MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Artificial Intelligence Project
Memo No. 158

MAC-M-369
March 1968

A Program for Drilling Students in Freshman Calculus Integration Problems

Joel Moses

Abstract

The SARGE program is a prototype of a program which is intended to be used as an adjunct to regular classroom work in freshman calculus. Using SARGE, students can type their step-by-step solution to an indefinite integration problem, and can have the correctness of their solutions determined by the system. The syntax for these steps comes quite close to normal mathematical notation, given the limitations of typewriter input. The method of solution is pretty much unrestricted as long as no mistakes are made along the way. If a mistake is made, SARGE will catch it and yield an error message. The student may modify the incorrect step, or he may ask the program for advice on how the mistake arose by typing "help". At present the program is weak in generating explanations for mistakes. Sometimes the "help" mechanism will just yield a response which will indicate the way in which the erroneous step can be corrected. In order to improve the explanation mechanism one would need a sophisticated analysis of students' solutions to homework or quiz problems. Experience with the behavior of students with SARGE, which is nil at present, should also help in accomplishing this goal.

SARGE is available as SARGE SAVED in T302 2517.

-

Introduction

Our aim in this project was to use the programs available at M.I.T. in the area of algebraic manipulation in order to produce a sophisticated teaching program for freshman calculus students. As a matter of fact, we were not interested in teaching symbolic or indefinite integration as such, since the normal courses performed this function quite well. We were interested in a program which could be used to supplement regular instruction with supervised practice sessions. Normally, human teaching assistants would be used for this purpose. Thus our goal can be said to be a simulation of the interaction of a student and a teaching assistant. In order to capture certain aspects of such an interaction two major design considerations were imposed.

1) Freshman calculus students are not computer programmers. Therefore the communication with the teaching system should be as natural as possible.

2) There should be no impediment to unusual solutions of a problem. Such unusual solutions include ingenious solutions as well as solutions with superfluous or erroneous steps. A student must be capable of giving the complete answer at any step. This capability will guarantee that gifted students who can solve the problem in their heads will not be forced to give a full step-by-step solution of a problem. Likewise a student who makes a false attempt* at solving a problem must be allowed to backtrack and attempt another path to a solution.

*We wish to distinguish between false attempts and incorrect attempts. In

$$\int x^2 \, dx,$$

the substitution $y = x$ is bad even though the result of this substitution, i.e.,

$$\int \frac{1}{5} y^{-2/5} \, dy$$

is given correctly. In the same problem the substitution $y = x^2$ is incorrect when it results in

$$\int \frac{1}{2} y \, dy.$$

The SARGE program meets the above requirements fairly well. Below we shall describe the manner in which one communicates a solution to SARGE. We will also indicate how the system operates. Later we shall criticize the present version of the system and present suggestions for its improvement.

Communicating with SARGE

We will present the syntax of the language that the student must use to communicate a solution to SARGE through an interaction between an imaginary student and the program (see figure 1). Let us suppose that the problem to be integrated is

$$\int (x + 1)\ (x + 2)\ dx.$$

This problem would be typed by SARGE as follows:

```
    0    PROBLEM INT (X + 1) (X + 2) DX
```

The above line is representative of the statements in SARGE's language. All statements in the language have the following form:

line-number   command   (command argument)   (expression)

(The latter two components of a statement have been parenthesized to indicate that they need not be present in each statement.)

The first component of an input line is an unsigned integer which is taken as a line number. Line numbers are chosen by the student, except that the problem is always presented as line 0. Line numbers should be distinct, and are used to identify steps in a solution.

Commands are used to indicate how the expression arose in the solution process. The current set of commands is PROBLEM, SUBST, TRANSF, ANS, PARTOF, and RETURNTO. The PROBLEM command is used only in defining a new problem, and may not be used later on in the solution. The PROBLEM command has no command argument. Thus the next component of the line is taken to be an expression. Line 0 indicates some interesting features of the expression syntax. For example, concatenation implies multiplication except when numbers are involved. This is the standard mathematical convention, and it forces the convention that all variables are single characters. Furthermore, an attempt was made to keep the usual syntax of the indefinite integral. Thus

$\int$ f(x) dx becomes int f(x) dx.

In line 1 the command used is TRANSF.

```
1    transf int x$2 + 3x +2 dx
```

This command is used when the expression is derived from the previous line by an equivalence preserving transformation. Line 1 is derived from line 0 by an expansion. Note that no justification need be given for the equivalence.

In line 1 we note that the dollar sign signifies exponentiation. The convention regarding the dollar sign is that if the exponent or base are single syntactic units (unsigned numbers or alphabetic characters) then no parentheses are required. If more complex bases or exponents are used (e.g., x+1), then parentheses must be used. This commonly-used convention applies also to the slash which signifies division.

The first statement which has the line number 2 is a transformation which involves certain identities regarding the integration operator.

```
2    transf  int x$2dx + int 2x dx + int 2dx
```

SARGE is aware of these linearity identities and moreover recognizes an error in transcription. This is noted by SARGE's next response.

```
AN ERROR OCCURRED AFTER LINE 1 WAS ANALYZED
```

An examination of figure 1 will indicate that SARGE's response up to now was a number which equaled the line number of the previous statement.

This is called the *slow* mode of interaction. In the *fast* mode, SARGE will make no responses unless an error occurs. The normal mode is slow. A change in mode can be made by executing SLOW NIL or FAST NIL prior to working on a problem. FAST NIL will cause the mode to change to fast.

After indicating that an error occurred the system anticipates a new set of key words. The keyword 'help' which is used in this case causes SARGE to examine a list which contains probable explanations for the error. Clearly the help mechanism should be used only as a last resort by the student. Frequently SARGE has no reasonable idea to transmit to the user, in which case the response is

```
          SORRY, NO IDEA WHY ERROR OCCURRED.
```

In the present situation it can only offer the student a means for correcting his last step. It therefore types

```
          THE DIFFERENCE BETWEEN THE EXPECTED EXPRESSION AND YOURS IS
                                         X
```

Other keywords which are recognized following an error are 'quit' and 'eval'. The quit mechanism allows the user to stop working on the current problem. The eval mechanism, used during the debugging of SARGE, allows one to execute any LISP function available in the system. Any line which follows an error and which does not begin with these keywords is ignored.

The second occurrence of line 2 indicates that the student

```
          2    transf  int x$2dx + int 3x dx + int 2dx
```

has accepted the advice offerred by SARGE. In line 3 the user is preparing to tackle a subproblem of line 2. He identifies the part of the expression he wants to work on next with the PARTOF command. The command argument here is a line number.

```
          3    partof 2   int x$2dx
```

In line 4 the user is attempting to make a substitution on the subproblem selected in the previous line. The command

```
        4    subst y=x$2,  int y dy
```

argument is an equation relating the new variable to the old one. This operation is restricted to having only one occurrence of the new variable ('y' in this case). For syntactic purposes a comma must follow this equation and precede the expression of the integral. As it turns out the result of the substitution is erroneous. SARGE now prints its standard error comment. A call for help at this point yields a very appropriate reply.

```
        IN YOUR SUBSTITUTION YOU FORGOT TO DIVIDE
        BY THE DERIVATIVE OF THE TRANSFORMATION
```

The second line 4 shows a correction

```
        4    sub, y=x$2,  int 0.5 y$0.5 dy
```

of this error. We note a new syntactic device -- a comma following one or more of the left-most characters in a command can be used to abbreviate the command name.

Line 5 was included here as an example of the RETURNTO mechanism. This mechanism allows a student who is frustrated in his current attempt at a solution to continue from a previous line.

Our student is determined to solve this problem in line 3 (which admittedly is a trivial problem) by a substitution. This analysis explains the first line 6.

```
        6    subst y=x$(1/2), int y?
```

The question-mark is a CTSS device for deleting the input line. Actually an empty line is transmitted to SARGE and is ignored. In some situations a student may wish to stop processing of a solution in a more drastic fashion. This would occur if the student typed a number of lines in fast mode and recognized an error in them before the system had a chance to analyze the solution. An interrupt from the console would cause an error, and the message from SARGE will indicate the extent to which it analyzed the solution.

The second line 6 shows that our poor student has finally seen the light and recognized the solution of the subproblem.

```
6    ans 3  x$3/3
```

The ANS command has as its command argument the line number of the subproblem whose integral appears as the expression.

We are now approaching the end of the interaction. In line 7 we go back to line 2 to pick up another subproblem.

```
7    partof 2   3 int x dx
```

Line 8 contains the integral of that subproblem.

```
8    ans 7   3/2 x$2
```

Line 9 claims to be the solution to the original problem (line 0), which it is indeed. Note that 'line(n)' causes the expression of line n to be substituted in its position in the expression.

We give in figures 2 and 3 two further examples of interactions with SARGE. No new linguistic features are introduced except for the use of trigonometric functions in the obvious manner.

How does SARGE work?

From a certain point of view the SARGE program is quite trivial. It took us only two weeks to get most of the system running and in partially debugged shape. However the size of the program is quite large (about 7000 compiled instructions and an equivalent of 3000 interpreted ones in the usual 32K LISP 1.5 system on CTSS). As we indicated earlier we intended to rely on the large store of available LISP programs in the area of algebraic manipulation. The most important of these programs, for our purposes, was William Martin's equivalence matching routines (see 1, Chapter 4). This program determines whether two expressions are equivalent by a very clever use of hash coding. Martin's program is deadly in recognizing equivalence of rational functions or rational function of trigonometric functions. For example, this program allows SARGE to recognize the equivalence of 1/2 sin(2x) and sin(x) cos(x) in figure 2. This routine is currently weak in recognizing equivalence of expressions involving square-roots. One could, however, design special routines which would recognize such situations and overcome them in many instances. Should the matching routine fail to recognize an equivalence, then SARGE would claim that an error was made when, in fact, none was. Currently there exists no machinery which allows the user to argue against the decisions of SARGE. Though errors of this nature occur very rarely, it would be useful to give the user the capability of overcoming the decisions of the system.

Another one of Martin's programs is used for parsing the input expression into prefix notation. The input line is read, character by character, by one of our programs which transforms the expression somewhat before giving it to Martin's parsing routine (e.g., special attention must be paid to the integral sign syntax). The result of the parse of

```
int 2x+1 dx
```

is

(integral (plus (times 2 x) 1) x).

If the expression is a sum of integrals then it is transformed into an integral of a sum. Hence line 2 of figure 1 is stored internally exactly as line 1 is. This transformation simplifies the coding. It should be noted that the system currently requires that an expression reduce either to a simple integral or to an expression involving no integrals. Hence integration-by-parts is an illegal transformation. The changes necessary to overcome this restriction can be easily made.

It should be clear that, in addition to the routines mentioned above, we require a symbolic differentiation program (used, for example, in determining the correctness of a proposed answer), a simplification program (an important component of the substitution checking mechanism), and a "solve" program (also needed in substitutions). These routines were stolen from our work on SIN(2), the infamous symbolic integrator. For the operation of these routines we required the SCHATCHEN pattern matching program. The SCHATCHEN language allows one to express concepts like "a quadratic in x" quite concisely. We have not made much use of SCHATCHEN in the current SARGE system, although we expect that an improved version will find its presence quite useful.

The main routine in SARGE, also called SARGE, is a fairly straight-forward interpreter of the input lines. Whenever an error occurs a check is made to determine whether any rules exist about how to treat the input line. If rules exist and the input line fits the expected error condition, a note is made in a location called 'help'. Otherwise 'help' is unchanged.

The main program is executed interpretively during the operation of the system because there is no space available in which to compile it and because it changes so drastically. The three interactions described in figures 1 - 3 each require about twenty seconds of execution time and a similar amount of swapping time. Hence the answer to the question posed in the beginning of this section is -- SARGE works slowly. It is likely that a three-fold improvement in speed could be had if the entire system were compiled.

There are, at present, two modes in which a problem is proposed to the user. In the slave mode the program proposes one of a set of built-in problems. This mode is activated by typing 'SLAVE NIL'. In the master mode, the user decides the problem on which he wants to work. This mode is activated by typing 'MASTER NIL'. In either case, control reverts to LISP following an interaction.

Critique and Future Directions

Below we shall consider some criticisms of the present SARGE system.

1) The syntax of SARGE is one dimensional. It should be two dimensional. The use of a typewriter as the input medium hinders a natural interaction with the computer. A RAND Tablet or some similar device is clearly called for. Programs which parse mathematical expressions written on a RAND Tablet have been designed by Anderson (3) and Martin (4). Such programs should be operational soon. However, there are some difficulties with the use of a RAND Tablet for this application. The typewriter input forces on the user a line-by-line format. Given the freedom possible with a RAND Tablet, the user is likely to generate much input which does not conform to SARGE's syntax (e.g., side calculations for the determination of derivatives, doodles, etc.). This will make the parsing problem a good deal more complicated. Furthermore a user station which includes a scope and a RAND Tablet is a good deal more expensive and consumes more computer time than a station which simply has a typewriter console.

2) The system is too inefficient for practical use by a large body of students. This is principally due to the scavenger approach we adopted in designing the system. A specially designed version should run substantially faster.

3) The error diagnosing capability of the system is not very powerful. What appears to be necessary here is a sophisticated analysis of the mistakes that students frequently make in integration problems. Such research could proceed in several stages. At one stage one might determine a set of commonly made errors (such as the error in substitution of figures 1 and 3). These errors would be known to the program and it would determine when such errors account for the input expression. In a second step, one would analyze the interactions students have with such a program. This analysis could be used

to improve the diagnostic capability. Finally one would want the program to build models of the behavior of each student so that the problems which are posed and the diagnosis given by the system becomes individualized. These are non-trivial objectives which face many difficulties. One obvious source of difficulty is the intelligent student who will try to 'program' the system in order to see what it will do under certain situations. The system should be clever enough to recognize when it is being 'hacked'.

The current error comments are not at all subtle. It would be preferable if one could proceed in stages by using hints at first. If a script were available to the program, then such hints could be built into the script for each problem. We have tried to avoid the script-based approach as much as possible. We have depended on the knowledge that the program has of the material to help it in diagnoses. Such knowledge is usually lacking in script-based teaching systems. However, one way of extending the present system would involve the use of scripts. For example, one could include in a script several methods of solution and several common mistakes made on each particular problem. In order to keep the system as flexible as possible one would have to determine how far the student has progressed in following a given solution method, and how to return to such built-in solutions from the solution developed by the student. This problem, too, is hardly trivial.

In summary, it would seem to us that a marriage of the script-based approach and the 'knowledge-based' approach is preferable to either approach. However there are difficult problems in making such a marriage a viable proposition.

```
user          slave nil
computer       0 PROBLEM  INT (X+1)(X+2) DX
computer       0
user          1 transf   int x$2 +3x + 2 dx
computer       1
user          2 transf   int x$2 dx + int 2x dx + int2dx

computer        AN ERROR OCCURRED AFTER LINE 1 WAS ANALYZED

user          help

computer        THE DIFFERENCE BETWEEN THE EXPECTED EXPRESSION AND YOURS IS
computer                     X
computer       1
user          2 transf   int x$2 dx + int 3x dx + int 2 dx
computer       2
user          3 partof 2   int x$2 dx
computer       3
user          4 subst y=x$2,   int y dy

computer        AN ERROR OCCURRED AFTER LINE 3 WAS ANALYZED

user          help

computer        IN YOUR SUBSTITUTION YOU FORGOT TO DIVIDE BY THE DERIVATIVE
computer        OF THE TRANSFORMATION
computer       3
user          4 subst y=x$2,    int 0.5 y$0.5 dy
computer       4
user          5 ret,3
computer       5
user          6 subst y=x$(1/2),  int y?
user          6 ans 3   x$3/3
computer       6
user          7 partof 2   int  3 x dx
computer       7
user          8 ans7     1.5x$2
computer       8
user          9 ans 0    line(6) +line(8) +2x


computer      CORRECT
```

Figure 1

```
computer        0 PROBLEM INT SIN(X)COS(X) DX
computer        0
user           1 transf    int 0.5sin(2x)dx
computer        1
user           2 transf   0.5 int sin(2x) dx
computer        2
user           3 subst y=2x,   1/2 int 1/2 sin(y) dy
computer        3
user           4 ans3  (1/2)(1/2)(-cos(y))
computer        4
user           5 ans0 -1/4cos(2x) ( sin(x)$2 + cos(x)$2 )
computer        CORRECT
```

Figure 2

```
user           fast nil
computer        NIL
user           master nil
computer        0 PROBLEM
user                          int e$(2x)/( 1 + e$(4x))  dx
user           1 subst y=e$x,   int y$2/(1+y$4) dy

computer        AN ERROR OCCURRED AFTER LINE 0 WAS ANALYZED

user           oh,yes
user           1 subst y=e$x,   int y/(1+y$4) dy
user           2 sub, z=y$2,   int 0.5/(1+z$2) dz
user           3 ans 2             1/2 arctan(z)
user           4 ans 0      1/2 arctan((e$x)$2)
computer        CORRECT
```

Figure 3

References

1) Martin, W. A., "Symbolic Mathematical Laboratory", Report MAC-TR-36 (Thesis), Project MAC, M.I.T., January 1967.

2) Moses, J., "Symbolic Integration", Report MAC-TR-47 (Thesis), Project MAC, M.I.T., December 1967.

3) Anderson, R. H., "Syntax Directed Recognition of Hand-Printed Two-Dimensional Mathematics", doctoral dissertation, Harvard University, January 1968.

4) Martin, W. A., "A Fast Parsing Scheme for Hand-Printed Mathematical Expressions", MAC-M-360, Project MAC, M.I.T., October 1967.