| | |
|---|---|
| AUTHOR | Davis, Alan; And Others |
| TITLE | Interactive Error Diagnostics for an Instructional Programming System. |
| INSTITUTION | Illinois Univ., Urbana. Dept. of Computer Science. |
| SPONS AGENCY | National Science Foundation, Washington, D.C. |
| PUB DATE | 74 |
| NOTE | 14p. |
| EDRS PRICE | MF-$0.75 HC-$1.50 PLUS POSTAGE |
| DESCRIPTORS | *Computer Assisted Instruction; *Computer Programs; *Computers; Computer Science; *Computer Science Education; *Programed Tutoring; Tutorial Programs |
| IDENTIFIERS | Error Analysis; *PLATO IV; University Of Illinois |

ABSTRACT

The development of an interactive error analysis system for a highly interactive programing language compiler is explored. A project is underway at the University of Illinois to automate the teaching of elementary computer science programing language courses by utilizing the PLATO IV interactive computer system. One goal of the project is to provide a highly interactive compiler/interpreter that will allow beginning programers to write, debug, and run fairly simple programs using newly-learned language constructs. The error analysis and diagnostic routines for the interactive compiler are divided into two separate systems. The first is concerned exclusively with the syntactic and semantic errors associated with program preparation and entry into the compiler. The second error system is concerned with the execution of the student's program and the error analysis and debugging that is initiated by the detection of an execution error. (WH)

# Interactive Error Diagnostics for an Instructional Programming System*

Alan Davis
Michael H. Tindall
Thomas R. Wilcox

Department of Computer Science
University of Illinois
Urbana, Illinois

## Introduction

This paper is concerned with the development of an interactive error analysis system for a highly interactive programming language compiler. Although the proposed system should have reasonably wide applicability to various time-sharing systems and interactive compilers, the following is a description of the goals and constraints of a particular compiler in which the error system is to be implemented.

A project is underway at the University of Illinois at Urbana-Champaign to automate the teaching of elementary computer science programming language courses by utilizing the PLATO IV interactive computer system [2]. One goal of this project is to provide a highly interactive compiler/interpreter that will allow beginning programmers to write, debug, and run fairly simple programs using newly-learned language constructs [4]. A requirement for the compiler is that it be able to parse and diagnose subsets of different programming languages (currently, PL/I, FORTRAN, BASIC, COBOL) with a minimal amount of redesign required. To change to a new

* This work is supported in part by the National Science Foundation under Grant No. US-NSF-EC-41511, and by IBM Corporation.

language, only the new lexical and syntactical tables must be redesigned (a relatively straight-forward process); the bulk of the compiler's on-line editing capabilities and internal structure can remain unchanged.

The overwhelming goal and emphasis for this compiler is that it behave like a "consultant", that is, be able to diagnose programming errors and help the student understand his errors. A good description of this "consultant" or "tutor" capability is that of a diagnostic system that points out different interpretations or ways of correcting an error situation, e.g., a diagnostic "prompter". The goal is not to have the system attempt to "recover" from the student's errors, but to interactively inform the student of an error and attempt to prompt him with suggestions about ways of correcting the error, requiring the student to actually analyze the situation and repair the program.

The error analysis and diagnostic routines for our interactive compiler are divided into two separate systems. The first is concerned exclusively with the syntactic and semantic errors associated with program preparation and entry into the compiler [3]. The second error system is concerned with the execution of the student's program and the error analysis and debugging that is initiated by the detection of an execution error [1]. The remainder of this paper will discuss these two error analysis systems.

## The Compile-time Error Analysis and Diagnostic System

The environment in which the compile-time error analysis system is invoked is the following. As the student enters the program, each word or "token" that is typed in is examined immediately by the syntax parser. As soon as an error is detected, the error routine is entered; this routine attempts to explain the error to the student, as described later in this paper. When the student understands the error, he must back up in the input string and repair the error appropriately; he is then allowed to continue entering the remainder of the program.

The overall philosophy of the approach developed for this error system is that effective error diagnostics can be given by informing the student about the current internal state of the compiler. Of interest to the student might be symbol table information, such as the attributes of a variable or whether or not a particular word is a reserved (pre-defined) word; dope vector information about defined arrays in the program; and syntactic requirements, such as what symbols the compiler is able to accept at a particular place in the user's program. The assumption is that if the compiler's current internal state can be adequately understood by the student, then the syntactic error (or semantic error, such as trying to re-declare an identifier in PL/I) will become apparent and the student can correct the program in the proper manner.

The problem in trying to implement this philosophy is that it is virtually impossible to know exactly what language construct the student thought he was using when an error is detected. Different parts of the compiler's state information may be important to the student in different error situations; also, different students in an identical error situation may very well need different information to understand their error. In an effort to provide the proper information for most situations, this paper suggests an interactive diagnostic system that tries to present to the student "intelligent guesses" as to the cause of the error. These "guesses" can be determined by examining the current input string, the syntactic requirements, the symbol table, and any other information comprising the state of the compiler; any discrepancies found in the state information can be reported to the student as a possible reason for the error (note that there will always be at least one discrepancy, since an error was originally detected). A given error situation may cause three of four (or more) system-generated "guesses" to be presented; the hope is that at least one of these "guesses" will be close enough to the actual error to reveal the error to the student.

It is instructive to consider a few examples of syntactic/semantic error situations and some appropriate diagnostic messages that are generated. All the examples are taken from the PL/I language, which has been used for our first implementation of this diagnostic system. Work is in progress to extend the system to FORTRAN. Since the compiler is intended for elementary programmers, only a subset of the most common features are accepted (in particular, for these examples the conditional expression in an "IF" statement is assumed to be a relational-type expression).

Example 1:

Assume the following string is input to the compiler:

IF A - B/C THEN

An initial diagnostic message is given:

IF A - B/C $\boxed{\text{THEN}}$

************Possible Correction************

Replace "THEN" with an arithmetic operator.

If the student requests further analysis, another message is:

IF A - B/C $\boxed{\text{THEN}}$

************Possible Correction************

Replace "THEN" with a relational operator.

A further request for analysis would yield:

IF $\boxed{\text{A - B/C}}$ THEN

************Possible Correction************

Replace this arithmetic expression with a conditional expression.

Example 2:

Assume the following string is input to the compiler:

I = J * 10 TO

One message that is given is:

I = J * 10 $\boxed{\text{TO}}$

************Possible Correction************

Replace "TO" with an arithmetic operator or a ";".

Another message is:

$\boxed{\text{I}}$ = J * 10 TO

************Possible Correction************

Insert "DO" in front of the identifier "I".

From these examples it is apparent that the error system is trying to give the student well-directed hints about possible causes for the error situation. Note also that these messages first make suggestions about possible "local" corrections to the program, assuming that everything that was input before the token at which the error was detected is correct. If the student requests further suggestions, then the error analysis system considers corrections of text to the left of the error token. Besides being perhaps the most natural order of analysis, this method also is good from a computational point of view: as more and more left context is considered, the amount of computing required to determine a possible suggestion increases rapidly; the hope is that the first few initial messages will cause the student to discover the error, and that the more expensive suggestions, while always available when needed, will not be required much of the time.

Consider another example:

Example 3:

Assume that the following declaration is made at the beginning of the program:

```
DECLARE    A(10), B;
```

Then suppose that the student later types in:

```
B = A ;
```

One message that is given is:

$B = A\ \boxed{;}$

```
************Possible Correction************
Insert the array subscripts in front of the ";".
```

Another message is:

$B = \boxed{A};$

************Possible Correction************

Replace array reference "A" with an ordinary identifier.

This example illustrates another feature of the diagnostic system. The analysis routines and corresponding diagnostic messages are able to manipulate and display information about important language constructs, such as "array subscripts", "expressions", "statements", etc. These high-level constructs are em[illegible]zed during the teaching of a programming language and should be referred to at appropriate times by the diagnostic system. It is essential that the diagnostic system be capable of making suggestions using terminology that is understandable to the student.

One final requirement for the compile-time diagnostic system is that as much of this "prompter" as is possible be built into the compiler and provided automatically for different programming languages. The key to achieving this is to have the diagnostic system be driven by the syntax parser table and compiler symbol tables. Then, constructing compiler parser and symbol tables for a new language will be sufficient to allow the proper operation of the interactive diagnostic prompter.

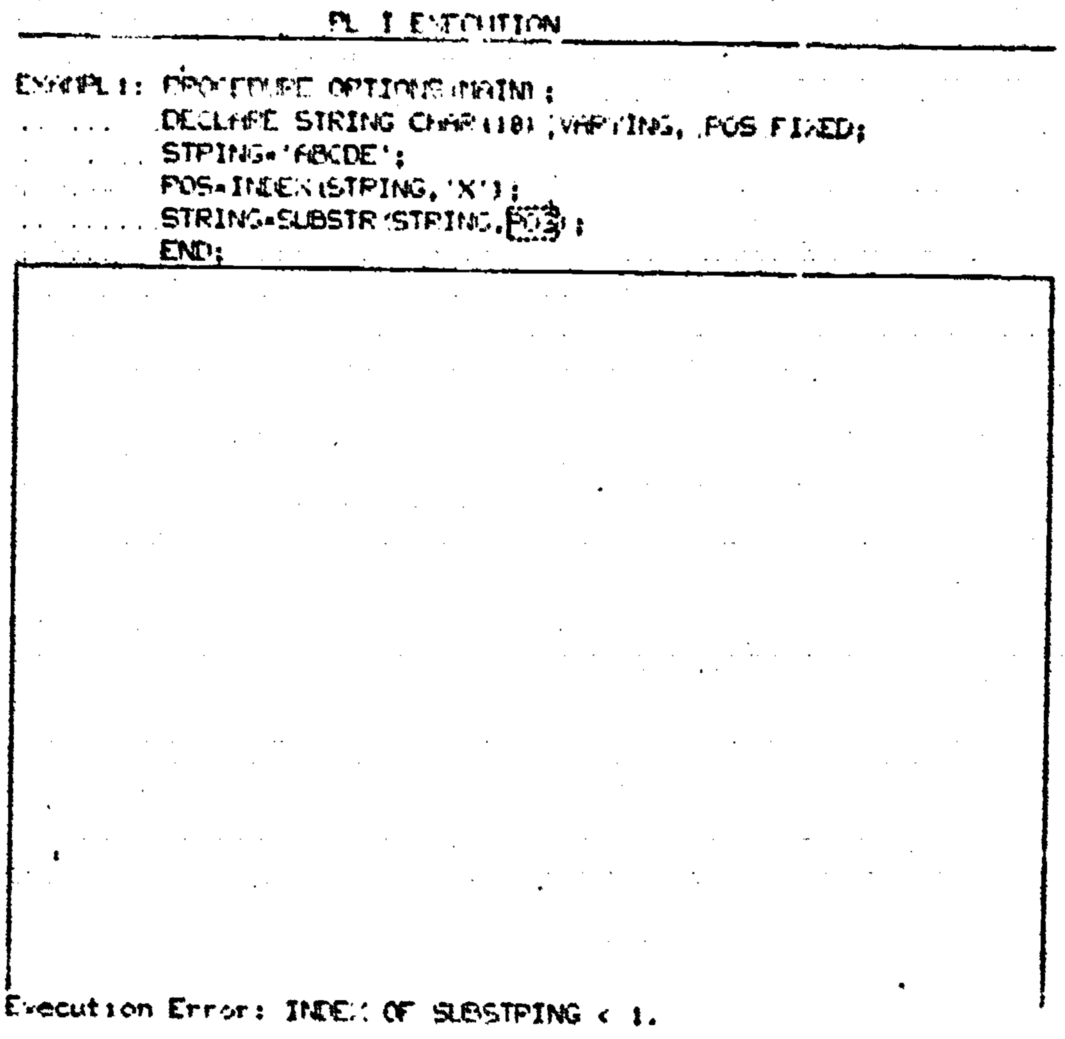
Let's now examine the error analysis and diagnostic system that is used to assist the student with program execution errors.

## Run-time Error Analysis

For program execution, it is assumed that the program being given to the execution package is a syntactically correct program for the language being written. This is insured by the process of immediate detection and analysis of syntax errors as described in the previous section. The run-time supervisor is an interpreter which interprets a tokenized version of the original source language program. During interpretation of each statement, error conditions are tested and, if located, an appropriate error message is displayed to the student. If the student requests help, control then passes to the execution-time error analysis program.

The error analysis program scans the student's program looking for conditions which it knows are common causes of errors. Note the distinction between an error and a cause of an error. An execution error is any illegal condition which the interpreter can find by examining various data structures in the run-time environment (e.g., subscript out of range). The cause of an error is the mistake in logic or in the use of a programming language feature that resulted in that error (e.g., improper declaration of an array). The scan for these causes is done by reverse executing the program, statement by statement, starting at the position of the error. This reverse execution insures that the analysis routines are examining each statement in the same execution environment in which it was executed during forward execution.

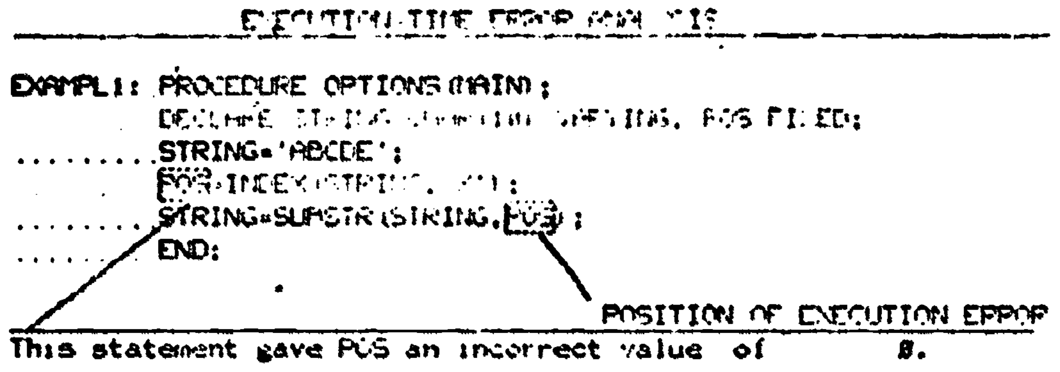
When a potential cause for an error is located, a discussion with the student is initiated concerning that condition. For example, when the following simple PL/1 program is executed, the indicated error is detected:



```
                    PL I EXECUTION
EXAMPL1: PROCEDURE OPTIONS(MAIN);
.. ...   DECLARE STRING CHAR(10) VARYING, POS FIXED;
   . ..  STRING='ABCDE';
         POS=INDEX(STRING,'X');
.. ..... STRING=SUBSTR(STRING,POS);
         END;

Execution Error: INDEX OF SUBSTRING < 1.
```

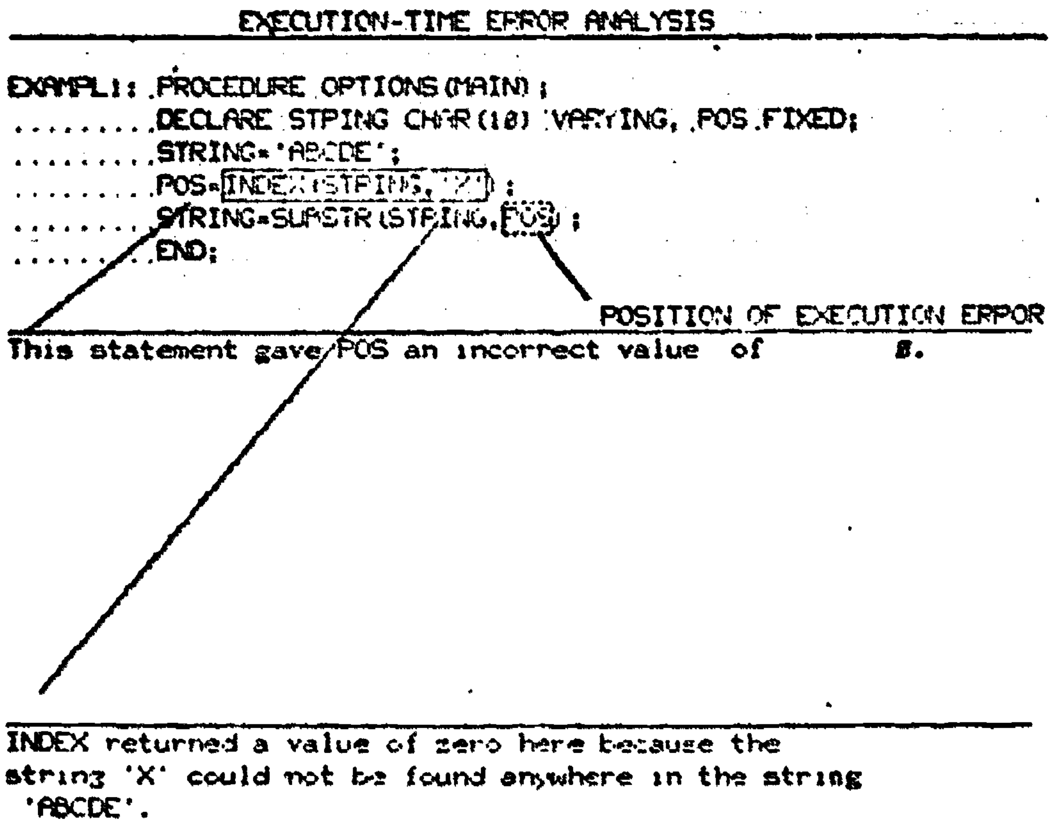
If the student types HELP, then the error analysis system commences its reverse execution. When a statement is located which is relevant to the error in question, that statement if flagged and brought to the attention of the student:



```
               EXECUTION-TIME ERROR ANALYSIS
EXAMPL1: PROCEDURE OPTIONS(MAIN);
         DECLARE STRING CHAR(10) VARYING, POS FIXED;
.........STRING='ABCDE';
         POS=INDEX(STRING,'X');
.........STRING=SUBSTR(STRING,POS);
.......  END;

                              POSITION OF EXECUTION ERROR
This statement gave POS an incorrect value of       0.
```

Next, that statement is examined for possible conditions which might have caused the error. In this particular case, a built-in function, INDEX, has returned a special-case result. This fact, if described to the student may help the student to understand why his error has occurred:



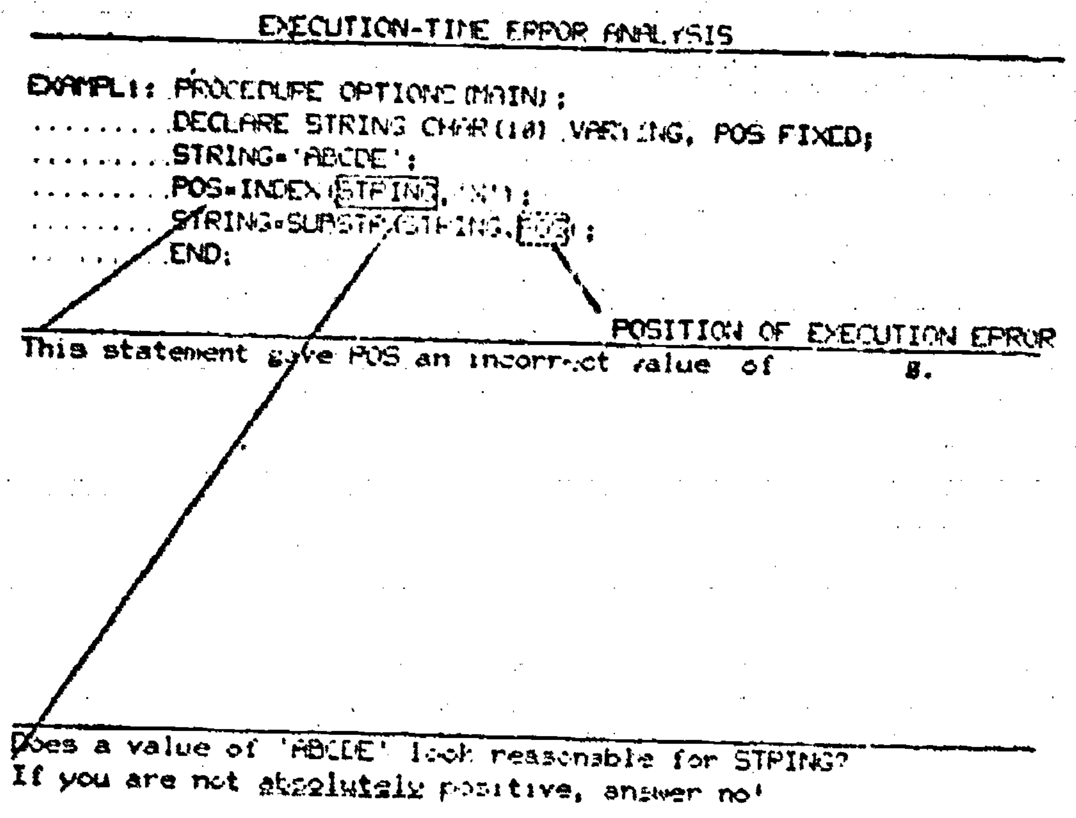
```
                EXECUTION-TIME ERROR ANALYSIS

EXAMPL1: .PROCEDURE OPTIONS(MAIN);
.........DECLARE STRING CHAR(10) VARYING, .POS.FIXED;
.........STRING='ABCDE';
.........POS=INDEX(STRING,'X');
.........STRING=SUBSTR(STRING,POS);
.........END;

                                 POSITION OF EXECUTION ERROR
This statement gave POS an incorrect value  of         0.

INDEX returned a value of zero here because the
string 'X' could not be found anywhere in the string
 'ABCDE'.
```

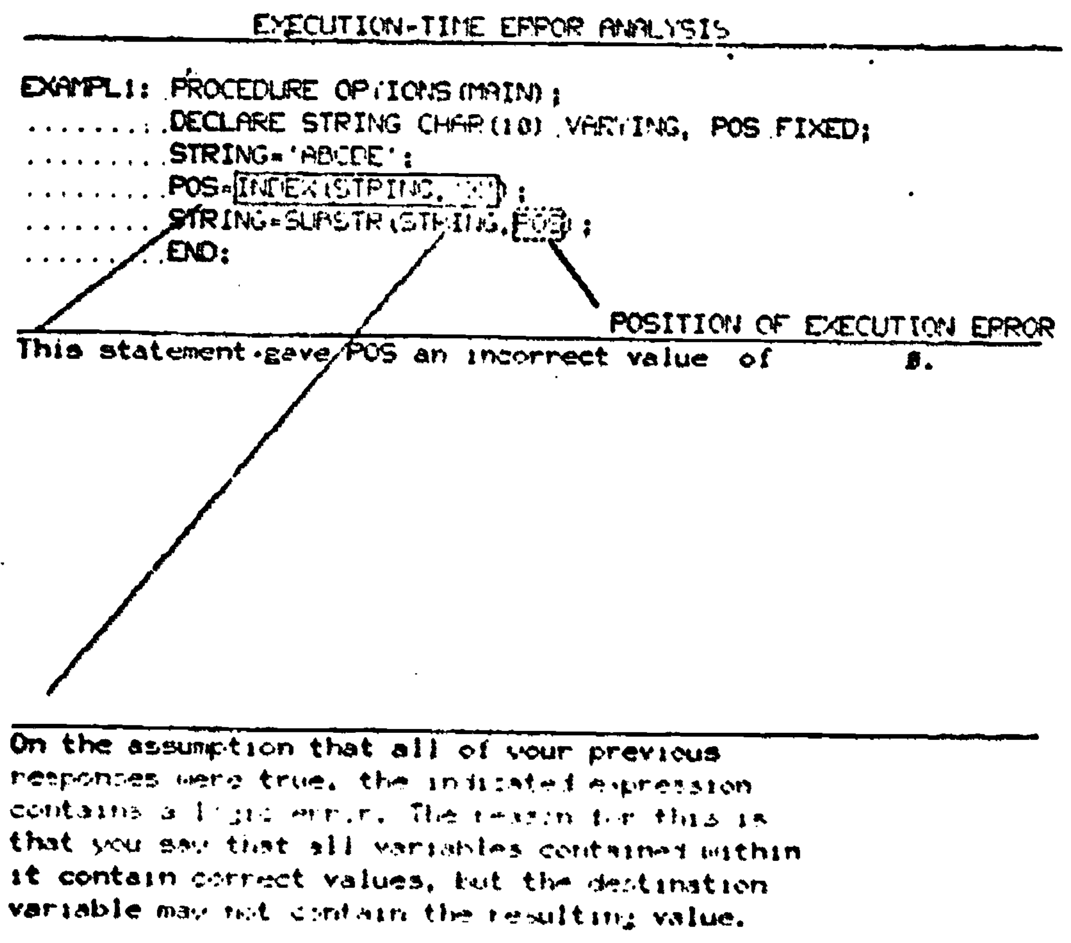
If the student now understands what caused his error, he may choose to edit and re-execute his program. If he chooses to request more help, the values of the operands involved are discussed and the student informs the analysis program whether they seem reasonable of not:



```
                    EXECUTION-TIME ERROR ANALYSIS
EXAMPL1: PROCEDURE OPTIONS(MAIN);
........DECLARE STRING CHAR(10) VARYING, POS FIXED;
........STRING='ABCDE';
........POS=INDEX(STRING,'X');
........STRING=SUBSTR(STRING,POS);
........END;
                                     POSITION OF EXECUTION ERROR
This statement gave POS an incorrect value of       0.

Does a value of 'ABCDE' look reasonable for STRING?
If you are not absolutely positive, answer no!
```

If the student now responds by saying that all the values are indeed correct then the present expression being examined must be incorrect:



```
                    EXECUTION-TIME ERROR ANALYSIS
EXAMPL1: PROCEDURE OPTIONS(MAIN);
........DECLARE STRING CHAR(10) VARYING, POS FIXED;
........STRING='ABCDE';
........POS=INDEX(STRING,'X');
........STRING=SUBSTR(STRING,POS);
........END;
                                     POSITION OF EXECUTION ERROR
This statement gave POS an incorrect value of       0.

On the assumption that all of your previous
responses were true, the indicated expression
contains a logic error. The reason for this is
that you saw that all variables contained within
it contain correct values, but the destination
variable may not contain the resulting value.
```

If the student had responded by saying that some of the variables had questionable values, then error analysis would continue in the reverse direction looking for statements which effect those new variables.

It is hoped that this simple example demonstrates the type of analysis performed. By giving the student numerous intelligent comments about positions in his program which are relevant to the execution error, it is hoped that the student will comprehend why his error occurred. As during compile-time error analysis, numerous suggestions are made to the students, but it is they who must decide how to correct their programs since only they can know what their programs were supposed to do.

## Conclusion

In summary, this paper has discussed an approach to program error analysis that emphasizes the activities of a programming language "tutor" or "consultant". When a compile-time syntactic error is detected, the diagnostic system tries to "prompt" the student into analyzing the error situation by giving the student hints in the form of "possible corrections" to the program. For an execution error, the diagnostic system tries to direct the student in analyzing the error situation by showing how various relevant sections of the program were actually executed.

At the present time, the two diagnostic systems described have been implemented in prototype form in the interactive compiler on the PLATO IV computer-aided instruction system. Work is in progress to refine the initial versions and to provide more complete terminology for communicating with the student.

## REFERENCES

[1] Davis, A. M., "An Analysis System For Execution-Time Errors," Ph.D. Thesis. Department of Computer Science, University of Illinois at Urbana-Champaign, Jan. 1975.

[2] Nievergelt. J., Reingold, E. M., and Wilcox, T. R., "The Automation of Introductory Computer Science Courses," A. Gunther et al. (editors), International Computing Symposium 1973, North-Holland Publishing Co., 1974.

[3] Tindall, M. H., "Interactive, Table-Driven Compiler Error Analysis," forthcoming Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.

[4] Wilcox, T. R., "The Interactive Compiler As A Consultant In The Computer Aided Instruction of Programming," 7th Annual Princeton Conference on Information Sciences and Systems, 1973.