Massachusetts Institute of Technology
Artificial Intelligence Laboratory

A.I. Memo No. 577 September, 1980

A Session With TINKER: Interleaving Program Testing With Program Design

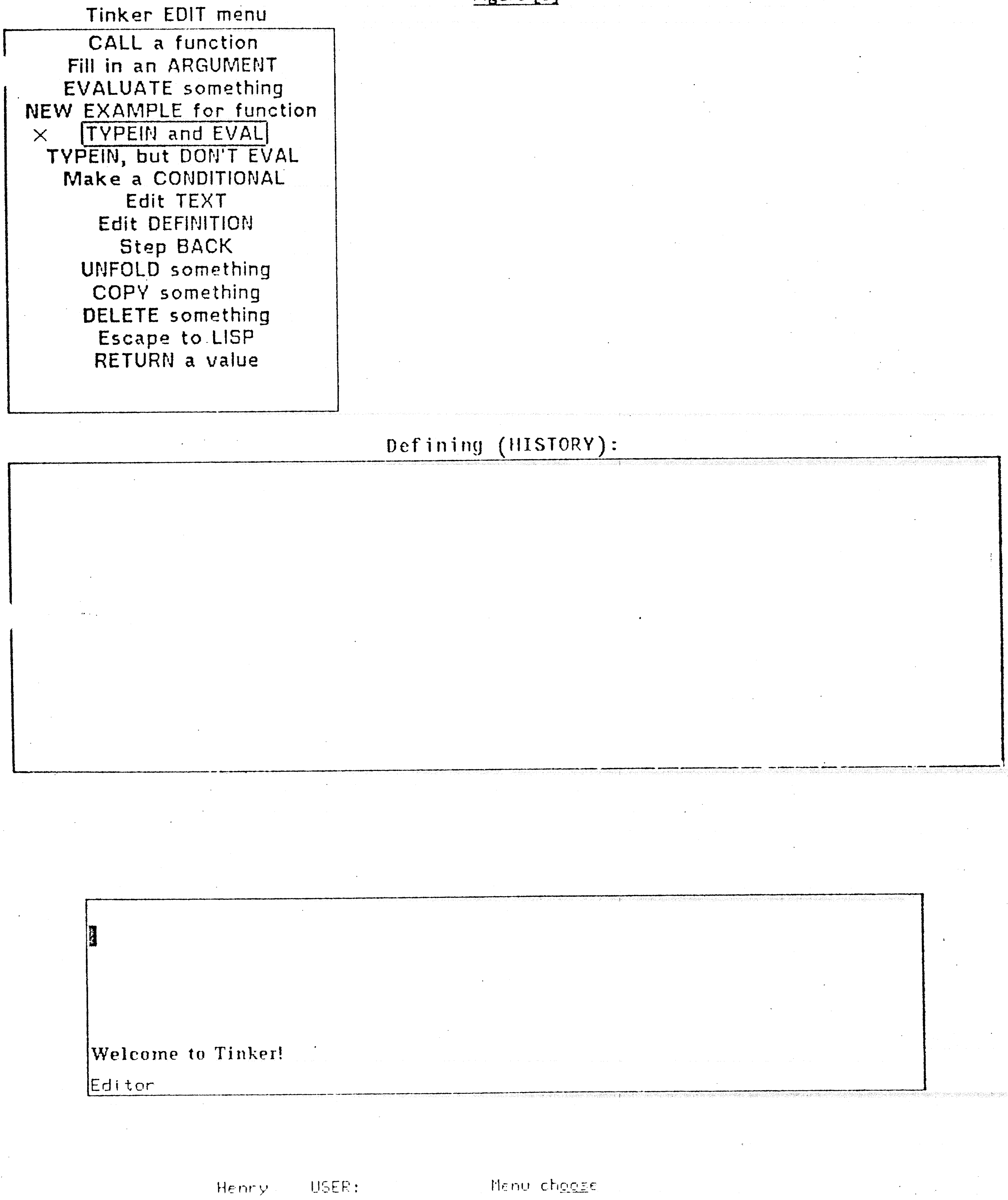
Henry Lieberman
Carl Hewitt

Abstract

Tinker is an experimental interactive programming system which integrates program testing with program design. New procedures are created by working out the steps of the procedure in concrete situations. Tinker displays the results of each step as it is performed, and constructs a procedure for the general case from sample calculations. The user communicates with Tinker mostly by selecting operations from menus on an interactive graphic display rather than by typing commands. This paper presents a demonstration of our current implementation of Tinker.

Acknowledgements: This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Office of Naval Research under Office of Naval Research contract N00014-75-C-0522, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

Figure [1]



Tinker EDIT menu

CALL a function
Fill in an ARGUMENT
EVALUATE something
NEW EXAMPLE for function
× TYPEIN and EVAL
TYPEIN, but DON'T EVAL
Make a CONDITIONAL
Edit TEXT
Edit DEFINITION
Step BACK
UNFOLD something
COPY something
DELETE something
Escape to LISP
RETURN a value

Defining (HISTORY):

Welcome to Tinker!
Editor

Henry USER: Menu choose

A Session With TINKER: Interleaving Program Testing With Program Design

Henry Lieberman and Carl Hewitt

Artificial Intelligence Laboratory
and Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

Tinker is an experimental interactive programming system which integrates program testing with program design. New procedures are created by working out the steps of the procedure in concrete situations. Tinker displays the results of each step as it is performed, and constructs a procedure for the general case from sample calculations. The user communicates with Tinker mostly by selecting opeations from menus on an interactive graphic display rather than by typing commands. This paper presents a demonstration of our current implementation of Tinker.

1. Introduction

Tinker is our first attempt at building an experimental programming environment for Lisp which explores two new directions in programming methodology:

Programs are tested with examples as they are being written: In conventional systems, the user writes a complete program, then tests it as a whole. In Tinker, new functions are defined by supplying *examples*, and working out the steps of the procedure on examples. As each step of the program is introduced, the effect of that step on an example is displayed immediately. Tinker remembers the steps and constructs a procedure for the general case. Program writing and program testing happen *simultaneously*.

Menu selection can replace much typing in constructing programs: Instead of specifying operations and operands by typing, the system displays on the screen a *menu* of available choices whenever possible, and the user simply points to one to select it.

The user doesn't have to remember or look up in a manual what commands are available or what the syntax of the language is. We are experimenting with Tinker to see to what extent menu selection can replace typed commands in designing and debugging programs.

We chose the name Tinker to suggest the process of building or fixing something in small steps, incrementally making a small change, seeing what happens, then making another change.

In contrast to previous research labelled as *programming by example*, Tinker doesn't attempt to try to *guess* the definition of the procedure from simply a statement of what values the function returns for given arguments. Instead, we must explicitly demonstrate the steps which the procedure must take to compute the value. Guessing the procedure definition from input-output histories is a much harder problem. It is an automatic programming task which is currently beyond the state of the art for non-trivial examples. Tinker's contribution lies in *integrating* the use of examples with program construction.

Rather than describe the capabilities of Tinker in the abstract, we feel that a good way to convey our ideas about programming methodology is to show Tinker in action. The body of this paper will consist of a *demonstration* of Tinker, presenting it to the reader as we would to a new user of the system. The illustrations depict what appears on the user's display screen. We will explain features of Tinker as they are encountered in the demonstration.

Our session with Tinker will consist of three parts, starting out very simply and working up toward more complicated tasks. The first part will discuss Tinker's user interface, and show how to build up and evaluate Lisp expressions using Tinker. The second will show how to introduce new functions into Tinker, illustrating the role of examples. The third part will present a more realistic scenario, one which would be plausible for an experienced user. This will show how to define functions with conditionals through the use of multiple examples, and recursive functions using partially specified definitions.

2. Getting started

The first illustration shows what Tinker looks like when it's first started. The screen starts out with three windows.

The large window in the center of the screen is the *snapshot* window. The snapshot window shows the current state of the computation. Whenever we introduce a new object or piece of program text, it will appear in the snapshot window. The snapshot window always contains a *partial definition* of the body of some function. (Initially, we're defining a special function called HISTORY, which contains top level evaluations.) Later, when we define a new function, the name of the function will appear in the label of the snapshot window, and we will construct a definition for it in the snapshot window.

At the top of the screen is the *Tinker Edit menu.* This shows the list of available operations at the top level of Tinker. These operations edit the contents of the snapshot window. They may create, delete, move or modify objects in the snapshot window. We'll explain what each editing operation does as we go along.

We can select an item from the menu by pointing to its name and pressing a button. Tinker runs on the MIT Lisp Machine, a personal computer equipped with a *mouse*, a pointing device. The current position of the mouse is indicated on the screen by an *X*. When the position of the mouse touches the name of an operation, the name is highlighted on the screen.

The window at the bottom of the screen is the *editor window.* Occasionally, Tinker will ask the user to type something, such as names for functions, or code to be executed. Everything typed on the keyboard goes into the editor window. Informative messages [like *Welcome to Tinker!*] and error messages also appear in the editor window. The editor window is connected to Zwei, a sophisticated real-time display text editor. This gives the user access to a wide range of editing operations whenever *anything* is typed. The editing operations include motion across characters, words, sentences, and expressions; search; cut and paste; expansion of abbreviations; and many more. This is in contrast to many systems which only allow a few trivial editing operations when typing input, and require the user to use a separate editor program for more extensive changes.

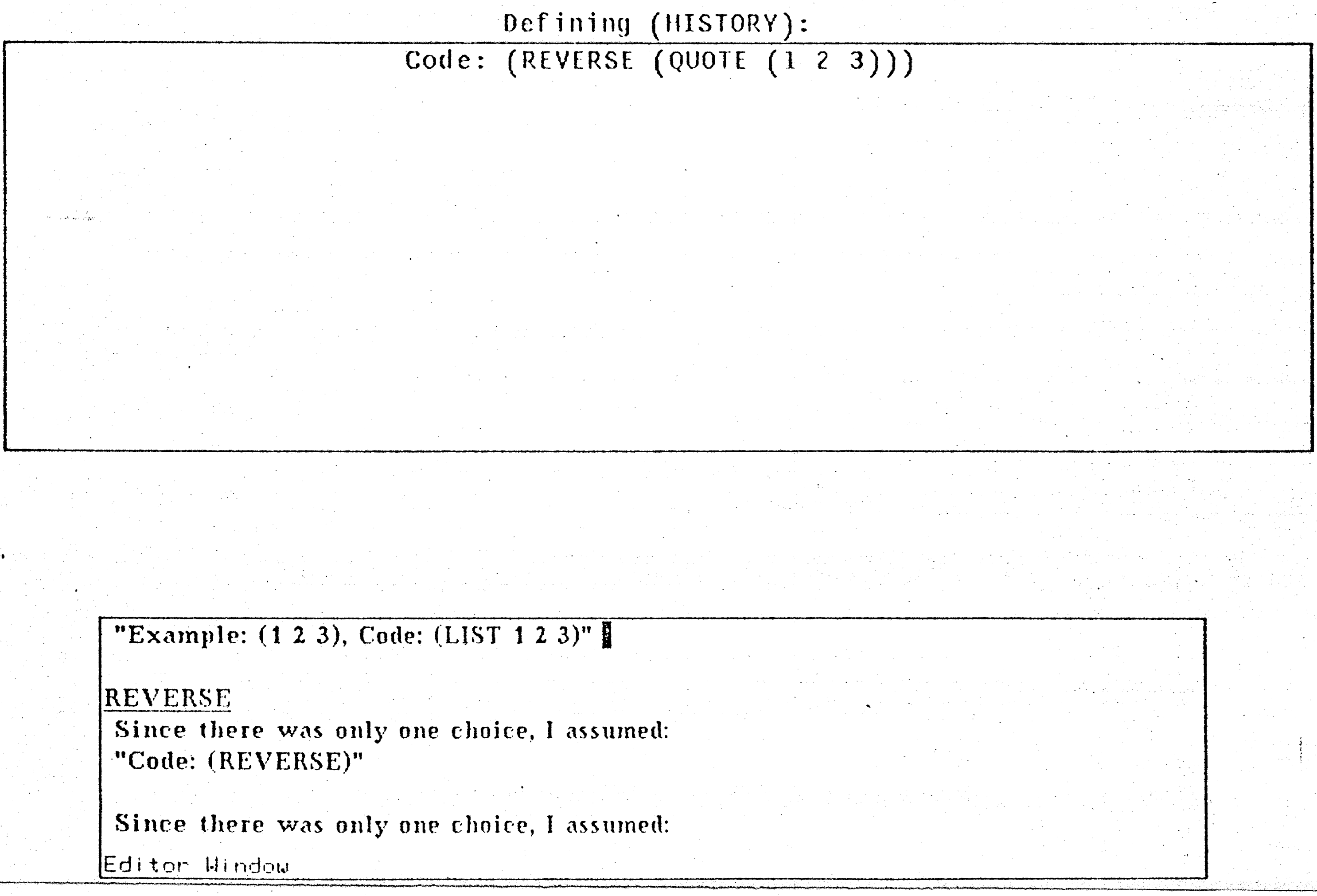
3. When Tinker evaluates some code it remembers both the value and the code

We will first show how to use Tinker as a kind of *desk calculator* for Lisp, building up Lisp expressions and evaluating them. Whenever Tinker evaluates an expression, it *remembers* the code that produced that value. Whenever that value is used as an ingredient in a subsequent computation, the code corresponding to the value is carried along. This allows us to *incrementally* build up complicated expressions.

Tinker could warn us right away, and we could fix it immediately. In conventional systems, the bug would go undetected until much later.

What appears now on the screen is a *partial* description of some code, a call to REVERSE. It isn't complete yet, because we haven't filled in the arguments to REVERSE, and evaluated it.

Next, we select `Fill in an ARGUMENT`. Selecting the list `(1 2 3)` chooses it as the argument to REVERSE, *moving* the list inside the call to REVERSE. Tinker *automatically* makes this selection for us, rather than stopping to ask, since there was only *one* piece of code that needed arguments, and only *one* object on the screen that could possibly be an argument to REVERSE.



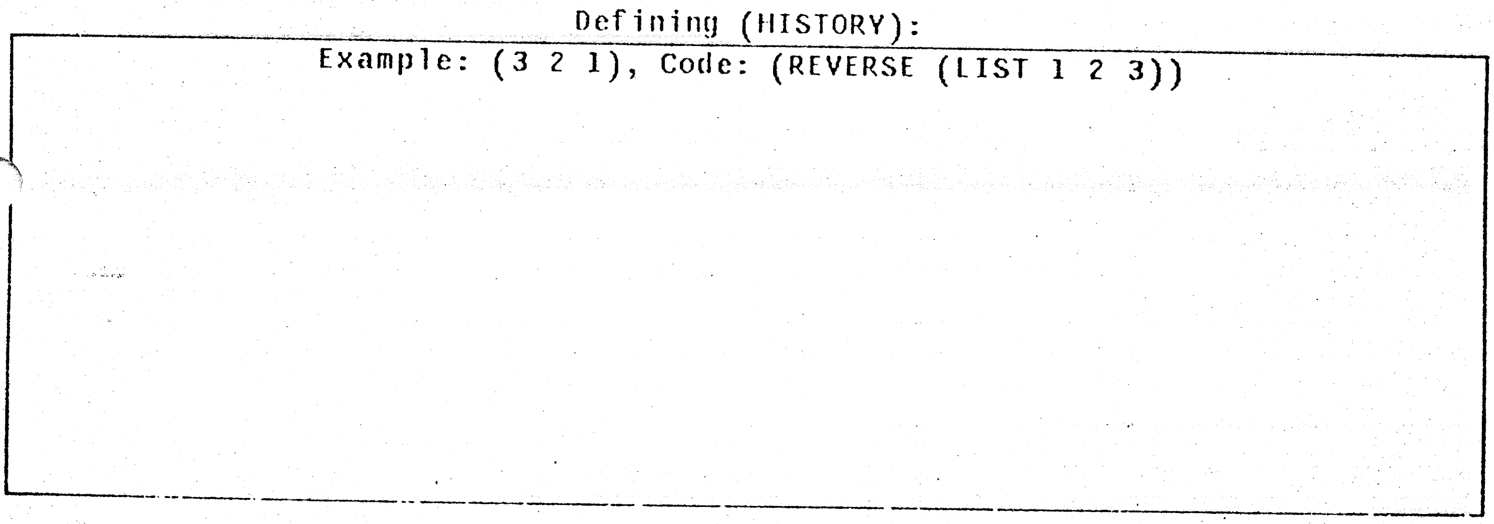
```
Defining (HISTORY):
Code: (REVERSE (QUOTE (1 2 3)))
```

```
"Example: (1 2 3), Code: (LIST 1 2 3)"

REVERSE
 Since there was only one choice, I assumed:
 "Code: (REVERSE)"

 Since there was only one choice, I assumed:
Editor Window
```

Figure 4. Selecting the menu operation "`Fill in an ARGUMENT`".

Tinker has a policy of bothering the user as little as possible; if it can determine that there's only one reasonable choice to be made, it goes ahead and makes the choice. (The choice can always be retracted, of course, if it had any unwanted consequences.) Tinker informs us of this in the editor window by saying `Since there was only one choice, I assumed: ....` If there were more than one possibility, Tinker would ask us which one we wanted. We shall see this later. Automatically making the *obvious* choice in the current context is very helpful, especially to experienced programmers who find that this speeds up interaction with the system. This automatic choice feature can be disabled for naive users, or those who don't find it to their taste.

Now that we've filled in all the arguments, we choose EVALUATE `something`, to find out the value of the call to REVERSE. (Tinker could be made to realize that since REVERSE takes only one argument, the code could be evaluated as soon as that argument is filled in.) This produces the list `(3 2 1)` from `(1 2 3)`.



```
Defining (HISTORY):
Example: (3 2 1), Code: (REVERSE (LIST 1 2 3))
```

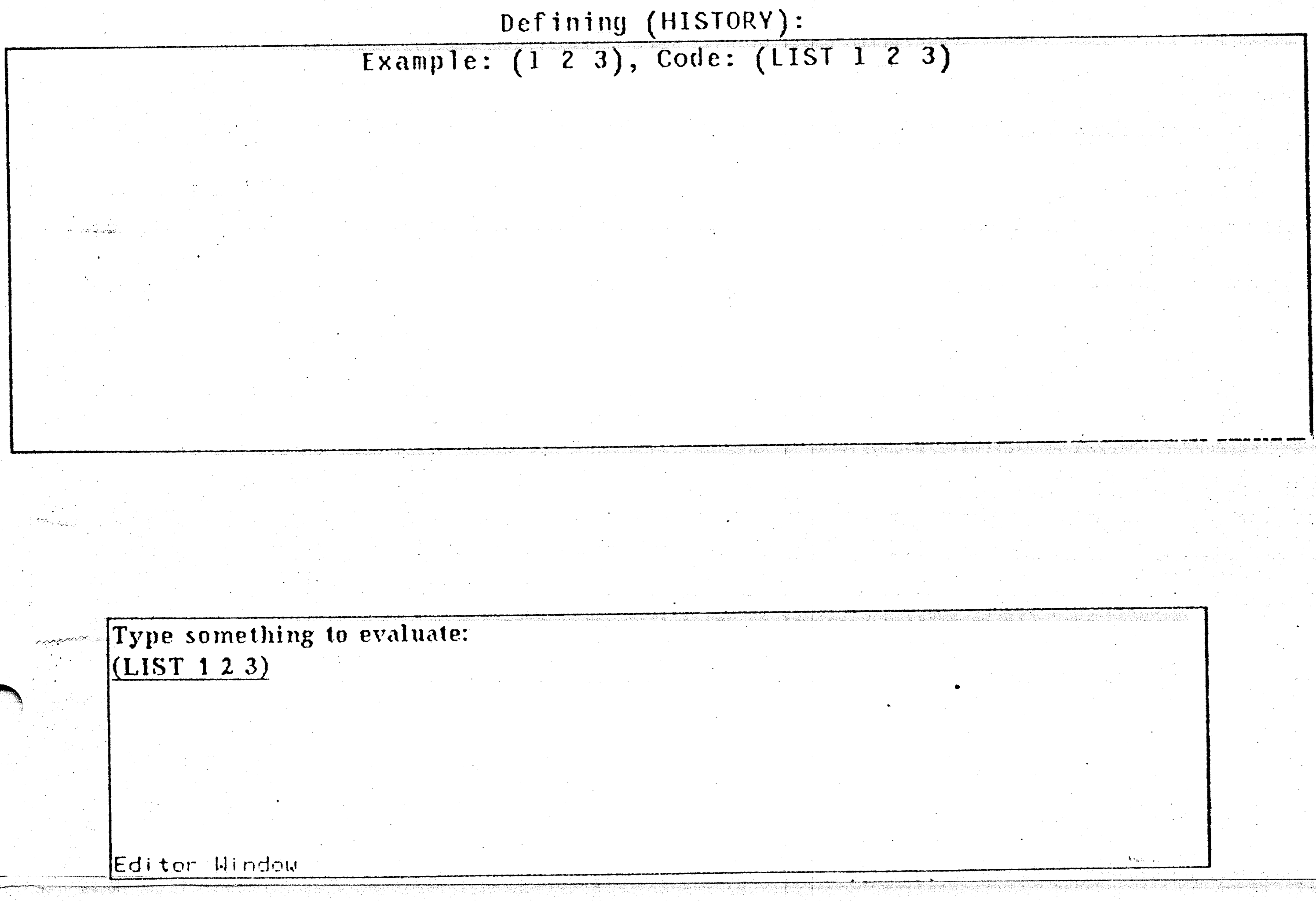
Figure 5. Selecting the menu operation "EVALUATE `something`".

Notice that the code `(LIST 1 2 3)` that produced the argument to REVERSE appears in the code portion. When we used the list as a component of a larger expression, the code which produced that list is carried along as part of the code for the larger expression. We can build up large expressions a little bit at a time, and Tinker makes visible all the intermediate steps of the evaluation, so we can verify that each step had the effect we intended.

Since we get to see all the intermediate results along the way, we can immediately verify the accuracy of each step before proceeding to the next.

One way of introducing new expressions to Tinker is by typing. Tinker provides two operations for introducing something new into the snapshot window by typing. The `TYPEIN and EVAL` operation is like the `READ-EVAL-PRINT` loop of Lisp. It reads a Lisp expression in the editor window (prompting with *Type something to evaluate:*), and calls `EVAL` on it. `TYPEIN, but DON'T EVAL` is similar, but doesn't evaluate the expression. It just reads an expression and puts it in the snapshot window.

The first thing we do is select `TYPEIN and EVAL`, and type in the code `(LIST 1 2 3)` in the editor window. This evaluates to the list `(1 2 3)`. In the snapshot window appears a message telling us that the value `(1 2 3)` was produced by the code `(LIST 1 2 3)`.



Defining (HISTORY):
Example: (1 2 3), Code: (LIST 1 2 3)

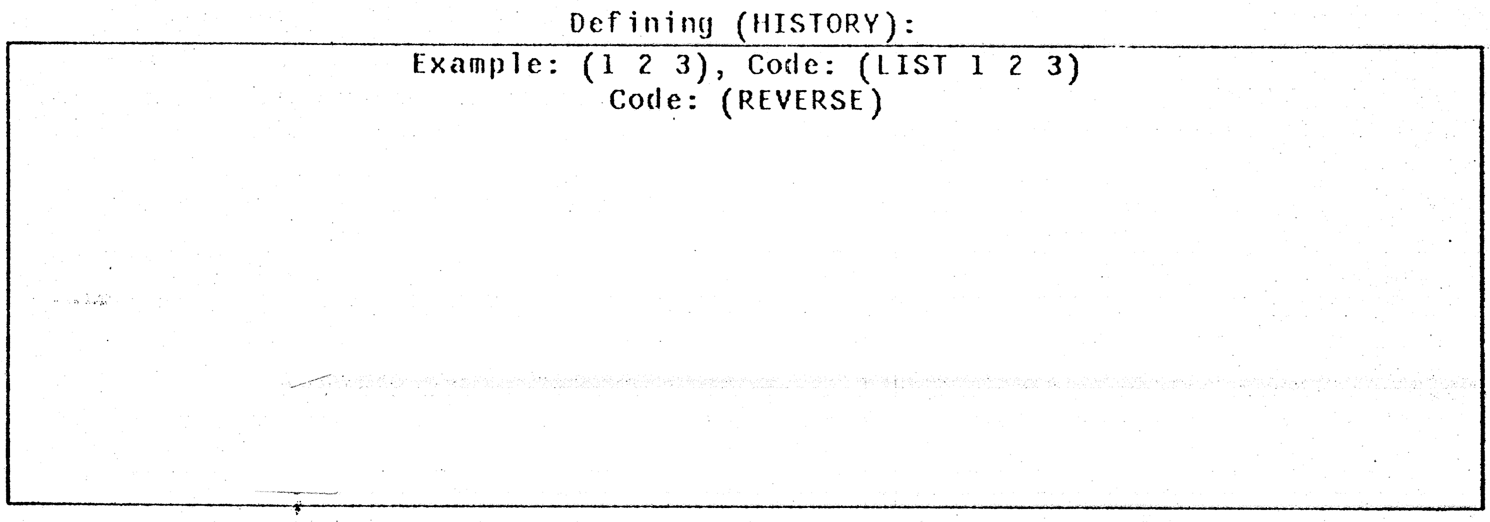
Type something to evaluate:
(LIST 1 2 3)

Editor Window

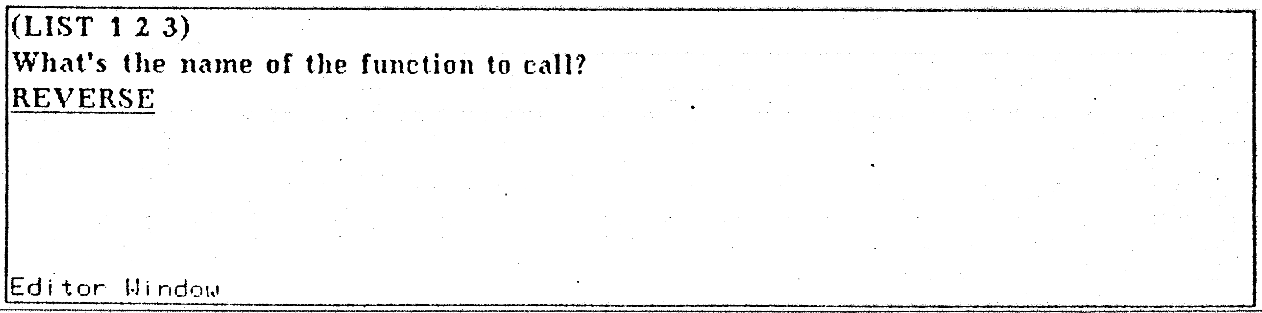
Figure 2. Selecting the menu operation "TYPEIN and EVAL".

(In this case, the code is always going to evaluate to the same value, but in general the code might have variables, and the value shown may only be an *example* of what the code could evaluate to.)

To show what happens when we make further use of the value (1 2 3), let's try reversing the list. We select the operation CALL a function, and Tinker asks us for the name of the function in the editor window. We type in REVERSE.



```
Defining (HISTORY):
Example: (1 2 3), Code: (LIST 1 2 3)
Code: (REVERSE)
```



```
(LIST 1 2 3)
What's the name of the function to call?
REVERSE
```

Figure 3. Selecting the menu operation "CALL a function".

It's not strictly necessary that we access the function by typing its name. We would like to extend Tinker so that we could point to an operand like the list, and Tinker would produce a menu of possible operations on it, based on the type of the data. For lists, it could know that common operations on lists are CAR, CDR, CONS, NULL, and one of these could be chosen, or another menu containing a larger number of less common functions could be selected. Tinker could also interactively check the types of arguments. If we inadvertently selected an argument of the wrong type, In this way, Tinker allows us to build up parts of a program like pieces of a *jigsaw puzzle*. We're free to piece together small parts of a program in any order, then combine them into larger chunks. The order in which we evaluate expressions while constructing the program doesn't have to be the same as the order in which the final program will evaluate expressions. We can write the program in the left-to-right order as we would write it conventionally, or we can follow the bottom-up order of evaluation. We believe this added flexibility will make it much easier to incrementally construct and modify programs.

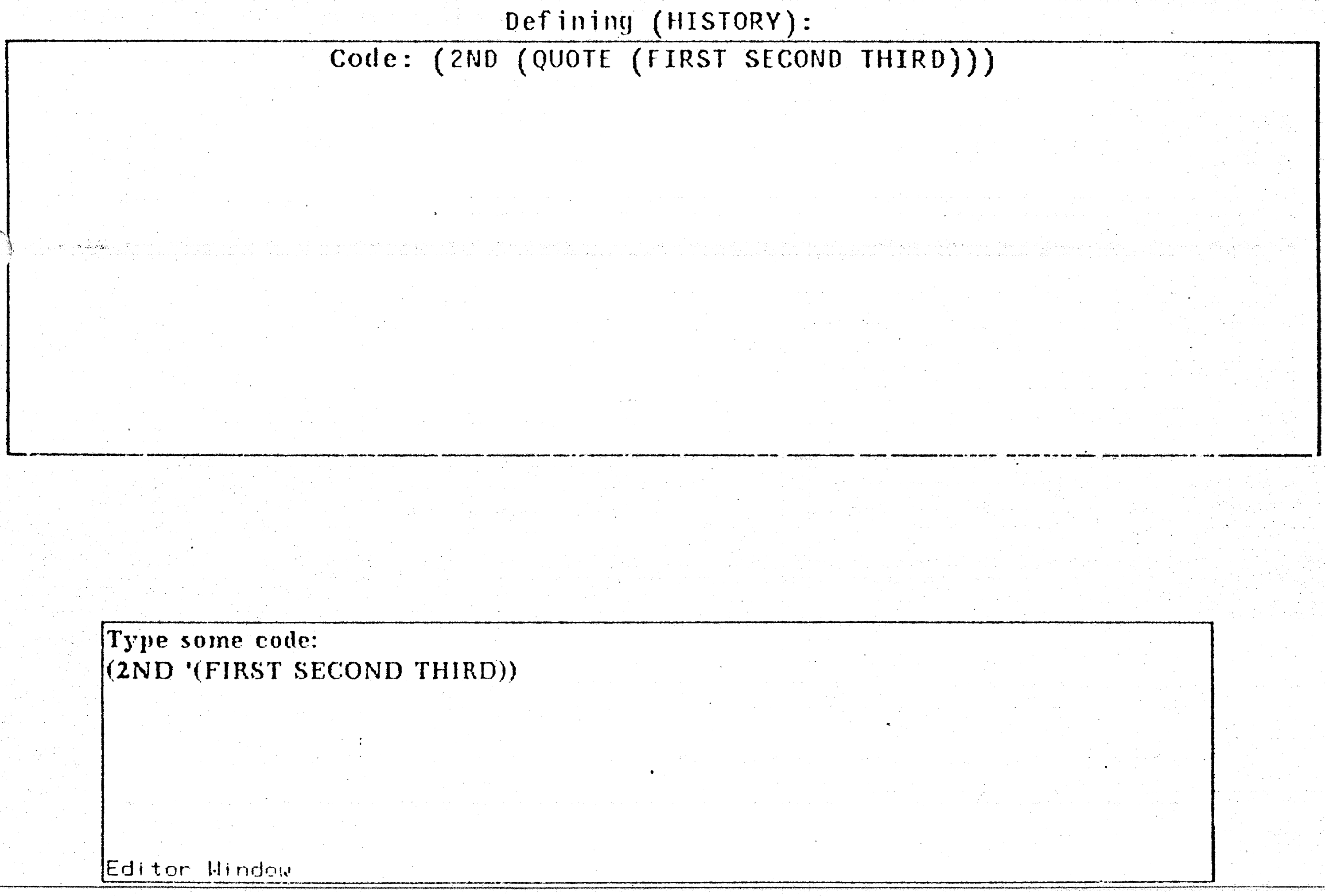
4. Defining a new function using an example

We will now demonstrate how to tell Tinker about a new function. Tinker will learn about a new function by watching us work out an *example* of a typical use of the new function. We will show Tinker how to perform each step of the definition of the function. Tinker will remember each step, showing us the result of each step on the data we supplied it in the example. When we've finished working out the example, Tinker will generalize that example and construct a procedure for the general case.

We will start with a very simple example, so the reader can get the flavor of our approach, and we will then proceed to more complex examples. We will now define a function 2ND which extracts the second element of a list using the operations CAR and CDR.

To begin, we pretend we already have the definition of the function 2ND, and we show Tinker a sample of how we would like the procedure to be used. We feed it the list (FIRST SECOND THIRD), and we would like it to evaluate to SECOND.

We use TYPEIN, but DON'T EVAL to enter the code (2ND '(FIRST SECOND THIRD)) then use NEW EXAMPLE for function, to identify this as a new example for the function 2nd.

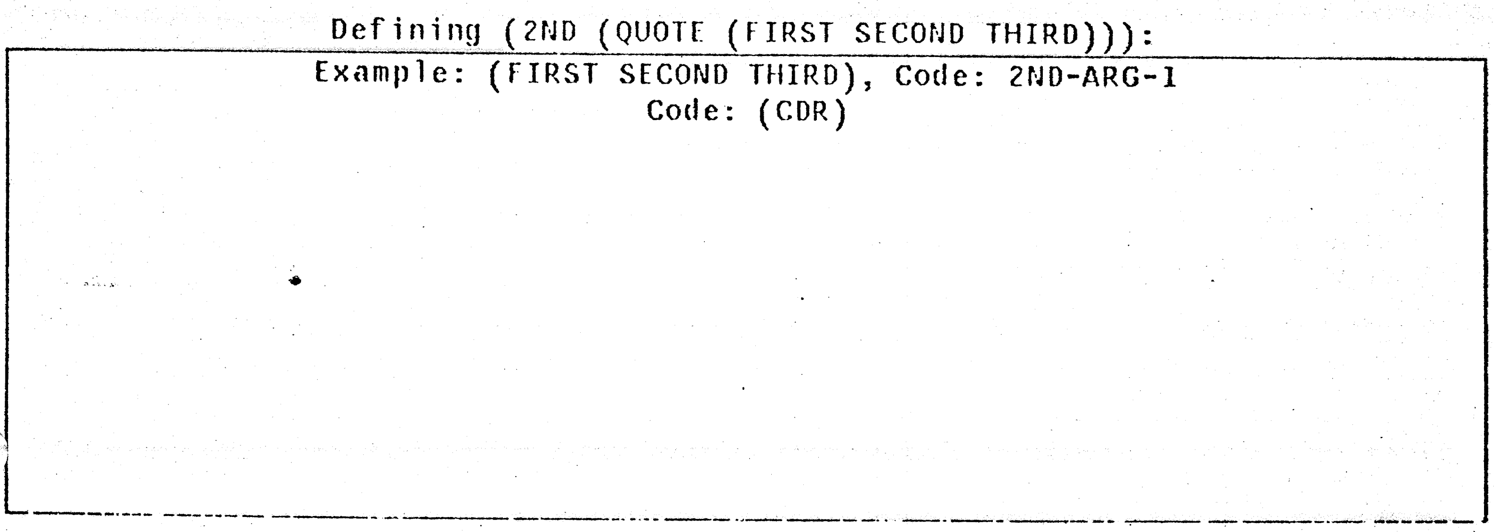


```
Defining (HISTORY):
Code: (2ND (QUOTE (FIRST SECOND THIRD)))
```

```
Type some code:
(2ND '(FIRST SECOND THIRD))

Editor Window
```

Figure 6. Selecting the menu operation "NEW EXAMPLE for function".

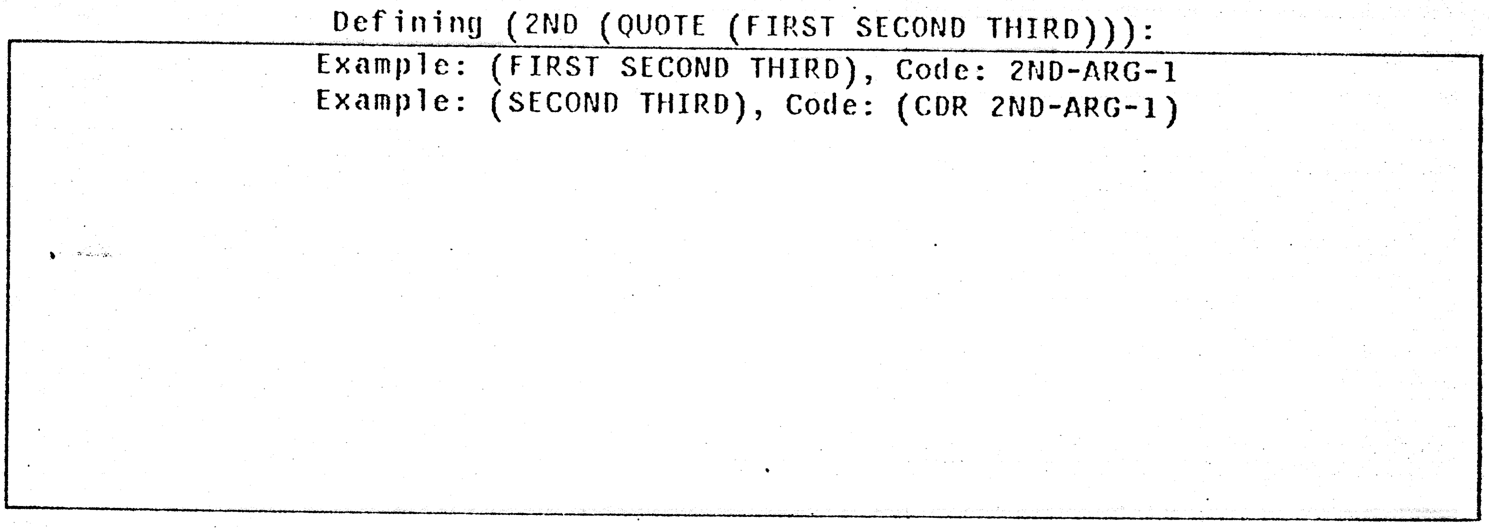
The label of the snapshot window informs us that we are now defining what the code (2ND '(FIRST SECOND THIRD)) should evaluate to. Tinker has *saved* the state of the snapshot window. Inside the snapshot window, we see that Tinker manufactured a new variable, which it called 2ND-ARG-1, to represent the single argument to the procedure 2ND. (We could, if we like, rename the variable.) Tinker tells us that the list (FIRST SECOND THIRD) is an *example* of the argument to the function 2ND. Given the arguments as raw material, we compute the value of the procedure in the snapshot window.



```
                    Defining (2ND (QUOTE (FIRST SECOND THIRD))):
                    Example: (FIRST SECOND THIRD), Code: 2ND-ARG-1
                                   Code: (CDR)
```

Figure 7. Selecting the menu operation "CALL a FUNCTION".

To extract the second element of the list, we must first chop off the first element using the function CDR. We perform the menu operations CALL a function, type in CDR, Fill in an ARGUMENT, then EVALUATE something. After these three operations, we're shown that the CDR of the argument to 2ND is the list (SECOND THIRD).

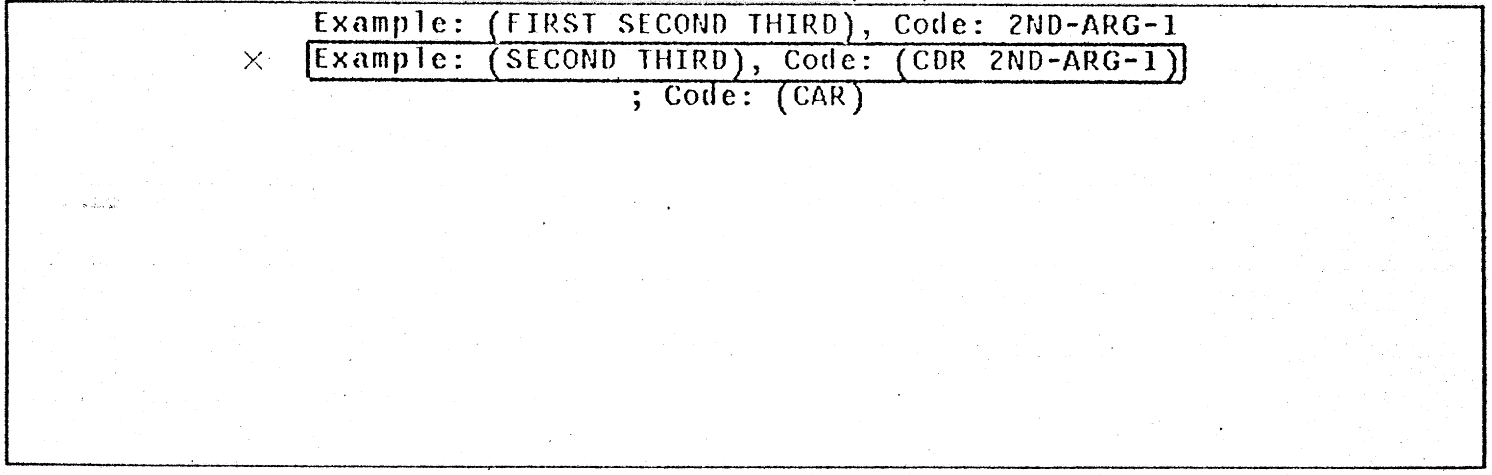


```
Defining (2ND (QUOTE (FIRST SECOND THIRD))):
Example: (FIRST SECOND THIRD), Code: 2ND-ARG-1
Example: (SECOND THIRD), Code: (CDR 2ND-ARG-1)
```

Figure 8. Selecting the menu operation "EVALUATE something".

Now, we're getting closer. We can see that the symbol SECOND is the first element of that list, and we can obtain it by taking the CAR. Tinker shows us that the answer we want, SECOND, is obtained by taking first the CDR, then the CAR of the argument to 2ND.

Next argument to (CAR)?

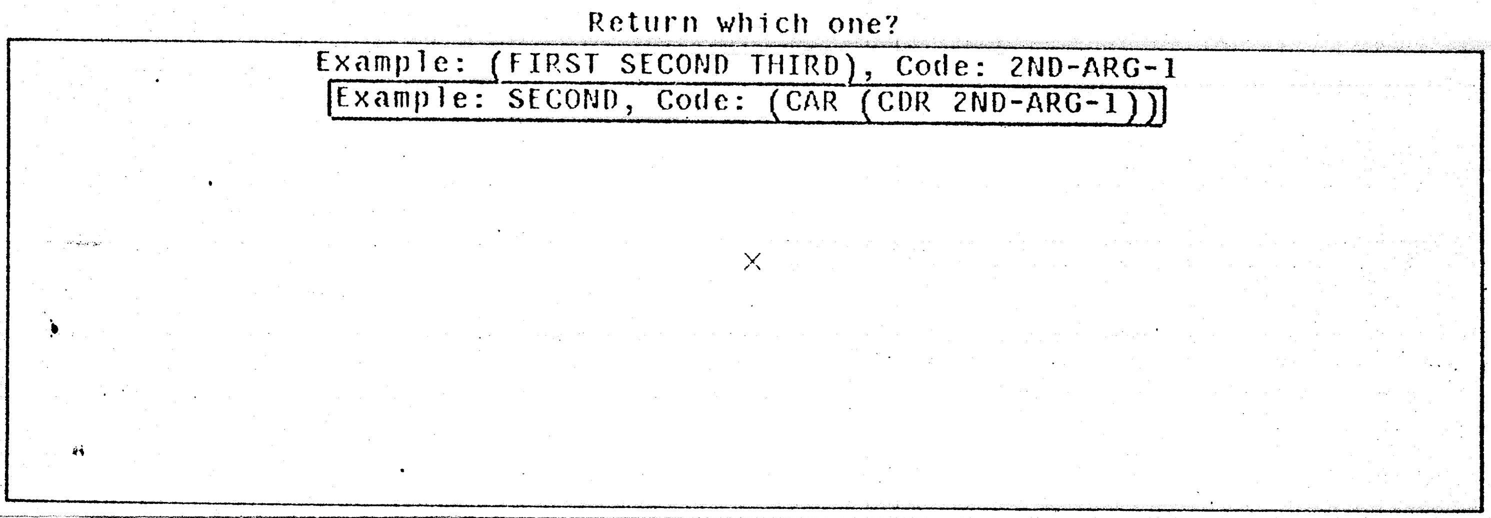


```
          Example: (FIRST SECOND THIRD), Code: 2ND-ARG-1
       X  Example: (SECOND THIRD), Code: (CDR 2ND-ARG-1)
                             ; Code: (CAR)
```

Figure 9. Selecting the menu operation "Fill in an ARGUMENT".

Although this function is really simple, it illustrates an important point about Tinker. In conventional programming, extracting elements of a list with CARs and CDRs is usually an error-prone process. It's very easy to specify one operation too few or too many. This is because when we specify a path through a data structure, we have to imagine where the path will lead. Since Tinker shows us all the intermediate results when the path is specified, the immediate visual feedback makes it easy to verify that a sequence of operations has the desired result.

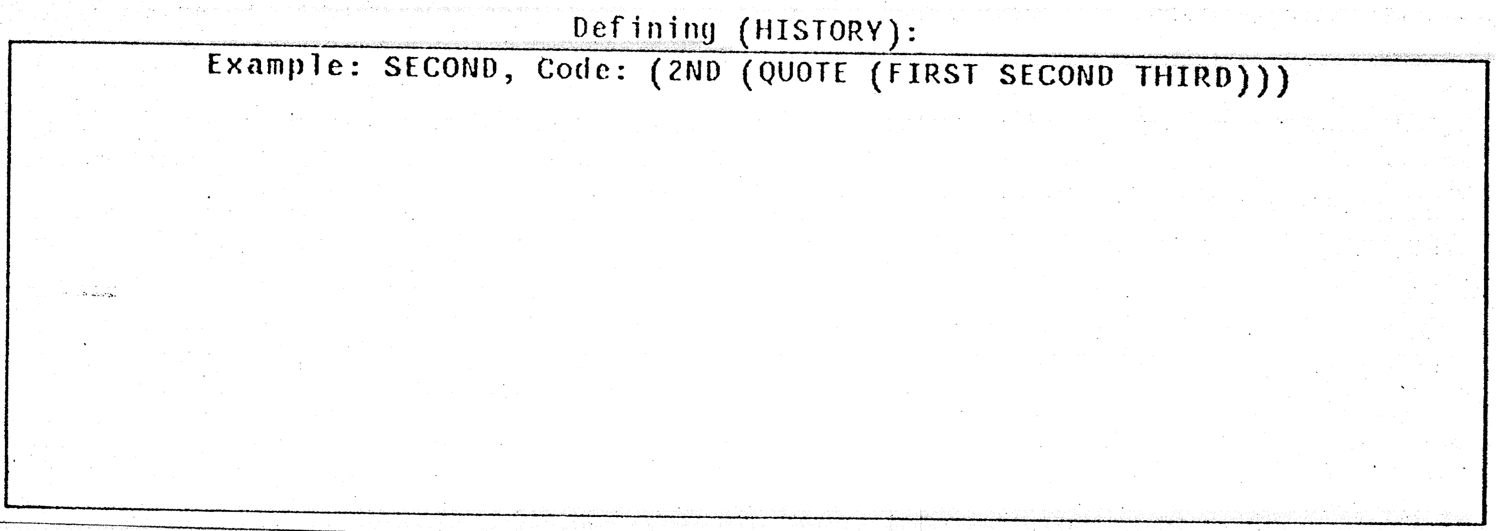
Since we've now obtained our desired result for the function, we select the RETURN a value operation, saying we want to return SECOND as the result of (2ND '(FIRST SECOND THIRD)).


Return which one?

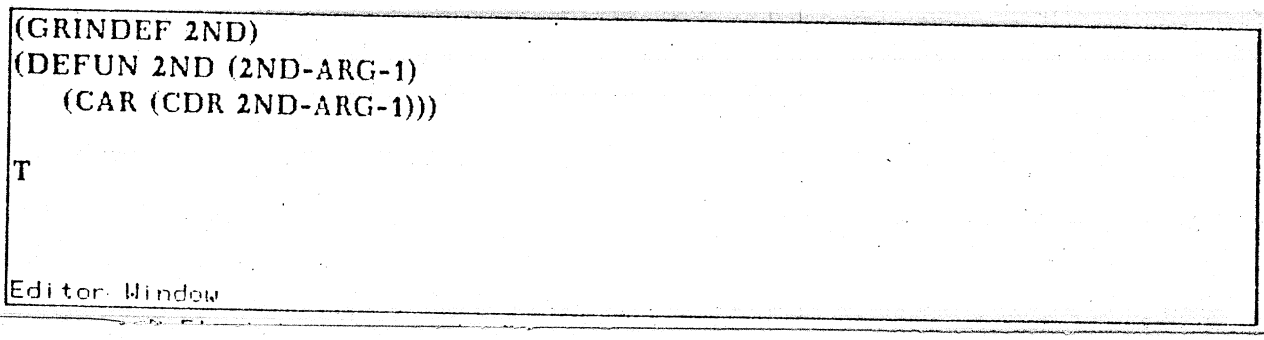
```
Example: (FIRST SECOND THIRD), Code: 2ND-ARG-1
Example: SECOND, Code: (CAR (CDR 2ND-ARG-1))
```

Figure 10. Selecting the menu operation "RETURN a value".

Tinker *restores* the state of the snapshot window, returning us to HISTORY. In the editor window, we examine the definition of 2ND. This shows us that Tinker has written the code for 2ND, defining it as a function of one argument.



```
Defining (HISTORY):
Example: SECOND, Code: (2ND (QUOTE (FIRST SECOND THIRD)))
```



```
(GRINDEF 2ND)
(DEFUN 2ND (2ND-ARG-1)
   (CAR (CDR 2ND-ARG-1)))

T
```

Figure 11. Selecting the menu operation "RETURN a value".

The output produced by Tinker is a ordinary Lisp program, indistinguishable from a definition typed in using the more conventional methodology. When we're writing a program, Tinker maintains its own data representation for programs, and has its own evaluator, so it can provide services the ordinary Lisp system does not provide. But since Tinker also writes ordinary Lisp code, functions produced by Tinker can be compiled with the Lisp compiler. This means there's no penalty for using Tinker during program development. The resulting code runs just as efficiently as if it were written conventionally.

Notice that as soon as our procedure appears, we are *guaranteed* that it works for the test case we've given it! There's no separation between *writing* a program and *testing* it. Instead, both writing and testing are interleaved and performed incrementally.

Tinker's approach is more robust than conventional systems. When a piece of erroneous code is introduced in a conventional program, it is usually buried deep within many other operations before we get a chance to see whether the code has the desired result for typical test cases. The symptoms appear only when the *entire* program is tested, and an error message or incorrect result is produced. We are then faced with the arduous task of *isolating* the erroneous code from many other operations, most of which are irrelevant. With Tinker, we can see that an incorrect piece of code fails to work for a test case as soon as it is introduced to the system.

5. Defining a function with conditionals and recursion

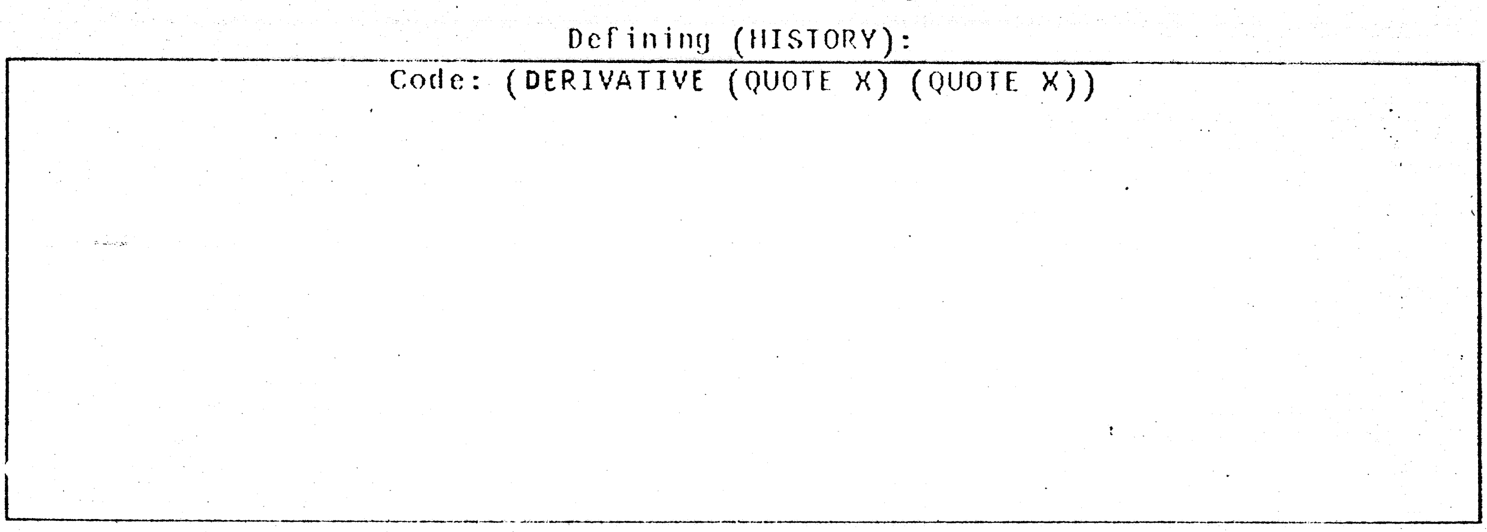
Now that we've shown how to define a simple function with Tinker, we will proceed to a more realistic scenario. This will show how to introduce functions with *conditionals* and *recursion.*

A folk wisdom among programmers says that in order to thoroughly test a program, at least *one example for each branch* of a conditional must be tried. So, to define a program with a conditional, we must provide Tinker with several examples, each one illustrating an important path for the resulting program.

We will also use a set of examples to build up functions with *recursive* calls. At any time, the examples we've given so far for a function will generate a *partial definition* of that function. It may not completely define the function we want, but the partial definition will work for the examples given it so far. We can then extend the function's behaviour with new examples which enable it to handle new cases, building upon the partial definition.

Our demonstration will show defining a *symbolic differentiation* function. The function will accept a list representing a symbolic expression in prefix form, and return the symbolic derivative in the same form. The DERIVATIVE function will have several cases, based on the type of the expression, and may need to recursively compute the derivative of a subexpression.

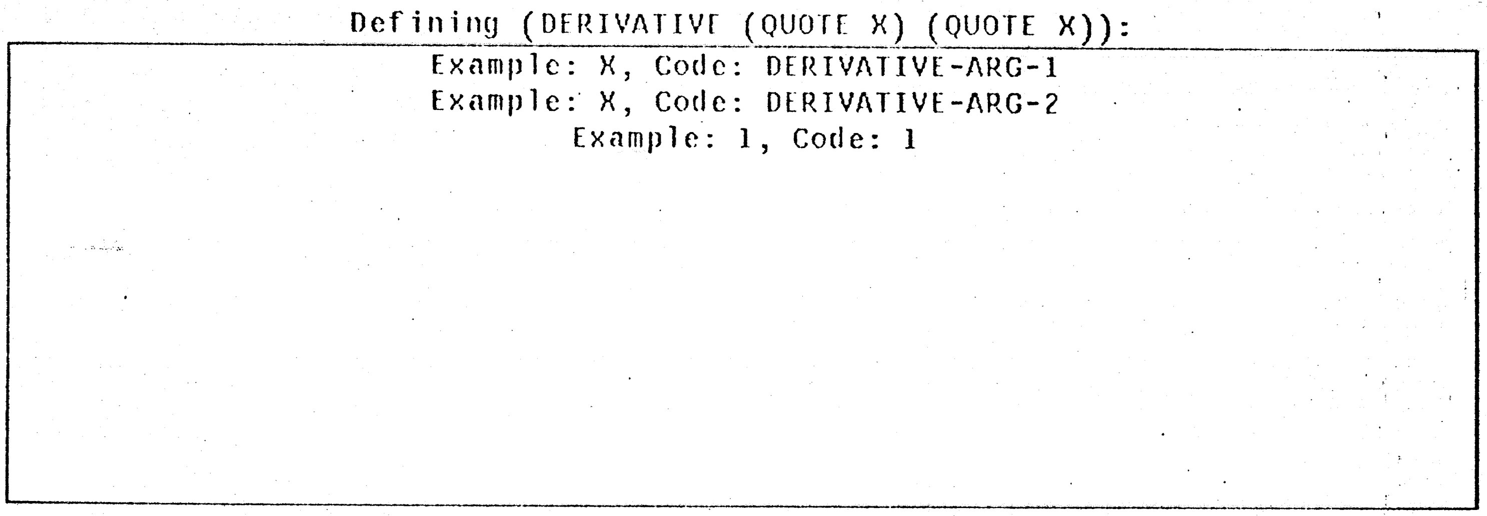
First, let's tell Tinker about a very simple case. The derivative of a variable is 1 with respect to itself. For this, we give Tinker the example (DERIVATIVE 'X 'X).



```
Defining (HISTORY):
Code: (DERIVATIVE (QUOTE X) (QUOTE X))
```

Figure 12. Selecting the menu operation "NEW EXAMPLE for function".

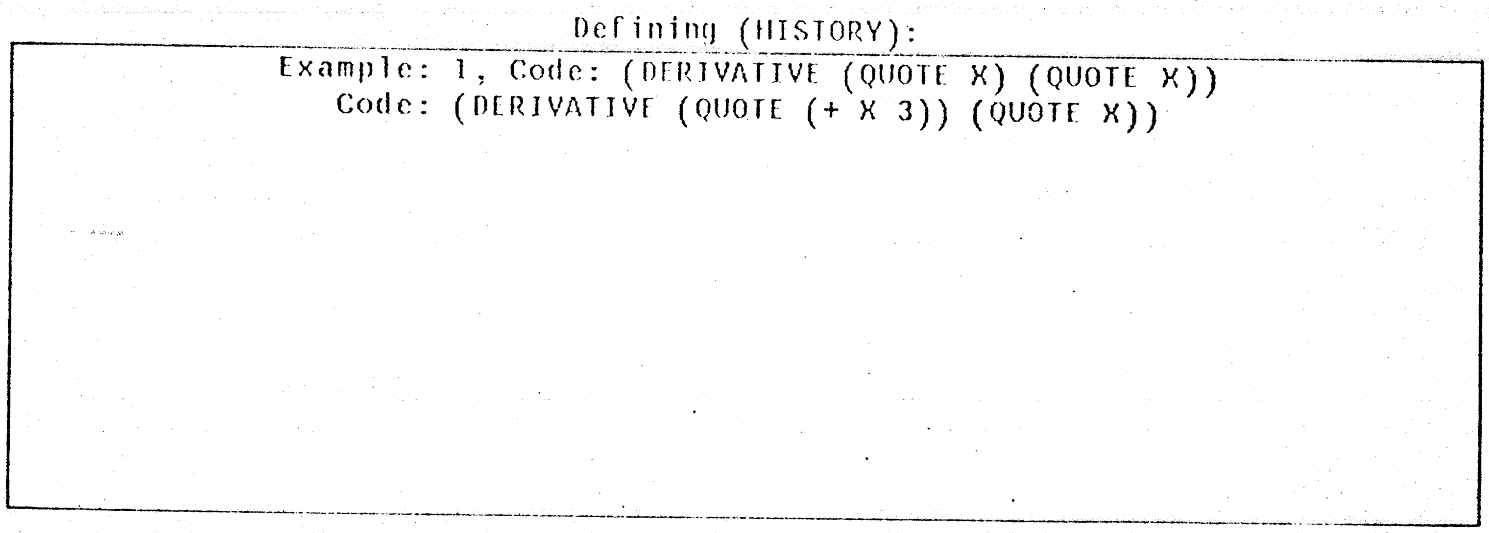
We type in the constant 1 using TYPEIN and EVAL, and return it with RETURN a value.



```
Defining (DERIVATIVE (QUOTE X) (QUOTE X)):
Example: X, Code: DERIVATIVE-ARG-1
Example: X, Code: DERIVATIVE-ARG-2
Example: 1, Code: 1
```

Figure 13. Selecting the menu operation "RETURN a value".

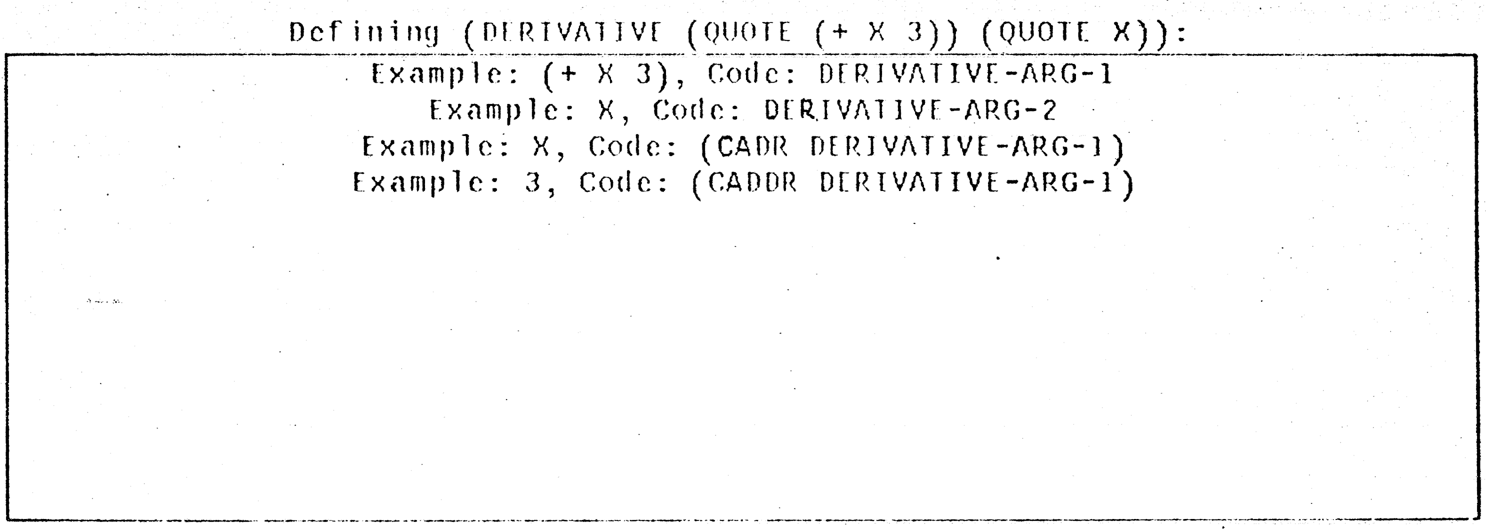
Next, we present a more complicated example. Let's try the derivative of the expression `(+ X 3)`. This example leads to considering *two* more cases: we shall tell Tinker how to take the derivative of an expression which is a sum, and also how to take the derivative of a constant.



```
Defining (HISTORY):
Example: 1, Code: (DERIVATIVE (QUOTE X) (QUOTE X))
      Code: (DERIVATIVE (QUOTE (+ X 3)) (QUOTE X))
```

Figure 14. Selecting the menu operation "**`NEW EXAMPLE for function`**".

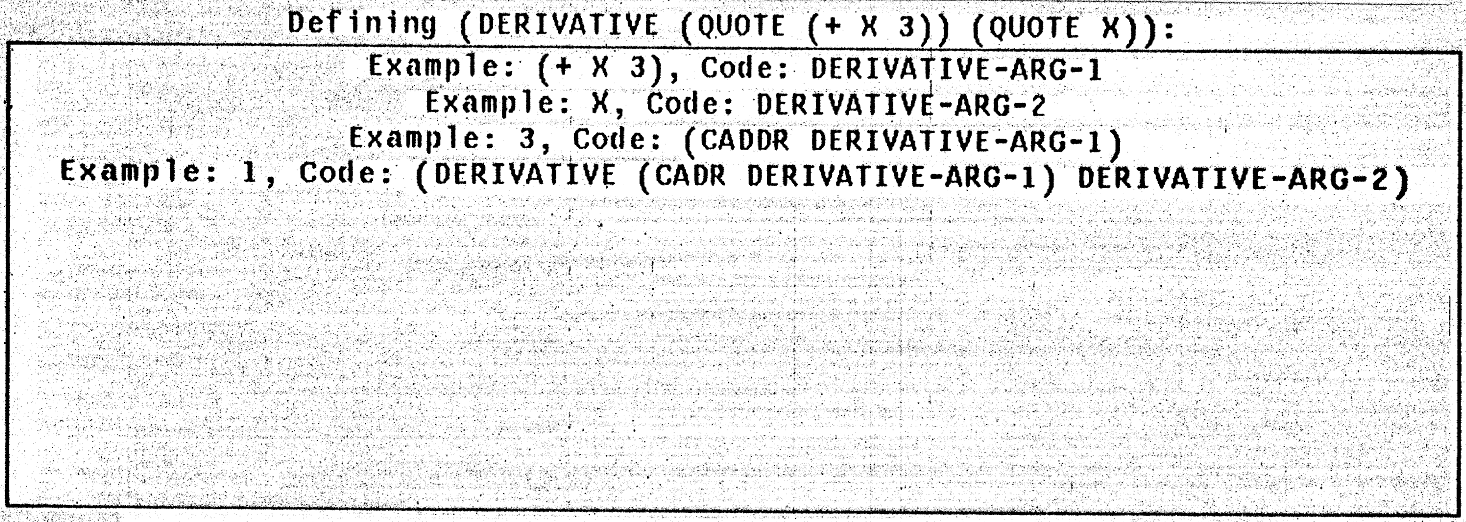
The recursive rule for finding the derivative of a sum says that the derivative of a sum is the sum of the derivatives of the subexpressions. We extract the subexpressions `X` and `3` from the sum `(+ X 3)`. Tinker remembers that `X` is the second element of the first argument to `DERIVATIVE`, and that `3` is the third element.



```
Defining (DERIVATIVE (QUOTE (+ X 3)) (QUOTE X)):
Example: (+ X 3), Code: DERIVATIVE-ARG-1
Example: X, Code: DERIVATIVE-ARG-2
Example: X, Code: (CADR DERIVATIVE-ARG-1)
Example: 3, Code: (CADDR DERIVATIVE-ARG-1)
```

Figure 15. Selecting the menu operation "EVALUATE something".

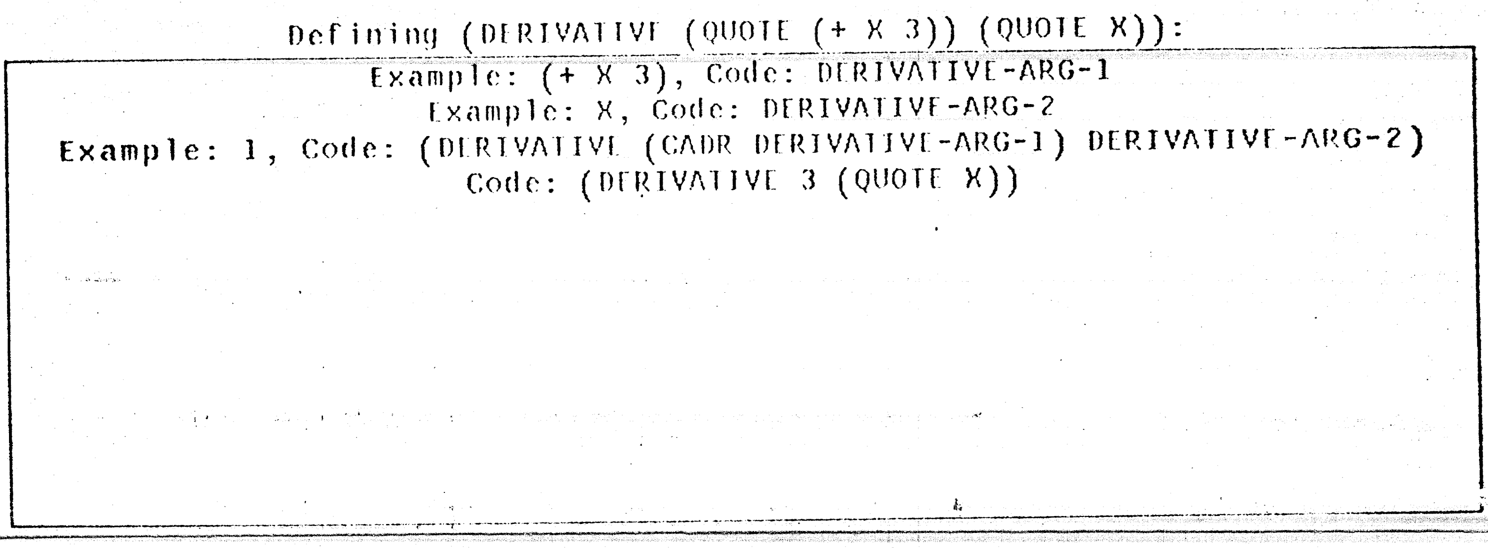
Now, we have to take the derivative of X, so we CALL a function, type DERIVATIVE, and feed it the subexpression X as an argument. Since taking the derivative of a variable is something Tinker already knows how to do, we simply EVALUATE it, returning the answer 1.



```
Defining (DERIVATIVE (QUOTE (+ X 3)) (QUOTE X)):
Example: (+ X 3), Code: DERIVATIVE-ARG-1
Example: X, Code: DERIVATIVE-ARG-2
Example: 3, Code: (CADDR DERIVATIVE-ARG-1)
Example: 1, Code: (DERIVATIVE (CADR DERIVATIVE-ARG-1) DERIVATIVE-ARG-2)
```

Figure 16. Selecting the menu operation "EVALUATE something".

Now, we try the same thing with the other subexpression, 3 But wait -- Tinker doesn't know how to take the derivative of a constant like 3 yet. So, we have to tell it. Instead of evaluating the call `(DERIVATIVE 3 'X)`, we select `NEW EXAMPLE for function`, telling Tinker we wish to define this case.



```
                  Defining (DERIVATIVE (QUOTE (+ X 3)) (QUOTE X)):
                    Example: (+ X 3), Code: DERIVATIVE-ARG-1
                       Example: X, Code: DERIVATIVE-ARG-2
Example: 1, Code: (DERIVATIVE (CADR DERIVATIVE-ARG-1) DERIVATIVE-ARG-2)
                         Code: (DERIVATIVE 3 (QUOTE X))
```

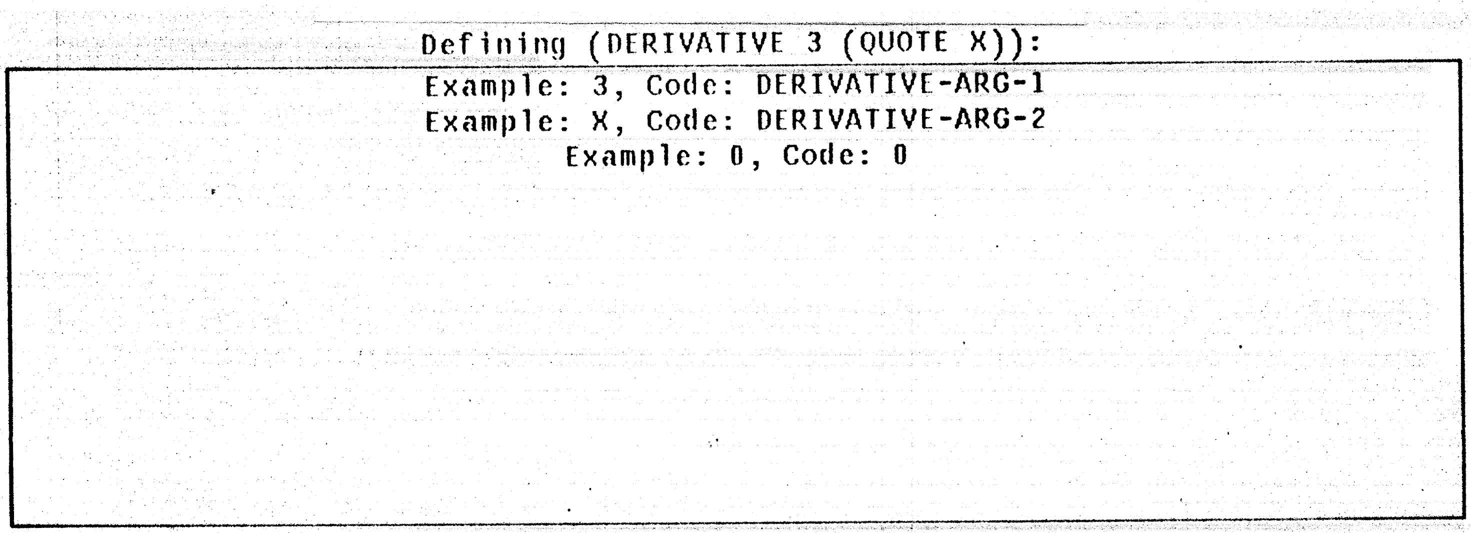
Figure 17. Selecting the menu operation "`NEW EXAMPLE for function`".

We leave temporarily the process of defining the derivative of `(+ X 3)` to define the derivative of 3. This shows a kind of *top down* program development process. In the course of developing our `DERIVATIVE` procedure by stepwise refinement, we discovered a new case that had to be handled. We can produce a definition of that new case, then return to the caller to continue with the original definition. This would also be useful if we discovered that it would be helpful to have an *auxiliary* function. We could define the auxiliary function, make sure it works in the cases needed by its caller, then return to continue the definition of the caller.

We would like Tinker to support a *top down debugging* methodology. What tempts most practical programmers to a bottom up programming style rather than the conceptually cleaner top down programming style advocated by experts? To a great extent, it is the fact that debugging is usually performed bottom up. Subprocedures are tested before the calling procedures can be debugged. One well-known way to encourage top down debugging is the use of *dummy subprocedures.* To test a calling procedure, a partial definition of the subprocedure is constructed, which need only be sufficient to test the cases used by the calling procedure.

Tinker makes top down debugging very convenient. When we decide we would like to use a subprocedure we haven't written yet, we write the call to it, and define it using the NEW EXAMPLE operation, just having it return the answer for a specific case rather than compute the value in general. This allows us to test the calling procedure. Later, when we are ready to write the definition of the subprocedure, we already have a set of appropriate test cases to guide us in writing the definition.

Now, we type in 0 as the answer for the derivative of the constant 3, and select RETURN a value.



```
Defining (DERIVATIVE 3 (QUOTE X)):
Example: 3, Code: DERIVATIVE-ARG-1
Example: X, Code: DERIVATIVE-ARG-2
      Example: 0, Code: 0
```

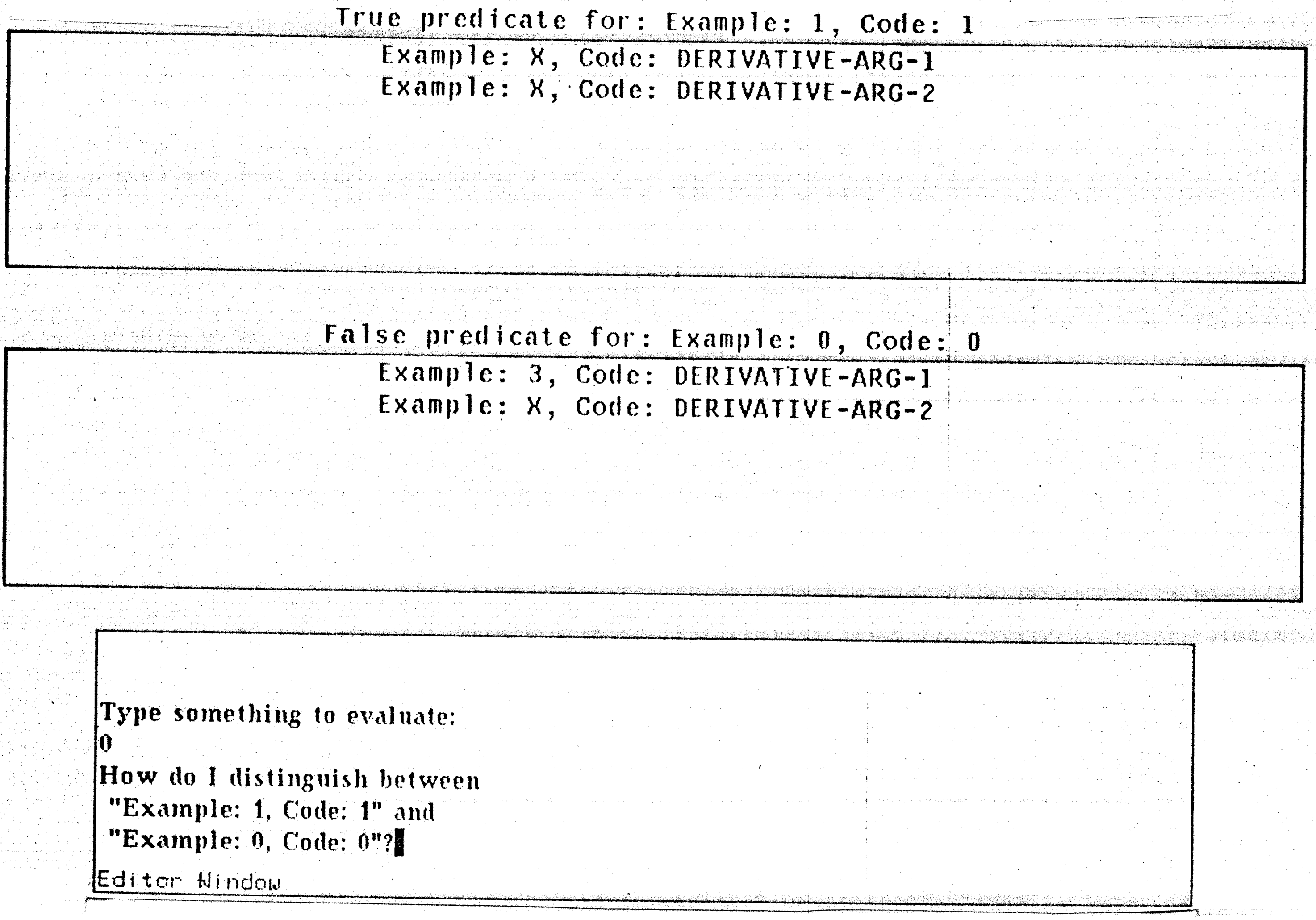
Figure 18. Selecting the menu operation "RETURN a value".

This gives Tinker *two* examples for the DERIVATIVE function,

```
(DERIVATIVE 'X 'X) evaluates to 1
(DERIVATIVE 3 'X) evaluates to 0
```

Tinker *compares* the code for the two cases, and notices that they're different. So Tinker decides that the code for the DERIVATIVE function must contain a conditional. But how should the two cases be distinguished?

Tinker asks us to distinguish between the two cases by defining a *predicate* for the conditional. When defining a predicate, Tinker displays *two* snapshot windows. One will correspond to the *true* case of the conditional, one to the *false* case.

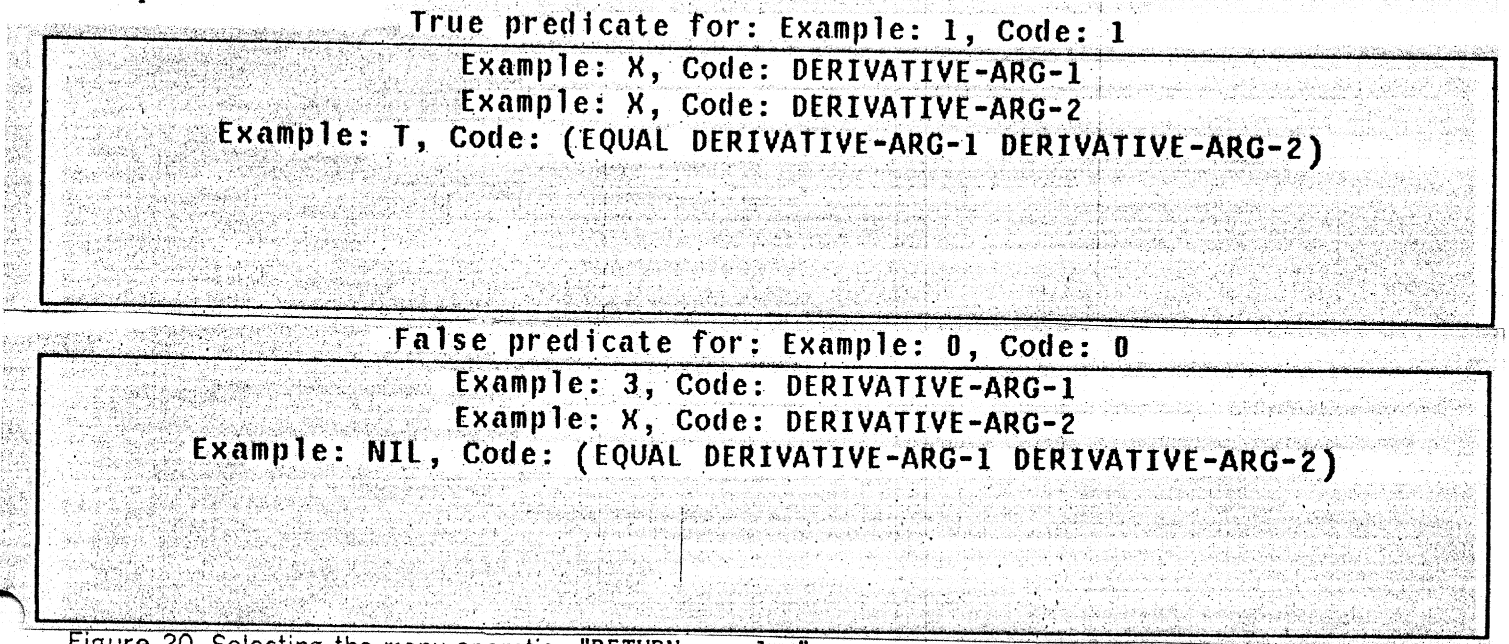


```
True predicate for: Example: 1, Code: 1
     Example: X, Code: DERIVATIVE-ARG-1
     Example: X, Code: DERIVATIVE-ARG-2

False predicate for: Example: 0, Code: 0
     Example: 3, Code: DERIVATIVE-ARG-1
     Example: X, Code: DERIVATIVE-ARG-2

Type something to evaluate:
0
How do I distinguish between
 "Example: 1, Code: 1" and
 "Example: 0, Code: 0"?
Editor Window
```

Figure 19. Selecting the menu operation "RETURN a value".

As we define the predicate, code will appear simultaneously in both windows. Since the values of the arguments to DERIVATIVE are different in the two cases, the code for the predicate may evaluate differently in one window than the other. Hopefully, if we've successfully defined a predicate to distinguish between the two cases, the predicate will evaluate to T (Lisp's way of saying *true*) in the top window, and NIL (Lisp's *false*) in the other.

How do we decide how to choose between the two cases? In the first case, the distinguishing property is that the two arguments to DERIVATIVE are equal. So, we enter the code (EQUAL DERIVATIVE-ARG-1 DERIVATIVE-ARG-2), and see that it evaluates to T in the top window, NIL in the bottom window. We return this as the value of the predicate.

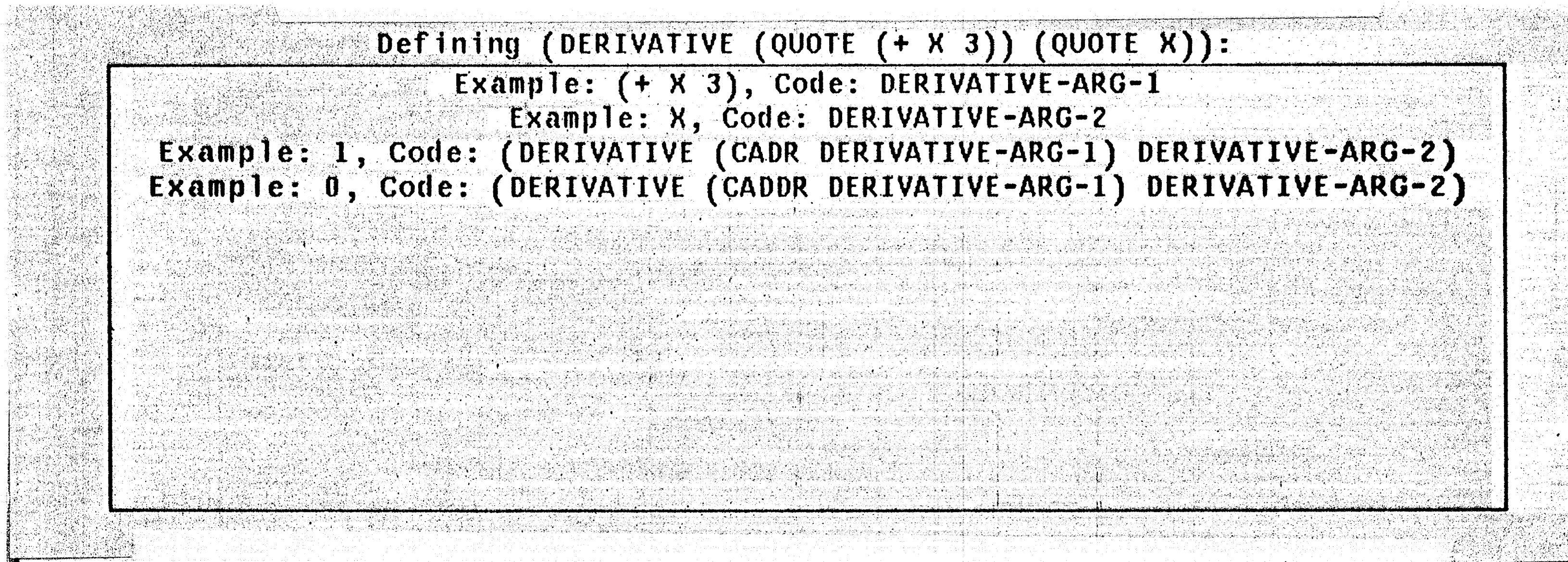


```
                 True predicate for: Example: 1, Code: 1
                   Example: X, Code: DERIVATIVE-ARG-1
                   Example: X, Code: DERIVATIVE-ARG-2
         Example: T, Code: (EQUAL DERIVATIVE-ARG-1 DERIVATIVE-ARG-2)

                 False predicate for: Example: 0, Code: 0
                   Example: 3, Code: DERIVATIVE-ARG-1
                   Example: X, Code: DERIVATIVE-ARG-2
         Example: NIL, Code: (EQUAL DERIVATIVE-ARG-1 DERIVATIVE-ARG-2)
```

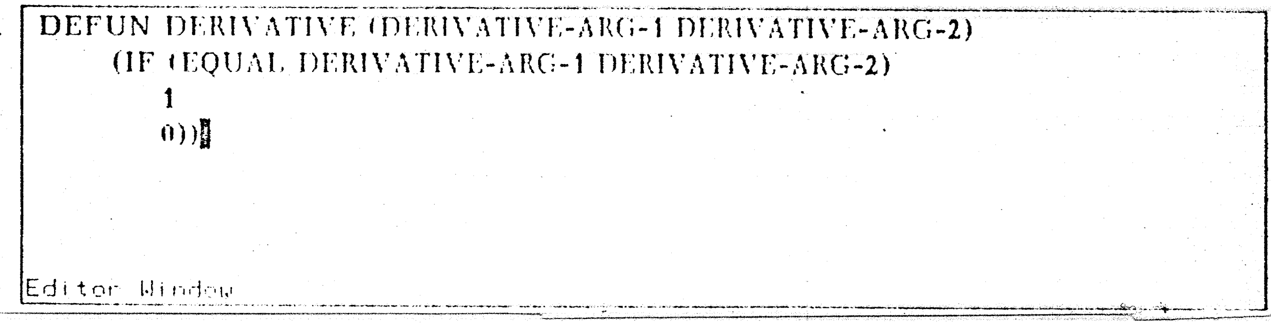
Figure 20. Selecting the menu operation "RETURN a value".

Tinker's displaying both cases of a conditional is valuable in that it helps to assure that the predicate performs its desired function of separating the two cases. Often, the distinguishing predicate is more easily determined after the code for the cases appears. Conditionals can also be introduced explicitly in a more conventional manner with the Make a CONDITIONAL menu operation. We would also like to provide the option of leaving one branch of the conditional undefined while the code for the other branch is worked out.

Tinker returns to the case of defining the derivative of the sum, `(+ X 3)` We examine the code for the `DERIVATIVE` function that Tinker's built up so far. We can see that the code consists of an `if` with the predicate and the two cases we've defined.



```
            Defining (DERIVATIVE (QUOTE (+ X 3)) (QUOTE X)):
              Example: (+ X 3), Code: DERIVATIVE-ARG-1
                 Example: X, Code: DERIVATIVE-ARG-2
Example: 1, Code: (DERIVATIVE (CADR DERIVATIVE-ARG-1) DERIVATIVE-ARG-2)
Example: 0, Code: (DERIVATIVE (CADDR DERIVATIVE-ARG-1) DERIVATIVE-ARG-2)
```

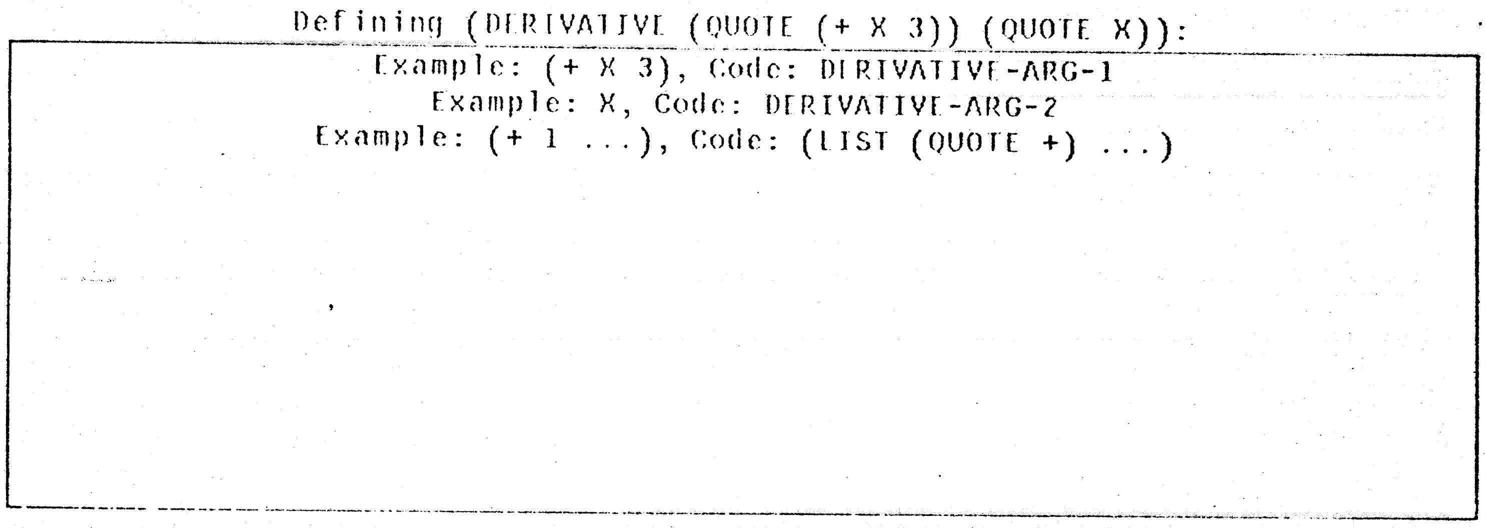


```
DEFUN DERIVATIVE (DERIVATIVE-ARG-1 DERIVATIVE-ARG-2)
    (IF (EQUAL DERIVATIVE-ARG-1 DERIVATIVE-ARG-2)
        1
        0))
```

Figure 21. Selecting the menu operation "RETURN a value".

At this stage, Tinker thinks that in every case where the arguments to `DERIVATIVE` aren't equal, the answer is zero. This isn't correct, of course, but it constitutes a partial definition, correct for what we've shown it so far, and we will continue to extend the definition.

We've computed the derivatives of the subexpressions, now it's time to combine them to form the derivative of the sum. Since the derivative of a sum is the sum of the derivatives, we construct a list of the symbol `+` and the expressions for the derivatives of the subexpressions. Evaluating this results in the list `(+ 1 0)`.

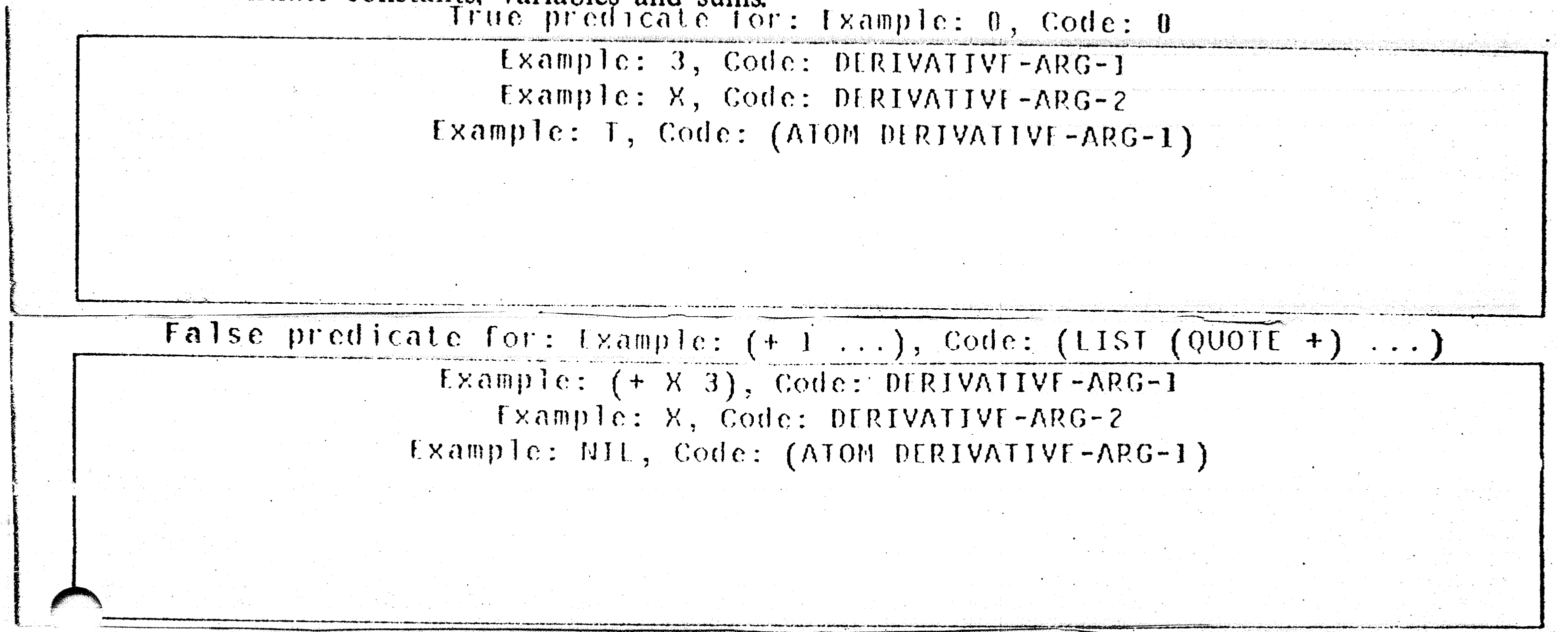


```
Defining (DERIVATIVE (QUOTE (+ X 3)) (QUOTE X)):
      Example: (+ X 3), Code: DERIVATIVE-ARG-1
          Example: X, Code: DERIVATIVE-ARG-2
   Example: (+ 1 ...), Code: (LIST (QUOTE +) ...)
```

Figure 22. Selecting the menu operation "`EVALUATE something`".

(We won't bother trying to simplify the expression for the derivative, so that it could recognize that `(+ 1 0)` is the same as 1.) Since this is the correct answer for the derivative of `(+ X 3)`, we return it.

This provides Tinker with yet another case for the DERIVATIVE function. Again, it shows us two snapshot windows and asks us to define a predicate, contrasting the sum with the last case we showed it, (DERIVATIVE 3 'X). We enter the code (ATOM DERIVATIVE-ARG-1) to distinguish the constant case and return it. Now, Tinker can differentiate constants, variables and sums.

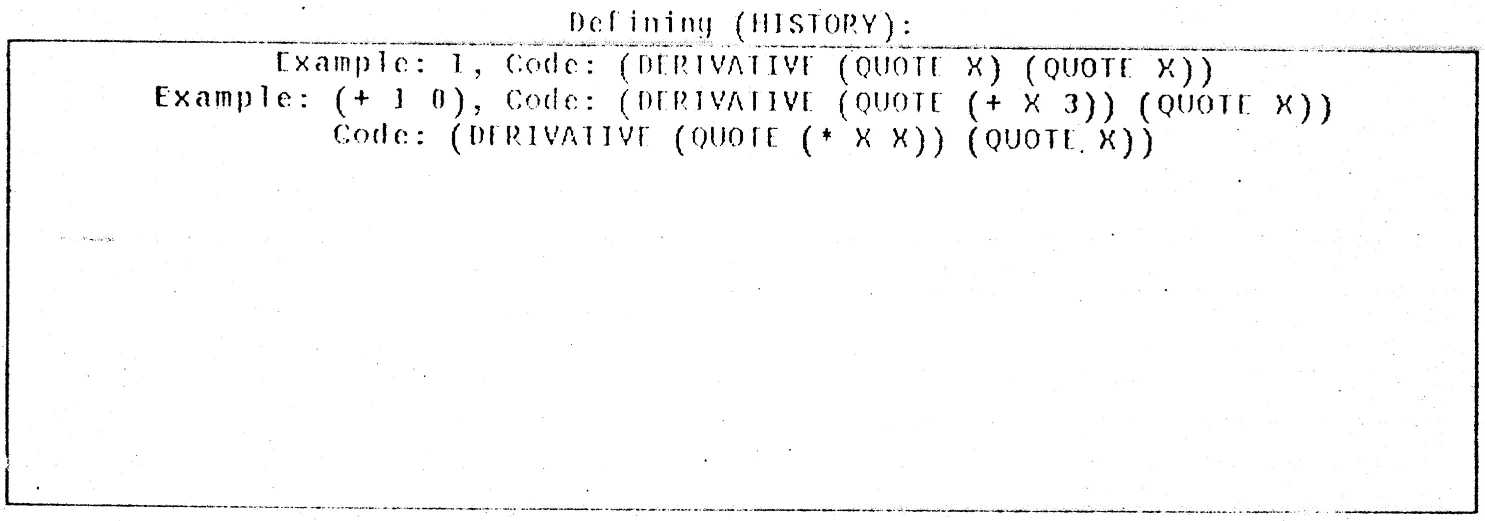


```
True predicate for: Example: 0, Code: 0
         Example: 3, Code: DERIVATIVE-ARG-1
         Example: X, Code: DERIVATIVE-ARG-2
      Example: T, Code: (ATOM DERIVATIVE-ARG-1)

False predicate for: Example: (+ 1 ...), Code: (LIST (QUOTE +) ...)
       Example: (+ X 3), Code: DERIVATIVE-ARG-1
         Example: X, Code: DERIVATIVE-ARG-2
     Example: NIL, Code: (ATOM DERIVATIVE-ARG-1)
```

Figure 23. Selecting the menu operation "RETURN a value".

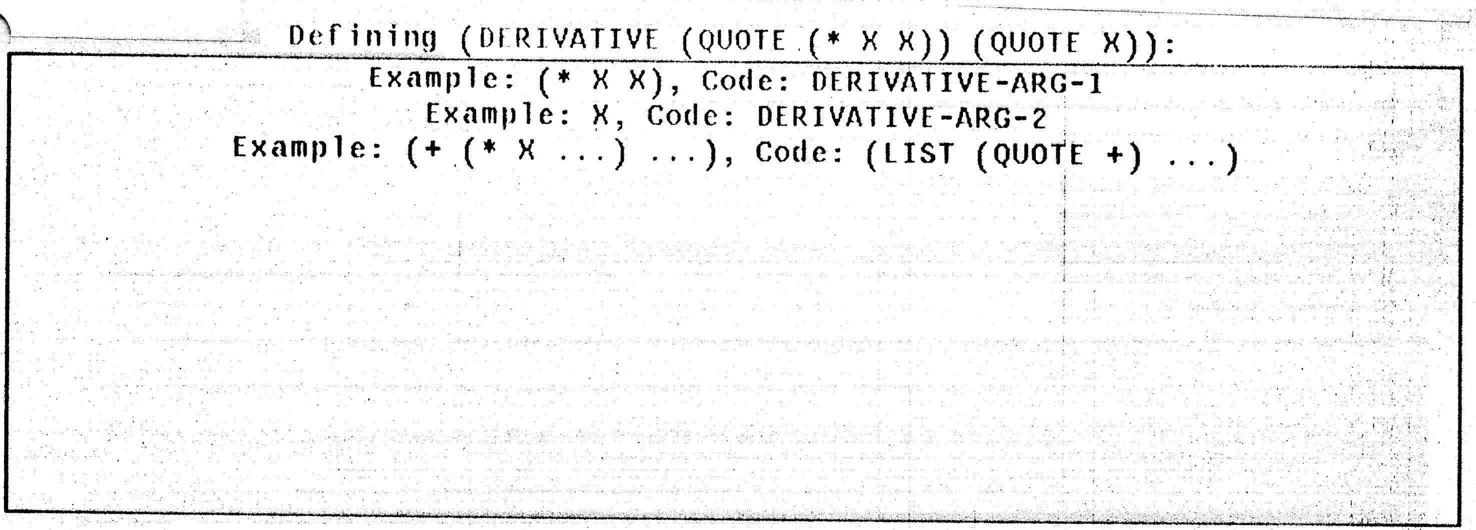
Tinker currently treats multi-way branches or dispatches as nested two-way branches. It is possible to modify it to produce a dispatch directly.

To round out our DERIVATIVE program, we could add another case telling Tinker how to do *products*, using the *chain rule*.



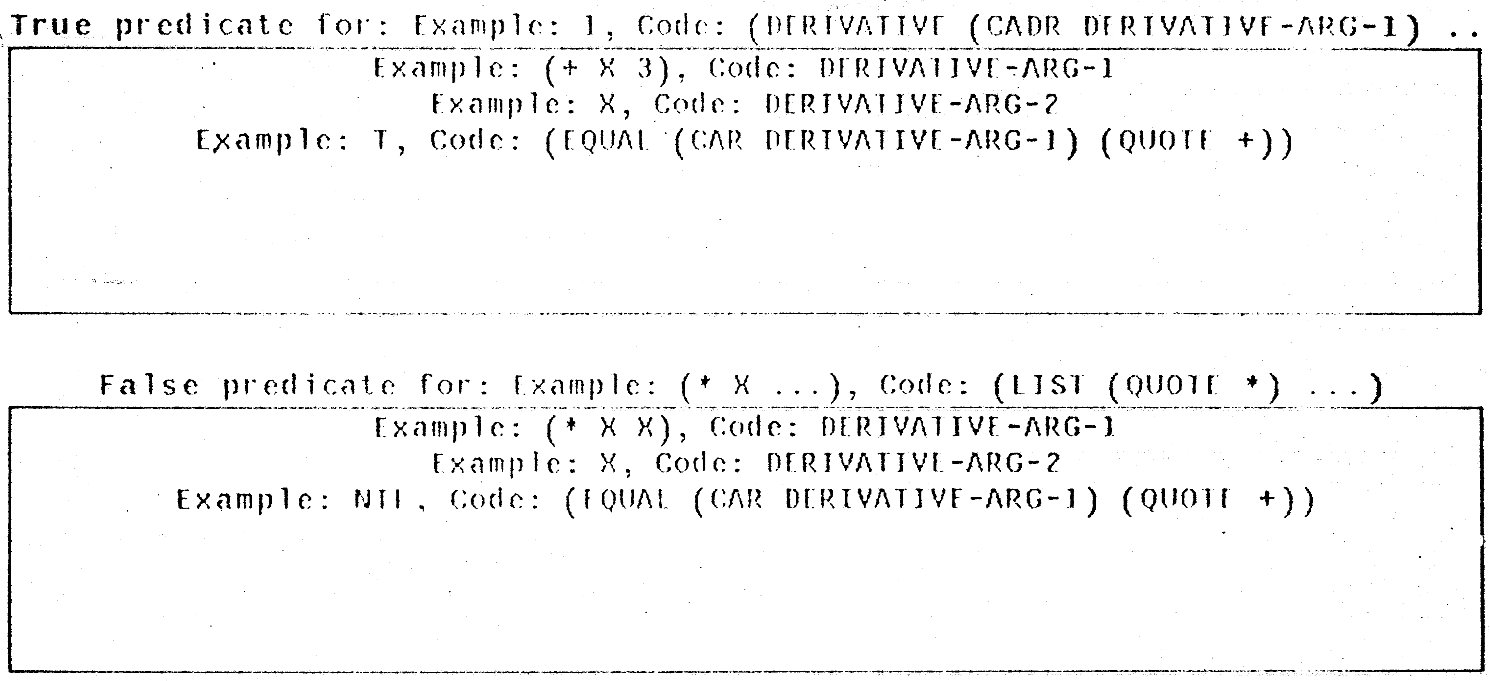
```
                  Defining (HISTORY):
          Example: 1, Code: (DERIVATIVE (QUOTE X) (QUOTE X))
Example: (+ 1 0), Code: (DERIVATIVE (QUOTE (+ X 3)) (QUOTE X))
             Code: (DERIVATIVE (QUOTE (* X X)) (QUOTE X))
```

Figure 24. Selecting the menu operation "NEW EXAMPLE for function".



```
     Defining (DERIVATIVE (QUOTE (* X X)) (QUOTE X)):
          Example: (* X X), Code: DERIVATIVE-ARG-1
             Example: X, Code: DERIVATIVE-ARG-2
Example: (+ (* X ...) ...), Code: (LIST (QUOTE +) ...)
```

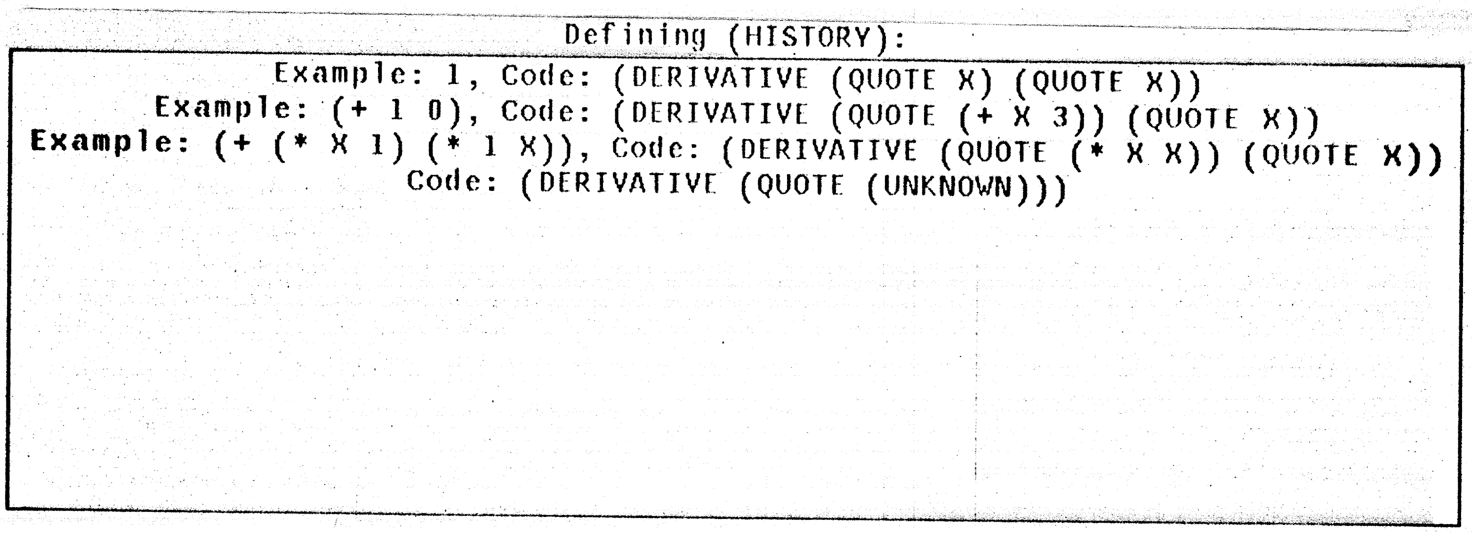
Figure 25. Selecting the menu operation "RETURN a value".



```
True predicate for: Example: 1, Code: (DERIVATIVE (CADR DERIVATIVE-ARG-1) ..
                  Example: (+ X 3), Code: DERIVATIVE-ARG-1
                     Example: X, Code: DERIVATIVE-ARG-2
        Example: T, Code: (EQUAL (CAR DERIVATIVE-ARG-1) (QUOTE +))

     False predicate for: Example: (* X ...), Code: (LIST (QUOTE *) ...)
                  Example: (* X X), Code: DERIVATIVE-ARG-1
                     Example: X, Code: DERIVATIVE-ARG-2
       Example: NIL, Code: (EQUAL (CAR DERIVATIVE-ARG-1) (QUOTE +))
```

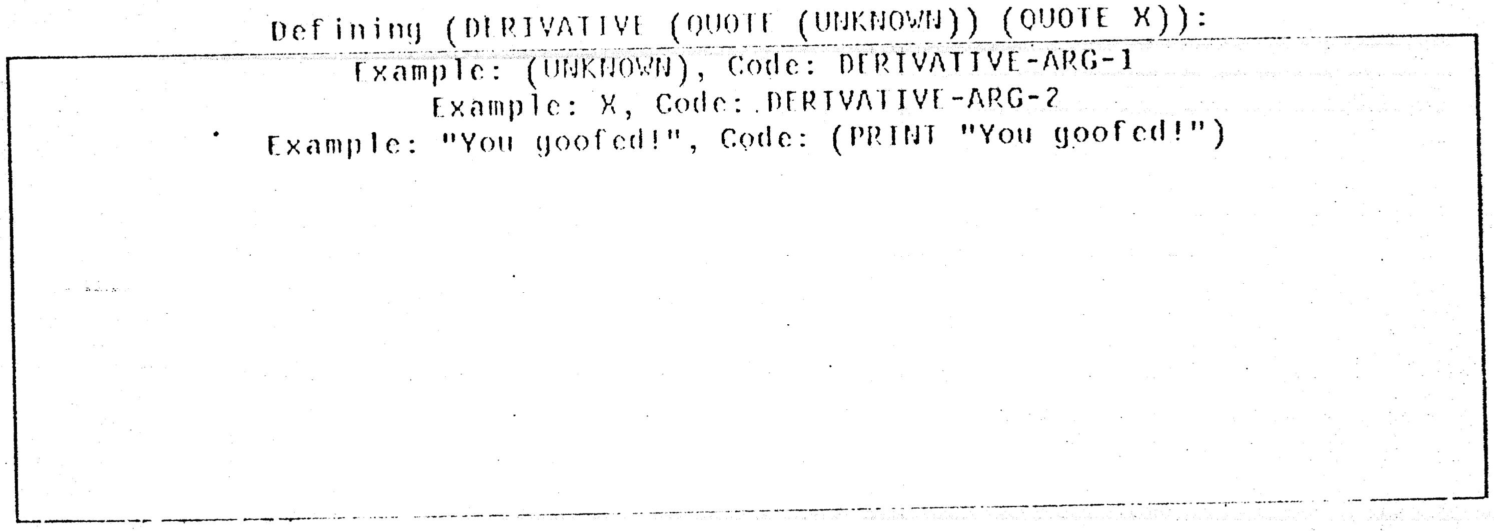
Figure 26. Selecting the menu operation "RETURN a value".

And, if we're really going to be conscientious about our derivative program, we should also define a case where we *can't* take the derivative! (Programmers often neglect to include *negative* examples as well as positive ones when testing a program. This is a prime cause of the fragility of many current software systems.) We illustrate a case where our DERIVATIVE function should produce an error message.



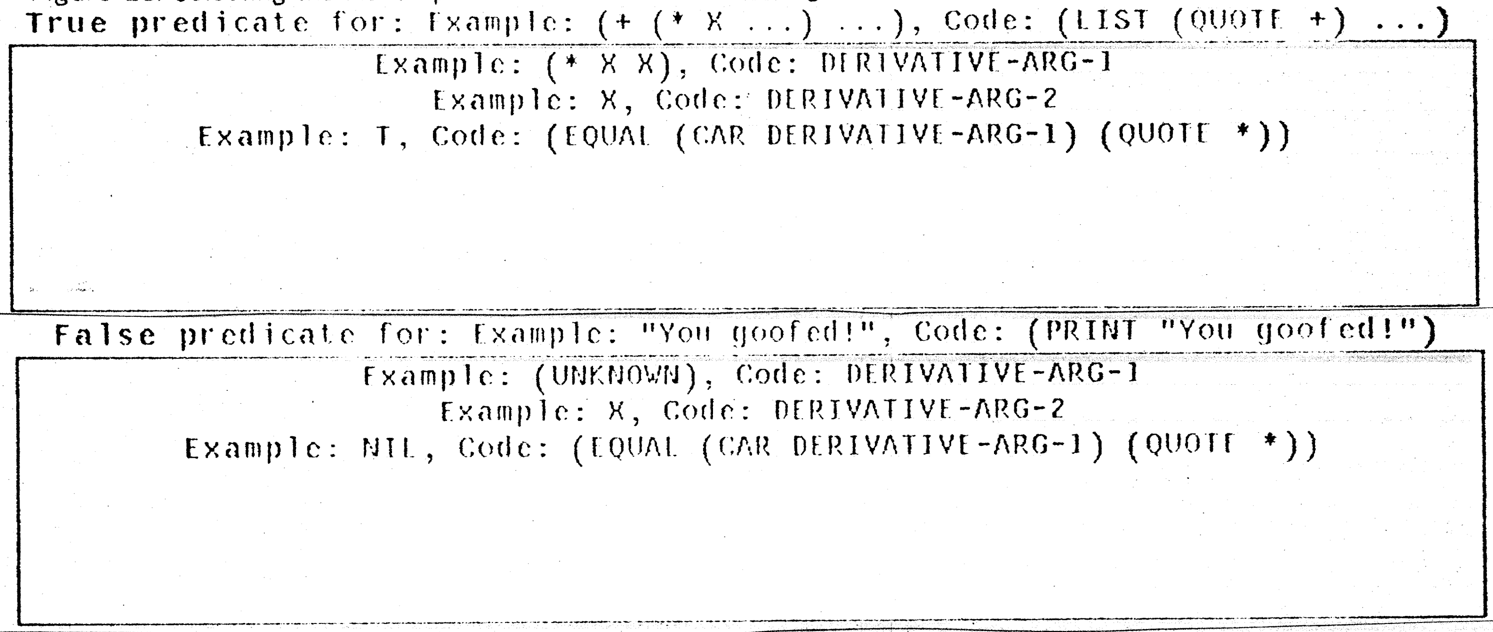
```
                          Defining (HISTORY):
            Example: 1, Code: (DERIVATIVE (QUOTE X) (QUOTE X))
       Example: (+ 1 0), Code: (DERIVATIVE (QUOTE (+ X 3)) (QUOTE X))
Example: (+ (* X 1) (* 1 X)), Code: (DERIVATIVE (QUOTE (* X X)) (QUOTE X))
                  Code: (DERIVATIVE (QUOTE (UNKNOWN)))
```

Figure 27. Selecting the menu operation "NEW EXAMPLE for function".



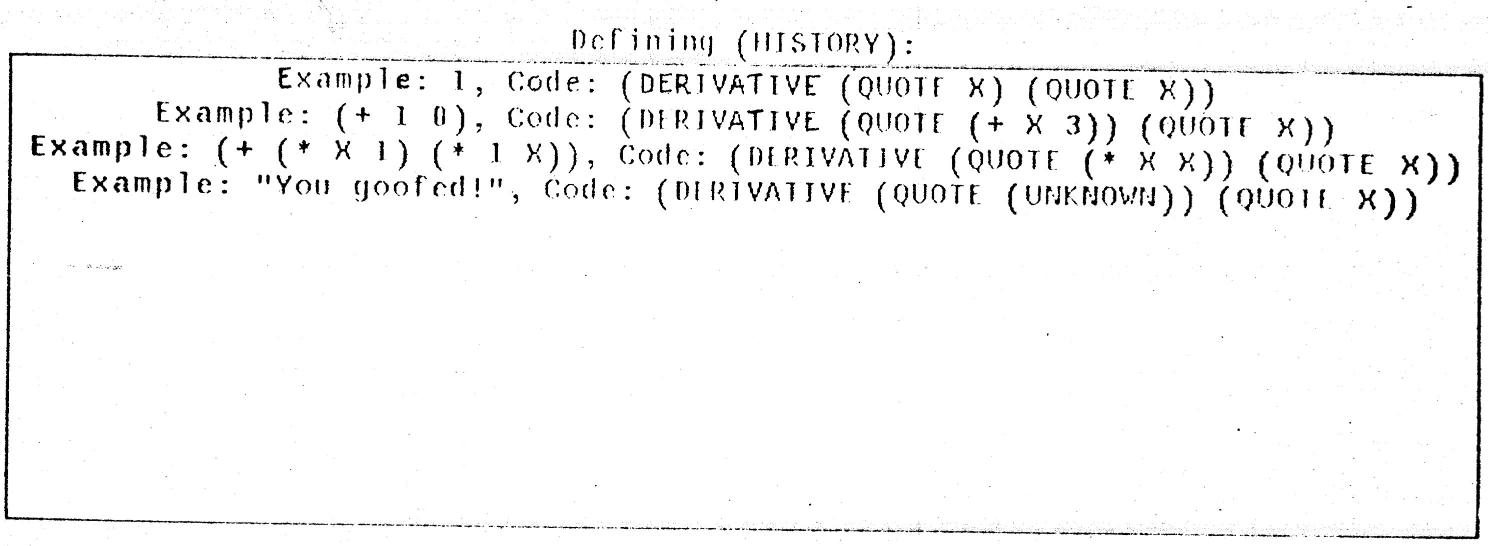
```
Defining (DERIVATIVE (QUOTE (UNKNOWN)) (QUOTE X)):
           Example: (UNKNOWN), Code: DERIVATIVE-ARG-1
                Example: X, Code: DERIVATIVE-ARG-2
     Example: "You goofed!", Code: (PRINT "You goofed!")
```

Figure 28. Selecting the menu operation "EVALUATE something".



```
True predicate for: Example: (+ (* X ...) ...), Code: (LIST (QUOTE +) ...)
              Example: (* X X), Code: DERIVATIVE-ARG-1
                 Example: X, Code: DERIVATIVE-ARG-2
   Example: T, Code: (EQUAL (CAR DERIVATIVE-ARG-1) (QUOTE *))

False predicate for: Example: "You goofed!", Code: (PRINT "You goofed!")
              Example: (UNKNOWN), Code: DERIVATIVE-ARG-1
                 Example: X, Code: DERIVATIVE-ARG-2
   Example: NIL, Code: (EQUAL (CAR DERIVATIVE-ARG-1) (QUOTE *))
```

Figure 29. Selecting the menu operation "RETURN a value".



```
                          Defining (HISTORY):
                Example: 1, Code: (DERIVATIVE (QUOTE X) (QUOTE X))
         Example: (+ 1 0), Code: (DERIVATIVE (QUOTE (+ X 3)) (QUOTE X))
Example: (+ (* X 1) (* 1 X)), Code: (DERIVATIVE (QUOTE (* X X)) (QUOTE X))
   Example: "You goofed!", Code: (DERIVATIVE (QUOTE (UNKNOWN)) (QUOTE X))
```

Figure 30. Selecting the menu operation "RETURN a value".

Collecting all these cases results in the following final program:

```
(DEFUN DERIVATIVE (DERIVATIVE-ARG-1 DERIVATIVE-ARG-2)
       (IF (EQUAL DERIVATIVE-ARG-1 DERIVATIVE-ARG-2) 1
           (IF (ATOM DERIVATIVE-ARG-1) 0
               (IF (EQUAL (CAR DERIVATIVE-ARG-1) '+)
                   (LIST '+
                         (DERIVATIVE (CADR DERIVATIVE-ARG-1)
                                     DERIVATIVE-ARG-2)
                         (DERIVATIVE (CADDR DERIVATIVE-ARG-1)
                                     DERIVATIVE-ARG-2))
                   (IF (EQUAL (CAR DERIVATIVE-ARG-1) '*)
                       (LIST '+
                             (LIST '*
                                   (CADR DERIVATIVE-ARG-1)
                                   (DERIVATIVE (CADDR DERIVATIVE-ARG-1)
                                               DERIVATIVE-ARG-2))
                             (LIST '*
                                   (DERIVATIVE (CADR DERIVATIVE-ARG-1)
                                               DERIVATIVE-ARG-2)
                                   (CADDR DERIVATIVE-ARG-1)))
                       (PRINT "You goofed!"))))))
```

Associating examples with function definitions will yield important benefits during *program maintenance* as well as program construction. When the definition of a function is *edited*, Tinker can run through all the test cases again, automatically, and issue a warning if any examples work out differently with the new definition. Tinker can make sure that changes intended to *extend* the behaviour of some function don't have the effect of *breaking* previously correct code.

6. Other features of Tinker

We will present brief explanations of other features of Tinker which appear on the *Tinker edit menu*, but weren't encountered in the above scenario.

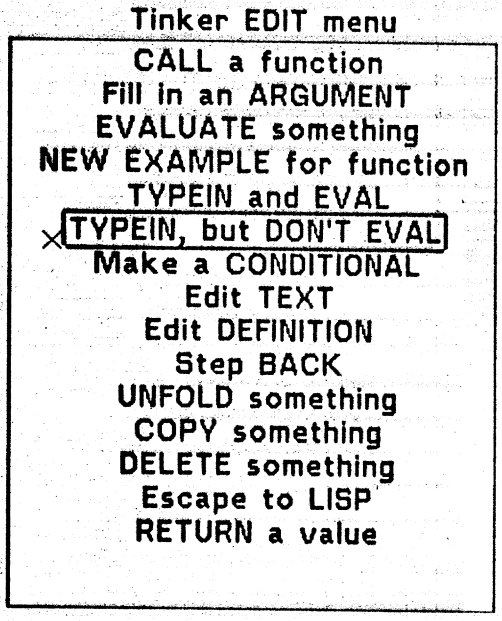


Figure [31]

UNFOLD is a kind of inverse to Fill in an ARGUMENT. It takes apart expressions, removing the arguments from a piece of code. COPY duplicates an expression in the snapshot window. DELETE simply removes an expression from the snapshot window.

Edit DEFINITION is the operation for *modifying* definitions of functions. It chooses a specific example for a function defined with Tinker, and returns to a snapshot window defining that example. Commands can be used to edit the definition which appears in the window, and the definition of the function is suitably changed. We would like to have Tinker run through all the examples again to check to see if they have changed as a result of the edit.

`Step BACK` is a debugging tool. Since Tinker always remembers the code that produced every value which appears on the screen, the Tinker interpreter is completely *reversible!* (Normally, the expense of remembering everything might be prohibitive, but this is only done for functions being debugged with Tinker. After a function is completed, it is compiled, and the overhead disappears.)

When a bug is encountered, we can simply step backwards until the offending expression is found. Since we *select* which expression to examine from the menu, it's easy to skip over details of the evaluation of expressions which are irrelevant to the bug. We can *zoom in* on bugs by examining evaluations at progressively finer levels of detail. This is unlike conventional steppers which always step linearly forward or breakpoints and stack debuggers which always step backward and lose information about evaluated arguments.

We also provide operations which fall back on the more conventional programming tools, so that these can also be used in the cases where they're appropriate. `Edit TEXT` takes the Lisp code representation for an item in the snapshot window, and lets us edit it with the `Zwei` text editor. When we're finished editing the text, we can indicate an expression in the editor and return it to Tinker to replace the originally selected item. Sometimes it's easier to fix things by typing than by menu selection.

`Escape to LISP` puts us in an ordinary Lisp `READ-EVAL-PRINT` loop in the editor window. Any Lisp expression can be evaluated there, and the result is printed back into the editor window. And, as we have seen, any typed Lisp expression can be included in Tinker's snapshot window using the `TYPEIN and EVAL` operation. Thus Tinker can be used to *supplement* existing facilities rather than *replace* them. Tinker provides compatible interfaces to the conventional programming system which make all of its features available as well.

7. Previous work

The most direct ancestor to our work was Smith's *Pygmalion*, which pioneered the idea of menu-oriented programming. Our techniques of displaying examples and source code for programs simultaneously, our direct production of Lisp code, our method of abstracting conditionals, and the use of a set of test cases for each function are some of the contributions which distinguish our work from his. Curry's system is similar in spirit to Pygmalion, and both these systems made interesting use of graphical rather than textual representations of programs, which we have not explored. Biermann has an interesting system which applies automatic programming techniques to synthesizing programs from demonstrations using examples. We see this as a fruitful area for further research, and we intend to experiment with hooking up Tinker to the description system *Omega* of Attardi, Simi and Hewitt to perform reasoning about the structure of programs. Tinker might be useful as a basis for a user interface to program understanding systems such as that of Rich, Shrobe, and Waters. While we are interested in incrementally interleaving program construction with *testing*, they have explored interleaving program construction with *verification*. Ultimately, we would like to integrate all of these.

Acknowledgements

We would like to thank Kenneth Kahn, Giuseppe Attardi, and Maria Simi for many important ideas and interesting discussions on the ideas in this paper.

We would like to thank the MIT Lisp Machine group: Richard Greenblatt, David Moon, Mike McMahon, Howard Cannon, Richard Stallman, Tom Knight, and Jack Holloway, for their excellent work in developing the hardware and software which supports Tinker.

We would like to thank David Canfield Smith for demonstrating his Pygmalion system to us and discussing it with us on a visit to Xerox PARC, and G. J. Curry for lending us a videotaped demonstration of his system.

Hal Abelson, Andy DiSessa, Marc LeBrun, J. C. R. Licklider, Walter Lieberman, Chuck Rich, Howie Shrobe, Luc Steels, Dick Waters, Jerry Barber, William Kornfeld, John McCarthy, Danny Hillis, Jonl White, Seymour Papert, Benoit Mandelbrot, Ron Becker, Robert Grafton, Robert Englemore, John Buxton, Douglas Hofstadter, Scott Kim, deserve thanks for taking the time to participate in demonstrations of Tinker, sharing their reactions with us, and giving us many helpful suggestions, criticisms, and comments.

Bibliography

Biermann, A. W. and Ramachandran K., "Constructing Programs from Example Computations", IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, September 1976.

Curry, G. J. "Programming by Abstract Demonstration" Technical Report No. 78-03-02. Dept. of Computer Science. University of Washington. March 1978.

DeJong, S. P., Zloof, M. "System for Business Automation: Programming Language" Communications of the ACM, May 1978.

DeJong, S. P., Zloof, M. "Query by Example" IBM T. J. Watson Research Center Technical Report

Hewitt, C. "Viewing Control Structures as Patterns of Passing Messages" A.I. Journal. Vol. 8. No. 3. June 1977. pp. 323-364.

Hewitt, C. E. "Procedural Semantics: Models of Procedures and Teaching of Procedures", in Natural Language Processing, Randall Rustin Ed., Algorithmics Press, 1972.

Hewitt, C. and Smith, B. "Towards a Programming Apprentice" IEEE Transactions on Software Engineering. SE-1, #1. March 1975. pp 26-54.

Hewitt, C. "Evolutionary Programming with the Aid of a Programmers' Apprentice" MIT AI Lab Working Paper 188. May 1979.

Hewitt, C.; Attardi, G.; and Lieberman, H. "Security and Modularity in Message Passing" MIT AI Lab Working Paper 180. December 1978. Revised April 1979.

Hewitt, C.; Attardi, G.; and Lieberman, H. "Specifying and Proving Properties of Guardians for Distributed Systems" MIT AI Lab Working Paper 172. December 1978. Revised April 1979. Proceedings of International Symposium on the Semantics of Concurrent Computation. Evian les bains, France. July 1979.

Hewitt, C. and Baker, H. "Laws for Communicating Parallel Processes" Proceedings of IFIP Congress 77, Toronto, August 8-12, 1977. pp. 987-992.

Meyers, Glenford, "The Art of Software Testing", Wiley, 1979.

Rich, C., Shrobe, H.E. and Waters, R.C., Sussman, G.J., and Hewitt, C.E., "Programming Viewed as an Engineering Activity", MIT AI Memo 459, January 1978.

Rich, C., Shrobe, H.E. and Waters, R.C., "Computer Aided Evolutionary Design for Software Engineering", MIT AI Memo 506, January 1979.

Shrobe, H., "Logic and Reasoning for Complex Program Understanding", MIT PhD. Thesis, October 1978.

Simon, H., "The Heuristic Compiler" Memorandum RM-3588-PR. The Rand Corporation. May 1963.

Smith, D. C., "PYGMALION: A Creative Programming Environment", Stanford AIM-260, June 1975.

Waters, R.C., "Automatic Analysis of the Logical Structure of Programs", MIT AI Laboratory TR-492, December 1978.

Weinreb, D., and Moon, D. "The Lisp Machine Manual" MIT AI Laboratory, November 1978.