PROGRAM DEBUGGING BY NEAR-MISS RECOGNITION AND SYMBOLIC EVALUATION

Rudi Lutz

This paper was given at the 6th European Conference on Artificial Intelligence (ECAI-84) held in Pisa 1984.

Cognitive Science Research Paper

Serial No. CSRP. 044

The University of Sussex,
Cognitive Studies Programme
School of Social Sciences
Falmer
Brighton BN1 9QN

This paper was given at the 6th European Conference on Artificial Intelligence (ECAI-84) held in Pisa 1984.

Program Debugging By Near-miss Recognition and Symbolic Evaluation.

Rudi Lutz
Cognitive Studies Programme
University of Sussex
Brighton
England

Debugging accounts for at least 25% of the total time involved in the programming process [1] and thus accounts for a highly significant fraction of the cost of software projects. My aim is to build intelligent tools to aid reasonably expert programmers with the task of debugging logical errors, and this could form one component of a larger "programmer's apprentice"-like system [9]. Most other systems attacking this problem have used either "cliche" recognition [e.g. 8] or symbolic evaluation [6]. The problem with the use of "cliche recognition" alone is that it leaves the system unable to deal well with pieces of code which do not correspond to any of the system's known cliches (plans). Symbolic evaluation techniques can deal with unknown code but the resulting expressions describing the program tend to be unmanageably large. The project described here hopes to show that a combination of these techniques can result in a useable general purpose debugging system, and indeed a similar viewpoint underlies Laubsch and Eisenstadt's debugging system for novice SOLO programmers [3, 5].

An overview of the system currently being implemented is shown in Figure 1. The main point about this is that the evaluation and recognition modules do not work on the source code of the program but on an internal representation of it, known as a surface plan. This is to maintain language independence so that, although this project is

currently intended as a tool for Pascal programmers, the system should be easy to modify for other languages by writing a new front- and back-end (to read programs in a given language, and to output a debugged program). The surface plan representation chosen for this project is that developed by Rich, Schrobe, and Waters for the MIT "Programmer's Apprentice" project and has also been used by Laubsch and Eisenstadt [5, 3].

A surface plan is a representation of the program in terms of its control and data flow, and can be thought of as a graph consisting of boxes joined by arcs representing the control and data flow within the program. Each box can be thought of as a processor which is activated when all of its input control- and data-flow lines are active, and which then produces the appropriate outputs on its output lines. Such processors can be grouped together to form higher level processes, which can in turn form part of other processes. Figure 2 shows part of such a graph where the spiral line indicates recursive nesting. More information on this can be found in [8]. Internally these control- and data-flow graphs are represented by frame-like structures similar to those in [10]. Each basic operation (and constant) is represented by a frame, as are more complex segments built up out of the primitive operations and other segments.

Plans in the library are also represented in the same formalism. However in addition to the control- and data-flow information each plan also has associated with it the following:

1) Preconditions:- conditions that must be satisfied by the inputs to a plan for a use of it to be valid.

2) Postconditions:-conditions satisfied by the outputs of a plan given that its preconditions have been satisfied.

3) "Canned text" for interacting with the programmer.

4) "Canned text" for generating explanations, comments, etc.

Using this representation for programs and library plans the system's strategy is:

(a) Translate the program into its surface plan.

(b) Recognise all occurrences of library plans. Make a note of any "near" matches.

(c) Symbolically evaluate any remaining bits of code.

(d) Check for broken preconditions of any of the recognised plans.

(e) Use "correspondences" between near matches and broken preconditions, and/or answers to questions put to the programmer to attempt to fix bugs.

(f) Translate the debugged surface plan back into the source language.

The current front-end translator from Pascal into the surface plan formalism can deal with a large subset of Pascal, including the structured looping constructs, conditional statements, assignment statements, procedures and functions, and data structures involving records and pointers. The current implementation works by a single-pass recursive descent method, maintaining a table stating where each variable was last updated. When a variable's value is used the system can then generate a dataflow arc from where that variable was last updated to the current consumer of that value. However because the system is single-pass it cannot at the moment translate recursive or forwardly defined procedures but it is currently being extended to use the techniques of [4].

Cliche recognition is a hard problem. Since the cliches are also held in the form of control- and data-flow graphs the problem reduces to that of finding occurrences of one graph (the cliche) in another (the program). However because the programs the system is looking at are allowed to have bugs in them an exact match to a given cliche may not be present. Accordingly, the system has to solve a

variant of the maximal common subgraph problem. In this particular application there is a complication in that parts of the program may belong to more than one cliche (i. e. they fill more than one role). The current version of the graph matcher is a variant of an algorithm due to McGregor [7] and is essentially a best-first search. It begins by assuming that any arc can match any other arc (subject to restrictions on type of source and destination node) and then as nodes are tentatively paired, the possible matches become increasingly restricted. In the longer term it is hoped that it will be possible to use web grammars [11] to parse these graphs (perhaps using a modification of Earley's algorithm [2] for parsing context free grammars) and thus achieve more efficient matching by lessening the need for searching.

Symbolic evaluation of surface plans is done by allowing evaluation effects to propagate themselves throughout the surface plan's frame system by means of if-added demons attached to the generic frame representing each of the primitive actions known to the system.

An Example

Consider the Pascal program fragment shown below:

```
while not eof do
   begin
     while not eoln do
      begin
       read(n);
       findplace(n, p);
       addtolist(n, p);
     end;
     readln;
   end;
```

This is part of a Pascal program to sort a set of numbers held in a file by reading each one in turn and inserting it into a linked list (initially empty). Procedure findplace finds the item after which

the new number is to be inserted, and procedure addtolist is supposed to do the inserting.

Although this sorting method is standard enough to be in the system's library of "cliches", it is interesting to see how well the system can cope if the plan for this is not in the library since it is not realistic to expect the system to have a plan for every possible program. However it is assumed that the system does have knowledge of various sub-plans of the program.

Now suppose that addtolist does its pointer manipulations in the wrong order, resulting in the behaviour shown in Figure 3. How does the system find this bug? The system begins by recognising everything it can about the program. In particular it recognises that findplace is an implementation of its library plan for finding the first item in a list satisfying some property (in this case being larger than the number just read in). During this recognition phase it also notes that part of procedure addtolist is quite close to the standard plan for splicing in a new element to a list. Of course this does not neccessarily mean there is a bug — the near match could just be a coincidence.

After the recognition process has been completed the system symbolically evaluates any unrecognised fragments of code. In particular it notes that the effect of the unrecognised part of addtolist can be described by:

Successor(Successor(P))=Successor(P). . . . . .Relation (1)

where P is the node after which the new item is to be inserted.

After symbolic evaluation the system attempts to verify the preconditions for each of the componenents of the program. From the precondition slot of the library plan corresponding to findplace the system retrieves the condition that the list input to findplace must

be a thread i. e. it must have a directed graph structure with no cycles and unique first and last nodes. This is clearly true at point A in Figure 2, but because of the recursive nature of the graph it must also be true at point B since this is the input to the call of findplace in the recursive invocation of the plan. However relation (1) implies that this input does have a cycle and thus violates the precondition.

There is thus an inconsistency between the output of addtolist and its subsequent use as input to findplace in the next loop cycle, and this would not have been detected using cliche recognition or symbolic evaluation on their own. This inconsistency between parts of the program indicates a domain-independent bug and the system can now attempt to repair it. For this particular bug the system notes that if the near match to a splicing plan were actually a splicing plan the precondition would not have been broken and the system can use this conjunction of a broken precondition and a near match to repair the program (possibly after asking the programmer for confirmation).

It is hoped that this combination of cliche recognition and symbolic evaluation will lead to the creation of powerful tools to aid programmers in the debugging task. A Pascal-to-surface plan translator and a first version of the graph matcher have already been implemented, and work is currently in progress on the symbolic evaluation module. In the future it is hoped that such things as meaningful variable name analysis (to facilitate indexing into the relevant parts of the plan library), and the ability to reason backwards from incorrect output to the error can be incorporated leading to a tool which people will find both helpful and natural to use.

References

1) Delaney W. A. Predicting the Costs of Computer Programs. Data Processing Magazine 32. (1966)

2) Earley J. An Efficient Context-free Parsing Algorithm. CACM 6:8, 451-455 (1970)

3) Eisenstadt M. , Laubsch J. Domain Specific Debugging Aids for Novice Programmers. Proc. 7th Int. Joint Conf. on Artificial Intelligence (IJCAI-81). Vancouver BC Canada. (1981).

4) Hecht M. S. Flow Analysis of Computer Programs. Elsevier North-Holland, Inc. New York (1977)

5) Laubsch J. , Eisenstadt M. Towards an Automated Debugging Assistant for Novice Programmers Proc. Artificial Intelligence and Simulated Behavior Conference Amsterdam (1980)

6) King J. C. Symbolic Execution and Program Testing. CACM 19:7 July (1976)

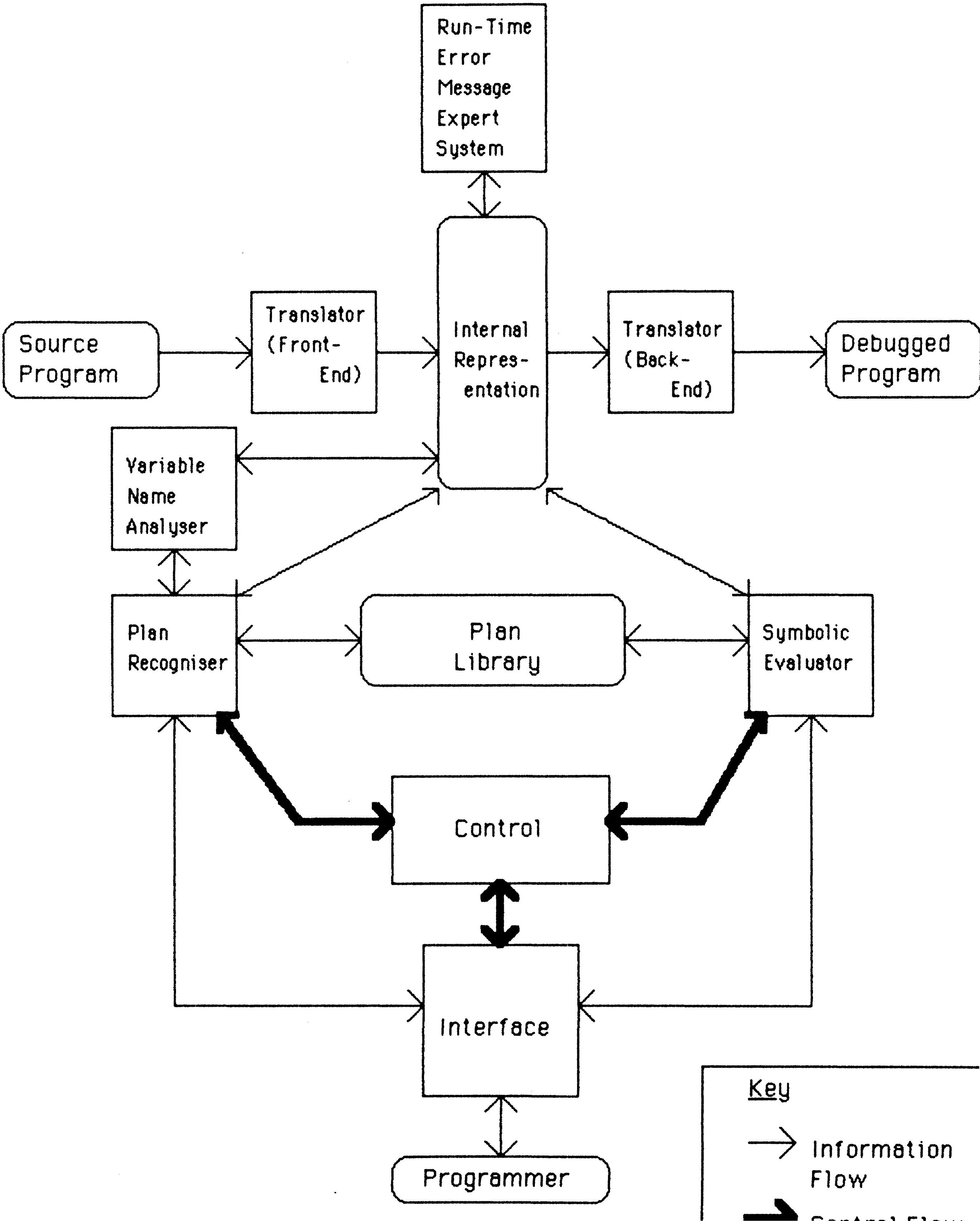
7) McGregor J. J. Backtrack Search Algorithms and the Maximal Common Subgraph Problem. Software-Practice and Experience 12, 23-24 (1982).

8) Rich C. Inspection Methods in Programming M. I. T. Artificial Intelligence Laboratory AI-TR-604 June 1981

9) Rich C. , Shrobe H. Initial Report on a LISP Programmers' Apprentice. IEEE transactions on Software Engineering. SE-4:6 1978. pp. 450-467

10) Goldstein I.P. and Roberts R.B. NUDGE, A Knowledge-Based Scheduling Program. Proc. 5th Int. Joint Conf. on Artificial Intelligence (IJCAI-77). Cambridge Massachusetts USA (1977).

11) Rosenfeld A. and Milgram D. Web Automata and Web Grammars. Machine Intelligence 7, 307-324 (1972).


Run-Time Error Message Expert System
Source Program
Translator (Front-End)
Internal Repres-entation
Translator (Back-End)
Debugged Program
Variable Name Analyser
Plan Recogniser
Plan Library
Symbolic Evaluator
Control
Interface
Programmer
Key
Information Flow

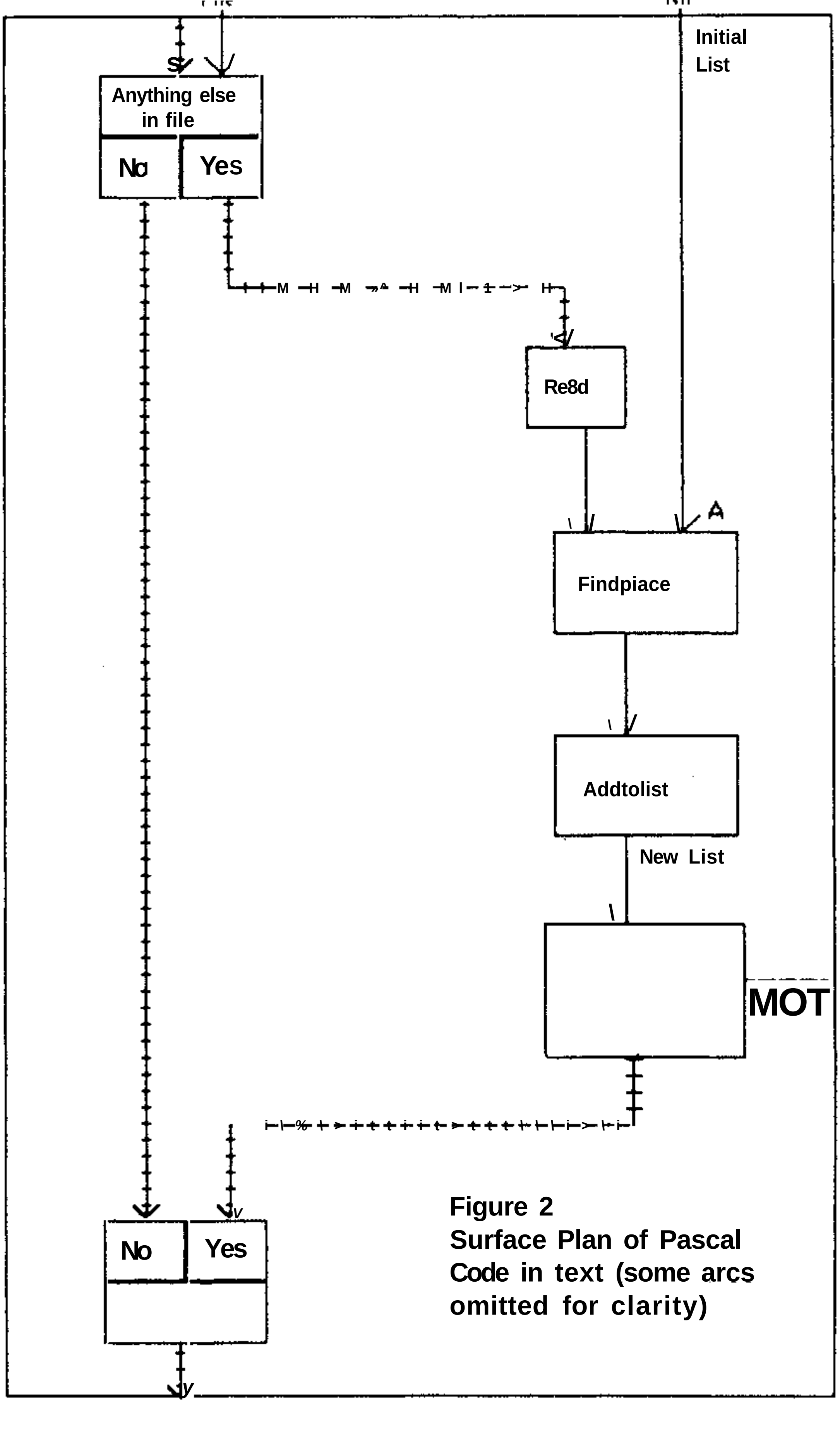
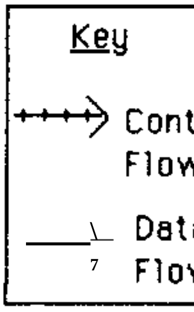


Figure 2
Surface Plan of Pascal Code in text (some arcs omitted for clarity)



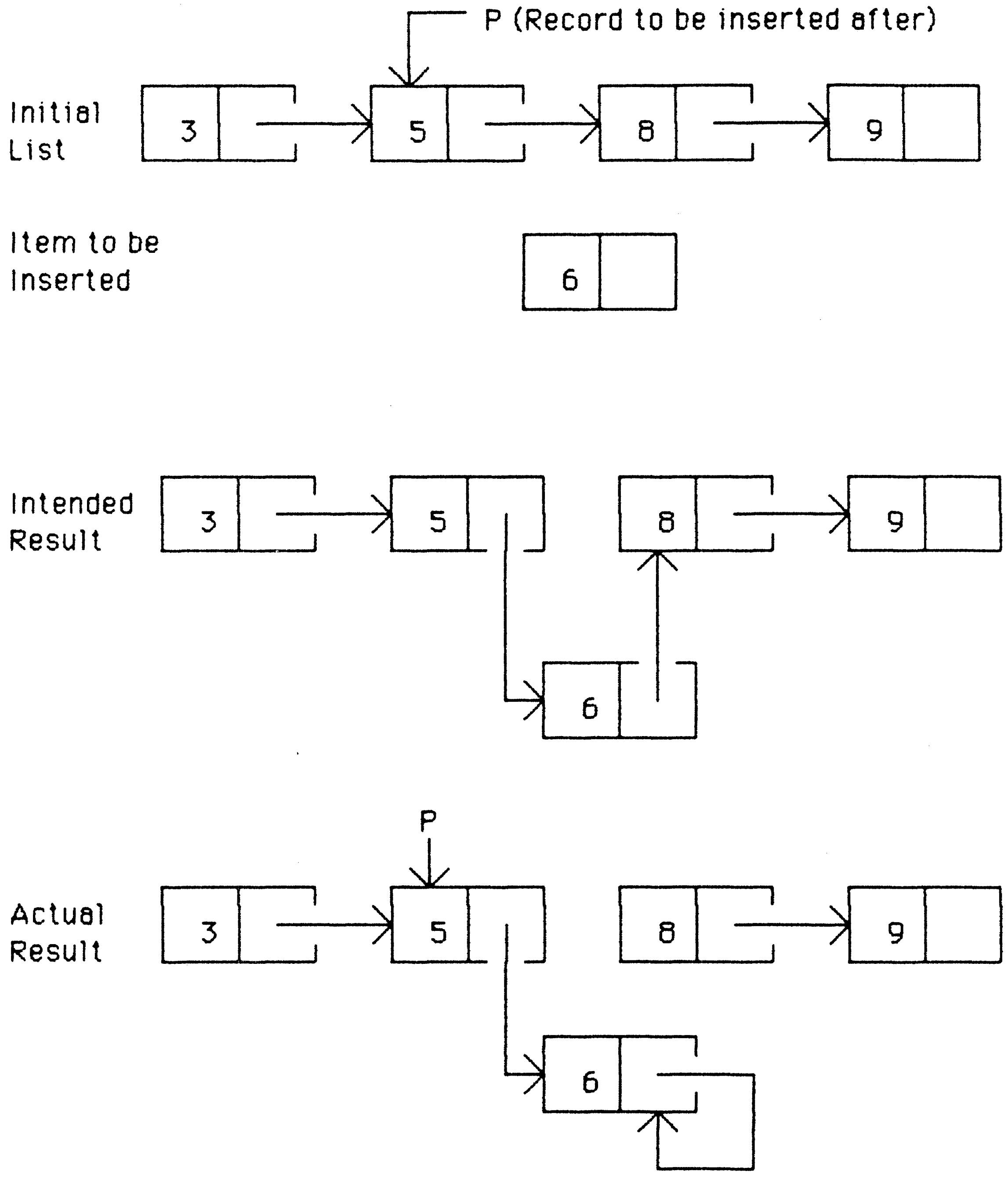

P (Record to be inserted after)
Initial List
3
5
8
9
Item to be Inserted
6
Intended Result
3
5
8
9
6
P
Actual Result
3
5
8
9
6

Figure 3