  END;
IF Val < > 99999 THEN
  ...
```

```
(Define-Rule WHILE-for-IF
  Statement-Type IF
  Error-Pattern (IF . WHILE)
  Bug (WHILE-for-IF Confusion (FoundStrt "MRet")
    (HistStrt "HistoryNode")))
```

Figure 7: The WHILE-for-IF bug rule invoked by PROUST to explain the plan difference between the faulty part of the program of figure 1 and the correct implementation of this part in listing 1.

(a)

Write a Pascal program that will prompt the user to input numbers from the terminal; each input stands for the amount of rainfall in New Haven for a day. Note: Since rainfall cannot be negative, the program should reject negative input. Your program should compute the following statistics from this data:

1. the average rainfall per day
2. the number of rainy days
3. the number of valid inputs (excluding any invalid data that might have been read in)
4. the maximum amount of rain that fell on any one day

The program should read data until the user types 99999; this is a sentinel value signaling the end of input. Do not include the 99999 in the calculations. Assume that if the input value is nonnegative, and not equal to 99999, then it is valid input data.

(b)

Total number of programs:	206	
Number of programs with bugs:	183	(89 percent)
Number of programs receiving full analyses:	161	(79 percent)
Total number of bugs:	570	
Bugs recognized correctly:	533	(94 percent)
Bugs not recognized:	29	(6 percent)
False alarms:	55	
Number of programs receiving partial analyses:	35	(17 percent)
Total number of bugs:	191	
Bugs recognized correctly:	71	(37 percent)
Bugs deleted from analysis:	70	(37 percent)
Bugs not recognized:	50	(26 percent)
False alarms:	19	
Number of programs PROUST did not analyze:	9	(4 percent)

Figure 8: (a) The Rainfall Problem was assigned to a class of novice programmers to test the effectiveness of PROUST. (b) This shows the results of running PROUST on the Rainfall Problem.



The Silver Fox
 The Silver Fox is a powerful, portable, and powerful personal computer. It is the first of its kind to be designed and built from the ground up for the personal computer market. It is the first of its kind to be designed and built from the ground up for the personal computer market. It is the first of its kind to be designed and built from the ground up for the personal computer market.

MORE HARDWARE

The Silver Fox is a powerful, portable, and powerful personal computer. It is the first of its kind to be designed and built from the ground up for the personal computer market. It is the first of its kind to be designed and built from the ground up for the personal computer market. It is the first of its kind to be designed and built from the ground up for the personal computer market.

FREE SILVERWARE

The Silver Fox is a powerful, portable, and powerful personal computer. It is the first of its kind to be designed and built from the ground up for the personal computer market. It is the first of its kind to be designed and built from the ground up for the personal computer market. It is the first of its kind to be designed and built from the ground up for the personal computer market.

REPRODUCTION

The Silver Fox is a powerful, portable, and powerful personal computer. It is the first of its kind to be designed and built from the ground up for the personal computer market. It is the first of its kind to be designed and built from the ground up for the personal computer market. It is the first of its kind to be designed and built from the ground up for the personal computer market.

ColorFax \$1647

The Silver Fox is a powerful, portable, and powerful personal computer. It is the first of its kind to be designed and built from the ground up for the personal computer market. It is the first of its kind to be designed and built from the ground up for the personal computer market. It is the first of its kind to be designed and built from the ground up for the personal computer market.

PROUST

*Since PROUST first
 generates a possible
 implementation and
 then matches it
 against the program,
 it is performing
 analysis by synthesis.*

process that it went through in selecting the Sentinel-Process-Read-While plan. It first substitutes all pattern variables in the goal expression that have bindings. Since ?New has Val as a binding, the subgoal expression becomes (Input Val). PROUST then retrieves plans from the plan database that implements Input. One such plan is the READ PLAN, which employs a Pascal Read statement to input the value. This plan matches the Read statements in the program.

This example shows how PROUST analyzes programs by predicting the plans that might be used and then testing these predictions. By selecting from a range of different plans and subplans for each goal, PROUST is able to generate a variety of different ways of implementing each goal. Since PROUST first generates a possible implementation and then matches it against the program, it is performing analysis by synthesis. In general, generating plan hypotheses and matching them against programs is rather more complex than the scenario presented here; for more information, see reference 3.

IDENTIFYING BUGS

When the Sentinel-Process-Read-While plan was matched against the program in figure 1a, the plan matched exactly. Since there were no match errors, there must not have been any bugs in that particular plan. It is frequently the case, however, that none of the plans that PROUST

predicts matches the program. When this happens PROUST must look for bugs that account for the mismatches in one of the plans. In this section we will discuss one of these mismatches in connection with the WHILE-for-IF example in figure 2a and show how it leads to the discovery of a bug.

The bug in the WHILE-for-IF example is discovered in processing the Input-Validation goal. One of the plans that PROUST suggests for implementing this goal is the so-called Bad Input Loop Test plan. This plan consists of a WHILE statement that tests the input to see if it is out of range, an error message inside the WHILE loop, an Input subgoal that rereads the input if it is out of range, and a test to see if the exit condition for the main loop has been satisfied.

Listing 2 illustrates a correct implementation of this plan (solving the Averaging Problem).

The Bad Input Loop Test plan matches the WHILE-for-IF example of figure 2a in all but one respect: there is no test for the exit condition of the main loop, such as IF Val < > 99999 THEN Where an IF statement is expected, a WHILE statement appears instead. PROUST has thus encountered a *plan difference*, i.e., a difference between the expected plan and the code. When PROUST encounters plan differences it does not give up on the plan; instead, it tries to find a way of interpreting the plan differences as bugs.

In most cases plan differences are explained by means of *bug rules*. Each bug rule has a test part, which examines the plan differences to see whether the rule is applicable, and an action part, which explains the plan differences.

Figure 7 shows the bug rule that is invoked to explain the plan differences in the WHILE-for-IF example. The rule is written in slot-filler notation; one set of slots constitutes the test part of the rule, and another set constitutes the action part. In the WHILE-for-IF rule the test part consists of a Statement-Type slot and an Error-Pattern slot. The Statement-Type slot indicates that the plan com-

ponent
 gram m
 Error-P
 (IF . W
 WHILE
 IF state
 conditi
 WHILE
 part of
 tion par
 slot: the
 tion of
 plan d
 case is
 PROUST
 program
 such as
 its find
 each bu
 an Engl
 and, if a
 illustrat

TEST F
 PROUST
 number
 assigne
 mers the
 tion of t
 is show

We m
 our stu
 would s
 tically co
 piled. Th
 only the
 handed
 mediate
 Since th
 the bug
 under t
 possible

Figure
 ning PRO
 There a
 solution
 the test
 able to c
 ing of 7
 identify
 percenta
 able to
 dicates
 were nc
 false ala
 recogniz

PROUST substitutes any objects whose values are already known into the goal expression.

objects whose values are already known into the goal expression. At this point the only information available about ?New and ?Sentinel is what appears in the problem description. There the value of ?Sentinel is listed as 99999, but the value of ?New is not listed. Therefore, the value of ?Sentinel is substituted into the goal expression, but ?New is left unchanged. The resulting goal expression is (Sentinel-Controlled-Input ?New 99999).

PROUST must now retrieve from its programming knowledge base plans that could be used to implement the goal Sentinel-Controlled-Input. It retrieves the filler of the Instances slot of the definition of Sentinel-Controlled-Input shown in figure 4. This

filler is a list of five items: Sentinel-Process-Read-While, Sentinel-Read-Process-While, Sentinel-Read-Process-Repeat, Sentinel-Process-Read-Repeat, and Bogus-Counter-Controlled-Loop. Each of these is the name of a plan. PROUST selects the first plan from the list, Sentinel-Process-Read-While. This will be PROUST's initial hypothesis of how the program implements the goal Sentinel-Controlled-Input.

Just as known values of objects were substituted into the goal expression (Sentinel-Controlled-Input ?New ?Sentinel), these same substitutions must now be performed on the selected plan. To see what substitutions must be made, PROUST examines the Form slot of the definition of Sentinel-Process-Read-While (Sentinel-Controlled-Input ?Input ?Stop). The Form slot indicates which pattern-variable names are used in the plans that implement the goal. By comparing the Form slot to the goal being analyzed, PROUST determines that each occurrence of ?Input in the selected plan should be replaced by the value of ?New. Each occurrence of ?Stop should be replaced by the value of ?Sentinel or 99999. Because the value of ?New is not known,

PROUST simply replaces ?Input with the variable name ?New. PROUST assumes that the process of matching the plan against the program will determine what the value of ?New is.

Figure 6 shows how the Sentinel-Process-Read-While plan is matched against the program example in figure 1a. Matching starts with the WHILE loop. The pattern in the plan for the WHILE loop is (WHILE (<> ?New 99999) ...). There are two WHILE loops in this program: WHILE Val <> 99999 DO ... and WHILE Val <= 0 DO ... PROUST tries to match each pattern against each of these statements. (WHILE (<> ?New 99999) ...) matches WHILE Val <> 99999 DO ... provided that Val is substituted for ?New. (WHILE (<> ?New 99999) ...) does not match WHILE Val <= 0 DO ... because the statement has a <= test instead of a <> test, and because it tests against 0 instead of 99999. Therefore PROUST selects WHILE Val <> 99999 DO ... as the match for the plan pattern. Since Val must be substituted for ?New so that the pattern matches, Val is recorded as the binding for ?New. Afterward, any component of the plan that has ?New in it will have Val substituted for ?New.

The next plan component that PROUST matches against the program is (BEGIN ...). There are several different BEGIN statements in the program that could be matched against this pattern. However, in the plan template the (BEGIN ...) pattern appears inside of the WHILE Val <> 99999 DO ... statement. Therefore, there is only one BEGIN statement that has an appropriate match.

When PROUST tries to match the (SUBGOAL (Input ?New)) components, a different type of processing is required. These plan components are goals; to match them against the program, PROUST must go through the same plan-selection

(continued)

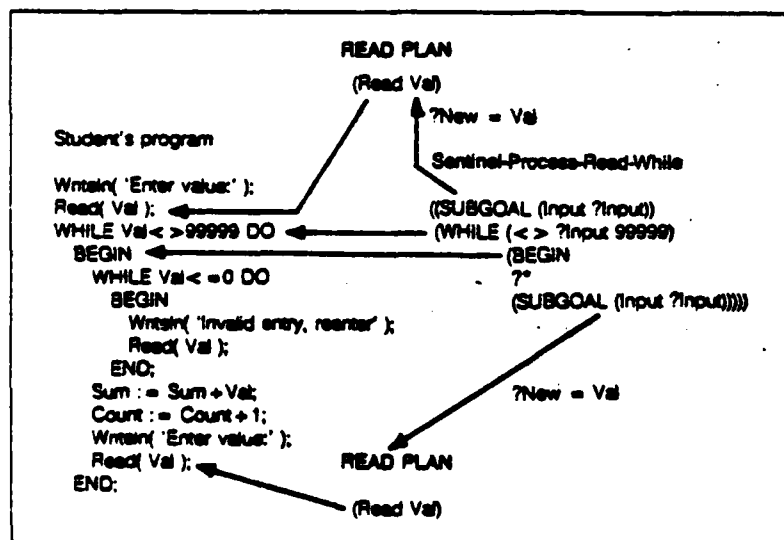


Figure 6: This shows how the Sentinel-Process-Read-While plan is matched against the program in figure 1.

COMPUTERS

IBM SYSTEM SPEC:
256K, 2 Drives, color
adapter & PGS M2
256K, 1 Drive & 10 MB
TWO USER SYSTEM
Diplex PC 86 & Term
COLUMBIA
MPC 4220 256K, 2 D,
PROFESSIONAL, 10M,
VP 2220, Portals 42,
CDROM/PRG 10, 10 L



CORONA PC-22 256K
Portals PC-22 256K
PLATIN MICRO 16
BROWNS DESIGNS
MOE MOY
NEC APC-8 PACKAGE
APC-8 486 Dk, 10MB
80486 with 350 37-
APC-8 486, 10MB,
and P370 Processor
APC-8 486 Dk, 10MB
80486 with 350 37-
SAPRO
NEC 386-2 T800
NEC 386-2 350-2 Plus
Mainframe, Secondary
NEC/DA Channel
SAP 80486 Computer
Co-Processor 486 Base, 40-
Pin LAPPRO 2, 4 & 16
TELEVIDEO
TELEPC-4 10MB, 256-
LEADING EDGE PC
ZENITH Z-151-S2 80486
ZV 151-S2 1 Dr 10 MB

FOR IBM PC/XT

ADVANCED DIGITAL
AST RESEARCH INC.
ADVANTAGE Multi
MEGA PLUS II 154K
30 Pins Plus 54K 50-
MONO GRAPH PLUS
GRABRAM
EXPANDED QUADRO
54K
QUAD 512+ (Serial P-
54K
QUADCOLOR I (Video)
HERCULES Graphics
Color Card (RGB, Corr
HYTEL 0067/0028) M
KEYTRONIC Denset
MA SYSTEMS PC Pe-
Parallel Port)
MINICOLOR Baby Blue
Serial Ports, Clock
ORION PC Turbo (80-
PARADIGM JA 501-2
PARADIGM SYSTEMS
Master Graphics Card
Module 4/8
PLANTRONICS Color-
STD SYSTEMS Green
Super R4 80486
TRACON TM 100-2 (CS-
TEAC PD-288 (CSDO)
TECHNAR Graphics Mk
The Captain (and Of
PC/Compat 10028K, 30
TREND LABS Ultra Pro

problem descriptions. Plans are stereotypic methods for implementing goals. A large part of writing programs consists of identifying goals that must be satisfied and selecting plans to implement these goals. Similarly, PROUST retrieves plans from its knowledge base for each goal referred to in the problem description. It compares these plans to the student's program to determine which fits the program best.

Figure 4 shows PROUST's definition for the Sentinel-Controlled-Input goal. The goal definition contains a series of slots: InstanceOf, Form, MainSegment, etc., together with fillers for each of these slots: Read&Process,

MainLoop, ?New, etc. These slots serve various functions, only some of which we will discuss here. The most important slots are the Instances and InstanceOf slots. The Instances slot lists the various plans in PROUST's knowledge base for implementing this goal. This slot's filler is a list of five items, each of which is the name of a plan. The InstanceOf slot indicates the class to which this goal belongs. The goal class in this case is Read&Process, which is the class of all goals that involve reading a sequence of values and processing them.

Figure 5 shows a plan, the Sentinel-Process-Read-While plan. This is one of the instances of the Sentinel-

Controlled-Input goal. This plan is a simplified version of the one PROUST actually uses. Plans are also defined in terms of slots and fillers. The most important slot is the Template slot, which describes the form the Pascal code implementing this plan should take. Plan templates consist of Pascal statements, subgoals, and labels. The Pascal statements are written in list notation rather than ordinary Pascal syntax; for example, the form (WHILE (<> ?Input ?Stop) ...) in Pascal syntax would appear as WHILE ?Input <> ?Stop DO ... Symbols that are preceded by question marks are pattern variables; these are substituted when the plan is used. ?New is substituted by a Pascal variable containing the input data, and ?Stop is substituted by a constant, the sentinel value. The "?" statement is a "wild card" pattern that can be substituted by an arbitrary sequence of Pascal statements; this is just a placeholder in the plan. Subgoals are indicated by (SUBGOAL ...) forms in the template; these are goals that must in turn be implemented using other plans.

```
((Define-Program Average)
 (Define-Object ?New)
 (Define-Object ?Sentinel Value 9999)
 (Define-Goal (Sentinel-Controlled-Input ?New ?Sentinel))
 (Define-Goal (Input-Validation ?New (<= ?New 0)))
 (Define-Goal (Output (Average ?New))))
```

Figure 3: The Averaging Problem translated into PROUST's problem-description language.

```
(Goal-Definition Sentinel-Controlled-Input
 InstanceOf      Read&Process
 Form            (Sentinel-Controlled-Input ?Input ?Stop)
 MainSegment     MainLoop
 MainVariable    ?New
 NamePhrase      "sentinel-controlled-loop"
 OuterControlPlan T
 Instances       (Sentinel-Process-Read-While
                  Sentinel-Read-Process-While
                  Sentinel-Read-Process-Repeat
                  Sentinel-Process-Read-Repeat
                  Bogue-Counter-Controlled-Loop))
```

Figure 4: The definition of the goal Sentinel-Controlled-Input in PROUST's problem-description language.

```
(Plan-Definition Sentinel-Process-Read-While
 Constants      (?Stop)
 Variables      (?Input)
 Template       ((SUBGOAL (Input ?Input))
                  (WHILE (<> ?Input ?Stop)
                     (BEGIN
                      ?*
                      (SUBGOAL (Input ?Input))))))
```

Figure 5: A plan for implementing the goal Sentinel-Controlled-Input.

MATCHING PLANS

Let's look at how plans and goals are used to understand a program. The plan in listing 1 has been implemented correctly. You will see how PROUST hypothesizes a plan that the program might use and then matches this plan against the program. In this case the match succeeds because the plan is implemented correctly. In the next section we will examine what happens when plans fail to match because the student's code has bugs.

The first step, before any analysis of goals and plans takes place, is to parse the student's Pascal program. This results in a parse tree. All subsequent analysis of the program is performed on the parse tree rather than on the original program text.

When PROUST analyzes a program, it selects goals from the problem description one at a time. Let's suppose that the goal that is selected first is (Sentinel-Controlled-Input ?New ?Sentinel). PROUST substitutes any

(continued)



mon bugs to see if it can explain the discrepancies.

PROUST'S 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; sets of data objects are the data that the program must manipulate.

The first step in translating an English-language problem statement into PROUST's problem-description language is to make the various goals that are mentioned in the problem statement explicit. Recall that the text of the Averaging Problem is the following:

Write a program that reads in a sequence of positive numbers, stopping when 99999 is read. Compute the average of these numbers. Do not include the 99999 in the average. Be sure to reject any input that is not positive.

Solutions to this problem operate on a sequence of input data; let us call this sequence *New*. The following goals can be extracted from the problem statement:

- Read successive values of *New*, stopping when a sentinel value, 99999, is read.
- Make sure that the condition $New \leq 0$ is never true.
- Compute the average of *New*.
- Output the average of *New*.

We must now take these goals and use them to generate a problem description for PROUST. Each data object that the goals refer to is named and declared. Each goal extracted from the problem statement is recorded in the problem description. The resulting problem description is shown in figure 3.

Like all the data structures that we discuss in this article, problem descriptions are in list notation and every statement and expression is enclosed in parentheses. The name of the program is indicated with a Define-Program statement. Objects

Listing 1: Yet another way to implement the input validation for the Averaging Problem.

```
Read( Val );
WHILE Val <= 0 DO
  BEGIN
    Writein( 'Invalid entry, reenter' );
    Read( Val );
  END;
WHILE Val < > 99999 DO
  BEGIN
    Sum := Sum + Val;
    Count := Count + 1;
    Writein( 'Enter value:' );
    Read( Val );
    WHILE Val <= 0 DO
      BEGIN
        Writein( 'Invalid entry, reenter' );
        Read( Val );
      END;
  END;
END;
```

are named using Define-Object statements. Goals are indicated using Define-Goal statements.

Object names are preceded by question marks. There are two objects defined in the Averaging Problem description, ?Sentinel and ?New. The question-mark notation is used frequently in artificial-intelligence (AI) programs; it indicates that the variable is not a literal value but is a parameter that must be substituted when the data structure is used. For example, the input-data object ?New will be substituted with the name of the Pascal variable that the student uses for storing the input data. The object ?Sentinel has the value 99999; wherever ?Sentinel appears in the problem description it can be substituted with 99999.

Objects can be either constant-valued or variable-valued. In this example, ?Sentinel is a constant, with the value 99999, and ?New is a variable. In PROUST's general problem-description language objects can have a variety of properties associated with them; however, we will not use any such properties in this simple example.

Goal statements consist of a name of a type of goal, followed by a list of arguments. In the form (Average ?New) for example, Average is a type of goal (to compute an average), and ?New is the argument of the goal. This form requires that the program compute the average of ?New.

Arguments to goal expressions can take a variety of forms. They can be objects, predicates or even other goal expressions. In the expression (Input-Validation ?New (<= ?New 0)), one argument is an object (?New), and the other is a predicate ?New <= 0. In LISP, function names and operators precede their arguments, which is why the <= precedes the ?New and 0 in the expression (<= 0). If goals are nested, as in (Output (Average ?New)), the outer goal refers to the value computed by the inner goal. Thus this goal requires that the program output the average of ?New.

In this example PROUST's problem descriptions are a reasonable approximation of the original English-language problem statements. These problem descriptions describe what the programs must do but not how they are supposed to do it. PROUST must analyze each individual program and determine how it is intended to satisfy the problem requirements.

PROGRAMMING KNOWLEDGE

Programming knowledge in PROUST is frame-based (see reference 5). In frame-based systems knowledge is organized into frames, each of which corresponds to a particular concept that the system "knows" about. Frames are similar to records in relational databases, although the operations that can be performed on frames are somewhat different. Knowledge in frames is organized into slots, which function as record field names, and fillers, which are the values assigned to each slot.

The two kinds of programming knowledge that we will consider here are goals and plans (other types of programming knowledge are discussed in reference 6). Goals are problem requirements that appear in

(continued)

Data-flow analysis checks for clear anomalies in the pattern of data definition and for use of data in a program. It can determine when a variable is defined and never used or when a variable is never defined. However, if there are no anomalies in data flow, data-flow analysis will not detect any bugs. Neither example in the preceding section has data-flow anomalies, so this method would not detect the bugs.

You might also try analyzing the structure of the program itself to see whether it suggests the presence of bugs. You could build a library of templates for common bugs, such as missing sentinel tests or WHILE statements in place of IF statements, and then match these templates against the program to identify the bugs. The problem with this approach is that you have no way of knowing where to match the bug templates in the program. For example, the WHILE-for-IF example has three different WHILE loops. How could you tell which WHILE loop really should be an IF statement? You could try to make the bug template more specific by making it apply only when there are two loops with the same exit test, one inside the other. But that would make the template too specific; it would not apply to other cases where WHILE statements appear instead of IF statements.

All of these approaches to debugging attempt to identify bugs without any understanding of what the program is supposed to do, and any such approach does little more than make guesses as to what bug is involved. In order to do better, a debugging system has to be able to infer the programmer's intentions and relate them to the code.

PROUST'S APPROACH

PROUST is written in T, a dialect of LISP. The full system contains roughly 15,000 lines of LISP code and runs on a VAX-11/750. A stripped-down version called Micro-PROUST has been developed in conjunction with Courseware Inc., of San Diego, Califor-

MICRO-PROUST FOR THE IBM PC

Micro-PROUST is a subset of the larger implementation of PROUST. It is capable of dealing with a limited range of novice programs and is currently set up to handle only those example solutions to the Averaging and Rainfall Problems provided with it. Micro-PROUST runs in Gold Hill Computers Inc. Golden Common LISP on an IBM Personal Computer with 512K bytes of memory. The source code and example programs are available for downloading from BYTEnet Listings. The telephone number is (603) 924-9820. The file PRSTREAD.ME contains directions on how to run Micro-PROUST.

nia (see the text box "Micro-PROUST for the IBM PC" above for more information). Micro-PROUST is capable of recognizing the kinds of bugs that are described in this article; however, there are a variety of tricky bugs that PROUST can identify but Micro-PROUST cannot. (If you are interested in PROUST's full diagnostic capabilities, consult reference 3.)

PROUST's analysis of programs is based on knowledge of the programming problem. Students may solve the problem in a variety of ways and their programs may have a variety of bugs, but they are all trying to solve the same problem. Knowledge of the problem makes the variability of novice solutions more manageable. It also provides important information about the programmer's intentions.

To provide PROUST with descriptions of the programming problems, we devised a problem-description language. We described each problem in this language and provided PROUST with a library of the descriptions. Each problem description in PROUST's problem-description language is a paraphrase of the English-language problem statement that we

hand out to students.

To understand the students' programs, PROUST also needs to know how to solve the problem. Solutions to a given programming problem may be implemented in a variety of different ways. Suppose that there was only one way to test input for validity in a Pascal program, namely, to insert a WHILE loop at the top of the main loop, such as in figures 1a and 2a. Once PROUST knew that a program must validate input, it would know to look for such a loop, as well as for the sentinel test that must follow. However, there are several ways of validating input. Listing 1 shows a loop that tests input in a different way. Instead of there being one input validation loop, there are two; one is at the bottom of the loop and the other precedes the loop. No additional sentinel test is required when this method is used, because, as soon as input is validated, control flows to the main exit test of the WHILE loop. Therefore, without knowing what method the programmer is using for validating input, PROUST cannot tell whether to look for a sentinel test within the body of the loop. In figure 1a it is an error not to have such a sentinel test, but in listing 1 it is not. PROUST needs knowledge about programming so that it can understand how each student designed and implemented his or her solution. Once it understands the programmer's intentions, it can then use knowledge about common bugs to identify them in the student's program.

PROUST analyzes programs by synthesis. When PROUST examines a program, it looks up the corresponding problem description in its library. It makes hypotheses about the methods programmers may use to satisfy each requirement in the problem description. Each hypothesis is a possible correct implementation of the corresponding requirement. If one of these hypotheses fits the student's code, then PROUST infers that the requirement is implemented correctly. If PROUST's hypotheses do not fit the student's program, then PROUST checks its database of com-

mon bug discrepancies.

PROUST'S DESCRIPTION

Problem description consists of problem data objects, the principles to be satisfied, the data manipulation.

The first English-language into PROUST language that are a statement of the A following:

Write a procedure to compute the average of a sequence of numbers, printing where the average is not included.

Solutions to a sequence of this sequence goals can be a problem statement.

- Read success
- 99999, is not
- Make sure
- <= 0 is not
- Compute
- Output the

We must use them to describe the object that is and declare from the problem recorded in the. The result is shown in figure 1.

Like all the discuss in descriptions every statement enclosed in the program. Define-Program

ments and does not understand how the control flow in a WHILE loop works. As long as the body of the loop is straight-line code, the student has no problem. However, if the body of the loop contains tests, the student thinks that the tests should be written as WHILE statements to ensure that they repeat when the body of the loop does. We will refer to this misconception henceforth as the WHILE-for-IF misconception. PROUST's output for this example, shown in figure 2b, takes the misconception into account and explains it to the student.

The bugs in figures 1a and 2a illustrate the following points. First, bugs frequently cannot be detected if you don't know what the program is supposed to do. Both of the programs shown run no matter what input is read; to determine that there is a bug, you must recognize that the programs output different results than they should. Bugs such as these are not unusual; the missing sentinel-test bug occurs in 18 percent of novice programmers' solutions to the Averaging Problem.

Second, novice programmers need help identifying such bugs. These bugs cause the programs to fail only after unusual inputs—ones that novice programmers are unlikely to test. In the case of the WHILE-for-IF misconception, even if the programmer tests the case in question, he or she will probably not understand why the program fails because he or she expects the WHILE statement to perform a different function than it actually does.

ALTERNATIVES TO INTENTION-BASED DEBUGGING

To support our claim that debugging requires knowledge of the programmer's intentions, we will examine the principal alternatives to intention-based debugging and show why they fall short. The methods we have considered are analysis of I/O (input/output) behavior, analysis of data flow, and recognition of patterns of buggy code.

Debugging by analyzing I/O behavior involves determining when the

```
(a)
1 PROGRAM Average( Input, Output );
2   VAR Sum, Count, Val, Avg: REAL;
3   BEGIN
4     Sum := 0;
5     Count := 0;
6     WriteLn( 'Enter value:' );
7     Read( Val );
8     WHILE Val < > 99999 DO
9       BEGIN
10        WHILE Val < = 0 DO
11          BEGIN
12            WriteLn( 'Invalid entry, reenter' );
13            Read( Val );
14          END;
15        WHILE Val < > 99999 DO
16          BEGIN
17            Sum := Sum + Val;
18            Count := Count + 1;
19            WriteLn( 'Enter value:' );
20            Read( Val );
21          END;
22        END;
23        IF Count = 0 then
24          WriteLn( 'No data entered' )
25        ELSE BEGIN
26          Avg := Sum/Count;
27          WriteLn( 'The average is', Avg );
28        END;
29      END;
30    END;
```

(b)

PROUST's output:

You are using a WHILE statement at line 15 where you should have used an IF statement. You probably want the code starting at line 15 to execute once each time through the loop; your code will make it execute many times.

The statement in question is:
WHILE Val < > 99999 DO ...

Figure 2: (a) Another novice programmer's attempt at implementing the Averaging Problem. (b) PROUST once again explains what the problem with the program is, what the programmer wanted to do, and what he actually did.

output of the program is incorrect and suggesting bugs that might have caused the faulty behavior (see reference 2). This approach treats debugging as similar to medical diagnosis (see reference 4). The faulty behavior can be thought of as the symptoms of the program, and the bugs can be thought of as the diseases. There are two problems with this approach: A program's symptoms cannot always be determined, and these symptoms cannot always be related to the bugs. The bugs in the programs in figures 1a and 2a affect the output of the program only occasionally; recognizing

when this happens requires knowledge about what the output should look like. Since the WHILE-for-IF example fails to test the input for validity after the first positive value is read, it appears that this program is missing an input-validation test. It is only after inspecting the code that it becomes clear that the bug is not in the input-validation test but in the sentinel test.

Another debugging approach you might try is data-flow analysis (see reference 1). This is the approach many error-checking compilers use.

(continued)

programmer's intentions assists debugging, we will present two examples of "buggy" programs and discuss why alternative approaches to automatic debugging fail to identify such bugs. Then we will describe how PROUST analyzes such programs. Finally, we will present some statistics showing PROUST's performance on large numbers of students' solutions to a typical assignment in an introductory programming class. This will help support our claim that PROUST's approach is adequate for the majority of novice programmers' programs.

EXAMPLES OF PROGRAM BUGS

Here is a simple programming problem called the Averaging Problem:

Write a program that reads in a sequence of positive numbers, stopping when 99999 is read. Compute the average of these numbers. Do not include the 99999 in the average. Be sure to reject any input that is not positive.

The student's program must compute the average of a series of positive numbers. It must ensure that the input to the program is in fact positive. The input terminates when a specific value—99999—is read. Values such as this, which signal the end of input, are called *sentinel values*.

Figure 1a shows a sample solution to the Averaging Problem. This program works except for the following

bug: if you type 99999 immediately after typing a nonpositive value, the program will continue to prompt for data after the 99999 is read. When the program finally does terminate, the average will be incorrect. For example, suppose that you input 5, -5, 99999. Instead of terminating when the 99999 is read, the program requests another input. If the user then entered another 99999, the program would not print the average as 5, but instead would print $(5+99999)/2$, or 50002.

The program interprets 99999 as data when the sequence 5, -5, 99999 is read because when the program reads the -5, it enters the input-validation loop, which starts with line 10, WHILE Val <= 0 DO. This loop is intended to iterate until a positive value is typed in; 99999 is positive, so when the 99999 is read, control leaves the input-validation loop. However, the program was written with the assumption that when the input-validation loop is exited, the current value of Val is valid input data. In this case, Val is not valid data; it is 99999, the sentinel value. The loop nevertheless processes 99999 as if it were data. To guard against this case, there should be a test for the sentinel after the input-validation loop.

Figure 1b is PROUST's output describing the missing sentinel-test bug. The error is described in two ways: First it is described in English; then PROUST generates an example of data that causes the program to fail.

Now look at the program in figure 2a. This is another solution to the Averaging Problem, and the bug in this program is also fairly obscure. If you type a positive value followed by a negative value, the negative value will be included in the average. Thus if you type -2, 2, 99999, the average will be 2, but if you type 2, -2, 99999, the average will be 0.

Unlike the example in listing 1a, the programmer has not left out the sentinel test but has written the test in the form of a WHILE statement instead of an IF statement. The student probably has a misconception about the distinction between the two state-

```
(a)
1 PROGRAM Average( Input, Output );
2   VAR Sum, Count, Val, Avg: REAL;
3   BEGIN
4     Sum := 0;
5     Count := 0;
6     Writeln( 'Enter Value: ' );
7     Read( Val );
8     WHILE Val <> 99999 DO
9       BEGIN
10        WHILE Val <= 0 DO
11          BEGIN
12            Writeln( 'Invalid entry, reenter' );
13            Read( Val );
14          END;
15          Sum := Sum + Val;
16          Count := Count + 1;
17          Writeln( 'Enter value: ' );
18          Read( Val );
19        END;
20      IF Count > 0 THEN
21        Writeln( 'No data entered' );
22      ELSE BEGIN
23        Avg := Sum/Count;
24        Writeln( 'The average is, ' Avg );
25      END;
26    END.
```

(b)

PROUST's output:

You're missing a sentinel test. If a sentinel value is input immediately following a nonpositive value, your program will treat it as valid data.

To see this, try the following data in your program:
5 -5 99999

Figure 1: (a) One novice programmer's attempt at implementing the Averaging Problem. (b) PROUST explains the bug lurking in the program in concise English sentences and even offers data illustrating the error.

PROUST

BY W. LEWIS JOHNSON AND ELLIOT SOLOWAY

An automatic debugger for Pascal programs

PROUST (Program Understander for Students) is a knowledge-based system that finds nonsyntactic bugs in Pascal programs written by novice programmers. When students compile a program successfully, PROUST is automatically invoked to analyze it. PROUST reports any bugs that are in the program to the student.

PROUST is not merely a tool that helps programmers find bugs, nor is it confined to a narrow class of bugs, such as uninitialized variables. It is designed to find every bug in most beginners' programs. PROUST is currently capable of correctly identifying all of the bugs in over 70 percent of the programs that students write when we assign them moderately complex programming problems. When PROUST finds a bug, it does not simply point to the lines of code that are wrong; instead, it determines how the bug can be corrected and suggests why the bug arose in the first place. Our aim is to build an instructional system around PROUST that assigns programming problems to students, reads their work, and gives them helpful suggestions.

In designing PROUST we found it necessary to deal directly with the

variability of bugs in beginners' programs. If a programming problem is assigned to a class of 200, the students will write 200 different programs (assuming that they do not cheat). There is variability both in their programs' designs and bugs. Some bugs, such as missing variable initializations, are accidental omissions that can be easily recognized and corrected. Other bugs result when the programmer fails to reason through the interactions between components. In isolation, each piece of the program may appear correct, but when combined, the program doesn't work. Still other bugs result from misconceptions about programming. The code may appear correct to the programmer, but it doesn't do what he or she expects, for reasons he or she does not understand. Bugs resulting from misconceptions are the most serious; students stand to benefit the most from having such problems pointed out to them.

If a debugging system is to cope with the various types of errors that programmers make, it must understand what the programmer is trying to do. Debugging systems usually don't concern themselves with what

the program is supposed to do, they only analyze what the program actually does (see references 1, 2, and 3). Figuring out how a program is supposed to work is not easy; to do it a debugger requires information about the programming problem and knowledge about how to write programs. Nevertheless, identifying the programmer's intentions is worth the effort, because this knowledge makes it possible to identify more bugs, as well as to understand their causes.

To show how knowledge of the pro-

(continued)

W. Lewis Johnson (POB 2158, Yale Station, New Haven, CT 06520) is a research associate at Yale. He has a B.A. from Princeton University and a Ph.D. from Yale University. His interests are artificial intelligence, software engineering, and computer-aided instruction. Dr. Johnson has been pursuing research in artificial intelligence at Yale since 1978.

Elliot Soloway (Department of Computer Science, Yale University, New Haven, CT 06520) is an assistant professor at Yale. He has a B.A. in philosophy and a Ph.D. in computer science from the University of Massachusetts at Amherst. Dr. Soloway heads a group at Yale that is exploring the cognitive underpinnings of programming.

2. PROUST/Micro-PROUST: From BYTE Magazine

1. Introduction

In this document we present the inner workings of Micro-PROUST. Our intent is to enable those who so are inclined to see at a nuts and bolts level how a system like PROUST actually works. We reprint here a paper on PROUST/Micro-PROUST that recently appeared in BYTE Magazine that should provide a good overview of the operation of the system. Next, we present the design document that was used to code to Micro-PROUST; the actual implementor of Micro-PROUST found this document thorough and comprehensive. The third document is a preliminary comparisons of the costs/benefits and techniques employed by PROUST and Micro-PROUST. The actual LISP code follows these papers.

Micro-PROUST was written for the IBM PC, in Golden LISP. This version of LISP is available from Golden Hill LISP, Inc., Cambridge, Mass. Micro-PROUST requires 512K to run. The source code for Micro-PROUST is available on diskette from us for the cost of the diskette and handling; those interested should write to the Cognition and Programming Project, Yale Computer Science Department. The source is also available on the BYTE network; please refer to the article on PROUST that appeared in BYTE for information on how to download the code; that article is reproduced elsewhere in this document.

We are currently porting Micro-PROUST to the VAX in order to convert it to Franz LISP. Thus, we plan to distribute the source to Micro-PROUST on magnetic tape; again, if you are interested in receiving Micro-PROUST on this medium write to the address given above.

As noted expressly in the code itself, we, nor the actual implementors make no promises for Micro-PROUST. While we are pleased with Micro-PROUST as a demonstration vehicle, we do not guarantee that it is bug free. We would, of course, be interested in receiving bug reports. Moreover, there is no additional documentation for the code that is not included here; that's all there is. Good luck!

ABSTRACT

PROUST is a system that can identify, for a class of moderately complex introductory looping assignments, the non-syntactic bugs in novices' programs. PROUST is a 15,000 LISP program and runs on a VAX. Micro-PROUST is a program meant to capture the essence of PROUST. Micro-PROUST is a 1500 line LISP program and runs on an IBM PC (with 512K). In this document we present the inner workings of Micro-PROUST. Our intent is to enable those who so are inclined to see at a nuts and bolts level how a system like PROUST actually works.

Table of Contents

1 Introduction	1
2 PROUST/Micro-PROUST: From BYTE Magazine	2
3 Design Document: Micro-PROUST	3
4 PROUST/Micro-PROUST: A Preliminary Comparison	4
5 Micro-PROUST Implementation: The LISP Code	5

3. Design Document: Micro-PROUST

Design for Micro-PROUST

W. Lewis Johnson

Elliot Soloway

Cognition and Programming Project

Department of Computer Science

Yale University

P.O. Box 2158 Yale Sta.

New Haven, Ct. 06520

(203) 436-0606

1. Introduction

This document describes the design of Micro-PROUST, a stripped-down version of PROUST [1, 2]. Micro-PROUST is a knowledge-based system which identifies a certain class of non-syntactic bugs in novice programs. It uses a single, uniform mechanism for analyzing buggy programs, unlike the full PROUST system, which employs various reasoning techniques in order to understand students' programs. The following are included in this description of Micro-PROUST:

- sample programming problems and solutions which Micro-PROUST should be able to handle;
- the design of Micro-PROUST; and
- a description of the knowledge bases required in order to make Micro-PROUST run on the examples given.

This design has been constructed with a Common Lisp environment in mind. Modifications may be required to get this system to run in some other environment.

1.1. Overview of the system

Micro-PROUST analyzes programs using an analysis-by-synthesis approach. The system takes as input a description of a programming problem, and a program. The problem description is a list of goals which the program must satisfy. Micro-PROUST uses the problem description to try to build an implementation model for the program. It does this by using a database of programming knowledge to suggest ways in which the goals in the problem description might be implemented. A separate knowledge base of bugs is used to identify bugs given the implementation model.

The following knowledge bases are required:

- a knowledge base of goals, indicating how these goals might be implemented using plans;
- a knowledge base of plans; and
- a rule base of rules for identifying bugs.

The system itself has the following main modules:

- a lexer and parser for Pascal;

- a goal selection and implementation mechanism;
- a plan matcher;
- a bug rule applier; and
- a bug description generator.

2. Test examples for Micro-PROUST

2.1. Problems for Micro-PROUST to analyze

Figures 2-1 and 2-2 show two different programming problems which Micro-PROUST should be able to handle. The first, the Average Problem, is extremely simple. The second, the Rainfall Problem, is an elaboration of the Average Problem.

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

Figure 2-1: The Average Problem

Noah needs to keep track of rainfall in the New Haven area in order to determine when to launch his ark. Write a program which he can use to do this. Your program should read the rainfall for each day, stopping when Noah types "99999", which is not a data value, but a sentinel indicating the end of input. If the user types in a negative value the program should reject it, since negative rainfall is not possible. Your program should print out the number of valid days typed in, the number of rainy days, the average rainfall per day over the period, and the maximum amount of rainfall that fell on any one day.

Figure 2-2: The Rainfall Problem

2.2. Programs for Micro-PROUST to analyze

2.2.1. Example 1

This example, shown in Figure 2-3, is shown in Figure 2-3. The output which Micro-PROUST should generate is shown in Figure 2-4.

2.2.2. Example 2

Here is another example averaging program.

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

Figure 2-3: Averaging program #1

1. It appears that you were trying to use line 12 to read the next input value. Incrementing NEW will not cause the next value to be read in. You need to use a READ statement here.
2. You need a test to check that at least one valid data point has been input before line 14 is executed. The average is not defined when there is no input.
3. You need a test to check that at least one valid data point has been input before line 15 is executed. The average is not defined when there is no input.

Figure 2-4: Output for averaging program #1

```
1  PROGRAM Average2( input, output );
2  VAR Sum, Count, New: INTEGER;
3      Avg: REAL;
4  BEGIN
5      Sum := 0;
6      Count := 0;
7      New := 0;
8      WHILE New<>99999 DO
9          BEGIN
10             Read( New );
11             Sum := Sum+New;
12             Count := Count+1;
13         END;
14         IF Count=0 THEN
15             Writeln( 'No data entered' )
16         ELSE
17             BEGIN
18                 Avg := Sum/Count;
19                 Writeln( 'The average is ', avg );
20             END
21         END;
```

Figure 2-5: Averaging program #2

1. You're missing a sentinel guard. When your program reads the sentinel, it processes it as if it were data.

Figure 2-6: Output for averaging problem #2

2.2.3. Example 3

Here is a solution to the Rainfall Problem.

```
PROGRAM NOAH (INPUT ,OUTPUT);
```

```
VAR
```

```
    RAINFALL, LARGEST, SUM, COUNT1,  
    COUNT2, AVERAGE, RAINYDAYS : REAL;
```

```
BEGIN
```

```
    (* INITIALIZE VARIABLES *)
```

```
    LARGEST :=0;
```

```
    SUM :=0;
```

```
    COUNT1 :=0;
```

```
    COUNT2 :=0;
```

```
    AVERAGE :=0;
```

```
    RAINYDAYS :=0;
```

```
    (* READ THE RAINFALL AND CHECK FOR ERROR VALUES *)
```

```
    WRITELN ('ENTER RAINFALL, WHEN YOU ARE FINISHED ENTER 99999');
```

```
    READLN;
```

```
    READ (RAINFALL);
```

```
    WHILE RAINFALL <> 99999 DO
```

```
        BEGIN
```

```
            WHILE RAINFALL <0 DO
```

```
                BEGIN
```

```
                    WRITELN (RAINFALL :8, 'IS NOT A POSSIBLE RAINFALL, TRY AGAIN.');
```

```
                    WRITELN ('ENTER RAINFALL');
```

```
                    READLN;
```

```
                READ (RAINFALL)
```

```
                END;
```

```
                IF RAINFALL > LARGEST
```

```
                THEN
```

```
                    LARGEST := RAINFALL;
```

```
                IF RAINFALL > 0
```

```
                THEN
```

```
                    COUNT2 := COUNT2 + 1;
```

```
                COUNT1 := COUNT1 + 1;
```

```
                SUM := SUM + RAINFALL;
```

```
                READLN;
```

```
                READ (RAINFALL)
```

```
            END;
```

```
    AVERAGE := SUM/COUNT1;
```

```
    WRITELN (COUNT1 :8, 'VALID RAINFALLS WERE ENETERED.');
```

```
    WRITELN ('THE AVERAGE RAINFALL WAS', AVERAGE :8, 'INCHES PER DAY.');
```

```
WRITELN ('THE HIGHEST RAINFALL WAS', LARGEST :0:2, 'INCHES');  
WRITELN ('THERE WERE', COUNT2 :0:2, 'RAINYDAYS IN THIS PERIOD.')
```

END.

Micro-PROUST's output should look something like this:

1. You're missing a sentinel guard. If a sentinel value is input immediately following a negative value, your program will process it as if it were data.
2. You need a test to check that at least one valid data point has been input before line 50 is executed. The average is not defined when there is no input.
3. You need a test to check that at least one valid data point has been input before line 46 is executed. The average is not defined when there is no input.
4. You need a test to check that at least one valid data point has been input before line 51 is executed. The maximum is not defined when there is no input.

2.2.4. Example 4

Here is another example Rainfall Program.

```

PROGRAM RAINFALL ( INPUT,OUTPUT );

VAR
  RAIN, DAYS, TOTALRAIN, RAINDAYS, HIGHRAIN, AVERAIN: REAL;

BEGIN
  DAYS := 0;
  TOTALRAIN := 0;
  RAINDAYS := 0;
  HIGHRAIN := 0;
  REPEAT
    WRITELN ('ENTER RAINFALL');
    READLN;
    READ (RAIN);

    WHILE RAIN <> 99999 DO
      BEGIN
        DAYS := DAYS + 1;
        TOTALRAIN := TOTALRAIN + RAIN;
        IF RAIN > 0 THEN
          RAINDAYS := RAINDAYS + 1;
        IF HIGHRAIN < RAIN THEN
          HIGHRAIN := RAIN.
        END;
      UNTIL RAIN = 99999;

      AVERAIN := TOTALRAIN / DAYS;

      WRITELN;
      WRITELN ( DAYS:0:0, 'VALID RAINFALLS WERE ENTERED');
      WRITELN;
      WRITELN ('THE AVERAGE RAINFALL WAS', AVERAIN:0:2, 'INCHES PER DAY');
      WRITELN;
      WRITELN ('THE HIGHEST RAINFALL WAS', HIGHRAIN:0:2, 'INCHES');
      WRITELN;
      WRITELN ('THERE WERE', RAINDAYS:0:0, 'IN THIS PERIOD');
    END.

```

The output from Micro-PROUST should look something like this:

1. You used a WHILE statement at line 19 where you should have used an IF.
2. You need a test to check that at least one valid data point has been input before line 35 is executed. The average is not defined when there is no input.

3. You need a test to check that at least one valid data point has been input before line 30 is executed. The average is not defined when there is no input.

4. You need a test to check that at least one valid data point has been input before line 37 is executed. The maximum is not defined when there is no input.

5. Your program does not perform an input validation.

2.2.5. Example 5

```

PROGRAM TEST1(INPUT, OUTPUT);
VAR
  SUM, N, MAX, AVE: REAL;
  COUNT, RAINY: INTEGER;
BEGIN
  SUM:=0;
  COUNT:=0;
  RAINY:=0;
  MAX:=0;
  WRITELN('ENTER RAINFALL');
  READLN;
  READ(N);
  WHILE N<>99999 DO
    BEGIN
      IF N<0 THEN
        WRITELN(N:0:2, ' IS NOT A POSSIBLE RAINFALL, TRY AGAIN');
      ELSE
        BEGIN
          COUNT:=COUNT+1;
          SUM:=SUM+1;
          IF N>0 THEN
            RAINY:=RAIN+1;
            IF N>MAX THEN
              N:=MAX;
            END;
          WRITELN('ENTER RAINFALL');
          READLN;
          READ(N)
        END;
      WRITELN;
    IF COUNT=0 THEN
      WRITELN(COUNT:0, ' VALID RAINFALLS WERE ENTERED. ');
    ELSE
      BEGIN
        AVE:=SUM/COUNT;
        WRITELN('THE AVERAGE RAINFALL WAS ', AVE:0:2, ' INCHES PER DAY. ');
        WRITELN('THE HIGHEST RAINFALL WAS ', MAX:0:2, ' INCHES. ');
        WRITELN('THERE WERE ', RAINY:0, ' RAINY DAYS IN THIS PERIOD. ')
      END
    END.
  END.

```

PROUST's output:

You are using a counter update instead of a running-total update at line 22. You must add the new-value variable to SUM, not add 1 to it.

The assignment at line 26 is backwards. This line will assign to N; you need to assign to MAX.

2.2.6. Example 6

```

PROGRAM TEST1(INPUT, OUTPUT);
VAR
    SUM,N,MAX,AVE:REAL;
    COUNT,RAINY:INTEGER;
BEGIN
    WRITELN('ENTER RAINFALL');
    READLN;
    READ(N);
    WHILE N<>99999 DO
        BEGIN
            IF N<0 THEN
                WRITELN(N:0:2,' IS NOT A POSSIBLE RAINFALL,TRY AGAIN')
            ELSE
                BEGIN
                    SUM:=SUM+N;
                    COUNT:=COUNT+1;
                    IF N>MAX THEN
                        MAX:=N;
                    IF N>0 THEN
                        RAINY:=RAINY+1;
                END;
                WRITELN('ENTER RAINFALL');
                READLN;
                READ(N)
            END;
            WRITELN;
            IF COUNT=0 THEN
                WRITELN(COUNT:0,' VALID RAINFALLS WERE ENTERED. ');
            ELSE
                BEGIN
                    AVE:=SUM/COUNT;
                    WRITELN('THE AVERAGE RAINFALL WAS ',AVE:0:2,' INCHES PER DAY. ');
                    WRITELN('THE HIGHEST RAINFALL WAS ',MAX:0:2,' INCHES. ');
                    WRITELN('THERE WERE ',RAINY:0,' RAINY DAYS IN THIS PERIOD. ')
                END
            END;
        END.

```

PROUST's output:

1. You did not initialize a sum.
2. You did not initialize a counter.
3. You did not initialize a maximum computation.
4. You did not initialize a counter.

3. Top-level organization of Micro-PROUST

The top-level processing is as follows.

1. Take as input a problem description, and a file name containing the student's program.
2. Parse the file, generating a parse tree.
3. Load the goal agenda with the problem description. (The goal agenda is described in Section 6.)
4. Retrieve plans for realizing each goal.
5. Match against the plans against the program.
6. If a plan matches, add it to the list of plans that have been matched so far.
7. Otherwise apply bug rules in order to explain the errors in matching the plans.
8. When the goal agenda is empty, report the bugs to the student.

The following is a list of the main routines in Micro-PROUST. More detailed descriptions of some of these modules will appear in subsequent sections.

- (MPROUST *filename problem*) -- this is the top-level routine of Micro-PROUST. *filename* is the name of the student's program, and *problem* is the problem description.
- (PARSE *filename*) -- parse *filename*, and return a pointer to the root of the parse tree.
- (INIT-AGENDA *problem*) -- initialize the goal agenda.
- (PROCESS-NEXT-GOAL) -- select a goal from the agenda, retrieve plans for it, match the plans against the program, and identify bugs.
- (REPORT-BUGS) -- describe the bugs which have been found to the student.

The calling hierarchy of these routines is as follows:

MPROUST

PARSE INIT-AGENDA PROCESS-NEXT-GOAL REPORT-BUGS

These following global variables are used in this process:

- *PARSE-TREE* -- the parse tree
- *AGENDA* -- the goal agenda
- *ACTIVE-GOAL* -- the current goal
- *ANALYZED-GOALS* -- the goals which have been analyzed so far
- *BUG-REPORT* -- the bugs which have been found so far.

4. The parser

You will have to build a lexer and parser for Pascal. You may borrow the one in PROUST, but it is written in T, not Common Lisp, so you would have to translate it. Assuming that you write your own, it should contain the following routines:

- (PARSE *filename*) -- parse the program contained in the file named "*filename*". Generate a syntax tree for the program, and return the parse tree node for the root of the program.
- (LEX-INITIALIZE *filename*) -- open the file and initialize the input stream to the beginning of the file. Initialize the lexer. Return T if successful.
- (LEX-GETTOK) -- get a token. Return the token type, and set *TOK-VAL* to the token value. Set *TOK-LINE* to the line number in the program which the token appears on. Token types and token values are discussed in Section 4.1.

The calling hierarchy is as follows:

PARSE

LEX-INITIALIZE LEX-GETTOK

I will not discuss here the method for constructing the Pascal parser; there exist various books on compiler construction which discuss how this can be done. The method that we used in Proust was to modify the Unix yacc program to make it generate Lisp code. Note that it is not necessary to construct a complete parser for Pascal; in particular, there is no need to make the parser handle procedures or complex datatypes. Therefore the parser should be fairly simple. Whatever method you use for implementing your parser, make sure that it generates parse trees such as those described in Section 4.2.

```

1 PROGRAM NOAA (INPUT ,OUTPUT);
2
3 VAR
4
5     RAINFALL, LARGEST, SUM, COUNT1,
6     COUNT2, AVERAGE, RAINYDAYS : REAL;
7
8
9 BEGIN
10     (* INITIALIZE VARIABLES *)
11     LARGEST :=0;
12     SUM :=0;
13     COUNT1 :=0;
14     COUNT2 :=0;
15
16     AVERAGE :=0;
17     RAINYDAYS :=0;
18     (* READ THE RAINFALL AND CHECK FOR ERROR VALUES *)
19     WRITELN ('ENTER RAINFALL, WHEN YOU ARE FINISHED ENTER 99999');
20     READLN;
21     READ (RAINFALL);
22     WHILE RAINFALL < 99999 DO
23         BEGIN
24             WHILE RAINFALL < 0 DO
25                 BEGIN
26                     WRITELN (RAINFALL :8, 'IS NOT A POSSIBLE RAINFALL, TRY AGAIN. ');
27                     WRITELN ('ENTER RAINFALL ');
28                     READLN;
29                     READ (RAINFALL)
30                 END;
31             IF RAINFALL > LARGEST
32             THEN
33                 LARGEST := RAINFALL;
34             IF RAINFALL > 0
35             THEN
36                 COUNT2 := COUNT2 + 1;
37                 COUNT1 := COUNT1 + 1;
38                 SUM := SUM + RAINFALL;
39                 READLN;
40                 READ (RAINFALL)
41             END;
42
43     AVERAGE := SUM/COUNT1;
44
45     WRITELN (COUNT1 :8, 'VALID RAINFALLS WERE ENTERED. ');
46     WRITELN ('THE AVERAGE RAINFALL WAS', AVERAGE :8, 'INCHES PER DAY. ');
47     WRITELN ('THE HIGHEST RAINFALL WAS', LARGEST :0:2, 'INCHES ');
48     WRITELN ('THERE WERE', COUNT2 :0:2, 'RAINYDAYS IN THIS PERIOD. ')
49 END.

```

Figure 2: An example solution of the Rainfall Problem

Noah needs to keep track of rainfall in the New Haven area in order to determine when to launch his ark. Write a Pascal program that will help him do this. The program should prompt the user to input numbers from the terminal; each input stands for the amount of rainfall in New Haven for a day. Note: since rainfall cannot be negative, the program should reject negative input. Your program should compute the following statistics from this data:

1. the average rainfall per day;
2. the number of rainy days.
3. the number of valid inputs (excluding any invalid data that might have been read in);
4. the maximum amount of rain that fell on any one day.

The program should read data until the user types 99999; this is a sentinel value signaling the end of input. Do not include the 99999 in the calculations. Assume that if the input value is non-negative, and not equal to 99999, then it is valid input data.

Figure 1: The Rainfall Problem

in terms of its intended function, e.g., outputting the maximum, or checking for invalid input. PROUST also generates an example of input data which will cause the program to perform incorrectly, to ensure that the programmer sees that the intended function has not been properly realized. In this case the suggested input sequence is 5, -5, 99999.

4. Application of Intention-Based Diagnosis in PROUST

PROUST's methods for analyzing bugs based on intentions has been discussed in detail elsewhere [6, 3, 5, 4]. We will focus here on some of the key features of PROUST's approach. Then, in the next section, we will compare the way that these features are realized in PROUST with the way that they are implemented in Micro-PROUST.

PROUST bases its analysis of each program on a description of the problem that the student is working on. PROUST is intended to be used in an introductory programming course, where the students are expected to complete a series of programming assignments. PROUST is supplied with a description of each assignment that the students will be working on. Each problem description indicates the type of data that the program must process, and the primary goals that must be achieved by the program. They indicate what the programs must do, but not how they should do it. The problem descriptions frequently leave out some details, which the programmer is assumed to know to fill in. The problem descriptions thus supply essential information about

errors in the programmer's intentions, or as errors in the realization of those intentions.

It is intuitively obvious to most programmers that a good understanding of the intended function of a program is necessary in order to debug a program accurately. Some bugs may be identified without reading the code, but frequently a person must read a program carefully in order to diagnose bugs and suggest an appropriate correction for them. Nevertheless, automatic debugging systems tend to analyze program behavior, without reference to underlying intentions [8, 1, 2, 7, 11]. Such methods can fail when the program runs, but generates incorrect results. PROUST is unique in its reliance on an explicit description of the intentions underlying programs.

3. An illustration of intention-based diagnosis

In order to illustrate how knowledge of intentions is needed when debugging programs, we will discuss an example buggy program and show how it is debugged. The example is a solution to the problem shown in Figure 1, which we will refer to as the Rainfall Problem. This problem requires that a series of inputs be read, which signify the amount of rainfall per day. The end of input is signaled when the user types 99999; we will refer to this value as the *sentinel value*. The program must perform various computations on these data, such as finding the average and the maximum. It must also ensure that the data that is entered is valid, i.e., negative daily rainfall must be rejected.

Figure 2 shows an actual student solution to the Rainfall Problem. This program has several bugs. If the user types 99999 without first entering rainfall data, the program divides by zero at line 43, and then prints out meaningless results at lines 46 and 47. In addition, the test for invalid input, starting at line 24, has a bug. If the user types 99999 immediately after an invalid value, the 99999 will not be recognized as the signal of input termination. Instead, the 99999 will be processed as data. We claim that knowledge of the intended function of this program is needed in order to debug the program. In particular, we need to know what range of inputs the program is supposed to handle, and what results it should generate for these inputs. For example, if we did not know that 99999 should not be processed as data, we would not be able to determine that the program behaves incorrectly when 99999 is input after a negative input.

PROUST's analysis of the program in Figure 2 is shown in Figure 3. This analysis makes frequent reference to the intentions underlying the program. It describes each buggy line of code

1. Introduction

As a first step in developing tools that can aid novice programmers when they are learning to program, we have designed, built and classroom tested a program called PROUST that can identify the non-syntactic bugs in novices' programs. Currently, PROUST operates on a class of moderately complex looping programs in Pascal that are typical assignments in an introductory programming course. PROUST provides students with a list of the non-syntactic bugs in their programs and suggestions about the misconceptions that they may be laboring under that are responsible for the bugs. PROUST can correctly identify approximately 75% of the bugs in the students programs.

PROUST is a 15,000-line LISP program which runs on a VAX 750 with several megabytes of memory. As such, PROUST cannot run on a personal computer. In a project sponsored by Courseware, Inc. we undertook to design a micro version of PROUST that would be able to run on an IBM PC. Our purposes in undertaking this project were twofold. First, PROUST is a very complex program; we believed that it would be instructive to strip PROUST down to its barest essentials, so others would be able to see more clearly how PROUST works. Second, we wished to demonstrate the feasibility of developing an AI-based system for a personal computer. To these ends, our project has been a success: in this paper we briefly describe the architecture of Micro-PROUST. However, Micro-PROUST as it stands is not powerful enough to use as an educational tool. We will discuss in Section 7 what will be required to expand Micro-PROUST sufficiently to make it a useful product, while at the same time staying within the space limitations of personal computers.

The organization of this paper is as follows. First, we briefly describe the architecture of PROUST, and talk about the features of PROUST that have been incorporated into Micro-PROUST. A comparison of the two systems is made. We conclude by discussing the prospects of building a product version of Micro-PROUST that will perform at an acceptable level in a real educational setting.

2. Intention-Based Diagnosis

The key idea underlying PROUST's approach is *intention-based diagnosis*. That is, PROUST identifies the non-syntactic bugs in a program by determining what the program is intended to do, and then relating these intentions to the actual code. Bugs then become apparent either as

Table of Contents

1. Introduction	1
2. Intention-Based Diagnosis	1
3. An illustration of intention-based diagnosis	2
4. Application of Intention-Based Diagnosis in PROUST	3
5. PROUST and Micro-PROUST	7
5.1. Goal decompositions	7
5.2. The goal database	10
5.3. Plans	10
5.4. Plan-difference rules	11
6. Additional Comparisons of PROUST and Micro-PROUST	13
7. Scaling Up Micro-PROUST	14
8. Concluding Remarks	16

Micro-PROUST: Knowledge-Based Debugging on a Personal Computer

W. Lewis Johnson

Elliot Soloway

Department of Computer Science

Yale University

P.O. Box 2158 Yale Sta.

New Haven, Ct. 06520

(203) 436-0606

9 May 1985

The research described in this paper was co-sponsored by the Personnel and Training Research Groups, Psychological Sciences Division, Office of Naval Research and the Army Research Institute for the Behavioral and Social Sciences, under Contract No. N00014-82-K-0714, Contract Authority Identification Number 154-492. Approved for public release; distribution unlimited. Reproduction in whole or part is permitted for any purpose of the United States Government. The development of Micro-PROUST was sponsored by Courseware, Inc., San Diego, Ca.

4. PROUST/Micro-PROUST: A Preliminary Comparison

23. April 1985

19

(DPS LOOP-INPUT-VALIDATION
INSTANCES
(BAD-INPUT-SKIP-GUARD
BAD-INPUT-LOOP-G

INIT-AGENDA

PROCESS-NEXT-GOAL

GET-NEXT-GOAL INSTANTIATE PROCESS-PLANS

7. The goal database

In order to process each goal, PROUST must look the goal up in the goal-plan database, in order to determine how the student might have implemented the goal. Information about goals is stored on the property list of the goal. The following properties are used:

- INSTANCES -- a list of plans which can be used to implement the goal
- NAME-PHRASE -- a string indicating in English the function of the goal.

7.1. The DPS macro

The following macro is used to construct the goal knowledge base, as well as other knowledge bases.

- DPS(*var slot1 filler1 slot2 filler2 ...*) -- adds properties *slot1*, *slot2*, etc. to the property list of *var*. DPS does not evaluate its arguments.

Example:

```
(DPS FIDO
  SPECIES CANIS-FAMILIARIS
  AGE      8
  SEX      MALE)
```

⇒

```
(PUT 'FIDO 'SPECIES 'CANIS-FAMILIARIS)
(PUT 'FIDO 'AGE 8)
(PUT 'FIDO 'SEX 'MALE)
```

7.2. Example goal database

The following database is required in order to handle the examples described in Section 2.

```
(DPS SENTINEL-CONTROLLED-LOOP
  INSTANCES
  (SENTINEL-READ-PROCESS-WHILE
   SENTINEL-PROCESS-READ-WHILE
   SENTINEL-READ-PROCESS-REPEAT)
  NAME-PHRASE
  "input processing")
```

```

(MAXIMUM (NEW . (*VAR* NEW SENTINEL-CONTROLLED-LOOP)))
(OUTPUT (VAL . (*VAR* MAX MAXIMUM)))
(GUARD-EXCEPTION (CODE . (*VAR* OUTPUT: OUTPUT))
  (PRED . (= (*VAR* COUNT COUNT) 0)))
(GUARD-EXCEPTION (CODE . (*VAR* UPDATE: AVERAGE))
  (PRED . (= (*VAR* COUNT COUNT) 0))))

```

Note that some variables are bound to an expression which contains a **VAR** form in it. These are used to refer to variables defined by other goals. For example, *(*VAR* NEW SENTINEL-CONTROLLED-LOOP)* refers to the variable *NEW* in the *SENTINEL-CONTROLLED-LOOP* goal.

6. Processing the goal agenda

During processing of the program the goal agenda is stored in the variable **AGENDA**. The goal currently being processed is in the variable **ACTIVE-GOAL**. The list of goals that have already been analyzed is called **ANALYZED-GOALS**. The processing of goals in Micro-PROUST is extremely simple; the program simply starts with the first goal in **AGENDA**, processes each goal in order, one at a time. This continues until **AGENDA** is empty.

After a goal has been matched, and a plan has been chosen, the goal and the plan are stored in **ANALYZED-GOALS**. This is discussed further in Section 10.3.

The following routines are used to manipulate the goal agenda:

- (INIT-AGENDA *problem description*) -- sets **AGENDA** to be the list of goals contained in the problem description
- (PROCESS-NEXT-GOAL) -- get the next goal and process it. This calls GET-NEXT-GOAL to get the next goal to process, then passes this to INSTANTIATE. PROCESS-PLANS is called in order to match the plans to the code.
- (GET-NEXT-GOAL) -- removes the goal at the head of **AGENDA**, and stores it in **ACTIVE-GOAL**. Returns **ACTIVE-GOAL**.
- (INSTANTIATE *goal*) -- instantiates a goal. Returns a list of plan instantiations. See Section 9.
- (PROCESS-PLANS *plans*) -- match the plans against the code. See Section 10.

The calling hierarchy of these routines is:

5. Problem descriptions

The problem descriptions for the two problems will appear as follows. These problem descriptions are a list of goals which will have to be satisfied in the student's program. For the sake of simplicity we are assuming that every goal that will be realized in the program will have to be included in the problem description.

Each goal is a list whose CAR is the type of goal which is required, and whose CDR is a list of variable bindings. Each goal has a specific set of free variables associated with it, not all of which may be listed in the bindings list. For example, SENTINEL-CONTROLLED-LOOP has two free variables: NEW, which is the new-value variable, and STOP which is the stop value. Only STOP is mentioned in the SENTINEL-CONTROLLED-LOOP goals that appear below. If a variable does not appear in the bindings list, it remains free, and must be bound during the process of matching the goal against the program. In other words, Micro-PROUST determines the binding of NEW in SENTINEL-CONTROLLED-LOOP when examines the student's program and discovers which Pascal variable in the program corresponds to NEW.

```
(DEFINE *AVG-SPEC* '(
  (SENTINEL-CONTROLLED-LOOP (STOP . 99999))
  (COUNT (NEW . (*VAR* NEW SENTINEL-CONTROLLED-LOOP)))
  (AVERAGE (NEW . (*VAR* NEW SENTINEL-CONTROLLED-LOOP)))
  (SUM (NEW . (*VAR* NEW SENTINEL-CONTROLLED-LOOP))
    (TOTAL . (*VAR* SUM AVERAGE)))
  (OUTPUT (VAL . (*VAR* AVG AVERAGE)))
  (GUARD-EXCEPTION (CODE . (*VAR* OUTPUT: OUTPUT))
    (PRED . (= (*VAR* COUNT COUNT) 0)))
  (GUARD-EXCEPTION (CODE . (*VAR* UPDATE: AVERAGE))
    (PRED . (= (*VAR* COUNT COUNT) 0)))))

(DEFINE *RAINFALL-SPEC* '(
  (SENTINEL-CONTROLLED-LOOP ((*VAR* NEW SENTINEL-CONTROLLED-LOOP) 99999))
  (LOOP-INPUT-VALIDATION ((*VAR* NEW SENTINEL-CONTROLLED-LOOP)
    (< (*VAR* NEW SENTINEL-CONTROLLED-LOOP) 0)))
  (AVERAGE (NEW . (*VAR* NEW SENTINEL-CONTROLLED-LOOP)))
  (COUNT (NEW . (*VAR* NEW SENTINEL-CONTROLLED-LOOP)))
  (OUTPUT (VAL . (*VAR* AVG AVERAGE)))
  (GUARD-EXCEPTION (CODE . (*VAR* OUTPUT: OUTPUT))
    (PRED . (= (*VAR* COUNT COUNT) 0)))
  (OUTPUT (VAL . (*VAR* COUNT COUNT)))
  (SUM (NEW . (*VAR* NEW SENTINEL-CONTROLLED-LOOP))
    (TOTAL . (*VAR* SUM AVERAGE)))
  (GUARDED-COUNT (NEW . (*VAR* NEW SENTINEL-CONTROLLED-LOOP))
    (PRED . (> (*VAR* NEW SENTINEL-CONTROLLED-LOOP) 0)))
  (OUTPUT (VAL . (*VAR* GUARDED-COUNT COUNT)))
```

BEGIN	a BEGIN-END block
CASE	a CASE statement
CASE-PART	a branch of a CASE statement
CONST-LIST	a list of constants referred to by a CASE-PART
WRITE	a call to WRITE
WRITELN	a call to WRITELN
READ	a call to READ
READLN	a call to READLN
:=	an assignment statement
STATEMENT-LIST	a list of statements; used in REPEAT and CASE statements
WHILE	a WHILE statement.
REPEAT	a REPEAT statement
IF	an IF statement
<	relational expressions
>	
<=	
>=	
<>	
=	
AND	logical expressions
OR	
NOT	
+	arithmetic expressions
-	
*	
/	
DIV	
MOD	
strings	
numbers	
identifiers	

Parse tree nodes are structures containing the following fields:

- NAME -- the name of the parsetree node
- PARENT -- the parent of the node in the parse tree
- CHILDREN -- children of the node
- LINE -- the line number where the code appears in the student's program
- MARK -- T or NIL, indicates whether or not the statement has been matched yet.

4.2. Parse tree nodes

Generally speaking, there is a node in the syntax tree for each syntactic statement, expression and subexpression in the program. Each parse tree node is labeled according to the principal keyword, operator, or value in the corresponding statement or expression. The following example shows a fragment of Pascal text and the parse tree that is generated for it.

```

WHILE N<0 DO
  BEGIN
    WRITELN( N, 'IS INVALID, PLEASE REENTER' );
    READ( N );
  END;

      WHILE
      <
      BEGIN
      N  0
      WRITELN      READ
      N "IS INVALID, PLEASE REENTER"  N

```

Note that I have chosen to use as names of the parse tree nodes the token values such as N and 0, rather than the token types, TOK-IDENTIFIER and TOK-NUMBER. I also name nodes for operators using the actual operator: the top-most node denoting the expression N<0, for example, is <, not TOK-LESS-THAN. *Check the Common Lisp manual: you may have to change this.*

In the case of statements, the name of the parse tree node is the principal keyword in the statement. This means that a parse tree node for a WHILE statement is named WHILE, and the parse tree node for an assignment statement is named :=. There is no parse tree node corresponding to the keywords THEN, DO, or END, so these do not appear in the tree.

Here is a list of the kinds of parse tree nodes which are used in Micro-PROUST:

PROGRAM	the root node for the parse tree
PROG-HEADER	the I/O header of the program
DECLS	the declaration part of the program
CONST-DECLS	constant declarations
VAR-DECLS	variable declarations
CONST	a constant declaration
VAR	a variable declaration
ID-LIST	a list of identifiers

4.1. The lexer

Lexers are also described in detail in compiler construction texts. I leave the details of the lexer design to the implementor. However, I should point out that there may be ways to take advantage of the Lisp READ function when constructing the lexer for Micro-PROUST. After all, READ processes text and identifies numbers and atoms, which is a lexical analysis process. In principal you could lex your Pascal programs by repeatedly calling READ on the Pascal program, and looking at what READ returns to see if it is a number, an atom, or an S-expression. Note, however, that to do this will require definition of read macros for special characters such as commas, semicolons, and operators. Otherwise if expressions such as A+B will be lexed as a single atom rather than as A, +, and B.

Lexical tokens are denoted using two values: the token type and the token value. The token type is a code indicating the type of token which has been encountered. These codes are Lisp atoms. Thus we might have the following relationship between tokens and token types:

Token	Token Type
BEGIN	TOK-BEGIN
UNTIL	TOK-UNTIL
<	TOK-LESS-THAN
<>	TOK-NOT-EQ
;	TOK-SEMICOLON
,	TOK-COMMA
*	TOK-MULTIPLY
-	TOK-MINUS
a-pascal-variable	TOK-IDENTIFIER
4.75	TOK-NUMBER
'hello world'	TOK-STRING
<end of file>	TOK-EOF

Note that in the case of identifiers, numbers and strings, the token type is simply TOK-IDENTIFIER, TOK-NUMBER, and TOK-STRING, respectively. In these cases the token value is the actual token, i.e., the name of the identifier, the actual number or string. In other cases the token value is undefined, since the token type is sufficient to describe what kind of token it is.

Bug Report

1. You're missing a sentinel guard. If a sentinel value is input immediately following a negative value, your program will process it as if it were data.

Try the following data in your program to see this:

5 -5 99999

Other bugs:

2. You need a test to check that at least one valid data point has been input before line 50 is executed. The average is not defined when there is no input.
3. You need a test to check that at least one valid data point has been input before line 46 is executed. The average is not defined when there is no input.
4. You need a test to check that at least one valid data point has been input before line 51 is executed. The maximum is not defined when there is no input.

Figure 3: PROUST's output for the example in Figure 2

the problems, without predisposing PROUST toward specific types of solutions.

When a student submits a program for analysis by PROUST, PROUST retrieves the appropriate problem description from the problem description library. These problem descriptions give PROUST a start at understanding the intended function of the program. PROUST must still go through a considerable amount of effort in order to understand how each student program is supposed to work. However, given a description of the problem, and knowledge base of facts about programming, PROUST is able in most cases to identify the student's intentions, and use the intentions to find the bugs.

Program analysis is performed in PROUST using an analysis-by-synthesis process. PROUST synthesizes different ways of solving and implementing the assigned programming problem. PROUST must have enough programming knowledge that it could write the program itself. After PROUST synthesizes a possible problem solution, it compares the hypothesized solution to the student's code. If there is a match, then it is likely that the synthesized solution fits the student's intentions. If there is a mismatch, then either the student's intentions do not match the synthesized solution, or else there is a bug in the realization of those intentions. PROUST

then uses its knowledge of bugs to determine if the mismatches result from bugs. If so, these bugs are reported to the student.

Synthesis of possible problem solutions involves two steps. First, PROUST synthesizes alternative *goal decompositions* for the program. PROUST's problem descriptions consist in part of high-level descriptions of the goals which must be satisfied in solving the problem. Goal decompositions are the relationships between the goals in the problem statement and the goals which are actually implemented in the student's program. The more complex the programming problem is, the more reasoning about goals the programmer has to do in order to put them in a form that can be implemented in code. The goal decomposition records the additions, modifications, and deletions to the original set of goals which are listed in the problem description.

The second step in synthesizing possible solutions is to select *programming plans* to implement the goals which result after goal decomposition. Programming plans are stereotypic methods for accomplishing programming tasks. Expert programmers make extensive use of programming plans in writing and understanding programs [10]. In PROUST the possible plans that the student might use to implement a goal are all matched against the program, to see which fits the code best.

When a plan fails to match the code exactly, PROUST makes a note of the differences between the plans and the code. It then uses a database of *plan-difference rules* to explain the plan differences. These plan-difference rules suggest bugs which could cause the match failures. The bugs that the plan difference rules identify are then reported to the student.

To summarize, intention-based diagnosis in PROUST consists of the following steps:

- retrieving the appropriate problem description,
- hypothesizing goal decompositions for the program,
- hypothesizing plans to implement the goal decompositions,
- matching the plans against the program, and
- using plan-difference rules to explain the plan mismatches.

5. PROUST and Micro-PROUST

Micro-PROUST goes through nearly the same process that PROUST does in analyzing buggy programs. Some of PROUST's analysis steps have been eliminated, but the resulting mechanism still retains most of the essential features of PROUST. The features of PROUST which were eliminated are those which require substantial amounts of code to implement, and which are not necessary for analyzing most novice programs. In order to highlight the similarities and differences between PROUST and Micro-PROUST, we will describe step by step the knowledge and processing required by the two systems to analyze solutions to the Rainfall Problem.

5.1. Goal decompositions

The main difference between PROUST and Micro-PROUST is that whereas PROUST bases its analysis on problem descriptions, and generates goal decompositions for each program, Micro-PROUST bases its analysis on hand-coded goal decompositions written by us. These goal decompositions are organized into a library, just as PROUST's problem descriptions are organized into a library. When Micro-PROUST analyzes a program, it retrieves the appropriate goal-decomposition description, and uses it to understand the program.

Micro-PROUST assumes that every solution to a given programming problem will have the same goal decomposition. This assumption is untenable for harder programming problems, but it may be tolerable if the programs are small.

In order to understand the effect of assuming a unique goal decomposition per problem, let us compare PROUST's problem description for the Rainfall Problem with Micro-PROUST's goal-decomposition description for the same problem. Figure 4 shows, in a stylized form, PROUST's description of the Rainfall Problem. This problem description consists of two parts: a set of *objects* which the program manipulates, and a set of *goals* that must be satisfied. Objects are the quantities of data that the program should manipulate. Two objects are defined in this example: ?DailyRain, the series of rainfall amounts, and ?Sentinel, the sentinel value. Object names are indicated by a preceding question mark. The goals are requirements that the program must satisfy. For example, *Average* is the goal of computing the average of some set of values; *Output* is the goal of outputting a value to the terminal. Goal names appear in italics. An English paraphrase of each statement in the description of the Rainfall Problem, one line at a time, is shown in Figure 5.

```

Define-Program Rainfall;

Define-Object ?DailyRain ObjectClass ScalarMeasurement;
Define-Object ?Sentinel Value 99999;

Define-Goal Sentinel-Controlled-Input( ?DailyRain, ?Sentinel );
Define-Goal Input-Validation( ?DailyRain, ?DailyRain < 0 );
Define-Goal Output( Average( ?DailyRain ) );
Define-Goal Output( Count( ?DailyRain ) );
Define-Goal Output( Guarded-Count( ?DailyRain, ?DailyRain > 0 ) );
Define-Goal Output( Maximum( ?DailyRain ) );

```

Figure 4: The problem description for the Rainfall Problem

- This problem is called "Rainfall".
- Define an object called DailyRain; it is a scalar measurement, i.e., a non-negative real number.
- Define an object called Sentinel, whose value is 99999.
- Generate successive values of DailyRain by reading them from the terminal, stopping when Sentinel is read.
- Test DailyRain for validity, by ensuring that DailyRain < 0 is never true.
- Output the average of DailyRain.
- Output a count of the number of values that DailyRain takes.
- Output a count of the number of values in DailyRain for which DailyRain > 0 is true.
- Output the maximum of DailyRain.

Figure 5: A line-by-line paraphrase of Figure 4

Figure 6 shows Micro-PROUST's goal-decomposition description for the Rainfall Problem. There are a number of differences between this description and PROUST's problem descriptions. For example, there are no object definitions; object processing was dispensed with in Micro-PROUST. As a consequence, Micro-PROUST must assume that all students will attribute the same properties to the data being processed. For example, PROUST's description of the Rainfall Problem indicates that the object ?DailyRain is a *ScalarMeasurement*, i.e., it is a non-negative real number. PROUST's knowledge base includes a description of *ScalarMeasurement*, specifying that objects of this type must be nonnegative and real. In Micro-PROUST there is no knowledge base of types of objects. All relevant facts about the objects being processed are

incorporated into the goals which are listed in the goal decomposition, or else are omitted. Thus the requirement that ?DailyRain must be non-negative is enforced by the *InputValidation* goal which checks each input of ?DailyRain for validity. The fact that ?DailyRain should be a real number is omitted; if a student declared the input variable to be an integer, Micro-PROUST would fail to detect the error.

The most significant difference between PROUST's description of the Rainfall Problem and Micro-PROUST's description is that several goals have been added to Micro-PROUST's description which are absent from PROUST's description. A *Sum* goal has been added to compute the sum of the inputs, and three *Guard-Exception* goals have been added to check for boundary conditions, such as division by zero. These goals are added in order to fill out the goal decomposition. PROUST would add these goals automatically as needed, relying upon its knowledge about goals. In Micro-PROUST's descriptions every goal must be made explicit. Furthermore, Micro-PROUST assumes that every goal listed must be explicitly realized in the program using some plan. It is possible to solve the Rainfall Problem without summing the inputs, and without checking for division by zero; Micro-PROUST cannot understand such solutions because they do not fit the given goal decomposition.

```

Sentinel-Controlled-Input( ?New, 99999 );
Input-Validation( (?New in goal Sentinel-Controlled-Input),
                  (?New in goal Sentinel-Controlled-Input) < 0 );
Average( (?New in goal Sentinel-Controlled-Input) );
Count( (?New in goal Sentinel-Controlled-Input) );
Output( (?Avg in goal Average) );
Guard-Exception( (?Output: in goal Output),
                  (?Output: in goal Output) = 0 );
Output( (?Count in goal Count) );
Sum( (?New in goal Sentinel-Controlled-Input),
      (?Sum in goal Average) );
Guarded-Count( (?New in goal Sentinel-Controlled-Input),
                (?New in goal Sentinel-Controlled-Input) > 0 );
Output( (?Count in goal Guarded-Count) );
Maximum( (?New in goal Sentinel-Controlled-Input) );
Output( (?Max in goal Maximum) );
Guard-Exception( (?Output: in goal Output),
                  (?Count in goal Count) = 0 );
Guard-Exception( (?Update: in goal Average),
                  (?Count in goal Count) = 0 );

```

Figure 6: Micro-PROUST's goal decomposition for the Rainfall Problem

5.2. The goal database

PROUST and Micro-PROUST both have knowledge bases of programming goals. Each goal in Micro-PROUST's knowledge base is also in PROUST's knowledge base. However, Micro-PROUST's knowledge base contains less information about each goal. All knowledge needed in PROUST in order to construct goal decompositions has been removed, since Micro-PROUST cannot construct goal decompositions automatically. What is left are the names of plans which implement the goal, and an English phrase describing the goal. Figures 7 and 8 show the same goal, *Input-Validation*, as they appear in PROUST's and Micro-PROUST's databases, respectively.

Input-Validation

Form:	<i>Input-Validation(?Val, ?Pred)</i>
Name phrase:	<i>"input validation"</i>
Main component:	Guard:
Instances:	BAD INPUT SKIP GUARD
	BAD INPUT LOOP GUARD

Figure 7: A goal in PROUST's knowledge base

Input-Validation

Name phrase:	<i>"input validation"</i>
Instances:	BAD INPUT SKIP GUARD
	BAD INPUT LOOP GUARD

Figure 8: A goal in Micro-PROUST's knowledge base

5.3. Plans

There is a close relationship between plans in Micro-PROUST and plans in PROUST. Every plan in Micro-PROUST's knowledge base is also in PROUST's knowledge base. However, Micro-PROUST's plan knowledge contains less information about each plan, and the plans cannot be used in as wide a range of cases. These points will be illustrated via a specific example, the BAD INPUT LOOP GUARD PLAN, which is an implementation of the goal *Input-Validation*. This plan checks input data for validity, and repeatedly rereads the data until it is valid.

Figure 9 shows Micro-PROUST's version of the BAD INPUT LOOP GUARD PLAN. Like all plans in Micro-PROUST it consists of Pascal statements to match against the program, and pointers indicating where the plan should match. The Pascal statements consist of a WHILE loop

containing a WRITELN statement and a READ statement, followed by an IF statement testing for the sentinel value. The pointer indicating where to match the plan is at the top of the plan template; it indicates that the plan should be found in the part of the plan implementing the goal *Sentinel-Controlled-Input* which is labeled Process:. In other words, the plan should be in the part of the loop that processes the input data. Note that this plan is quite specific; it will only work in sentinel-controlled loops, and even then it will not match if a READLN statement appears in the program instead of a READ statement.

BAD INPUT LOOP GUARD

Template:

```

      (in component Process: of goal Sentinel-Controlled-Input)
Guard:      WHILE ?Pred DO
              BEGIN
                  WRITELN( ?* );
Input:      READ( ?Val )
              END;
InternalGuard: IF ?Val <> ?Stop THEN
Process:      ?*

```

Figure 9: Micro-PROUST's BAD INPUT LOOP GUARD PLAN

Figure 10 shows PROUST's rendition of the BAD INPUT LOOP GUARD PLAN. The main difference between the two plans is that PROUST allows subgoals to be inserted into the plan. These subgoals become part of the goal decomposition that PROUST constructs for the program. The subgoals can be implemented using different plans, thus increasing the variety of code that the plan can match. Instead of referring specifically to the goal *Sentinel-Controlled-Input* within the plan, PROUST's plan refers to a generic class of goals called *Read & Process*. This class denotes all goals which involve the inputting and processing of data.

5.4. Plan-difference rules

When a plan fails to match the student's program exactly, the differences between the plan and the code must be explained. By explaining a plan difference we mean suggesting a bug which would produce the detected plan difference, as well as misconceptions which can cause the bug. PROUST and Micro-PROUST both use plan-difference rules to explain plan differences. These rules are test-action pairs, where the test part examines the plan differences and the action indicates what if any bugs these plan differences suggest. There is a correspondence between

BAD INPUT LOOP GUARD

Variables: ?Val, ?Pred
 Template:
 (in component Process: of goal Read & Process)
 spanned by:
 Guard: WHILE ?Pred DO
 BEGIN
 subgoal Output Diagnostic()
 Next: subgoal Simple Input(?Val)
 END
 Process: ?*

Posterior goals:
 Sentinel Guard((?New from goal Read & Process),
 (?Stop from goal Read & Process))

Figure 10: A plan for implementing *Input-Validation*

plan-difference rules in the two systems, although the tests and actions of the rules tend to differ in certain respects.

To see how plan-difference rules are used in the two systems, we will examine how they explain one of the bugs in the example in Figure 2. This program uses the BAD INPUT LOOP GUARD to implement the goal *Input-Validation*, but there is no test for the sentinel value after the WHILE loop at line 24. PROUST discovers the match failure when it is processing the goal *Sentinel-Guard*, a subgoal of the BAD INPUT LOOP GUARD PLAN. The plan-difference rule which accounts for the match failure in PROUST is shown in Figure 11. The rule checks whether the sentinel guard is the only plan that is missing, or whether the input step is also missing. If the input step is missing, there is no reason to complain about the sentinel guard being missing, since the missing input is the more important bug. If the input step is present, the rule checks to see whether there are sentinel guards elsewhere in the loop which might serve the function of the missing sentinel guard. When no other sentinel guards are present, the rule indicates that the missing sentinel guard is a bug, and it may have been omitted inadvertently by the student.

IF no plans implementing *Sentinel-Guard* can be matched,
 AND the input step of either BAD INPUT LOOP GUARD or the
 main loop was found,
 AND no sentinel guards appear later on in the loop,
 THEN the sentinel guard may have been omitted by mistake.

Figure 11: Rule for recognizing missing sentinel guards

The corresponding rule in Micro-PROUST, shown in Figure 12, is much more direct. It simply checks to see whether the missing plan component is labeled `InternalGuard:`. If so, it may have been inadvertently omitted. The sentinel guard in the `BAD INPUT LOOP GUARD PLAN` is labeled `InternalGuard:`, as can be seen in Figure 9. Therefore the rule applies. Because Micro-PROUST's rule is simpler, it will apply in some cases where it should not. If the sentinel is tested in an unusual but correct way, Micro-PROUST is likely to indicate that the sentinel test is buggy. Micro-PROUST's rules are not sensitive to correlations between bugs, so it cannot detect when one bug might be the cause of the other.

IF a plan component of type `InternalGuard:` cannot be matched,
THEN it may have been omitted by mistake.

Figure 12: Micro-PROUST's rule for missing guards

Some plan-difference rules in PROUST not only explain the plan differences; they also correct them. For example, rules which identify when `BEGIN-END` pairs are missing insert the missing `BEGIN-END` pairs into PROUST's internal parse-tree representation of the code. Subsequent analysis of the program will then proceed as if the `BEGIN-END` pairs were present in the student's program. If PROUST did not insert the `BEGIN-END` pairs, it might misunderstand the intended data flow of the code, and signal bugs which are really side-effects of the buggy block structure. Micro-PROUST, on the other hand, makes no attempt to correct bugs. It therefore is more likely to signal more bugs than are really present in the program.

6. Additional Comparisons of PROUST and Micro-PROUST

In Figure 13 we present further comparisons of PROUST and Micro-PROUST along a number of key dimensions. PROUST has been under development for some 4 years, and is the successor to MENO, an earlier version of our bug identification system [9]. In contrast, Micro-PROUST was not a research effort but could draw directly on our experience in building bug analysis systems. PROUST is faster than Micro-PROUST as long as enough memory is available. However, PROUST needs two megabytes of memory to run comfortably; if a VAX is heavily loaded, PROUST slows down substantially, and Micro-PROUST is then faster. The difference in size of the two programs is dramatic; Micro-PROUST is only 10% of the size of PROUST! In the next section, we address the natural question: can Micro-PROUST be scaled up to the level of PROUST's performance, and still have acceptable time and space characteristics?

	PROUST	Micro-PROUST
Cost:	~ \$450,000	~ \$25,000
Time:	4 years	2 months
Coverage:	2 programming assignments	1 programming assignment
Bug Catalogue:	~ 50 ~ 10	
Size:	15,000 lines of LISP	1500 lines of LISP
Machine:	DEC VAX 750 (4 Meg)	IBM PC (512 K)
Run time:	.5 - 3 minutes	~90 seconds
Performance:	75% of bugs correctly identified	unknown

Figure 13: Fact Sheet: PROUST and Micro-PROUST

7. Scaling Up Micro-PROUST

We will now examine some of the issues which we expect will arise when Micro-PROUST is scaled up to approximate PROUST's performance in finding bugs. We will focus on two issues: the problems involved in expanding Micro-PROUST's knowledge bases, and the limitations that Micro-PROUST's simplified architecture impose.

In order for Micro-PROUST to cover a wide range of novice programs, its knowledge bases will have to be greatly expanded. In PROUST, 4000 lines of code are needed to specify goals, plans, and plan-difference rules. We have seen that Micro-PROUST's knowledge bases correspond closely to PROUST's. Since Micro-PROUST's knowledge is simplified, the same knowledge units would take up less space in Micro-PROUST than they would in PROUST. On the other hand, Micro-PROUST's plans are not as general as PROUST's, since they cannot contain

subgoals. This will probably mean that more plans will have to be added to Micro-PROUST than appear in PROUST. It is clear that Micro-PROUST's knowledge bases will be sizeable, and will press against the space limitations of the IBM PC.

It should be possible to organize Micro-PROUST's knowledge bases so that space limitations are avoided. Only a fraction of Micro-PROUST's knowledge need be available at any one time. For example, when a particular goal is being processed, the only plans that need be in memory are those which implement that goal. It therefore should be possible to store Micro-PROUST's knowledge on disk, and retrieve knowledge only as needed. Micro-PROUST's speed would be reduced because of this, so it is hard to say whether the resulting system would be fast enough to use in the classroom.

The more troubling problems with Micro-PROUST stem from the reduced power of its analytic techniques. Micro-PROUST assumes that all solutions to a given programming problem will have the same goal decomposition. It is prone to misdiagnosis of novice bugs. These shortcomings may limit Micro-PROUST's effectiveness, or may necessitate enhancements of Micro-PROUST's diagnostic capabilities.

The effect of restricting the goal decompositions of programs depends on the complexity of the problem. The more goals there are that must be satisfied in a program, the more ways there are that these goals can be combined. The range of possible goal decompositions therefore increases rapidly as problem complexity increases. Our empirical studies of novice programs indicate that 40% of novice solutions to the Rainfall Problem fail to conform to Micro-PROUST's goal decomposition for this problem. Micro-PROUST cannot avoid making frequent analysis errors as long as its model of students' intentions is faulty this often.

Even if Micro-PROUST's goal decomposition is correct, there may still be cases where Micro-PROUST misdiagnoses bugs. As indicated earlier, Micro-PROUST does not correct bugs when it finds them; therefore the presence of some bugs may cause Micro-PROUST to misdiagnose other bugs. Furthermore, unlike PROUST, Micro-PROUST has no way of choosing between alternative interpretations of the code. It does not notice if more than one plan can match the same lines of code. Instead, Micro-PROUST simply chooses the first account of the code that it comes across. Micro-PROUST's accuracy will suffer because of this, but the extent of the performance degradation is hard to predict. Empirical evaluations of Micro-PROUST will

be needed to determine exactly how accurate its analysis can be expected to be.

8. Concluding Remarks

As a demo, Micro-PROUST unquestionably allows one to quickly see the potential for computer-based enhancement of learning. However, as a product, it is still unclear whether or not Micro-PROUST can ever be even approximately as effective as its parent, PROUST, running on a large machine. However, we believe that Micro-PROUST's knowledge can be expanded without running into space limitations. The resulting system should be reasonably effective, at least on simple programming problems. Further work will be required to determine how effective Micro-PROUST's approach can be, and whether or not it will result in a viable educational tool.

References

1. Fosdick, L.D., and Osterweil, L.J. "Data flow analysis in software reliability". *Computing Surveys* 8, 3 (1976), 305-330.
2. Harandi, M.T. Knowledge-Based Program Debugging: a Heuristic Model. Proceedings of the 1983 SOFTFAIR, SoftFair, 1983.
3. Johnson, W.L., and Soloway, E. "PROUST: Knowledge-Based Program Understanding". *IEEE Transactions on Software Engineering SE-11*, 3 (1984), 267-275.
4. Johnson, W.L., and Soloway, E. Intention-Based Diagnosis of Programming Errors. Proc. of the Nat. Conf. on Art. Intel., AAAI, 1984, pp. 162-168.
5. Johnson, W.L. and Soloway, E. "PROUST: An Automatic Debugger for Pascal Programs". *Byte* 10, 4 (1985), 179-190.
6. Johnson, W.L. Intention-Based Diagnosis of Programming Errors. forthcoming, Yale University Department of Computer Sci., 1985.
7. Lukey, F.J. "Understanding and Debugging Programs". *Int. J. of Man-Machine Studies* 12 (1980), 189-202.
8. Shapiro, E.. *Algorithmic Program Debugging*. MIT Press, Cambridge, Mass., 1982.
9. Soloway, E., Rubin, E., Woolf, B., Bonar, J., and Johnson, W. L. "MENO-II: An AI-Based Programming Tutor". *Journal of Computer-Based Instruction* 10, 1 (1983).
10. Soloway, E. and Ehrlich, K. "Empirical Investigations of Programming Knowledge". *IEEE Transactions on Software Engineering SE-10*, 5 (1984).
11. Wertz, H. "Stereotyped program debugging: an aid for novice programmers". *Int. J. of Man-Machine Studies* 16 (1982), 379-392.

5. Micro-PROUST Implementation: The LISP Code

```

;; there are none left to try.
(defun plan-match (plans)
  (cond ((null plans) ())
        ((ins-status (car plans)) (plan-match (cdr plans)))
        (t (list-plan (car plans))
            (cond
              ((match-a-plan (car plans)) (car plans))
              (t (plan-match (cdr plans)))))))

;; Recursively try to match each line of the plan by calling match-component.
;; If all the components match, return T, else block the plan and return nil.
(defun match-a-plan (plan)
  (cond ((match-component plan (ins-address plan))
        (and exec-out (format t "Matched code!" (ins-name plan)))
        (cond ((equal (incf (ins-address plan))
                      (length (ins-matches plan)))
              (cond (exec-out
                     (format t "%Plan %a matched code!" (ins-name plan)))
                    t)
                (t (match-a-plan plan))))
        (t (setf (ins-status plan) t)
           (and exec-out
                (format t "%Match errors in plan %a." (ins-name plan))
                ())))))

;; Call context-nodeset to get a list of possible nodes and pass this list
;; along with the plan component to xrefer, xlabel, or xmatch, depending on
;; what type of match instruction it is.
(defun match-component (plan address)
  (let* ((plan-line (get-plan-element plan address))
        (nodeset (context-nodeset (caddr plan-line) plan)))
    (case (car plan-line)
      ((refer) (xrefer plan-line plan address))
      ((label) (xlabel plan-line plan address))
      ((match) (xmatch plan-line plan address)))))

;; Get the plan-line for match-component
(defun get-plan-element (plan address)
  (aref (get (ins-name plan) 'template) address))

;; Context-nodeset
;; .....
;; Context-nodeset uses the third argument in a plan component to construct
;; a list of nodes which can be matched. The third argument is a list of zero,
;; one, or two elements. Context-nodeset-1 creates a list of nodes based on
;; the first element if it exists and context-nodeset-2 modifies this list of
;; nodes based on the second element if it exists. See the Spec for element
;; descriptions.
(defun context-nodeset (context plan)
  (context-nodeset-2 (cdr context) plan
                    (context-nodeset-1 context plan)))

;; Get a list of nodes based on the first element of the context descriptor
;; or return the list of the highest level begin block for a null context
;; descriptor
(defun context-nodeset-1 (context plan)
  (cond
    ((null context) (get-begin-block (node-children *parse-tree*)))

```

```

      (and exec-out (format plan-window "%s" (aref tapit i)))
      (and (aref (ins-matches plan) i)
            (and exec-out (format cek-window "T%s")))))
    (and exec-out (any-key)))

;; List-goal lists the goal name in the goal-window.
(defun list-goal (goal)
  (and exec-out (send goal-window :clear-screen))
  (and exec-out (format goal-window "Current Goal Name: %s" (car goal)))
  (and exec-out (any-key)))

;; Any-key prints out a message and waits for a carriage return to continue.
(defun any-key ()
  (cond (exec-out
        (format exec-window "%nEnter Carriage Return to Continue ---"
                  (read-line exec-window))
        (t
         (format t "%nEnter Carriage Return to Continue ---"
                 (read-line)))))

;; Put is a function from older lisps that I was used to using.
(defun put (x y z)
  (setf (get x y) z))

;; DPS (var1 slot1 filler1 slot2 filler2 ...) adds properties slot1, slot2, etc.
;; to the property list of var1. It is used to build all of the property
;; lists.
(defmacro DPS (var1 &body slot-fill)
  (cond ((oddp (length slot-fill))
        (error "DPS: Bad number of arguments"))
        (t (DPS1 var1 slot-fill))))

;; Auxillary function to DPS
(defun DPS1 (var1 slot-fill)
  (cond ((null slot-fill) t)
        (t (progn () (put var1 (car slot-fill) (cadr slot-fill))
                          (DPS1 var1 (cddr slot-fill))))))

;; Load the Match group
;; Contains the following functions and their supporting functions:
;;   plan-match, match-a-plan, match-component, context-nodeset, xrefer,
;;   xlabel, xmatch, get-candidates, and match-stat.
;; Match sub-group of Micro-Proust MATCH2.lsp
;; Written by Bret Wallach      10/24/84
;;
;; This section of code follows the Micro-Proust design spec fairly well.
;; Refer to it for further details.

;; Define a match-frame
(defstruct match-frame
  (code ())
  (bindings ())
  (errors ()))

;; High level Match functions.
;; .....
;; Try to match each unblocked (ins-status is nil) plan until one matches or

```

```

;; nodes so they are not matched again later.
(defun close-goal (goal plan)
  (progn
    (push (cons goal plan) *analyzed-goals*)
    (mark-stats plan)))

;; Mark each node as matched unless the fourth argument of the plan line is T
;; or the first argument is a Refer or Label command.
(defun mark-stats (plan)
  (let* ((tplt (get (ins-name plan) 'template))
        (nodes (ins-matches plan))
        (n (length nodes)))
    (do*
      ((i 0 (+ i 1)))
      ((= i n))
      (cond ((equal (car (aref tplt i)) 'match)
              (setf (node-mark (car (aref nodes i)))
                    (not (car (cdddr (aref tplt i))))))
            (t ())))))

;; If none of the plans work for a goal, declare it missing
(defun missing-goal (goal)
  (and exec-out (format t "~%Goal ~a is missing" (car *active-goals*)))
  (setf *bug-report* (append *bug-report* (list (list (car *active-goals*)))))

;; Report the bugs
.....
;; For each bug call report1. Report1 determines if there is a reporting
;; function defined for that type of bug and calls it if it exists.
(defun report-bugs ()
  (format t "~%~%"          "*****Bug Report***** ")
  (mapc #'report1 *bug-report*)
  (format t "~%~%"          "*****End of Report*****~%~%~% 12)
  t)

;; The car of a bug is the bug name. If it has a 'report-fun property,
;; retrieve it and call it with the rest of the bug as an argument. See
;; the spec for details.
(defun report1 (bug)
  (let ((bug-func (get (car bug) 'report-fun)))
    (cond (bug-func (format t "~%~%"
                          (funcall bug-func (cdr bug)))
          (t ())))))

;; Utilities
.....
;; List-plan lists the plan components of plan in the plan-window.
(defun list-plan (plan)
  (and exec-out (send plan-window :clear-screen))
  (and exec-out (send chek-window :clear-screen))
  (and exec-out (format plan-window "Plan-name: ~a" (ins-name plan)))
  (let* ((tplt (get (ins-name plan) 'template))
        (nodes (ins-matches plan))
        (n (length nodes)))
    (do*
      ((i 0 (+ i 1)))
      ((= i n))
```

```

(process-next-goal)
(cond (exec-out
      (any-key)
      (send *standard-output* :set-size 80 25)
      (send *standard-output* :clear-screen)
      (with-open-file
        (ous prog :direction :input :element-type 'string-char)
        (pparse))))
(report-bugs))

;; Initialize the agenda
;;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;; Reads the problem file into the global variable *agenda*. Also initializes
;; *active-goals*, *analyzed-goals*, *bug-reports*.
(defun init-agenda (problem)
  (progn
    (with-open-file
      (ous problem
        :direction :input :element-type 'string-char)
      (setf *agenda* (read ous)))
    (setf *active-goals* ())
    (setf *analyzed-goals* ())
    (setf *bug-reports* ())))

;; Process the goals
;;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;; Pops a goal off the top of *agenda*, instantiates the goal, processes the
;; goal and repeats until there are no goals left. It also call list-goal
;; for each goal which lists the goal and bindings in goal window.
(defun process-next-goal ()
  (cond
    (*agenda*
      (list-goal (car *agenda*))
      (progn (process-plans (instantiate (get-next-goal)))
              (process-next-goal)))
    (t t)))

;; Pops a goal off the top of *agenda*.
(defun get-next-goal ()
  (setf *active-goals* (pop *agenda*)))

;; Processes each plan returned by instantiate until one of them matches the
;; program code or none of them matched and the mismatches can't be explained.
(defun process-plans (plans)
  (let ((good-plan ()) (new-plans plans))
    (do ()
      ((or good-plan (null new-plans)))
      (setf good-plan (plan-match plans))
      (cond ((null good-plan)
              (setf new-plans (explain-mismatches plans))
              (t ())))
      (cond (good-plan (close-goal (car *active-goals*) good-plan))
            (t (missing-goal *active-goals*))))))

;; Close the goal
;;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;; If the goal had a plan that matched, remember it and mark all the matched

```

```

:: Title-page
:: Prints title page in beginning of program
:: (defun Title-page ()
:: (format t
:: "~%~%~%
MICRO PROUST
A PASCAL Programming Tutor

Designed By:

Elliot Solovay & W. Lewis Johnson
Yale University

Implemented on the IBM PC By:

Bret Wallach, Advanced Processing
Leszek Izdebski, Courseware, Inc.

"))

:: Parser calling functions
:: This function displays the program file on the screen (using pparse),
:: and parses the file using the function parse. Due to a bug in GCLISP,
:: I was unable to specify a larger initial stack group. As a result, I
:: create a special stack just for parsing.
(defun parse-call (filename)
  (format t "~%Lexing ~s" filename)
  (with-open-file
    (ous filename :direction :input :element-type 'string-char)
    (pparse))
  (stack-group-preset *ns* 'parse filename)
  (format t "~%Parsing ~s" filename)
  (funcall *ns* nil)
  (and exec-out (any-key)))

:: Display the program on the screen.
(defun pparse ()
  (do ((x (read-line ous ()) t) (read-line ous ()) t))
      (i 1 (+ i 1)))
  ((equal x t))
  (format t "~%~d ~s" i x))

(setf *ns* (make-stack-group 'ns* :regular-pdl-size 3800
                             :special-pdl-size 3800))

:: This function initialized the goal agenda, processes the goals, and reports
:: the errors found.
(defun aproust (problem)
  (init-agenda problem)

```

```

        (any-key))
        (t (format t "~%Unable to parse file!?!?")
            (format t "~%Please fix syntax errors and try again.")
            (any-key))))))
    (send *standard-output* :set-size 80 25)
    (do-question))
;; (dribble)
(exit)))

(defun do-question ()
  (send *standard-output* :clear-screen)
  (format t "Your options are:")
  (format t "~% 0: Run Proust on EXAMPLEA (average problem)")
  (format t "~% 1: Run Proust on EXAMPLE1 (average problem)")
  (format t "~% 2: Run Proust on EXAMPLE2 (average problem)")
  (format t "~% 3: Run Proust on EXAMPLE3 (rainfall problem)")
  (format t "~% 4: Run Proust on EXAMPLE4 (rainfall problem)")
  (format t "~% 5: Run Proust on EXAMPLE5 (rainfall problem)")
  (format t "~% 6: Run Proust on EXAMPLER (rainfall problem)")
  (exec-output)
  (format t "~% 8: Exit.")
  (format t "~%Enter option: "))

(defun exec-output ()
  (cond (exec-out
        (format t "~% 7: Disable execution output (currently enabled)")
        (t
         (format t "~% 7: Enable execution output (currently disabled)"))))

;; This function filters out reader errors that would otherwise cause the
;; program to bomb.
(defun safe-read ()
  (multiple-value-bind (what err)
    (ignore-errors (read)))
  (cond ((null err) what)
        (t (format t "~%Input error, please re-enter.~%"
                    (safe-read)))))

;; This function filters out reader and string function errors that would
;; otherwise cause the program to bomb.
(defun safe-string-read ()
  (multiple-value-bind (what err)
    (ignore-errors (string (read))))
  (cond ((null err) what)
        (t (format t "~%Input error, please re-enter.~%"
                    (safe-string-read)))))

;; This function filters out reader and string function errors that would
;; otherwise cause the program to bomb.
(defun safe-p-read ()
  (multiple-value-bind (what err)
    (ignore-errors (read)))
  (cond ((and (null err)
              (integerp what)
              (>= what 0)
              (<= what 8))
        what)
        (t (format t "~%Input error, please re-enter.~%"
                    (safe-p-read)))))

```

I.3 MPROUST.LSP

```

::: ttfaproust.lsp
::: Micro-PROUST calling functions  MPROUST.LSP
::: Written by Bret Wallach of Advanced Processing on 10/23/84
:::   for Courseware, Inc.
:::
::: Welcome to Micro-PROUST.  This code follows the design specification
::: written by W. Lewis Johnson fairly well, so refer to that Spec when
::: necessary
:::
::: Authors do not guarantee the performance of this program in any way nor
::: will they accept responsibility for any problems resulting from the use
::: of this code.
:::
::: This is the top level function.  It sets up four windows.  Exec-window is
::: a two line window at the bottom of the screen used by any-key to wait for
::: a <CR> before proceeding.  Plan-window contains the plan currently being
::: matched, chek-window contains a T for each component of the plan that
::: matched, and goal window contains the current goal and a list of initial
::: bindings.  This function asks for the program file and a problem description
::: identifier.  It then calls parse-call to parse the program and mproust to
::: do the analysis assuming it was possible to parse the program.
(defun m-proust ()
  (let ((exec-window (make-window-stream :top 23 :height 2))
        (plan-window (make-window-stream :top 12 :height 11 :left 3))
        (chek-window (make-window-stream :top 13 :height 10 :width 3))
        (goal-window (make-window-stream :top 9 :height 3)))
    (close-all-files)
    (send *standard-output* :clear-screen)
    (delete-file (merge-pathnames "sysout.dat" ddev))
    ;; (dribble (merge-pathnames "sysout.dat" ddev))
    (do-question)
    (do ((p (safe-p-read) (safe-p-read)))
        ((equal p 8))
        (cond ((equal p 7)
              (setf exec-out (not exec-out)))
              ((and (setf prog (car (aref efiles p)))
                    (setf q (cadr (aref efiles p)))
                    (multiple-value-bind (ig err)
                        (ignore-errors (close (open prog :direction :input)))
                        err))
              (format t "~%File ~a not found." prog)
              (any-key))
              ((multiple-value-bind (ig err)
                  (ignore-errors (close (open q :direction :input)))
                  err)
              (format t "~%File ~a not found." q)
              (any-key))
              (t
               (parse-call prog)
               (cond (*parse-tree*
                     (cond (exec-out
                           (send *standard-output* :clear-screen)
                           (send *standard-output* :set-size 80 8))
                           (t
                            (format t "~%Analyzing program for errors..."))
                           (mproust q)

```

```
(MULTIPLE-VALUE-BIND
  (F A B C D)
  (%SYSINT @X11 0 0 0 0)
  (1+ (LSH (LOGAND A @X00C0) 6)))
)))))
```

```
(WHEN (OR (NOT (BOUNDP '*MEMORY-ALLOCATED-P*))
  (NOT *MEMORY-ALLOCATED-P*))
  (ALLOCATE #X800 3. 2. T) ; allocate all DS mem but 32K
  (SETQ *MEMORY-ALLOCATED-P* T))
```

```
(load "lisplib\\defaac.lsp")
(load "lisplib\\defstruct.lsp")
(LOAD "lisplib\\DRIBBLE.lsp") ; DRIBBLE function
```

```

(IF (PROBE-FILE (FIRST X))
  (APPLY 'OPEN X)
  (PROGN
    (FORMAT T ""&Insert diskette for file "A in drive "A.
Type any character when ready."
      (NAMESTRING (CAR X))
      (PATHNAME-DEVICE (CAR X)))
    (READ-CHAR)
    (APPLY 'SAFE-FILE-OPEN X))))

;;; its rather important that this is HERE, not autoloaded
;;; For closing all currently open files.
(DEFUN CLOSE-ALL-FILES ()
  (MAPCAR '(LAMBDA (FILE-STREAM)
    (PROG1 (FILE-STREAM :PATHNAME)
      (FILE-STREAM :CLOSE)))
    *OPEN-FILE-STREAMS*))

(SETQ *DEFAULT-PATHNAME-DEFAULTS* (MERGE-PATHNAMES (CD) *FDD.LSP*))

;; THE GCLISP toplevel help facility

(SETQ *DEFAULT-IE-OPTIONS*
  '(((IE-COMMANDS (#\C-B . (LAMBDA (&REST IGNORE)          ; ?B - backtrace
    (TERPRI T)
    (BACKTRACE)
    (TERPRI T)
    :REFRESH))
    (#\C-D . (LAMBDA (&REST IGNORE)          ; <ALT>-D DOS
    (FORMAT T ""&Going to DOS..."
      (DOS)
      :REFRESH))
    (#\C-G . (LAMBDA (&REST IGNORE)          ; ?G - CLEAN-UP-ERROR
    (CLEAN-UP-ERROR)))
    (#\C-L . (LAMBDA (&REST IGNORE)          ; ?L - CLEAR SCREEN
    (SEND *TERMINAL-IO* :CLEAR-SCREEN)
    :REFRESH))
    (#\C-P . (LAMBDA (&REST IGNORE)          ; ?P
    (CONTINUE)))
    (#\C-C . (LAMBDA (&REST IGNORE)          ; ?Z
    (STACK-GROUP-UNWIND)))
    (#\ESC . (LAMBDA (BUF IGNORE)
    (LENGTH BUF)))
    (#\RUBOUT . (LAMBDA (IGNORE IGNORE)      ; <RUBOUT> 1 CHAR
    1)))
  )))

(DEFUN FIND-FUNC (BUF)
  (LET ((IDX (1- (LENGTH BUF))))
    (WHEN (PLUSP IDX)
      (DOTIMES (I (LENGTH BUF))
        (IF (EQ #\ (CHAR BUF IDX)) (RETURN NIL))
        (DECF IDX))
      (READ-FROM-STRING BUF NIL NIL :START (1+ IDX)))))

(SETF *SINGLE-DISKETTE-DRIVE-SYSTEM*
  (EQ 1 (UNLESS *NUMBER-OF-DRIVES*
    (SETF *NUMBER-OF-DRIVES*

```

```

((NULL X) (NREVERSE RESULT))))))
(88. . . . . :#X
(LAMBDA (S &AUX (*READ-BASE+ 16.))
  (READ S NIL NIL T)))
(92. . . . . :#\
(LAMBDA (S)
  (FUNCALL S :UNTYI 92.) ; put escape char back
  (DO ((STR (STRING (READ S NIL NIL T)))
      (BITS 0)
      (I 0))
    (())
    (COND ((= (- (LENGTH STR) I) 1) ; 1 char left
      (RETURN (CODE-CHAR (CHAR STR I) BITS)))
      ((STRING-EQUAL "C-" STR :START2 I :END2 (+ I 2))
      (SETQ BITS (LOGIOR BITS 1)
        I (+ I 2)))
      ((STRING-EQUAL "M-" STR :START2 I :END2 (+ I 2))
      (SETQ BITS (LOGIOR BITS 2)
        I (+ I 2)))
      ((NAME-CHAR (SETQ STR (SUBSEQ STR I)))
      (RETURN (CODE-CHAR (NAME-CHAR STR) BITS)))
      (T
      (ERROR "Bad \"#\0\\\" name: "S" STR))))))
(124. . . . . :#|
(LAMBDA (S)
  (DO ((CNT 0)
      (CHAR (FUNCALL S :TYI) (FUNCALL S :TYI))
      (LCHAR 0 CHAR))
    ((AND (= CHAR 35.)(= LCHAR 124.)(ZEROP CNT)) NIL)
    (COND ((AND (= LCHAR 35.)(= CHAR 124.))
      (INCF CNT))
      ((AND (= LCHAR 124.)(= CHAR 35.)(> CNT 0))
      (DECF CNT))))))
))

:: Used by the #+ and #- macros
(DEFUN |#-FEATUREP| (F)
  (COND ((ATOM F)(MEMBER F *FEATURES*))
    ((EQ (CAR F) 'NOT) (NOT (|#-FEATUREP| (SECOND F))))
    ((EQ (CAR F) 'OR)
    (DOLIST (I (REST F) NIL)
      (WHEN (|#-FEATUREP| I) (RETURN T))))
    ((EQ (CAR F) 'AND)
    (DOLIST (I (REST F) T)
      (UNLESS (|#-FEATUREP| I) (RETURN NIL))))
    (T (ERROR "Bad #+/- feature syntax: "S" F))))

(DEFUN SHARP-MACRO (STREAM IGNORE &AUX X Y)
  (SETQ X
    (ASSOC (CHAR-UPCASE (SETQ Y (FUNCALL STREAM :TYI)))
      *SHARP-SIGN-MACROS*))
  (IF X
    (FUNCALL (CDR X) STREAM)
    (ERROR NIL "Undefined \# macro: "C" Y)))

(SET-MACRO-CHARACTER 35. 'SHARP-MACRO)

:: A SAFE WAY TO OPEN FILES
(DEFUN SAFE-FILE-OPEN (&REST X)

```

```

(IF (NOT (SYMBOLP (NTH 2 FRM))) FRM
  '(MACRO (NTH 1. FRM) (NTH 2. FRM)) ; Name and arglist
    (RPLACB (NTH 2. FRM) ; macro's arg
      (PROGN
        ;; pop off first element of arg list
        (SETQ (NTH 2. FRM) (CDR (NTH 2. FRM)))
        (NTHCDR 3. FRM)))))) ; splice in body

;; A simple DEFVAR macro
(DEFMACRO DEFVAR FRM
  '(UNLESS (BOUNDP (CAR FRM))
    (SETQ (CAR FRM) (IF (> (LENGTH FRM) 1.)
      (CADR FRM)
      NIL))))

;; A simple DEFCONSTANT
(DEFMACRO DEFCONSTANT FRM
  '(SETQ (CAR FRM) (CADR FRM)))

;; A simple DEFPARAMETER
(DEFMACRO DEFPARAMETER FRM
  '(SETQ (CAR FRM) (CADR FRM)))

;; The Sharp sign macro character handlers.
(DEFCONSTANT *SHARP-SIGN-MACROS*
  '(
    (39. . ;@
      (LAMBDA (S)
        (LIST 'FUNCTION (READ S NIL NIL T))))
    (40. . ;@
      (LAMBDA (S &AUX X)
        (FUNCALL S :UNTYI 40.)
        (SETQ X (READ S NIL NIL T))
        (APPLY 'VECTOR X)))
    (43. . ;@+
      (LAMBDA (S &AUX (X (READ S NIL NIL T)))
        (VALUES (READ S NIL NIL T) (NOT (|@-FEATUREP| X)))))
    (45. . ;@-
      (LAMBDA (S &AUX (X (READ S NIL NIL T)))
        (VALUES (READ S NIL NIL T) (|@-FEATUREP| X)))))
    (46. . ;@.
      (LAMBDA (S)
        (EVAL (READ S NIL NIL T))))
    (58. . ;@:
      (LAMBDA (S) (COPY-SYMBOL (READ S NIL NIL T))))
    (66. . ;@B
      (LAMBDA (S &AUX (*READ-BASE* 2.))
        (READ S NIL NIL T)))
    (68. . ;@D
      (LAMBDA (S &AUX (*READ-BASE* 10.))
        (READ S NIL NIL T)))
    (70. . ;@O
      (LAMBDA (S &AUX (*READ-BASE* 8.))
        (READ S NIL NIL T)))
    (83. . ;@S
      (LAMBDA (S &AUX (SPEC (READ S NIL NIL T)))
        (EVAL (CONS (INTERN (STRING-APPEND "MAKE-" (CAR SPEC)))
          (DO ((X (CDR SPEC) (CDR X))
              (RESULT NIL (CONS "" (CADR X) (CONS (CAR X) RESULT))))))

```

I.2 THEINIT.LSP

```

;;; ttttheinit.lsp
;;; (C) Copyright Gold Hill Computers, a Division of Apiary, Inc. 1984

;;; GOLDEN Common Lisp initialization file

(SETF *MONITOR-IS-COLOR*      NIL
      *SINGLE-DISKETTE-DRIVE-SYSTEM*  NIL
      *NUMBER-OF-DRIVES*        2
      *BREAK-EVENT* '(LAMBDA (&AUX INPUT-EDITOR)(BREAK)))

(SETQ *BQ-LEVEL* 0
      *BQ-COMMA* (GENSYM)
      *BQ-COMMA-AT* (GENSYM))

;; the famous back-quote macro
(DEFUN BQ (X &AUX)
  (COND ((NULL X) NIL)
        ((ATOM X) (LIST 'QUOTE X))
        ((EQ (CAR X) *BQ-COMMA*)
         (CADR X))
        (T
         (DO* ((HOOK (NCONS 'LIST))
               (Y HOOK (SNOC Y (BQ (CAR X)))))
               (X X (CDR X)))
         ((NULL X) HOOK)
         (COND ((ATOM X)
                  (RPLACA HOOK 'LIST*)
                  (SNOC Y (LIST 'QUOTE X))
                  (RETURN HOOK))
                ((EQ (CAR X) *BQ-COMMA-AT*)
                 (RPLACA HOOK 'LIST*)
                 (RPLACD Y (NCONS (LIST 'APPEND
                                         (CADR X)
                                         (BQ (CDDR X)))))
                 (RETURN HOOK))
                ((EQ (CAR X) *BQ-COMMA*)
                 (RPLACA HOOK 'LIST*)
                 (RPLACD Y (NCONS (CADR X)))
                 (RETURN HOOK)))))))

(DEFUN COMMA-MACRO (STREAM IGNORE &AUX X)
  (WHEN (<= *BQ-LEVEL* 0)
    (ERROR "Comma not inside backquote"))
  (IF (OR (EQ (SETQ X (FUNCALL STREAM :TYI)) 64.) ; is it ""?
          (EQ X 46)) ; or a "."?
      *BQ-COMMA-AT*
      (FUNCALL STREAM :UNTYI X) ; put it back
      (LIST *BQ-COMMA* (READ STREAM NIL NIL T))))

(DEFUN BQ-MACRO (STREAM IGNORE &AUX (*BQ-LEVEL* (1+ *BQ-LEVEL*)))
  (BQ (READ STREAM NIL NIL T)))

(SET-MACRO-CHARACTER 44. 'COMMA-MACRO)
(SET-MACRO-CHARACTER 96. 'BQ-MACRO)

(MACRO DEFMACRO (FRN)

```

I. The LISP Code

I.1 MPROUST.INI

```
:: tttaproust.ini
:: load the lisp init file
(load "b:theinit.lsp")

:: default drive
(setf ddev "b:")

:: default setting for execution output flag
(setf exec-out t)

:: files
(setf efiles '(
  ("examplea.pas" "average.prb")
  ("example1.pas" "average.prb")
  ("example2.pas" "average.prb")
  ("example3.pas" "rainfall.prb")
  ("example4.pas" "rainfall.prb")
  ("example5.pas" "rainfall.prb")
  ("exampler.pas" "rainfall.prb")
))

:: micro-proust code file
(load "b:files\\mproust")
```

Table of Contents

I The LISP Code	0
1.1 MPROUST.INI	0
1.2 THEINIT.LSP	1
1.3 MPROUST.LSP	6
1.4 DEFMAC.LSP	42
1.5 DEFSTRUC.LSP	45
1.6 DRIBBLE.LSP	49
II Sample Pascal Programs	51
II.1 EXAMPLE1.PAS	51
II.2 EXAMPLE2.PAS	52
II.3 EXAMPLE3.PAS	53
II.4 EXAMPLE4.PAS	54
II.5 EXAMPLE5.PAS	55
II.6 EXAMPLEA.PAS	56
II.7 EXAMPLER.PAS	57
III Problem Specifications	58
III.1 AVERAGE.PRB	58
III.2 RAINFALL.PRB	59

```

((equal (cadr context) 'at)
 (aref (ins-matches plan) (- (cadr context) 1)))
(equal (cadr context) 'containing)
 (get-ancestors (list (node-parent (context-node context plan))))
(equal (cadr context) 'above)
 (above (context-node context plan)
        (node-children (node-parent (context-node context plan))))
(equal (cadr context) 'below)
 (below (context-node context plan)
        (node-children (node-parent (context-node context plan))))
(t (get-begin-block (node-children *parse-tree*))))

;; Modify the node-list based on the second element of the context descriptor
;; if it exists.
(defun context-nodeset-2 (context plan node-list)
  (cond
    ((null context) node-list)
    ((equal (cadr context) 'top)
     (list (car (node-children (car node-list)))))
    ((equal (cadr context) 'bottom)
     (last (node-children (car node-list))))
    ((equal (cadr context) 'above)
     (above (context-node context plan) (node-children (car node-list))))
    ((equal (cadr context) 'below)
     (below (context-node context plan) (node-children (car node-list))))
    (t node-list)))

;; Context-nodeset supporting functions
;; .....
;; Context-node returns the list of nodes located in ins-matches [context].
(defun context-node (context plan)
  (car (aref (ins-matches plan) (- (cadr context) 1))))

;; This function returns all of the siblings in ls that occur above the
;; statement represented by node.
(defun above (node ls)
  (cond
    ((null ls) ())
    ((equal (car ls) node) ())
    (t (cons (car ls) (above node (cdr ls))))))

;; This function returns all of the siblings in ls that occur below the
;; statement represented by node.
(defun below (node ls)
  (cond
    ((null ls) ())
    (t (cdr (member node ls)))))

;; This function returns the node of the main begin block of the program
(defun get-begin-block (nodes)
  (cond
    ((null nodes) ())
    ((equal (node-name (car nodes)) 'begin) (list (car nodes)))
    (t (get-begin-block (cdr nodes)))))

;; This function returns a list of all direct ancestors of the nodes in nodes.
;; It is used for the Containing descriptor.
(defun get-ancestors (nodes)

```

```

(cond
  ((null (node-parent (car nodes))) nodes)
  (t (get-ancestors (cons (node-parent (car nodes)) nodes))))

;; Xmatch functions
;; .....
;; This function calls get-candidates to modify the nodeset created by
;; Context-nodeset and call xmatch-frame to set-up and execute a match-stat.
(defun xmatch (plan-line plan address)
  (cond
    ((equal (cadr plan-line) '(*var* *))
     (setf (ins-matches plan) nodeset)
     t)
    (t
     (xmatch-frame (cadr plan-line)
                   (get-candidates (caddr plan-line)
                                   (caddr plan-line)
                                   nodeset)
                   plan address))))

;; This function creates a match frame and calls match-stat for each possible
;; candidate in candidates until one matches or they all don't match. The
;; match frame for each candidate the doesn't match is pushed onto the
;; partials list of the plan. If a candidate is matched, then any bindings
;; accumulated during the matching process are added to the plan bindings.
(defun xmatch-frame (pattern candidates plan address)
  (cond
    ((null candidates)
     (setf (ins-status plan) t)
     ())
    (t (let ((mf (make-match-frame)))
         (setf (match-frame-code mf) (car candidates))
         (cond ((match-stat pattern mf)
                (setf (aref (ins-matches plan) address)
                      (list (car candidates)))
                (cond ((match-frame-bindings mf)
                       (setf (ins-bindings plan)
                             (append (match-frame-bindings mf)
                                     (ins-bindings plan))))
                t)
         (t (push mf (ins-partials plan))
            (xmatch-frame pattern (cdr candidates)
                          plan address))))))

;; Get candidates functions
;; .....
;; Get-candidates first calls modify-nodeset which may add the children of
;; the the context-nodeset to nodes depending on the context descriptor. See
;; the Spec for further details. Get-cand-live filters out nodes that have
;; already been marked or are the wrong type of node.
(defun get-candidates (stat des nodes)
  (get-cand-live (stat-table stat) (modify-nodeset des nodes)))

;; This function returns a list of nodes which are of the same statement type
;; as an element of stats and have not been marked as matched.
(defun get-cand-live (stats nodes)
  (cond

```

```

((null nodes) ()))
((and (member (node-name (car nodes)) stats)
      (null (node-mark (car nodes)))))
  (cons (car nodes) (get-cand-live stats (cdr nodes))))
(t (get-cand-live stats (cdr nodes))))

;; This function returns a list of statements which are similar or identical to
;; stat.
(defun stat-table (stat)
  (or (get 's-table stat) (list stat)))

;; Table of similar statements
(dps s-table
  if (if while)
  while (if while)
  read (read :=)
  := (read :=))

;; This function returns either nodes or nodes and the children of nodes
;; depending on des (the context descriptor).
(defun modify-nodeset (des nodes)
  (cond
    ((and (equal (caddr des) 'at)
          (or (equal (caddr des) 'top)
              (equal (caddr des) 'bottom)))
      nodes)
    ((equal (caddr des) 'containing) nodes)
    (t (add-children nodes))))

;; This function returns nodes and the children of nodes
(defun add-children (nodes)
  (cond
    ((null nodes) ()))
    ((atom nodes) (cons nodes (add-children (node-children nodes))))
    (t (mapcan #'add-children nodes))))

;; Xlabel
;; .....
;; This function set ins-matches of the current plan address to be the
;; current context-nodeset. It also inserts the label and nodeset into the
;; bindings list of the plan so it can be referred to later by other plans.
(defun xlabel (plan-line plan address)
  (setf (aref (ins-matches plan) address) nodeset)
  (push (cons (caddr plan-line) nodeset)
        (ins-bindings plan))
  t)

;; Xrefer
;; .....
;; This function retrieves a list of nodes from the bindings list of analyzed
;; plans. It uses the second element of the plan-line as the label.
(defun xrefer (plan-line plan address)
  (let ((var
        (cond
          ((null (caddr plan-line)))
          (cdr (mapcan #'(lambda (x) (member (car (caddr plan-line)) x))
                      (ins-bindings plan)))))))

```

```

        (t (resolve-var-ref (car (cdadr plan-line))
                             (cadr (cdadr plan-line))))))
      (setf (aref (ins-matches plan) address)
            (cond (var var)
                  (t (get-begin-block (node-children sparse-tree))))))
    t)

;; Match-stat
;; .....
;; This function gets the matching started by calling match-parent
(defun match-stat (pattern m-frame)
  (let ((match-frame m-frame))
    (match-parent pattern (match-frame-code m-frame))
    (cond ((match-frame-errors m-frame) ())
          (t t))))

;; Match-parent calls match-children for the children of the current
;; pattern and then compares the first element of the pattern with the
;; node-name. Note that we must attempt to match the entire pattern even
;; if there is a mismatch, since we have to construct a complete list of
;; mismatches for explain-mismatches. All mismatches are added to
;; match-frame-errors.
(defun match-parent (pattern node)
  (let ((b1 (match-children (cdr pattern) (node-children node))))
    (cond ((not (equal (car pattern) (node-name node)))
           (push (cons (car pattern) (node-name node))
                 (match-frame-errors match-frame))
           nil)
          (t b1))))

;; Match-children tries to match the children. Even if there is a mismatch
;; it continues on to try and find more mismatches. All mismatches are added
;; to match-frame-errors. See the Spec for more details.
(defun match-children (pattern node-list)
  (cond ((and (null pattern) (null node-list)) t)
        ((null pattern)
         (push (cons () (mapcar 'node-code node-list))
               (match-frame-errors match-frame))
         nil)
        ((equal pattern '(*var *)) t)
        ((null node-list)
         (push (cons pattern ())
               (match-frame-errors match-frame))
         nil)
        ((atom (car pattern))
         (let ((b1 (match-children (cdr pattern) (cdr node-list))))
           (cond ((not (equal (car pattern) (node-name (car node-list))))
                  (push (cons (car pattern) (node-name (car node-list)))
                        (match-frame-errors match-frame))
                  nil)
                 (t b1))))
        ((equal (car pattern) '(*var ?))
         (match-children (cdr pattern) (cdr node-list)))
        ((equal (car pattern) '(*var *))
         (let ((cur-bind (copy-tree (match-frame-errors match-frame))))
           (or (match-children (cdr pattern) (cdr node-list))
               (and (setf (match-frame-errors match-frame) cur-bind)
                    t))))))

```

```

        nil)
      (match-children (cdr pattern) node-list)
      (and (setf (match-frame-errors match-frame) cur-bind)
           nil)
      (match-children pattern (cdr node-list))))))
((equal (car pattern) 'evare)
 (let ((binds (check-bindings (cadr pattern)))
       (bool1 (match-children (cdr pattern) (cdr node-list))))
  (cond (binds
        (and (match-children (list binds)
                              (list (car node-list)))
              bool1))
        (t (add-bindings (cadr pattern)
                          (node-code (car node-list)))
            (match-children (cdr pattern) (cdr node-list))))))
(t (and (match-parent (car pattern) (car node-list))
        (match-children (cdr pattern) (cdr node-list)))))

;; Match utilities
;; .....
;; This function searches for var-ref in the bindings of the current plan,
;; and the bindings of the match-frame and returns the binding if found.
(defun check-bindings (var-ref)
  (cond ((equal var-ref '?) '?)
        ((equal var-ref '?*) '?*)
        (t (or (resolve-var var-ref
                              (ins-bindings plan))
                (resolve-var var-ref
                              (match-frame-bindings match-frame))))))

;; Add the bindings of var-name to var-ref to the current match-frame
(defun add-bindings (var-ref var-name)
  (push (cons var-ref var-name) (match-frame-bindings match-frame)))

;; Find the pair (var . bind) in the s-list list and return bind.
(defun resolve-var (var list)
  (cdr (assoc var list)))

;; Load the Instantiate group
;; Contains the following functions and their supporting functions:
;;   Instantiate, Instantiate-plan, Process-bindings-list, and
;;   resolve-var-ref.
;; Instantiate Sub-group of Micro-Proust    INS1.lsp
;; Written by Bret Wallach                  10/24/84
;;
;; Define the plan instantiation structure
(defstruct ins
  (name ())
  (bindings ())
  matches
  (partials ())
  (address 0)
  (states ())
  (bugs ()))

```

```

::Instantiate
.....
:: This function returns a list of plan instantiations for the plan
:: 'instances of the goal. The bindings list of the goal is first processed
:: by process-bindings-list. Instantiate-plan is called to create a plan
:: instantiation structure for each plan.
(defun instantiate (goal)
  (let ((bindings-list (process-bindings-list goal)))
    (cond (exec-out
           (format goal-window "%$Bindings: ")
           (mapcar 'lins bindings-list)))
      (setf *active-plans* (mapcar 'instantiate-plan
                                   (get (car goal) 'instances))))))

:: This function returns either the the binding for variable bindings or the
:: node-line for label bindings. This is used to display a meaningful
:: bindings list in the goal-window.
(defun lins (bin)
  (cond ((atom (cdr bin))
         (format goal-window "%s ==> %s" (car bin) (cdr bin)))
        ((node-p (cdr bin))
         (format goal-window "%s ==> line %s" (car bin) (node-line (cdr bin))))
        (t
         (format goal-window "%s ==> %s" (car bin) (cdr bin)))))

:: Process the bindings
.....
:: This function processes the bindings list of goal and returns the bound
:: bindings by calling the function process-bindings-l.
(defun process-bindings-list (goal)
  (process-bindings-l (car goal) (cdr goal)))

:: This function calls process-list for the binding specification of each
:: member of the bindings list. It conses them all together and returns
:: the processed bindings list.
(defun process-bindings-l (goal-name goal-list)
  (cond
    ((null goal-list) ())
    (t
     (cons (cons (car goal-list)
                 (process-list goal-name (cdr goal-list)))
           (process-bindings-l goal-name (cdr goal-list))))))

:: This function substitutes the real value in for each (*var* x y) triple
:: in the list var-val.
(defun process-list (goal-name var-val)
  (cond
    ((null var-val) ())
    ((atom var-val) var-val)
    ((equal (car var-val) '*var*)
     (resolve-var-ref (cadr var-val) (caddr var-val)))
    (t(cons (process-list goal-name (car var-val))
              (process-list goal-name (cdr var-val))))))

:: This function returns the value of var1 which was bound in goal.
(defun resolve-var-ref (var1 goal)

```

```

(let ((inst (resolve-var goal *analyzed-goals*)))
  (cond ((null inst) ())
        (t (resolve-var var1 (ins-bindings inst))))))

;; This function returns a plan instantiation structure for plan
(defun instantiate-plan (plan)
  (make-ins name plan
            bindings bindings-list
            matches (make-array (length (get plan 'template))
                               :initial-element ())))

;; Utility to display a plan instantiation structure
(defun list-ins (plan)
  (print (ins-name plan))
  (print (ins-bindings plan))
  (print (ins-matches plan))
  (print (ins-partials plan))
  (print (ins-address plan))
  (print (ins-status plan))
  (ins-bugs plan))

;; Load the goal knowledge base
;; Goal Knowledge Base for Micro-Proust      GOALS.kbs
;; Taken directly from Micro-Proust Design Spec 10/23/84
;;
;; INSTANCES refer to possible plan names.
;; NAME-PHRASE describes what the goal is doing. This is used during
;; bug reporting.
;;
(DPS SENTINEL-CONTROLLED-LOOP
  INSTANCES (SENTINEL-READ-PROCESS-WHILE
              SENTINEL-PROCESS-READ-WHILE
              SENTINEL-READ-PROCESS-REPEAT)
  NAME-PHRASE "input processing")

(DPS LOOP-INPUT-VALIDATION
  INSTANCES (BAD-INPUT-SKIP-GUARD
              BAD-INPUT-LOOP-GUARD)
  NAME-PHRASE "input validation")

(DPS AVERAGE
  INSTANCES (AVERAGE-PLAN)
  NAME-PHRASE "average")

(DPS OUTPUT
  INSTANCES (WRITELN-PLAN)
  NAME-PHRASE "output")

(DPS COUNT
  INSTANCES (COUNTER-PLAN)
  NAME-PHRASE "count")

(DPS SUM
  INSTANCES (RUNNING-TOTAL)
  NAME-PHRASE "sum")

(DPS GUARDED-COUNT

```

```

INSTANCES (GUARDED-COUNTER-PLAN)
NAME-PHRASE "count")

(DPS MAXIMUM
INSTANCES (MAXIMUM-PLAN)
NAME-PHRASE "maximum")

(DPS GUARD-EXCEPTION
INSTANCES (GUARD-EXCEPTION-PLAN)
NAME-PHRASE "boundary condition check")

:: Load the plans
:: Plan knowledge base for Micro-Proust   PLANS.kbs
:: Taken from Micro-Proust design spec with some modification.
::
:: A fourth element has been added to some lines.  If this element is T,
:: then the node which the statement matches is not marked as being matched.
:: If it is nil or non-existent, then the matched node is marked.  This
:: is only valid for MATCH instructions.
::
:: Some of the context descriptor line numbers were in error in the Spec
:: and are fixed in this file.
::
(DPS SENTINEL-PROCESS-READ-WHILE
TEMPLATE
@((MATCH (WHILE (<> (*VAR* NEW) (*VAR* STOP)) (*VAR* ?)) ()))
(LABEL MAINLOOP: ((AT 1)))
(MATCH (BEGIN (*VAR* *)) ((AT 1)))
(MATCH (READ (*VAR* NEW)) ((AT 3) (BOTTOM)))
(LABEL NEXT: ((AT 4)))
(LABEL PROCESS: ((AT 3) (ABOVE 5)))
(MATCH (READ (*VAR* NEW)) ((ABOVE 1)))
(LABEL INIT: ((AT 7))))

(DPS SENTINEL-READ-PROCESS-WHILE
TEMPLATE
@((MATCH (WHILE (<> (*VAR* NEW) (*VAR* STOP)) (*VAR* ?)) ()))
(LABEL MAINLOOP: ((AT 1)))
(MATCH (BEGIN (*VAR* *)) ((AT 1)))
(MATCH (READ (*VAR* NEW)) ((AT 3) (TOP)))
(LABEL NEXT: ((AT 4)))
(MATCH (IF (<> (*VAR* NEW) (*VAR* STOP)) (*VAR* ?)) ((AT 3) (BELOW 4)))
(LABEL PROCESS: ((AT 6)))
(LABEL INTERNAL-GUARD: ((AT 6)))
(MATCH (:= (*VAR* NEW) (*VAR* SEEDVAL)) ((ABOVE 1)))
(LABEL INIT: ((AT 9))))

(DPS SENTINEL-READ-PROCESS-REPEAT
TEMPLATE
@((MATCH (REPEAT (*VAR* ?) (= (*VAR* NEW) (*VAR* STOP))) ()))
(LABEL MAINLOOP: ((AT 1)))
(MATCH (STATEMENT-LIST (*VAR* *)) ((AT 1)))
(MATCH (READ (*VAR* NEW)) ((AT 3)))
(LABEL NEXT: ((AT 4)))
(MATCH (IF (<> (*VAR* NEW) (*VAR* STOP)) (*VAR* ?)) ((AT 3) (BELOW 4)))
(LABEL PROCESS: ((AT 6))))

```

```
;; (LABEL PROCESS:) was changed from ((AT 2)) to ((AT 3)) so as to
;; avoid a conflict between TEST: and PROCESS: during the explain
;; missing rules functions.
```

```
(DPS BAD-INPUT-SKIP-GUARD
```

```
  TEMPLATE
```

```
  #((REFER (*VAR* PROCESS: SENTINEL-CONTROLLED-LOOP) ()))
    (MATCH (IF (*VAR* PRED) (*VAR* ?) (*VAR* ?)) ((AT 1)))
    (LABEL TEST: ((AT 2)))
    (MATCH (WRITELN (*VAR* *)) ((AT 2)))
    (LABEL PROCESS: ((AT 3))))
```

```
(DPS BAD-INPUT-LOOP-GUARD
```

```
  TEMPLATE
```

```
  #((REFER (*VAR* PROCESS: SENTINEL-CONTROLLED-LOOP) ()))
    (MATCH (WHILE (*VAR* PRED) (*VAR* ?)) ((TOP) (AT 1)))
    (LABEL TEST: ((AT 2)))
    (MATCH (BEGIN (*VAR* *)) ((AT 2)))
    (MATCH (WRITELN (*VAR* *)) ((AT 4)))
    (MATCH (READ (*VAR* NEW)) ((AT 4) (BELOW 5)))
    (LABEL INPUT: ((AT 6)))
    (MATCH (IF (<> (*VAR* NEW) (*VAR* STOP)) (*VAR* *)) ((AT 1)(BELOW 2)))
    (LABEL PROCESS: ((AT 8))))
```

```
(DPS AVERAGE-PLAN
```

```
  TEMPLATE
```

```
  #((REFER (*VAR* MAINLOOP: SENTINEL-CONTROLLED-LOOP) ()))
    (MATCH (:= (*VAR* AVG) (/ (*VAR* SUM) (*VAR* COUNT))) ((BELOW 1)))
    (LABEL UPDATE: ((AT 2))))
```

```
(DPS WRITELN-PLAN
```

```
  TEMPLATE
```

```
  #((MATCH (WRITELN (*VAR* *) (*VAR* VAL) (*VAR* *)) ()))
    (LABEL OUTPUT: ((AT 1))))
```

```
(DPS COUNTER-PLAN
```

```
  TEMPLATE
```

```
  #((REFER (*VAR* PROCESS: SENTINEL-CONTROLLED-LOOP) ()))
    (MATCH (:= (*VAR* COUNT) (+ (*VAR* COUNT) 1)) ((AT 1)))
    (LABEL UPDATE: ((AT 2)))
    (REFER (*VAR* MAINLOOP: SENTINEL-CONTROLLED-LOOP) ()))
    (MATCH (:= (*VAR* COUNT) 0) ((ABOVE 4)))
    (LABEL INIT: ((AT 5))))
```

```
(DPS GUARDED-COUNTER-PLAN
```

```
  TEMPLATE
```

```
  #((REFER (*VAR* PROCESS: SENTINEL-CONTROLLED-LOOP) ()))
    (MATCH (IF (*VAR* PRED) (*VAR* ?)) ((AT 1)))
    (LABEL GUARD: ((AT 2)))
    (MATCH (:= (*VAR* COUNT) (+ (*VAR* COUNT) 1)) ((AT 3)))
    (LABEL UPDATE: ((AT 4)))
    (REFER (*VAR* MAINLOOP: SENTINEL-CONTROLLED-LOOP) ()))
    (MATCH (:= (*VAR* COUNT) 0) ((ABOVE 6)))
    (LABEL INIT: ((AT 7))))
```

```
(DPS RUNNING-TOTAL
```

```
  TEMPLATE
```

```
  #((REFER (*VAR* PROCESS: SENTINEL-CONTROLLED-LOOP) ()))
    (MATCH (:= (*VAR* TOTAL) (+ (*VAR* TOTAL) (*VAR* NEW))) ((AT 1)))
    (LABEL UPDATE: ((AT 2)))
```

```
(REFER (*VAR* MAINLOOP: SENTINEL-CONTROLLED-LOOP) ())
(MATCH (:= (*VAR* TOTAL) 0) ((ABOVE 4)))
(LABEL INIT: ((AT 5))))
```

```
:: This plan is messed up, but works sufficiently for now.
:: Both LABEL statements have been added for bug rule purposes.
:: Problems are:
:: 1. It will match IF count = 0 then ave = sum / count else writeln ('a');
::    This is clearly backwards.
::
:: 2. Usually when the IF statement is missing, the WRITELN statement is
::    missing also. Therefore, bug rules explain the missing IF, adding
::    something to *bug-report*. Then the WRITELN is also not matched
::    and more bug rules fire which should generate another error message.
::    Two messages for one error is two much.
```

```
(DPS GUARD-EXCEPTION-PLAN
```

```
  TEMPLATE
```

```
  @((REFER (*VAR* CODE) ()))
  (MATCH (IF (*VAR* PRED) (*VAR* ?) (*VAR* ?)) ((CONTAINING 1)) T)
  (LABEL EXCEPTION: ((AT 2)))
  (MATCH (WRITELN (*VAR* *)) ((AT 3)) T)
  (LABEL GUARD-ACTION: ((AT 4))))
```

```
(DPS MAXIMUM-PLAN
```

```
  TEMPLATE
```

```
  @((REFER (*VAR* PROCESS: SENTINEL-CONTROLLED-LOOP) ()))
  (MATCH (IF (> (*VAR* NEW) (*VAR* MAX)) (*VAR* ?)) ((AT 1)))
  (MATCH (:= (*VAR* MAX) (*VAR* NEW)) ((AT 2)))
  (LABEL UPDATE: ((AT 3)))
  (REFER (*VAR* MAINLOOP: SENTINEL-CONTROLLED-LOOP) ())
  (MATCH (:= (*VAR* MAX) (*VAR* MINVAL)) ((ABOVE 5))))
```

```
:: Load the Parse group
:: Contains the following functions and their supporting functions:
::   Parse.
:: Lex-initialize and Lex-gottok are not used. See parse.lsp for further
:: details.
:: Parse sub-group of MPROUST   PARSE LSP
:: Written by Bret Wallach     10/24/84
::
:: Only one week was allocated for the Parser and Lexer and there
:: were no compiler compilers or other such tools available. As
:: a result this is one of the more slow, inefficient, and kludgy
:: parser/lexers in existence today. Well, enough for excuses.
```

```
:: This function uses read-tokens to get a list of tokens and their line
:: numbers from the program file. It then makes an array out the list
:: and stuffs it in the global variable sentence. The global variable
:: s-ptr points to the current token in sentence. The function p-match
:: is then called to build the parse tree from the tokens.
```

```
(defun parse (filename)
```

```
  (let* ((*line-numbers* 1)
```

```
    (x (read-tokens filename))
```

```
    (sentence (make-array (+ (length x) 2) :fill-pointer 0))
```

```
    (s-ptr 0))
```

```

(mapcar #'(lambda (z) (vector-push z sentence)) z)
(vector-push '(nil 0) sentence)
(vector-push '(nil 0) sentence)
(setf *parse-tree* (car (p-match *program))))

;; The lexer
;;.....
;; Unfortunately for us, GCLISP version 0.98 has a bug making it impossible
;; to restore the read table after altering it to define various characters
;; such as + to be macro characters. As a result, I had to write my own
;; read function.
;;
;; Read-tokens opens the program file and calls read-tokens-1 to work on it.
(defun read-tokens (filename)
  (with-open-file
    (ous filename :direction :input :element-type 'string-char)
    (read-tokens-1)))

;; Read-tokens-1 creates a list of conses (symbol . *line-number*) where
;; symbol is gotten by calling read-pas and *line-number* is the current
;; line in the file
(defun read-tokens-1 ()
  (do* ((syb (read-pas ous) ()) (read-pas ous) ())
        (cs (list (cons syb *line-number*)))
        (append cs (list (cons syb *line-number*)))))
  ((equal syb ()) cs)))

;; Read-pas reads characters until it hits a macro-character. It then calls
;; Read-pas1 to process it.
(defun read-pas (stream eof-f eof-r)
  (setf (fill-pointer read-vector) 0)
  (do ((tchar (read-char stream eof-f eof-r))
        (read-char stream eof-f eof-r)))
    ((assoc tchar break-alist) (read-pas1 tchar))
    (vector-push tchar read-vector)))

;; Read-pas1 either returns the symbol just read or processes the macro char.
(defun read-pas1 (tchar)
  (cond ((equal (fill-pointer read-vector) 0)
        (funcall (cdr (assoc tchar break-alist)) tchar))
        (t
         (unread-char tchar stream)
         (read-from-string read-vector))))

;; This array is used to build the symbols
(setf read-vector (make-array 256 :element-type 'string-char :fill-pointer 0))

;; This a-list contains all the special characters
(setf break-alist '((#\ . ign) (13 . ign) (9 . ign) (12 . ign)
  (10 . lp) (#\+ . ren) (nil . endfun) (#\{ . con)
  (#\, . ren) (#\ ( . par) (#\) . ren)
  (#\~ . ren) (#\: . ren) (#\' . dq)
  (#\% . ren) (#\ / . ren) (#\= . ren)
  (#\< . ren) (#\> . ren) (#\ . ren)))

;; Character macros
;;.....
;; IGN ignores the character. It is used for spaces, tabs, etc.

```

```

(defun iga (char)
  (do ((x (read-char stream eof-f eof-r)
          (read-char stream eof-f eof-r)))
      ((not (equal x char))
       (progn (unread-char x stream)
              (read-pas stream eof-f eof-r)))))

;; Lp is called for a line-feed. It increments *line-number*
(defun lp (char) (incf *line-number*) (read-pas stream eof-f eof-r))

;; Rem returns the read character
(defun rem (char) (vector-push #\\ read-vector)
                  (vector-push char read-vector)
                  (read-from-string read-vector))

;; Par is the left-parenthesis macro. If a * follows immediately then
;; ignore everything until *) since it is a comment. Otherwise return '\\(
(defun par (char)
  (let ((n-char (read-char stream eof-f eof-r)))
    (cond ((equal n-char #\\*)
           (do ((ch #\\( x)
                 (x (read-char stream eof-f eof-r)
                    (read-char stream eof-f eof-r)))
               ((or (and (equal n-char #\\*) (equal x #\\))
                    (equal x eof-r))
                (read-pas stream eof-f eof-r))))
          (t
           (unread-char n-char stream)
           (vector-push #\\ read-vector)
           (vector-push #\\( read-vector)
           (read-from-string read-vector)))))

;; Com is the { macro. Ignore the comment
(defun com (char)
  (do ((x (read-char stream eof-f eof-r) (read-char stream eof-f eof-r))
      ((or (equal x #\\}) (equal x eof-r)) (read-pas stream eof-f eof-r)))

;; Dq is the single-quote macro. It turns the ' into " and returns a string.
(defun dq (char)
  (vector-push #\\" read-vector)
  (do ((x (read-char stream eof-f eof-r) (read-char stream eof-f eof-r))
      ((or (equal x #\\') (equal x eof-r))
       (progn (vector-push #\\" read-vector)
              (decf (fill-pointer read-vector))
              (vector-push #\\" read-vector)
              (read-from-string read-vector)))
      (vector-push x read-vector)))

;; Endfun returns () signifying the end of the file.
(defun endfun (char) eof-r)

;; Parse functions
.....
;; Define a parse-tree node
(defstruct node
  (name ())
  (parent ())
  (children ())
  (line 0)
  (mark ()))

```

```

;; Debug Utility to list the node information
(defun list-node (x)
  (print (node-name x))
  (print (node-parent x))
  (print (node-children x))
  (print (node-line x))
  (node-mark x))

;; Debug Utility to get the parenthesized PASCAL starting at parse node x.
(defun node-code (x)
  (cond ((null x) ())
        ((null (node-children x)) (node-name x))
        (t (cons (node-name x) (mapcar 'node-code (node-children x))))))

;; Parse matching
.....
;; This is the starting parse matcher. It uses the global variable sentence
;; and matches it to the EBNF specified by g-node. If g-node has no 'template
;; property and g-node is equal to the current token in sentence, then create
;; a new node with name g-node and return. If g-node has a template then
;; call p-match-1 with the template and try to match the template against
;; the sentence. If the match succeeds p-match-1 will return a list of
;; children nodes. If the 'parent property of g-node is nil, return that
;; list of children, otherwise create a new node with those children.
;; Otherwise return nil.
(defun p-match (g-node)
  (let ((tplt (get g-node 'template))
        (par-name (get g-node 'parent))
        (c-node ())
        (line 1)
        (par ()))
    (cond ((null plit)
           (cond ((equal g-node (car (aref sentence s-ptr)))
                  (incf s-ptr)
                  (list (make-node name g-node
                                   line (cdr (aref sentence s-ptr)))))
                 (t ())))
          (t (and (setf line (cdr (aref sentence s-ptr))
                        c-node (p-match-1 plit))
                  (cond ((and par-name (not (eq c-node t)))
                         (setq par (make-node name (get g-node 'parent)
                                                children c-node
                                                line line))
                         (mapcar #'(lambda (x)
                                     (setf (node-parent x) par))
                                c-node)
                         (list par))
                        (par-name
                         (list (make-node name (get g-node 'parent)
                                                line line)))
                        (t c-node)))))))

;; This function tries to match each option of a template with the sentence
;; tokens pointed to by s-ptr. If any of them match, return the children
;; created in the process, else return nil.
(defun p-match-1 (tplt &aux chil bool1 s-ptr1)
  (setq bool1 (null (car tplt))) chil ()))

```

```

(setf s-ptr1 s-ptr)
(doe ((x taplt (cdr x))
      (y (car x) (car x)))
      ((or (null x) (null y) bool1) (or (and bool1 (or chil bool1))
                                         (and (setf s-ptr s-ptr1) nil)))
      (setf s-ptr s-ptr1)
      (multiple-value-setq (chil bool1) (p-match-2 y))))

;; Option matcher
.....
;; This function tries to match a parse template against the current sentence
;; tokens. If it succeeds it returns two values. The first is a list of
;; children created while matching things and the second is T. If it fails
;; the first value is undefined and the second value is nil.
(defun p-match-2 (tap &aux t-node p-node fbool)
  (setf p-node () fbool t)
  (doe
    ((x tap (cdr x))
     (y (car x) (car x)))
    ((or (null x) (null fbool)) (values p-node fbool))
    (cond ((null y) ())
          ((atom y)
           (cond ((setf t-node (p-match y))
                  (setf p-node (append p-node (filt-t t-node))))
                 (t (setf fbool nil))))
           ((equal (car y) '*opt*)
            (setf p-node (append p-node (filt-t (p-match (cadr y))))))
           ((equal (car y) '*ignore*)
            (setf fbool (not (null (p-match (cadr y))))))
           ((and (equal (car y) '*func*)
                  (funcall (cadr y) (car (aref sentence s-ptr))))
            (setf p-node (append p-node
                                (list (make-node name (car (aref sentence s-ptr))
                                                line (cdr (aref sentence s-ptr))))))
           (incf s-ptr))
          ((and (equal (car y) '*para*)
                 (setf t-node (p-match (cadr y))))
           (multiple-value-bind (q-nodes bool) (p-match-2 (cdr x))
             (cond (bool
                    (setf (node-children (car t-node))
                          (append p-node q-nodes))
                    (setf x nil)
                    (setf p-node t-node)
                    (t (setf fbool nil))))
             ((equal (car y) '*func*) (setf fbool nil))
             ((equal (car y) '*mule*)
              (or (and (setf t-node (p-match-mul (cdr y)))
                       (setf p-node (append p-node t-node)))
                  (setf fbool nil)))
             ((equal (car y) '*null*)
              (or (and (setf t-node (p-match-null (cdr y)))
                       (setf p-node (append p-node t-node)))
                  (setf fbool nil)))
             (t (error "PARSE: Bad BNF option."))))))

;; Option matcher Utilities
.....
;; This function filters out all T's which are introduced by matches that

```

AD-A157 505

MICRO-PROUST(U) YALE UNIV NEW HAVEN CT DEPT OF COMPUTER 2/2
SCIENCE W L JOHNSON ET AL. JUN 85 YALEU/CSD/RR-402
N00014-82-K-0714

UNCLASSIFIED

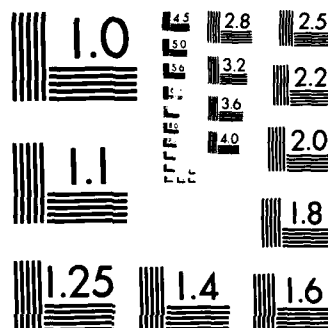
F/G 9/2

NL

END

FORMED

DTIC



MICROCOPY RESOLUTION TEST CHART
 NBS-1963-A

```

;; don't create any children.
(defun filt-t (l) (if (eq l t) nil l))

;; This function is called by p-match-2 to take care of the *null* template
;; option. The algorithm used is explained in PARSE.DAT.
(defun p-match-null (tap &aux p q)
  (cond ((null (setf p (p-match (car tap)))) (values () ()))
        (t (do ()
                  ((or (>= s-ptr (length sentence))
                      (null (member (car (aref sentence s-ptr)) (cadr tap))))
                    (values p t))
                 (incf s-ptr)
                 (setf p (list (make-node name (car (aref sentence (1- s-ptr)))
                                         children (setf q (append p (filt-t
                                                                    (p-match (car tap))))
                                                                    line (cdr (aref sentence (1- s-ptr))))))
                           (mapc #'(lambda (x) (setf (node-parent x) (car p))) q)))))))

;; This function is called by p-match-2 to take care of the *null* template
;; option. The algorithm used is explained in PARSE.DAT.
(defun p-match-null (tap &aux p q)
  (cond ((null (setf p (p-match (car tap)))) (values () ()))
        (t (do ()
                  ((or (and (cadr tap) (incf s-ptr) nil)
                      (>= s-ptr (length sentence))
                      (and (cadr tap)
                          (null (member (car (aref sentence (1- s-ptr))
                                          (cadr tap))))
                      (null (setf q (p-match (car tap))))))
                    (progn (and (cadr tap) (decf s-ptr)) (values p t)))
                 (setf p (append p (filt-t q)))))))

;; Load the parenthesized BNF for the subset of PASCAL that is needed.
;; Written by Bret Wallach          10/23/84
;;
;; This file contains the information necessary to parse and build
;; a parse-tree for the subset of PASCAL that Micro-Proust is using.
;;
;; Only one week was allocated for programming the parser. As a result
;; there certainly are bugs in my PASCAL language definition.
;;
;; Each time a parse var completely matches, a parse-tree node is
;; built using the 'parent property as the name. If the 'parent
;; property is nil, then no new nodes are created and the children
;; nodes are passed along instead. See PARSE.LSP for additional
;; details.
;;
;; The following options are allowed for a match.
;;
;; 1. *ignore* means that the object must match, but don't create
;;    a child node for it.
;;
;; 2. *opt* means try to match the object, but if it doesn't match,
;;    skip it.
;;
;; 3. *func* means apply the following lambda expression to the

```

```

::      current token and if T is returned, consider it a match
::      using the next token as the name of a new child node.

4      *opar* means that this item must match and then will be used
::      as the node-name of the parent node or all the rest of the
::      matched objects

5      *enul* is used to build the expression sub-trees. This is how
::      it is used
      Step 1. Try to match the first argument. If it matches
               create node x for it and go to Step 2, otherwise quit.
      Step 2. Try to match any of the elements of the second
               argument and then the first argument again. If they
               match create nodes y and z for both matches
               respectively. Make y the parent node with x as the
               left child and z as the right child. Set x = y. Go to
               Step 2. Else return x.

6      *enul1* is used to simplify right recursive rules. For example, the
::      following two templates for *en* with nil parent properties would
::      build the same parse sub-tree.

```

```
(((*en \, *en) (*en))) ; *en ::= *en "," *en | *en
```

```
(((*enul1 *en (\,))) ; *en ::= *en {"," *en}
```

The second argument is the item to be repeated, and the third argument is a list of possible separators.

Notes for writing new rules.

```

::      Do not use any left recursive rules. These will bomb the program.
::      If a rule has a right recursive option, put that option first.
::      Always create the rules so they match as much as possible.
::      *opt* is like [] in EBNF
::      *enl* is like {} in EBNF
::      *enul1* is also like {} in EBNF but precedence is not considered.

```

```

:: *program ::= "PROGRAM" *id [*prog-header] [*decls] *main-block *end
(dps  *program
      template
      ((((*ignore* PROGRAM) *id (*opt* *prog-header)
        (*opt* *decls) *main-block (*ignore* *end)))
       parent PROGRAM)

```

```

:: *main-block ::= "BEGIN" *statements | "BEGIN"
(dps  *main-block
      template
      ((((*ignore* BEGIN) *statements)
        ((*ignore* BEGIN)))
       parent BEGIN)

```

```

:: *end ::= "END;" | "END."
(dps  *end
      template
      ((((*ignore* END) (*ignore* \;)) ((*ignore* END.)))
       parent nil)

```

```

:: *prog-header ::= "(" *ph-list ")" ";"
(dps  *prog-header
      template
      (((*ignore* \() *ph-list (*ignore* \)) (*ignore* \;)))
      parent PROG-HEADER)

:: *ph-list ::= *id {" " *id}
(dps  *ph-list
      template
      (((*null* *id (\.))))
      parent nil)

:: *decls ::= *const-decls *var-decls | *const-decls | *var-decls
(dps  *decls
      template
      ((*const-decls *var-decls)(*const-decls)(*var-decls))
      parent DECLS)

:: *const-decls ::= "CONST" *const-list
(dps  *const-decls
      template
      ((CONST *const-list))
      parent CONST-DECLS)

:: *const-list ::= *const {"const"}
(dps  *const-list
      template
      (((*null* *const ())))
      parent nil)

:: *const ::= *id "=" *expr ";"
(dps  *const
      template
      ((*id (*ignore* =) *expr (*ignore* \;)))
      parent CONST)

:: *bool-expr ::= *expr *bool-op *expr
(dps  *bool-expr
      template
      ((*expr (*par* *bool-op) *expr))
      parent nil)

:: *bool-op ::= *bool-ne | *bool-ge | *bool-le | ...
(dps  *bool-op
      template
      ((*bool-ne)(*bool-ge)(*bool-le)
      (*bool-eq)(*bool-lt)(*bool-gt))
      parent nil)

:: *bool-eq ::= "="
(dps  *bool-eq
      template
      (((*ignore* \=)))
      parent \=)

:: *bool-lt ::= "<"
(dps  *bool-lt
      template
      (((*ignore* \<)))

```

```

parent \<)

:: *bool-gt ::= ">"
(dps  *bool-gt
  template
  (((*ignore* \>)))
  parent \>)

:: *bool-ne ::= "<>"
(dps  *bool-ne
  template
  (((*ignore* \<) (*ignore* \>)))
  parent \<\>)

:: *bool-ge ::= ">="
(dps  *bool-ge
  template
  (((*ignore* \>) (*ignore* \=)))
  parent \>\=)

:: *bool-le ::= "<="
(dps  *bool-le
  template
  (((*ignore* \<) (*ignore* \=)))
  parent \<\=)

:: *expr ::= *term { "+" *term | "-" *term | "OR" *term }
(dps  *expr
  template
  (((*rule* *term (\+ \- OR))))
  parent ())

:: *term ::= *fact { "*" *fact | "/" *fact | "DIV" *fact
::          | "MOD" *fact | "AND" *fact }
(dps  *term
  template
  (((*rule* *fact (\* \/ DIV MOD AND))))
  parent ())

:: *fact ::= "(" *expr ")" | *id-or-num | *u-minus-fact
::          | *u-plus-fact | *not-fact
(dps  *fact
  template
  (((*ignore* \() *expr (*ignore* \)))
  (*id-or-num)(*u-minus-fact)(*u-plus-fact)(*not-fact))
  parent ())

:: *u-minus-fact ::= "-" *fact
(dps  *u-minus-fact
  template
  (((*ignore* \-) *fact))
  parent \-)

:: *u-plus-fact ::= "+" *fact
(dps  *u-plus-fact
  template
  (((*ignore* \+) *fact))
  parent nil)

```

```

:: *not-fact ::= *NOT* *fact
(dps  *not-fact
      template
      (((*ignore* NOT) *fact)))
      parent NOT)

:: *expr-list ::= *expr-or-string {*,* *expr-or-string}
(dps  *expr-list
      template
      (((*null* *expr-or-string ( \.))))
      parent nil)

:: *expr-or-string ::= *expr [*w-format] | string
:: where string is predefined
(dps  *expr-or-string
      template
      ((*expr (*opt* *w-format)) ((*func* (lambda (var) (stringp var))))))
      parent nil)

:: *w-format ::= ":" *int ":" *int | ":" *int
(dps  *w-format
      template
      (((*ignore* \:) (*ignore* *int) (*ignore* \:) (*ignore* *int))
       ((*ignore* \:) (*ignore* *int))))
      parent nil)

:: *int is predefined
(dps  *int
      template
      (((*func* (lambda (var) (integerp var))))))
      parent nil)

:: *var-decls ::= *VAR* *var-list
(dps  *var-decls
      template
      (((*ignore* VAR) *var-list))
      parent VAR-DECLS)

:: *var-list ::= *var {*var}
(dps  *var-list
      template
      (((*null* *var ())))
      parent nil)

:: *var ::= *id-list ":" *var-type ":"
(dps  *var
      template
      ((*id-list (*ignore* \:) *var-type (*ignore* \:)))
      parent VAR)

:: *var-type ::= *REAL* | *INTEGER*
(dps  *var-type
      template
      ((REAL)(INTEGER))
      parent nil)

:: *begin-block ::= *END* | *statements* *END*
(dps  *begin-block
      template

```

```

      (((*ignore* END))
      (*statements (*ignore* END)))
    parent BEGIN)

;; *statements ::= *statement {"*" *statement}
(dps  *statements
      template
      (((*null* *statement (\:)))
      parent nil)

;; *semi ::= ";"
(dps  *semi
      template
      (((*ignore* \:)))
      parent nil)

;; *statement ::= *ass-statement | "IF" *if-statement | *begin-block ...
(dps  *statement
      template
      ((*ass-statement) ((*ignore* IF) *if-statement)
      ((*ignore* BEGIN) *begin-block)
      ((*ignore* WHILE) *while-statement)
      ((*ignore* REPEAT) *repeat-statement)
      ((*ignore* WRITELN) *writeln-statement)
      ((*ignore* WRITE) *write-statement)
      ((*ignore* READ) *read-statement)
      ((*ignore* READLN) *readln-statement)
      ((*ignore* CASE) *case-statement) (*null-statement))
      parent nil)

;; *null-statement ::= ""
(dps  *null-statement
      template
      (())
      parent nil)

;; *ass-statement ::= *id "=" *expr
(dps  *ass-statement
      template
      ((*id (*ignore* \:) (*ignore* \=) *expr))
      parent :=)

;; *while-statement ::= *bool-expr "DO" *statement
(dps  *while-statement
      template
      ((*bool-expr (*ignore* DO) *statement))
      parent WHILE)

;; *repeat-statement ::= *repeat-list "UNTIL" *bool-expr
(dps  *repeat-statement
      template
      ((*repeat-list (*ignore* UNTIL) *bool-expr))
      parent REPEAT)

;; *repeat-list ::= *statements
(dps  *repeat-list
      template
      ((*statements))
      parent STATEMENT-LIST)

```

```

:: *write-statement ::= *write-list
(dps  *write-statement
  template
  (((*opt* *write-list)))
  parent WRITE)

:: *writeln-statement ::= *write-list
(dps  *writeln-statement
  template
  (((*opt* *write-list)))
  parent WRITELN)

:: *write-list ::= "(" *expr-list ")"
(dps  *write-list
  template
  (((*ignore* \( *expr-list (*ignore* \))))
  parent nil)

:: *read-statement ::= *read-list
(dps  *read-statement
  template
  (((*opt* *read-list)))
  parent READ)

:: *readln-statement ::= *read-list
(dps  *readln-statement
  template
  (((*opt* *read-list)))
  parent READLN)

:: *read-list ::= "(" *list-id ")"
(dps  *read-list
  template
  (((*ignore* \( *list-id (*ignore* \))))
  parent nil)

:: *case-statement ::= *expr "OF" *case-list [";"] "END"
(dps  *case-statement
  template
  ((*expr (*ignore* OF)
    *case-list (*opt* *semi) (*ignore* END)))
  parent CASE)

:: *case-list ::= *case-part ";" *case-list | *case-part
(dps  *case-list
  template
  ((*case-part (*ignore* \;) *case-list) (*case-part))
  parent nil)

:: *case-part ::= *case-const ":" *statement
(dps  *case-part
  template
  ((*case-const (*ignore* \:) *statement))
  parent CASE-PART)

:: *case-const ::= *case-const-list
(dps  *case-const
  template

```

```

((case-const-list))
parent CONST-LIST)

;; *case-const-list ::= *id-or-num ".*" *case-const-list | *id-or-num
(dps  *case-const-list
  template
  ((id-or-num (*ignore* \.) *case-const-list) (*id-or-num))
  parent nil)

;; *id-or-num ::= *id | *num
(dps  *id-or-num
  template
  ((*id) (*num))
  parent nil)

;; *if-statement ::= *bool-expr "THEN" *statement [*else-p]
(dps  *if-statement
  template
  ((*bool-expr (*ignore* THEN) *statement (*opt* *else-p)))
  parent IF)

;; *else-p ::= "ELSE" *statement
(dps  *else-p
  template
  (((*ignore* ELSE) *statement))
  parent nil)

;; *id-list ::= *list-id
(dps  *id-list
  template
  ((*list-id))
  parent id-list)

;; *list-id ::= *id {".*" *id}
(dps  *list-id
  template
  (((*null* *id ( \.))))
  parent nil)

;; *id is predefined
(dps  *id
  template
  (((*func* (lambda (x)
    (and (>= (char-upcase (aref (string x) 0)) #\A)
    (<= (char-upcase (aref (string x) 0)) #\Z))))))
  parent nil)

;; *num is predefined
(dps  *num
  template
  (((*func* (lambda (x) (numberp x)))))
  parent nil)

;; Load the Explain mismatched group
;; Contains the following functions and their supporting functions:
;; Explain-mismatches, eval-malformed-rules, single-eval-malformed,

```

```

;; item-eval-malformed, eval-a-malformed-rule, eval-missing-rules, and
;; single-eval-missing.
;; Explain Mismatches functions      EXPLAIN.LSP
;; Written by Bret Wallach          10/23/84
;;
;;
;; Explain mismatches
;;
;; This function calls eval-malformed-rules and then eval-missing-rules
;; to try to explain mismatches in the plans. If one of those two functions
;; returns a plan (meaning that all mismatches for the plan are explained),
;; the plan is set to unblocked, and the component address is increased by 1.
(defun explain-mismatches (plans)
  (let ((new-plan
        (or (eval-malformed-rules plans)
            (eval-missing-rules plans))))
    (cond (new-plan
          (setf (ins-status new-plan) ())
          (incf (ins-address new-plan))
          new-plan)
          (t ())))))

;; Use the malformed rules to explain mismatches in the plans
;;
;; Eval-malformed-rules call Single-Eval-Malformed for each plan.
(defun eval-malformed-rules (plans)
  (cond
    (plans (or (single-eval-malformed (car plans))
               (eval-malformed-rules (cdr plans))))
    (t ())))

;; This function gets each match-frame from the partials slot of the plan
;; instantiation and passes it to item-eval-malformed so that the errors
;; can be explained. If the errors are explained the plan is returned.
(defun single-eval-malformed (plan)
  (let ((partial (pop (ins-partials plan))))
    (cond (partial
          (format t "~% Partial: ~a" (match-frame-errors partial))
          (format t "~% Plan-name: ~a" (ins-name plan))
          (or (and (item-eval-malformed plan partial)
                  (setf (aref (ins-matches plan)
                              (ins-address plan))
                        (list (match-frame-code partial)))
              plan)
              (single-eval-malformed plan)))
          (t ())))))

;; This function applies each malformed rule to a match-frame until
;; the errors are explained or there are no rules left.
(defun item-eval-malformed (plan match-frame)
  (do* ((x *malformed-rules* (cdr x))
        (y (car x) (car x)))
    ((or (null x) (eval-a-malformed-rule plan match-frame y))
     (cond (x
           (and exec-out
                (format t "~% Match error explained by ~a in plan ~a."
                        y (ins-name plan)))
           plan)
           (t ())))))

```

```
(t ())))))

;; This function makes sure that each part of a malformed rule is satisfied.
;; The bug rule action is performed if this is the case.
(defun eval-a-malformed-rule (plan match-frame rule)
  (let ((explained-errors 0))
    (and (plan-component plan rule)
          (expected match-frame rule)
          (plan-test plan rule)
          (test-code)
          (>= explained-errors (length (match-frame-errors match-frame)))
          (action))))

;; Use the missing rules to explain mismatches in the plans
;; .....
;; Eval-missing-rules calls Single-Eval-Missing for each plan.
(defun eval-missing-rules (plans)
  (cond
   (plans (or (single-eval-missing (car plans))
               (eval-missing-rules (cdr plans))))
   (t ())))

;; This function applies each missing rule to the plan-instantiation until
;; the errors are explained or there are no rules left. If one of the
;; rules fires, the plan-component is said to match the same thing as the
;; first plan component.
(defun single-eval-missing (plan)
  (do* ((x *missing-rules* (cdr x))
        (y (car x) (car x)))
       ((or (null x) (eval-a-missing-rule plan y))
        (cond (x (setf (aref (ins-matches plan)
                              (ins-address plan))
                        (aref (ins-matches plan) 0))
              (and exec-out
                    (format t "%Match error explained by %a in plan %a."
                            y (ins-name plan)))
              plan) (t ())))))

;; This function makes sure that each part of a missing rule is satisfied.
;; The bug rule action is performed if this is the case.
(defun eval-a-missing-rule (plan rule)
  (and (plan-component plan rule)
        (plan-test plan rule)
        (test-code)
        (action)))

;; Explain Utilities
;; .....
;; This function makes sure that the component that didn't match is referred
;; to by a label statement in a later plan component. If there is no
;; 'plan-component property (this is the label), then return T.
(defun plan-component (plan rule)
  (let ((cur-addr (+ (ins-address plan) 1))
        (len (length (ins-matches plan)))
        (templ (get (ins-name plan) 'template))
        (label (get rule 'plan-component)))
    (cond ((null label) t)
          (t (do ((i cur-addr (+ i 1)))
```

```

      ((or (equal i len)
            (and (equal (car (aref tmplt i)) 'label)
                  (equal (cadr (aref tmplt i)) label)
                  (equal (caddr (aref tmplt i))
                          cur-addr)))
            (not (equal i len))))))

;; This function analyzes the 'expected and 'found properties of the
;; bug rule. If both properties exist, then a cons of the form
;; (expected . found) is sought in the match-frame errors.
;; If there is no 'expected property, but there is a 'found property,
;; then there must be a cons of the form (x . found) where x can
;; be anything. If neither property exists then true is returned.
(defun expected (match-frame rule)
  (let ((exptd (get rule 'expected))
        (fnd (get rule 'found)))
    (cond ((and (null exptd) (null fnd)) t)
          ((and (null exptd)
                 (not (null (rassoc fnd (match-frame-errors
                                         match-frame)))))
           (incf explained-errors) t)
          (t (let ((arg (cdr (assoc exptd (match-frame-errors
                                         match-frame)))))
                (cond ((or (equal fnd arg) (member fnd arg))
                       (incf explained-errors) t)
                      (t ()))))))

;; This function makes sure the plan errors being explained are part of
;; a specified plan. If there is no 'plan property, return T.
(defun plan-test (plan rule)
  (let ((plan-name (get rule 'plan)))
    (or (null plan-name) (equal plan-name (ins-name plan)))))

;; This function executes the test part of the rule and returns its result.
;; It returns T if there is no test function associated with the rule.
(defun test-code ()
  (let ((func (get rule 'test-code)))
    (or (null func) (funcall func))))

;; This function fires the action part of the rule and returns T.
(defun action ()
  (setf *bug-report* (append *bug-report*
                              (list (funcall (get rule 'action)))) t))

;; Bug rules
BUGS.LSP
;; Written by Bret Wallach 10/23/84
;;
;; This file contains the test action pairs and the report functions
;; for the bug rules.
;;
;; The variable explained-errors contains the number of mismatches that
;; have been explained by the bug rules. All mismatches have to be
;; explained before the action part of a rule will fire.

;; Utilities
.....
;; Returns the node in the *parse-tree* that the bug is associated

```

```

(defun bug-statement (bug)
  (match-frame-code (cdr (assoc 'statement bug))))

;;Returns the component label of the bug
(defun bug-component (bug)
  (cdr (assoc 'component bug)))

;;Returns the goal label of the bug
(defun bug-goal (bug)
  (cdr (assoc 'goal bug)))

;;While-for-if rule
.....
(defun while-for-if-action ()
  '(while-for-if (statement . match-frame)))

(dps while-for-if expected if found while action while-for-if-action
  report-fun report-while-for-if)

(defun report-while-for-if (bug)
  (format t
    "You used a WHILE statement at line ~a where you should have used an IF."
    (node-line (bug-statement bug))))

;;Read-is-counter rule
.....
(defun check-read-is-counter ()
  (let ((var (node-name (car (node-children (match-frame-code match-frame))))))
    (incr (cadr (assoc () (match-frame-errors match-frame)))))
    (and (equal (length (match-frame-errors match-frame)) 2)
          (equal (car incr) '+)
          (equal (cadr incr) var)
          (equal (caddr incr) '1)
          (incf explained-errors)
          (incf explained-errors)
          t)))

(defun read-is-counter-action ()
  '(read-is-counter (statement . match-frame)))

(dps read-is-counter expected read found :=
  test-code check-read-is-counter action read-is-counter-action
  report-fun report-read-is-counter)

(defun report-read-is-counter (bug)
  (format t
    "It appears that you were trying to use line ~a to read the next input
    value. Incrementing ~a will not cause the next value to be read in. You
    need to use a read statement here."
    (node-line (bug-statement bug))
    (node-name (car (node-children (bug-statement bug))))))

;;Sum is counter rule
.....
(defun sum-is-counter-action ()
  '(sum-is-counter (statement . match-frame)))

```

II.2 EXAMPLE2.PAS

```
PROGRAM Average2( INPUT, OUTPUT );
VAR SUM, COUNT, NEW: INTEGER;
    AVG: REAL;
BEGIN
    SUM := 0;
    COUNT := 0;
    NEW := 0;
    WHILE NEW <> 9999 DO
        BEGIN
            READ( NEW );
            SUM := SUM + NEW;
            COUNT := COUNT + 1;
        END;
    IF COUNT = 0 THEN
        WRITELN( 'NO DATA ENTERED' )
    ELSE
        BEGIN
            AVG := SUM / COUNT;
            WRITELN( 'THE AVERAGE IS ', AVG );
        END
    END;
END;
```

II. Sample Pascal Programs

II.1 EXAMPLE1.PAS

```
PROGRAM Average (input,output);
VAR Sum, Count, New: INTEGER;
    Avg: REAL;
BEGIN
    Sum := 0;
    Count := 0;
    Read (New);
    WHILE New<>9999 DO
        BEGIN
            Sum := Sum+New;
            Count := Count+1;
            New := New + 1
        END;
    Avg := Sum/Count;
    WRITELN ('The average is ',avg);
END;
```

(OTHERWISE (APPLY @'DRIBBLE-IN ARGS))))

I.6 DRIBBLE.LSP

```

::: tftdribble.lsp
::: Copyright (c) Gold Hill Computers, Inc. 1984

::: The DRIBBLE facility (slurp)
::: The only toplevel function in this file is DRIBBLE.

(DEFVAR *DRIBBLE-STREAM* NIL)           ; the DRIBBLE output stream
(DEFVAR *DRIBBLE-TERMINAL* NIL)         ; during DRIBBLE this is terminal

(DEFUN DRIBBLE (&OPTIONAL PN)
  "DRIBBLE with a pathname argument start the dribble operation, with
  no pathname argument ends the dribble operation."
  (COND ((AND PN (NULL *DRIBBLE-STREAM*))           ; open dribble file
    (SETQ *DRIBBLE-STREAM* (OPEN PN :DIRECTION :OUTPUT)
      *DRIBBLE-UNTYI* NIL
      *DRIBBLE-TERMINAL* *TERMINAL-IO*
      *TERMINAL-IO* #'DRIBBLE-TERM)
    T)
    ((AND (NULL PN) *DRIBBLE-STREAM*)           ; close dribble file
      (SETQ *TERMINAL-IO* *DRIBBLE-TERMINAL*)
      (CLOSE *DRIBBLE-STREAM*)
      (SETQ *DRIBBLE-STREAM* NIL))
    ((AND (NULL PN) (NULL *DRIBBLE-STREAM*)) ; asked to close but not open
      (FORMAT T "~&DRIBBLE not in progress.~n"))
    ((AND PN *DRIBBLE-STREAM*)           ; asked to open but already open
      (FORMAT T "~&DRIBBLE is already in progress.~n"))
    ))

(DEFVAR *DRIBBLE-UNTYI* NIL)

:: This is the input stream handler during a dribble.
(DEFUN DRIBBLE-IN (MSG &REST ARGS)
  (CASE MSG
    (:TYI
      (COND (*DRIBBLE-UNTYI*
        (PROG1 *DRIBBLE-UNTYI*
          (SETQ *DRIBBLE-UNTYI* NIL)))
      (T
        (LET ((CHAR (SEND *DRIBBLE-TERMINAL* :TYI)))
          (SEND *DRIBBLE-STREAM* :TYO CHAR)
          CHAR))))
      (:UNTYI (SETQ *DRIBBLE-UNTYI* (CAR ARGS)))
      ;; forward to the real stream.
      (OTHERWISE
        (APPLY *DRIBBLE-TERMINAL* MSG ARGS))))

:: This is the output stream handler during a dribble.
(DEFUN DRIBBLE-OUT (&REST ARGS)
  (APPLY *DRIBBLE-STREAM* ARGS)
  (APPLY *DRIBBLE-TERMINAL* ARGS))

:: This is the stream for *TERMINAL-IO* during a dribble.
:: Depending on the message we decide to dispatch to the input
:: or output dribble streams.
(DEFUN DRIBBLE-TERM (&REST ARGS)
  (CASE (CAR ARGS)
    ((:TYO :STRING-OUT :LINE-OUT) (APPLY #'DRIBBLE-OUT ARGS))

```

```

(+ OFFSET INITIAL-OFFSET) ; offset to structure elements
NAMEDP                    ; wheter its named
PRINT-FUNCTION            ; the print function or NIL
CONSER                    ; name of the conser or NIL
PREDICATE                 ; name of predicate or NIL
)))

```

```

; The structure definition info is kept on the STRUCTURE-DESCRIPTOR property
; of the structure name. It consists of a list:
;   ( slot-alist type options)

```

```

(DEFMACRO DEFSTRUCT FRM
  (LET* ((SLOTS (MAPCAR '(LAMBDA (X) (IF (CONSP X) X (NCONS X))) (CDR FRM)))
        (S-NAME (IF (CONSP (CAR FRM)) (CAR FRM) (CAAR FRM)))
        (OPTIONS (IF (CONSP (CAR FRM)) (CDAR FRM) NIL))
        (RESULT '('.S-NAME))
        (SLOT-ALIST))
    (IF (STRINGP (CAAR SLOTS)) (POP SLOTS)) ; dump DOC string
    (MULTIPLE-VALUE-BIND (CONC-NAME STRUCT-TYPE OFFSET NAMEDP
                        PRINT-FUNCTION CONSER PREDICATE)
      (DEFSTRUCT-PROCESS-OPTIONS S-NAME OPTIONS)

      (SETQ SLOT-ALIST ; entries: (SLOT-NAME INDEX DEFAULT)
        (DO ((I OFFSET (1+ I))
            (SLOT SLOTS (CDR SLOT))
            (ALIST NIL (CONS (IF (CONSP (CDAR SLOT))
                              (LIST (CAAR SLOT) I (CDAR SLOT))
                              (LIST (CAAR SLOT) I)) ALIST)))
          ((NULL SLOT) (NREVERSE ALIST))))

      (PUTPROP S-NAME
        (LIST SLOT-ALIST ; alist of slots
              STRUCT-TYPE ; type of structure
              OPTIONS ; options
              (+ OFFSET (LENGTH SLOT-ALIST)) ; size of structure
              NAMEDP)
        'STRUCTURE-DESCRIPTOR)

      ;; hook into to the type mechanism
      (WHEN NAMEDP (SETF (GET S-NAME 'SUPER-TYPES)
                        (CONS S-NAME (GET 'STRUCTURE 'SUPER-TYPES))))

      (DOLIST (SLT SLOT-ALIST) ; define accessors
        (PUSH (DEF-ACCESSOR SLT CONC-NAME STRUCT-TYPE) RESULT))
      (WHEN CONSER ; define make macro
        (PUSH (MAKE-STRUCTURE S-NAME CONSER NAMEDP) RESULT))
      (WHEN PREDICATE (PUSH PREDICATE RESULT))
      (WHEN PRINT-FUNCTION (PUSH PRINT-FUNCTION RESULT))
      (CONS 'PROGN RESULT)
    )))

```

```

(DEFUN DEFSTRUCT-PROCESS-OPTIONS (STRUCT-NAME OPTIONS)
  (LET ((CONC-NAME (STRING-APPEND STRUCT-NAME #\~)) ; set up defaults
        (TYPE 'VECTOR)
        (OFFSET 1)
        (INITIAL-OFFSET 0)
        (EXP-NAMEDP) ; whether NAMED/UNNAMED specified
        (NAMEDP T)
        (PRINT-FUNCTION)
        (CONSER (INTERN (STRING-APPEND "MAKE-" STRUCT-NAME)))
        (PREDICATE T)
        )
    (DOLIST (OPT OPTIONS)
      (CASE (IF (CONSP OPT) (CAR OPT) 'ATOMIC)
        (:CONC-NAME
         (SETQ CONC-NAME (CADR OPT)))
        (:TYPE
         (UNLESS (MEMBER (SETQ TYPE (CADR OPT))
                        '(VECTOR LIST ARRAY-LEADER))
                  (ERROR "Illegal DEFSTRUCT type: "S" OPT))
         (UNLESS EXP-NAMEDP (SETQ NAMEDP NIL))
         (SETQ OFFSET (UPDATE-OFFSET TYPE NAMEDP))
         )
        (:NAMED
         (SETQ NAMEDP T EXP-NAMEDP T OFFSET (UPDATE-OFFSET TYPE NAMEDP)))
        (:CONSTRUCTOR
         (SETQ CONSER (CADR OPT)))
        (:PREDICATE
         (SETQ PREDICATE (CADR OPT)))
        (:PRINT-FUNCTION
         (SETQ PRINT-FUNCTION '(PUTPROP ',STRUCT-NAME
                                           ,(CADR OPT)
                                           ':PRINT-FUNCTION)))
        (:INITIAL-OFFSET (SETQ INITIAL-OFFSET (CADR OPT)))
        (ATOMIC
         (CASE OPT
           ((:CONSTRUCTOR :CONC-NAME) ; just ignore
            (:NAMED
             (SETQ NAMEDP T EXP-NAMEDP T
                   OFFSET (UPDATE-OFFSET TYPE NAMEDP)))
             (OTHERWISE
              (ERROR "Illegal DEFSTRUCT option: "S" OPT))))
           (OTHERWISE
            (ERROR "Illegal DEFSTRUCT option: "S" OPT))))
        (WHEN PREDICATE
         (IF (EQ PREDICATE T)
             (SETQ PREDICATE (IF NAMEDP
                                  (INTERN (STRING-APPEND STRUCT-NAME "-P"))
                                  NIL))
             (UNLESS NAMEDP
              (ERROR "Can't have PREDICATE with UNNAMED structure"))))
        (WHEN PREDICATE
         (SETQ PREDICATE
          '(DEFUN ,PREDICATE (X)
            ,(IF (EQ TYPE 'LIST)
                '(EQ (CAR X) ',STRUCT-NAME)
                '(TYPEP X ',STRUCT-NAME))))))
    (VALUES CONC-NAME ; prefix for accessor macros or NIL
            TYPE ; structure type

```

```

(TYPE (CADR (GET STRUCT 'STRUCTURE-DESCRIPTOR)))
(STRUCTURE-DESCRIPTOR (GET STRUCT 'STRUCTURE-DESCRIPTOR))
(OPTIONS (DEFST-OPTIONS (GET STRUCT 'STRUCTURE-DESCRIPTOR)))
(ST))
'(DEFMACRO ,NAME X
.(CASE TYPE
  (LIST
    '(LIST
      .DO ((S (DEFST-SLOTS (GET ',STRUCT 'STRUCTURE-DESCRIPTOR)
        (CDR S))
        TEMP
        (RES ,(WHEN (DEFST-NAMEDP STRUCTURE-DESCRIPTOR)
          '(NCONS ',STRUCT))
          (IF (SETQ TEMP
            (OR (GETF X (SLOT-NAME (CAR S)))
              (CAR (SLOT-DEFAULT (CAR S)))))
            (CONS TEMP RES)
            (CONS NIL RES))))
          ((NULL S)(NREVERSE RES))))))
      ((VECTOR ARRAY-LEADER)
        '(LET ((',(SETQ ST (MAKE-SYMBOL "VAR"))
          (MAKE-ARRAY ,(OR (CADR (ASSOC 'LENGTH OPTIONS))
            (PROG1
              (DEFST-SLOT-CNT STRUCTURE-DESCRIPTOR)
              (WHEN (EQ TYPE 'ARRAY-LEADER)
                (ERROR
                  "No length for array leader"))))
            ..(WHEN NAMEDP
              '(:NAMED-STRUCTURE-SYMBOL ',STRUCT))
            ..(WHEN (EQ TYPE 'ARRAY-LEADER)
              '(:LEADER-LENGTH ,(LENGTH SLOT-ALIST)))
            ..(WHEN (OR (MEMBER 'NAMED OPTIONS)
              (ASSOC 'NAMED OPTIONS))
              '(:NAMED-STRUCTURE-SYMBOL ',STRUCT))
            ..(CDR (ASSOC 'MAKE-ARRAY OPTIONS))))))
          .DO ((S (DEFST-SLOTS (GET ',STRUCT 'STRUCTURE-DESCRIPTOR)
            (CDR S))
            TEMP
            (OBJ (NCONS NIL))
            (RES NIL
              (PROGN
                (SETQ TEMP (GETF X (SLOT-NAME (CAR S)) OBJ))
                (WHEN (AND (EQ TEMP OBJ) (SLOT-DEFAULT (CAR S)))
                  (SETQ TEMP (CAR (SLOT-DEFAULT (CAR S)))))
                (IF (EQ TEMP OBJ)
                  RES
                  (CONS (LIST ',(IF (EQ TYPE 'VECTOR)
                    'ASET 'STORE-ARRAY-LEADER
                    TEMP ',ST (SLOT-POS (CAR S)))
                    RES))))))
            ((NULL S)(NREVERSE (CONS ',ST RES))))))
        ))))
      )))
(DEFUN UPDATE-OFFSET (TYPE NAMEDP)
  (COND ((AND (EQ TYPE 'ARRAY-LEADER) NAMEDP) 2)
    ((OR (AND NAMEDP (NEQ TYPE 'ARRAY-LEADER))
      (AND (EQ TYPE 'ARRAY-LEADER) (NOT NAMEDP)))
      1)
    (T 0)))

```

1.5 DEFSTRUC.LSP

```

:: t1defstruc.lsp
:: Copyright (c) Gold Hill Computers, Inc. 1984

:: The DEFSTRUCT package

:: Notes: The DEFSTRUCT package is not completely COMMON Lisp compatible yet.
:: The accessor and constructors are not functions as per COMMON Lisp but are
:: macros, this will be fixed. The slot names in the structure constructor
:: are not keywords but just symbols. Thus for the structure:
:: (DEFSTRUCT TURTLE HEADING X-POS Y-POS)
:: the make function would be:
:: (MAKE-TURTLE HEADING 90 X-POS 100 Y-POS 200).
:: COMMON Lisp would have the slotnames be keywords, eg. :HEADING. This will
:: be fixed in the next release.

:: The currently supported options are shown below:

:: (DEFSTRUC NAME-OR-OPTIONS &REST SLOTS)
:: Options:
:: :CONC-NAME
:: :CONSTRUCTOR
:: :INITIAL-OFFSET
:: :NAMED
:: :PREDICATE
:: :PRINT-FUNCTION
:: :TYPE {VECTOR | LIST}

:: Accessor macros
:: These macros take the object that is on the STRUCTURE-DESCRIPTOR
:: property of the structure name.
(DEFMACRO DEFST-SLOTS X '(CAR ,(CAR X))) ; returns structures slots
; hard wired in DESCRIBE
(DEFMACRO DEFST-TYPE X '(CADR ,(CAR X))) ; returns type
(DEFMACRO DEFST-OPTIONS X '(CADDR ,(CAR X))) ; returns all defstruct options
; hard wired in DESCRIBE
(DEFMACRO DEFST-SLOT-CNT X '(NTH 3 ,(CAR X))) ; number of slots (local?)
(DEFMACRO DEFST-NAMEDP X '(NTH 4 ,(CAR X))) ; whether named structure

:: These macros take a slot descriptor, member of the DEFST-SLOTS list.
(DEFMACRO SLOT-NAME X '(CAR ,(CAR X))) ; name of slot
; hard wired in DESCRIBE
(DEFMACRO SLOT-POS X '(CADR ,(CAR X))) ; structure position (local?)
(DEFMACRO SLOT-DEFAULT X '(CDDR ,(CAR X))) ; default form, returns a LIST

(DEFUN DEF-ACCESSOR (SLOT-D CONC-NAME TYPE)
  '(DEFMACRO ,(IF CONC-NAME
    (INTERN (STRING-APPEND CONC-NAME (CAR SLOT-D)))
    (CAR SLOT-D))
    X
    ,(CASE TYPE
      (VECTOR '(LIST 'AREF (CAR X) ,(CADR SLOT-D)))
      (ARRAY-LEADER '(LIST 'ARRAY-LEADER (CAR X) ,(CADR SLOT-D)))
      (LIST '(LIST 'NTH ,(CADR SLOT-D) (CAR X))))))

:: Returns the macro that will make the structure.
(DEFUN MAKE-STRUCTURE (STRUCT NAME NAMEDP)
  (LET ((SLOT-ALIST (CAR (GET STRUCT 'STRUCTURE-DESCRIPTOR)))

```

```
(SETQ *VALLIST* (CONS PATH *VALLIST*))  
(T  
 (DEFMACRO-COLLECT (CAR PATTERN) (LIST 'CAR PATH))  
 (DEFMACRO-COLLECT (CDR PATTERN) (LIST 'CDR PATH))))
```

```

(COND ((> STATE 0) (ERROR "Bad pattern to DEFMACRO: "S" EPAT))
  (T (DEFMACRO-PARSE-& (CDR PATTERN) PATH 1 EPAT))))
((MEMBER (CAR PATTERN) '(&REST &BODY))
  (AND (EQ (CAR PATTERN) ' &BODY)
    (SETQ *DEFMACRO-&BODY-FLAG* T))
  (AND (NULL (CDR PATTERN))
    (ERROR "Bad pattern to DEFMACRO: "S" EPAT))
  (COND ((> STATE 1) (ERROR "Bad pattern to DEFMACRO: "S" EPAT))
    (T (DEFMACRO-PARSE-& (CDR PATTERN) PATH 2 EPAT))))
((EQ (CAR PATTERN) ' &AUX)
  (COND ((> STATE 2) (ERROR "Bad pattern to DEFMACRO: "S" EPAT))
    (T (DEFMACRO-PARSE-& (CDR PATTERN) PATH 3 EPAT))))
(= STATE 0)
(DEFMACRO-COLLECT (CAR PATTERN) (LIST 'CAR PATH))
(LET ((PAIR (DEFMACRO-PARSE-&
  (CDR PATTERN) (LIST 'CDR PATH) 0 EPAT)))
  (WHEN (CDR PAIR) (INCF (CDR PAIR)))
  (INCF (CAR PAIR))
  PAIR))
(= STATE 1)
(COND ((ATOM (CAR PATTERN))
  (DEFMACRO-COLLECT (CAR PATTERN)
    '(CAR ,PATH)))
  (T
    (AND (CAR (CDDAR PATTERN))
      (PUSH (CAR (CDDAR PATTERN))
        *OPTIONAL-SPECIFIED-FLAGS*))
    (DEFMACRO-COLLECT (CAAR PATTERN)
      '(COND (.PATH
        .(AND (CAR (CDDAR PATTERN))
          '(SETQ ,(CAR (CDDAR PATTERN)) T))
        (CAR ,PATH)
        (T ,(CADAR PATTERN))))))
  (LET ((PAIR (DEFMACRO-PARSE-&
    (CDR PATTERN) (LIST 'CDR PATH) 1 EPAT)))
    (IF (NULL (CDR PAIR))
      PAIR
      (INCF (CDR PAIR)))
    PAIR))
(= STATE 2)
(DEFMACRO-COLLECT (CAR PATTERN) PATH)
(WHEN (CDR PATTERN)
  (WHEN (OR (ATOM (CDR PATTERN))
    (NOT (EQ (CADR PATTERN) ' &AUX)))
    (ERROR "Bad pattern to DEFMACRO: "S" EPAT))
    (DEFMACRO-PARSE-& (CDDR PATTERN) PATH 3 EPAT))
  (NCONS 0))
(= STATE 3)
(IF (ATOM (CAR PATTERN))
  (DEFMACRO-COLLECT (CAR PATTERN) NIL)
  (DEFMACRO-COLLECT (CAAR PATTERN) (CADAR PATTERN)))
(DEFMACRO-PARSE-& (CDR PATTERN) (LIST 'CDR PATH) 3 EPAT))
))

```

```

(DEFUN DEFMACRO-COLLECT (PATTERN PATH)
  (COND ((NULL PATTERN))
    ((ATOM PATTERN)
      ;; *** test of & keyword here
      (SETQ *VARLIST* (CONS PATTERN *VARLIST*)))
  )

```

I.4 DEFMAC.LSP

```

;;; ttfdefmac.lsp

;;; Copyright (c) Gold Hill Computers, Inc. 1984

;; NOTE: The &WHOLE option is not yet supported.

(MACRO DEFMACRO (X) (DEFMACRO1 (CDR X)))

(DEFVAR *OPTIONAL-SPECIFIED-FLAGS*)

;; whether &BODY was specified
(DEFVAR *DEFMACRO-&BODY-FLAG*)

;; X is the cdr of the DEFMACRO form.
(DEFUN DEFMACRO1 (X)
  (LET (*VARLIST* *VALLIST* *OPTIONAL-SPECIFIED-FLAGS* *DEFMACRO-&BODY-FLAG*)
    (LET ((PAIR (DEFMACRO-PARSE-&
                  (CADR X) '(CDR *MACROARG*) 0 (CADR X)))
          (BODY (CDDR X)))
      '(MACRO .(CAR X) (*MACROARG*)
        .IF (NOT (AND (ZEROP (CAR PAIR))
                     (NULL (CDR PAIR))))
          '((AND .(COND ((ZEROP (CAR PAIR))
                        '(> (LENGTH *MACROARG*)
                          .(1+ (CDR PAIR))))
                    ((NULL (CDR PAIR))
                     '(< (LENGTH *MACROARG*)
                       .(1+ (CAR PAIR))))
              (T ' (OR (< (LENGTH *MACROARG*)
                        .(1+ (CAR PAIR)))
                      (> (LENGTH *MACROARG*)
                        .(1+ (CDR PAIR))))))
            (ERROR "Wrong number of args to macro: "S"
                  *MACROARG*))))
      .(DEFMACRO2 (NREVERSE *VARLIST*) (NREVERSE *VALLIST*)
        *OPTIONAL-SPECIFIED-FLAGS* BODY))))))

;; Put together the various bindings and the body.
;; The VARS are bound sequentially since their initializations may depend
;; on each other (in left-to-right fashion).
(DEFUN DEFMACRO2 (VARS VALS FLAGS BODY)
  '(LET* (.@FLAGS
    .MAPCAR #'(LAMBDA (X Y) (LIST X Y)) VARS VALS)
    (*MACROARG1* .(IF (CDR BODY)
      '(PROGN .,BODY)
      (CAR BODY))))
    (IF (CONSP *MACROARG1*)
      (RPLACD *MACROARG* *MACROARG1*)
      *MACROARG1*)))

(DEFUN DEFMACRO-PARSE-& (PATTERN PATH STATE EPAT)
  (COND ((NULL PATTERN) (CONS 0 0))
        ((ATOM PATTERN)
         (COND ((> STATE 1) (ERROR "Bad pattern to DEFMACRO: "S" EPAT))
               (T (DEFMACRO-COLLECT PATTERN PATH)
                  (NCONS 0))))
        ((EQ (CAR PATTERN) '&OPTIONAL)

```

```
sum-is-counter assignment-backwards))
(setf *missing-rules* '(missing-exception-guard missing-internal-guard
missing-guarded-counter))
```

```
:: Load the files before we start for smoother operation.
(cond ((functionp 'make-window-stream) ())
      (t (load "lisplib\\vstream.lsp")))
::(cond ((functionp 'dribble) ())
::      (t (load "lisplib\\dribble.lsp")))
::
```

```
::: Some useful macros from COMMON Lisp
```

```
(DEFMACRO WITH-OPEN-STREAM X
  '(LET (. (CAR X))
    (UNWIND-PROTECT
      .(IF (CDDR X)
          '(PROGN .CDR X))
      (CADR X))
    (CLOSE .(CAAR X)))))
```

```
(DEFMACRO WITH-OPEN-FILE X
  '(WITH-OPEN-STREAM (. (CAAR X) (OPEN .CDAR X)))
  .CDR X))
```

```
(m-proust)
```

input before line "a is executed. The "a is not defined when there is no input."

```
(node-line (or (car (bug-component bug)) *parse-tree*))
(get (bug-goal bug) 'name-phase)))
```

::Missing-internal-guard rule

```
(defun missing-internal-guard-action ()
  '(missing-internal-guard (component . .(cdr (assoc 'mainloop:
    (ins-bindings plan))))
    (goal . .(car *active-goal*))))
```

```
(dps missing-internal-guard plan-component internal-guard:
  action missing-internal-guard-action
  report-fun report-missing-internal-guard)
```

```
(defun report-missing-internal-guard (bug)
  (format t
    "You're missing a sentinel guard in the loop beginning at line "a. When
    your program reads the sentinel, it processes it as if it were data."
    (node-line (or (car (bug-component bug)) *parse-tree*))))
```

::Missing-guarded-counter rule

```
(defun missing-guarded-counter-action ()
  '(missing-guarded-counter (component . .(cdr (assoc 'input:
    (ins-bindings plan))))))
```

```
(dps missing-guarded-counter plan-component process:
  plan bad-input-loop-guard
  action missing-guarded-counter-action
  report-fun report-missing-guarded-counter)
```

```
(defun report-missing-guarded-counter (bug)
  (format t
    "You're missing a sentinel guard. If a sentinel value is input immediately
    following a negative value at line "a, your program will process it
    as if it were data."
    (node-line (or (car (bug-component bug)) *parse-tree*))))
```

::Miss-internal-guard report

```
(defun report-miss-internal-guard (bug)
  (format t
    "Your program does not perform an input validation."))
```

```
(dps loop-input-validation report-fun report-miss-internal-guard)
```

::Rule list

```
(setf *malformed-rules* '(read-is-counter while-for-if
```

```
(dps sum-is-counter found 1 plan running-total action sum-is-counter-action
  report-fun report-sum-is-counter)
```

```
(defun report-sum-is-counter (bug)
  (format t
    "You are using a counter update instead of a running-total update at
    line ~a. You must add the value of ~a to the variable ~a, not add 1 to it."
    (node-line (bug-statement bug))
    (resolve-var-ref 'new 'sentinel-controlled-loop)
    (node-name (car (node-children (bug-statement bug))))))
```

```
::Assignment-backwards rule
```

```
.....
(defun assignment-backwards-test ()
  (and (equal (length (match-frame-errors match-frame)) 2)
    (equal (node-name (match-frame-code match-frame)) ':=)
    (equal (length (node-children (match-frame-code match-frame))) 2)
    (equal (cadr (match-frame-errors match-frame))
      (caddr (match-frame-errors match-frame)))
    (equal (cadr (match-frame-errors match-frame))
      (caddr (match-frame-errors match-frame)))
    (incf explained-errors)
    (incf explained-errors)))
```

```
(defun assignment-backwards-action ()
  '(assignment-backwards (statement . .match-frame)))
```

```
(dps assignment-backwards plan-component update:
  test-code assignment-backwards-test
  action assignment-backwards-action
  report-fun report-assignment-backwards)
```

```
(defun report-assignment-backwards (bug)
  (format t
    "The assignment at line ~a is backwards. This line will assign to ~a;
    you need to assign to ~a."
    (node-line (bug-statement bug))
    (node-name (car (node-children (bug-statement bug))))
    (node-name (cadr (node-children (bug-statement bug))))))
```

```
::Missing-exception-guard-rule
```

```
.....
(defun missing-exception-guard-action ()
  '(missing-exception-guard (component . .(cdr (assoc 'code
    (ins-bindings plan))))
    (goal . .(cdr (assoc 'goal
    (ins-bindings plan))))))
```

```
(dps missing-exception-guard plan-component exception:
  action missing-exception-guard-action
  report-fun report-missing-guard-exception)
```

```
(defun report-missing-guard-exception (bug)
  (format t
    "You need a test to check that at least one valid data point has been
```

II.3 EXAMPLE3.PAS

```
PROGRAM MOAH (INPUT ,OUTPUT);
```

```
VAR
```

```
    RAINFALL, LARGEST, SUM, COUNT1,  
    COUNT2, AVERAGE, RAINYDAYS : REAL;
```

```
BEGIN
```

```
    (* INITIALIZE VARIABLES *)
```

```
    LARGEST :=0;
```

```
    SUM :=0;
```

```
    COUNT1 :=0;
```

```
    COUNT2 :=0;
```

```
    AVERAGE :=0;
```

```
    RAINYDAYS :=0;
```

```
    (*READ THE RAINFALL AND CHECK FOR ERROR VALUES *)
```

```
    WRITELN ('ENTER RAINFALL, WHEN YOU ARE FINISHED ENTER 9999');
```

```
    READLN;
```

```
    READ (RAINFALL);
```

```
    WHILE RAINFALL <> 9999 DO
```

```
        BEGIN
```

```
            WHILE RAINFALL <0 DO
```

```
                BEGIN
```

```
                    WRITELN (RAINFALL :8, 'IS NOT A POSSIBLE RAINFALL, TRY AGAIN.');
```

```
                    WRITELN ('ENTER RAINFALL');
```

```
                    READLN;
```

```
                    READ (RAINFALL)
```

```
                END;
```

```
            IF RAINFALL > LARGEST
```

```
                THEN
```

```
                    LARGEST :=RAINFALL;
```

```
            IF RAINFALL > 0
```

```
                THEN
```

```
                    COUNT2 :=COUNT2 + 1;
```

```
            COUNT1 := COUNT1 + 1;
```

```
            SUM := SUM + RAINFALL;
```

```
            READLN;
```

```
            READ (RAINFALL)
```

```
        END;
```

```
    AVERAGE := SUM/COUNT1;
```

```
    WRITELN (COUNT1 :8, 'VALID RAINFALLS WERE ENTERED.');
```

```
    WRITELN ('THE AVERAGE RAINFALL WAS', AVERAGE :8, 'INCHES PER DAY.');
```

```
    WRITELN ('THE HIGHEST RAINFALL WAS', LARGEST :0:2, 'INCHES.');
```

```
    WRITELN ('THERE WERE', COUNT2 :0:2, 'RAINYDAYS IN THIS PERIOD.')
```

```
END.
```

II.4 EXAMPLE4.PAS

```
PROGRAM RAINFALL ( INPUT,OUTPUT );
```

```
VAR
```

```
RAIN, DAYS, TOTALRAIN, RAINDAYS, HIGHRAIN, AVERAIN: REAL;
```

```
BEGIN
```

```
DAYS := 0;
```

```
TOTALRAIN := 0;
```

```
RAINDAYS := 0;
```

```
HIGHRAIN := 0;
```

```
REPEAT
```

```
WRITELN ('ENTER RAINFALL');
```

```
READLN;
```

```
READ (RAIN);
```

```
WHILE RAIN <> 9999 DO
```

```
BEGIN
```

```
DAYS := DAYS + 1;
```

```
TOTALRAIN := TOTALRAIN + RAIN;
```

```
IF RAIN > 0 THEN
```

```
RAINDAYS := RAINDAYS + 1;
```

```
IF RAIN > HIGHRAIN THEN
```

```
HIGHRAIN := RAIN
```

```
END
```

```
UNTIL RAIN = 9999;
```

```
AVERAIN := TOTALRAIN / DAYS;
```

```
WRITELN;
```

```
WRITELN ( DAYS:0:0, 'VALID RAINFALLS WERE ENTERED');
```

```
WRITELN;
```

```
WRITELN ('THE AVERAGE RAINFALL WAS',AVERAIN:0:2,'INCHES PER DAY');
```

```
WRITELN;
```

```
WRITELN ('THE HIGHEST RAINFALL WAS',HIGHRAIN:0:2,'INCHES');
```

```
WRITELN;
```

```
WRITELN ('THERE WERE',RAINDAYS:0:0,'IN THIS PERIOD');
```

```
END.
```

II.5 EXAMPLE5.PAS

```
PROGRAM TEST1(INPUT, OUTPUT);
```

```
VAR
```

```
    SUM,N,MAX,AVE:REAL;
```

```
    COUNT,RAINY:INTEGER;
```

```
BEGIN
```

```
    SUM:=0;
```

```
    COUNT:=0;
```

```
    RAINY:=0;
```

```
    MAX:=0;
```

```
    WRITELN('ENTER RAINFALL');
```

```
    READLN;
```

```
    READ(N);
```

```
    WHILE N<>9999 DO
```

```
    BEGIN
```

```
        IF N<0 THEN
```

```
            WRITELN(N:0:2,' IS NOT A POSSIBLE RAINFALL, TRY AGAIN')
```

```
        ELSE
```

```
            BEGIN
```

```
                COUNT:=COUNT+1;
```

```
                SUM:=SUM+1;
```

```
                IF N>0 THEN
```

```
                    RAINY:=RAINY+1;
```

```
                    IF N>MAX THEN
```

```
                        N:=MAX;
```

```
            END;
```

```
            WRITELN('ENTER RAINFALL');
```

```
            READLN;
```

```
            READ(N)
```

```
        END;
```

```
    WRITELN;
```

```
    IF COUNT=0 THEN
```

```
        WRITELN(COUNT:0,' VALID RAINFALLS WERE ENTERED.')
```

```
    ELSE
```

```
        BEGIN
```

```
            AVE:=SUM/COUNT;
```

```
            WRITELN('THE AVERAGE RAINFALL WAS ',AVE:0:2,' INCHES PER DAY.');
```

```
            WRITELN('THE HIGHEST RAINFALL WAS ',MAX:0:2,' INCHES.');
```

```
            WRITELN('THERE WERE ',RAINY:0,' RAINY DAYS IN THIS PERIOD.')
```

```
        END
```

```
END.
```

II.6 EXAMPLEA.PAS

```
PROGRAM Average (input,output);
VAR Sum, Count, New: INTEGER;
    Avg: REAL;
BEGIN
    Sum := 0;
    Count := 0;
    Read (New);
    WHILE New<>9999 DO
        BEGIN
            Sum := Sum+New;
            Count := Count+1;
            Read (New)
        END;
    IF Count = 0
    THEN WRITELN ('No average')
    ELSE BEGIN
        Avg := Sum/Count;
        WRITELN ('The average is ',avg);
    END
END;
```

II.7 EXAMPLER.PAS

```
PROGRAM TEST1(INPUT, OUTPUT);
VAR
  SUM,N,MAX,AVE:REAL;
  COUNT,RAINY:INTEGER;
BEGIN
  SUM:=0;
  COUNT:=0;
  RAINY:=0;
  MAX:=0;
  WRITELN('ENTER RAINFALL');
  READLN;
  READ(N);
  WHILE N<>9999 DO
    BEGIN
      IF N<0 THEN
        WRITELN(N:0:2,' IS NOT A POSSIBLE RAINFALL, TRY AGAIN')
      ELSE
        BEGIN
          COUNT:=COUNT+1;
          SUM:=SUM+N;
          IF N>0 THEN
            RAINY:=RAINY+1;
          IF N>MAX THEN
            MAX:=N;
          END;
          WRITELN('ENTER RAINFALL');
          READLN;
          READ(N)
        END;
      WRITELN;
      IF COUNT=0 THEN
        WRITELN(COUNT:0,' VALID RAINFALLS WERE ENTERED.')
      ELSE
        BEGIN
          AVE:=SUM/COUNT;
          WRITELN(COUNT:0,' VALID RAINFALLS WERE ENTERED. ');
          WRITELN('THE AVERAGE RAINFALL WAS ',AVE:0:2,' INCHES PER DAY. ');
          WRITELN('THE HIGHEST RAINFALL WAS ',MAX:0:2,' INCHES. ');
          WRITELN('THERE WERE ',RAINY:0,' RAINY DAYS IN THIS PERIOD.')
        END
      END;
    END;
  END.
```

III. Problem Specifications

III.1 AVERAGE.PR8

```
:: Average Problem                                AVERAGE.pr8
:: Taken from Micro-Proust Design Spec    10/23/84
::
:: (goal . average) was added to guard exception to make
:: bug reporting for missing-exception-guard nicer
::
((sentinel-controlled-loop (stop . 9999))
 (average)
 (count (count . (*var* count average)))
 (sum (new . (*var* new sentinel-controlled-loop))
      (total . (*var* sum average)))
 (output (val . (*var* avg average)))
 (guard-exception (code . (*var* output: output))
                  (pred . (= (*var* count count) 0))
                  (goal . average))
 (guard-exception (code . (*var* update: average))
                  (pred . (= (*var* count count) 0))
                  (goal . average)))
```

III.2 RAINFALL.PRB

```

:: Rainfall Problem Description          RAINFALL.prb
:: Taken from Micro-Proust Design Spec  10/23/84
::
:: (goal . x) added to guard-exception to make bug reporting nicer
::
((sentinel-controlled-loop (stop . 9999))
 (loop-input-validation (new . (*var* new sentinel-controlled-loop))
                        (pred . (< (*var* new sentinel-controlled-loop) 0)))

 (average)
 (count (count . (*var* count average)))
 (output (val . (*var* avg average)))
 (guard-exception (code . (*var* output: output))
                  (pred . (= (*var* count count) 0))
                  (goal . average))
 (output (val . (*var* count count)))
 (sum (new . (*var* new sentinel-controlled-loop))
      (total . (*var* sum average)))
 (guarded-count (pred . (> (*var* new sentinel-controlled-loop) 0)))
 (output (val . (*var* count guarded-count)))
 (maximum (new . (*var* new sentinel-controlled-loop)))
 (output (val . (*var* max maximum)))
 (guard-exception (code . (*var* update: average))
                  (pred . (= (*var* count count) 0))
                  (goal . average))
 (guard-exception (code . (*var* output: output))
                  (pred . (= (*var* count count) 0))
                  (goal . maximum)))
:)
```

END

FILMED

9-85

DTIC