Constructing Graphical User Interfaces By Example
La Création des Interfaces Graphiques Par Exemple

Henry Lieberman

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

Tinker is an experimental programming environment for Lisp which makes use of graphics in two novel ways: First, a high resolution display and a mouse allows Tinker to replace most typed commands with menu selection operations, and use multiple windows to display different viewpoints on a developing program simultaneously. Second, Tinker uses an example-oriented approach to programming which is especially suited to writing graphics programs. Tinker allows the implementor to *preview* the graphic output of a program *as the program is being written*, by examining example pictures illustrating the effects of each graphics command. As each request for user input is introduced, the implementor supplies example input. Tinker's unique approach to the design of interactive graphical user interfaces is illustrated by showing how to write a portion of a *VisiCalc*-like constraint system.

Tinker est un système expérimental de programmation en language Lisp utilisant des graphiques de deux nouvelles manières. Premièrement, au lieu de taper des directives, un écran de haute resolution et une "souris" permettent au Tinker d'employer la technique de sélection de menu et employer plusieurs fenêtres pour afficher a l'écran des differents points de vue simultanément en cours. Deuxièmement, Tinker utilise en programmation une approche orienteé vers des exemples specialement appropriеés pour écrire les programmes graphiques. Tinker permet au programmeur d'observer les affiches graphiques tout en écrivant des programmes. A chaque demande de l'utilisateur, le programmeur presente un exemple des donnees en entrée. Ce papier presente un exemple de programmation avec Tinker; la création d'une petite partie d'un système orienté vers des contraintes, semblable à *VisiCalc*.

1. VisiCalc dynamically enforces constraints between variables

Currently, one of the most successful programs in the microcomputer community is Software Arts, Inc.'s *VisiCalc* [2]. VisiCalc ("visible calculator") presents the user with a two-dimensional grid of *boxes*, and each box may contain either a concrete value like a number or text string, or an *expression*. The expression computes the value of the box in terms of the contents of other boxes and arithmetic or other operations. For example, a box `C1` might be constrained to be the sum of two other boxes `A1` and `B1`. The user may then *edit* values in the boxes, and when a value is changed, VisiCalc *recomputes* the values of all boxes which depend upon the changed box, to maintain the constraint. This is a powerful feature, as it allows the user to anticipate the effects of changes in a complex system of relationships between variables, by asking the system what would happen if certain values were changed.

2. Tinker exploits some of the same user interface principles as VisiCalc

Why has VisiCalc been so successful in the software market? Part of the reason for the tremendous success of VisiCalc is to be found in certain characteristics of the user interface that make the system easy to learn and use. VisiCalc's grid of boxes continuously provides a display of the user's *current state*, so the user can always answer the question "Where am I?" in the midst of solving a problem.

Since VisiCalc displays the *value* of a symbolic expression stored in a box, the user can verify that a symbolic expression had the correct result in the particular case in which it is being used. VisiCalc thus allows a kind of *programming with examples*, which contributes to its ease of use, since people often find it easier to think about specific examples than to reason about the properties of abstract expressions.

Furthermore, as soon as any change is made to a box, any boxes affected by the change are immediately redisplayed to reflect the change. Thus, the user gets *immediate graphical feedback* on the result of an action. The user can verify right away that the action had the intended result, by checking that the new "current state" conforms to expectations. If an error is made, it can be corrected on the spot, before further actions are taken.

These principles of user interface design:

> *Always display the current state of a process.*
> *Use concrete values to show examples of symbolic expressions.*
> *Provide immediate graphical feedback to the user of the result of each action.*

which make VisiCalc so congenial to use can profitably be applied to environments for general programming as well.

Tinker is an environment for writing and debugging programs in Lisp which uses these principles to help make programming easier and more reliable. Tinker encourages programming using an *example-based* strategy. When you want to define a new function using Tinker, you present an example of the kind of input data you would like the function to accept, and work out the steps of the

procedure on the example data. Each time a step of the procedure is performed, Tinker does two things. First, Tinker displays the result of the step in the example case supplied. Tinker also remembers the code which was responsible for each step. When the desired result for the function has been computed, Tinker abstracts a program for the general case from the examples demonstrated.

Tinker's display always tracks the "current state" of a procedure in the process of being defined, since it shows the result of each step in the example situation as that step is introduced into the program. Showing the intermediate states the program goes through in typical examples can make it much easier for the programmer to decide what the next step in the program should be. Every Lisp expression introduced into the program is accompanied by concrete values which express the result of evaluating that expression in specific situations. Looking at concrete examples often reflects the behavior of the procedure more accurately than trying to infer the general behavior of the program from looking at the code alone. Every action taken by the programmer is reflected immediately in updates to the display, showing both values computed by that action and side effects such as graphical output caused by that action. Seeing the effect of an action immediately makes it easier to verify that the action was behaving properly, and gives the opportunity to change that action while the matter is still fresh in the programmer's mind.

3. Tinker is especially helpful in designing programs with interactive graphical interfaces

The remainder of this paper will show an example of programming with Tinker. Appropriately enough, we will illustrate Tinker's methodology by demonstrating how to program a small portion of a VisiCalc-like constraint system. Tinker's example-based programming style should prove especially important in constructing interactive systems which adhere to the user interface principles we discussed above.

What do we mean by an "example" of an interactive system? When programming an interactive system with Tinker, the programmer is supplied with a *window* on the screen which represents *the user's view of the system.* This window will show exactly what a hypothetical user might see during a typical session with the completed system. Whenever code is written to display graphic output to the user, samples of that output will appear. Whenever code is written to request input from the user, the implementor is requested to supply sample input to the program. Since the implementor is required to ***play the role of the user*** while developing the program, the implementor can more easily verify that the user interface "feels right".

4. A simplified problem: enforcing a sum constraint

We will vastly simplify the problem in order to give a crisp illustration of the basic ideas without getting too bogged down in details in such a short paper. Instead of implementing an entire VisiCalc system, we will implement a much less ambitious project. We will display on the screen a set of three boxes, which the user can edit to contain numbers. The boxes might represent the fields of some office form, if the application were an office information system. Our program will maintain a *sum* constraint, so that the value in the third box is always the sum of the other two. If the user changes a number in a box by editing, our system will recompute the total to keep the constraint satisfied.

Although our example project is simple, it illustrates a number of important concepts in the design of interactive user interface programs. Many user interface programs are characterized by a top level *command loop*, which interprets commands input by the user. Several different methods of obtaining input from the user for arguments to commands may be used, including typing and selection with a pointing device such as the mouse. Typically, initialization must be performed before the command loop is started and cleanup must follow exit from the command loop. Commands which alter the state of objects being displayed on the screen must redisplay their state after each iteration of the command loop, so that the user always sees the "current state" of the environment.

We will define our system to Tinker by working out an example session with our proposed system. Part of Tinker's display layout is always one window which shows the *user's-eye view* of the program being written. If we write code to display something on the screen, an example of what the display should look like will appear in that window. If we write code requesting input from the user, we will provide example input in that window, either by typing or mouse actions. The examples we will use will demonstrate displaying the initial screen layout, interpreting commands from the user, and displaying state information on the screen.

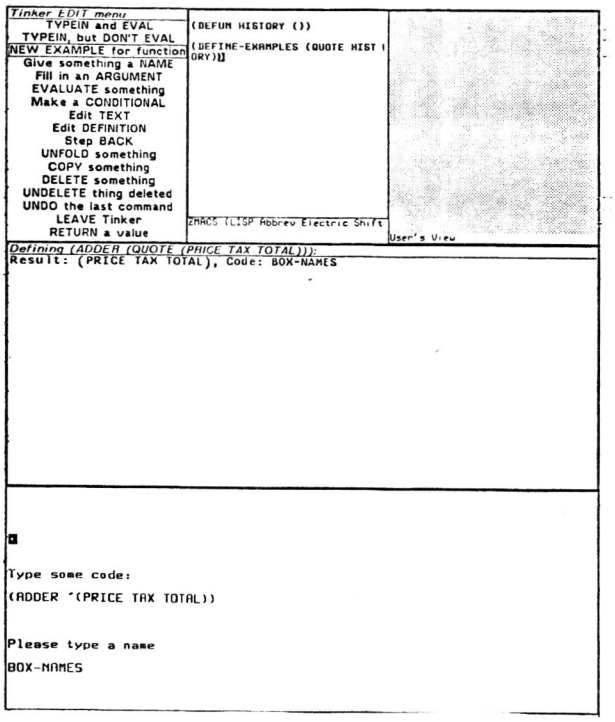
Our system will be implemented in a top-down fashion. We will start with the top level functions, and when an example which motivates the need for a new subroutine is encountered, we will introduce lower level subroutines.

5. We display a set of boxes on the screen to initialize the system

In Tinker, a function is defined by first presenting an *example.* Suppose we decide that our top level function is to be called ADDER. The ADDER function is to take one argument, a list of names for the boxes to be displayed on the screen. Had we already defined the ADDER function, we could test it out creating three boxes named PRICE, TAX, and TOTAL by evaluating Lisp code like this: (ADDER '(PRICE TAX TOTAL)). Instead, we give this same code to Tinker as a new example for the function ADDER.

At the upper left corner of the screen, Tinker always displays a menu of commands. We start off the new example by choosing the menu operation named Typein, but DON'T EVAL. At the bottom of the screen, Tinker prompts us to type in some code, and we respond with the Lisp expression (ADDER '(PRICE TAX TOTAL)). Then, we select the menu operation NEW EXAMPLE for function to indicate that this is the first example for the function ADDER. We use the command Give something a NAME to name the argument to ADDER BOX-NAMES.

The screen now looks like this:

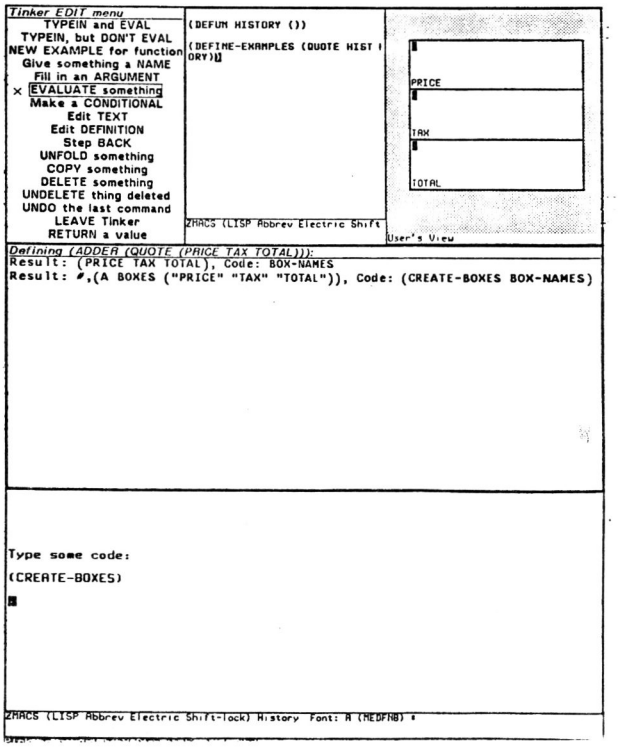


In the center of the screen is the *snapshot window*, whose title line informs us that the example we're currently defining is (ADDER '(PRICE TAX TOTAL)). Inside the snapshot window are objects which represent a piece of Lisp code, and its associated value *in the current example*. We see an object representing the argument to ADDER, whose RESULT part is (PRICE TAX TOTAL), and whose CODE part is a variable called BOX-NAMES.

The first action we'd like the ADDER function to take is to display some boxes on the screen, using the given names as labels for the boxes. We assume that there has already been defined a function named CREATE-BOXES which can create and display boxes on the screen, given a list of names, and a specification of a rectangle on the screen. The rectangle is constructed by using the mouse to indicate two points on the screen, which fix the upper right and lower left corners of the boxes. We can make a design choice whether to include the size of the boxes as a constant in the program, or let the user specify the size of the boxes when the program is run.

Now, we would like to supply the CREATE-BOXES function with the example list (PRICE TAX TOTAL) so it can display the boxes on the screen. But we don't mean that the ADDER program should *always* use the specific list (PRICE TAX TOTAL), but rather that this list is just to be considered a place-holder for *whatever list is given as the argument to* ADDER. So we would like the CREATE-BOXES function to use the *value* of the argument variable, the list (PRICE TAX TOTAL), when displaying the boxes, but the *name* of the argument variable BOX-NAMES should appear in the code for the call to CREATE-BOXES. Tinker's rules for constructing expressions say that you can use the RESULT part of something displayed in the snapshot window as an argument to some other function, and the CODE part will appear in the function being defined.

We select the operations Fill in an ARGUMENT and EVALUATE something to complete the call to CREATE-BOXES. The boxes appear on the screen, labelled with their names, and a list of objects representing the boxes on the screen is returned. The screen shows us an example of what the completed system might look like to the user upon startup.



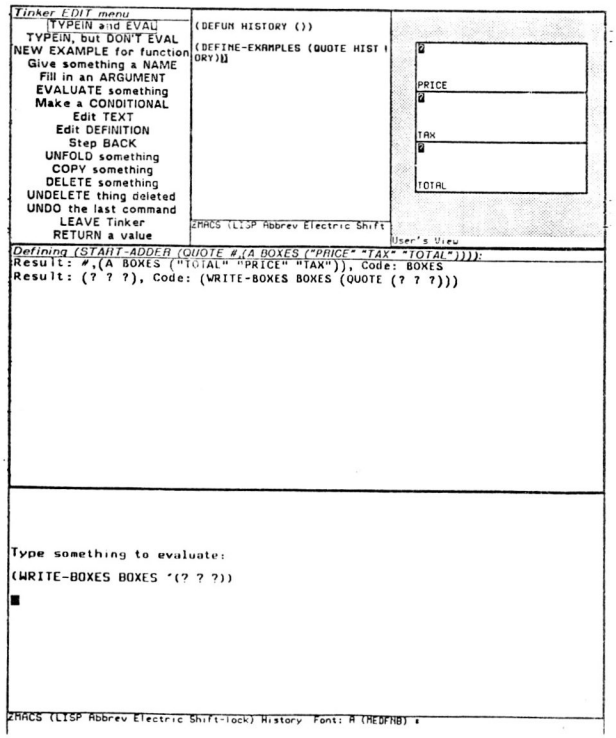
6. We define an initialization function by recursively presenting a new example

The next step we will take is to define an auxiliary function to perform initialization. Tinker has the ability to create new subroutines by *recursively presenting new examples*. We can construct a call to a subroutine which has not yet been written, and use this as a new example to define the subroutine. We'll call our initialization function START-ADDER, and pass it the list of boxes as an argument, which will be named BOXES. We now choose the menu operation NEW EXAMPLE for function.

We move from working on the definition of ADDER to working on the definition of START-ADDER. When the definition of START-ADDER has been complete, Tinker will return us to defining ADDER.

The first action taken by START-ADDER will be to initialize the boxes with "empty" or "unknown" values. The user will later fill in values by *editing* the contents of the boxes. An already-existing procedure named CREATE-BOXES takes a list of boxes and a list of contents and writes the contents into the boxes, displaying the contents in the boxes on the screen. Needing to represent empty boxes in some way, we adopt the convention that a box containing a question mark will be considered to have no value.

The next picture shows the screen at this point, defining the function START-ADDER, which so far has just written question marks into the boxes to initialize them.



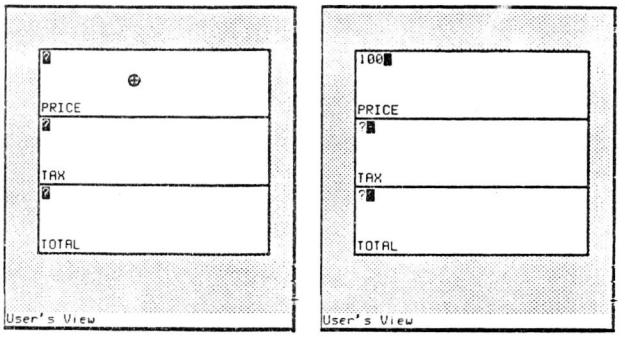
7. The implementor plays the role of the user in editing the boxes

The main body of the our program will consist of a *command loop*, in the style common to many interactive programs such as editors [9], [3]. The program will first accept input from the user, then update the data structures, redisplay the new state, and repeat this loop continually until the user decides to signal an exit from the program.

Since Tinker defines functions by using examples, we must provide examples of the user's input to the program. In our case, that means whenever code which asks the user to edit boxes is introduced into the program, Tinker asks *us* to edit the example boxes we have supplied. The result of that editing operation supplies the example input. Not only can we see an example of how the system will appear from the user's viewpoint, but we also get to use the system by providing example input and working out the steps showing how the system will respond to the example input. Tinker's ability to let the implementor preview the user interface as the program is being written should sharpen the implementor's ability to empathize with the needs of the user.

We call the function EDIT-BOXES, which asks us to choose one of the three boxes to edit by pointing to the desired box with the mouse and pressing a button. The user can use a text editor [11] to change the text in the box. When the user signals that the editing has been completed, the new state of the boxes is returned.

The next two pictures show choosing a box to edit with the mouse, and replacing a value in one of the boxes.



We have changed the value of the top box labelled PRICE to 100. The value of the EDIT-BOXES operation is a list of the values of the boxes, two of which still have unknown values, so the result is (100 ? ?). We shall refer to this as the STATE of the boxes.

8. The command loop enforces the sum constraint and updates the boxes

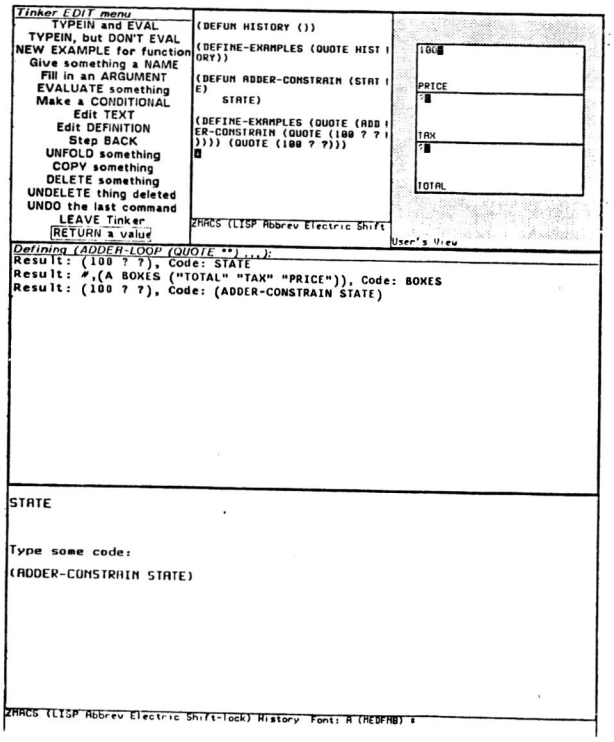
We now construct the function which is going to be the command loop of the system, named ADDER-LOOP. ADDER-LOOP takes as argument the list of boxes, along with the current state as returned by EDIT-BOXES.

The command loop now must take two actions: It must examine the state and enforce the sum constraint between the values in the boxes, and it must new state back into the boxes and update the display.

We invent a new *state transition* subroutine which we will call ADDER-CONSTRAIN, whose job it is to compute a new state given the current state of the boxes. Computing the new state may need to be done in various ways, depending on the state itself. We will have to present *several* examples for the function ADDER-CONSTRAIN, not just one. Each example will illustrate an important case for the resulting function, whose code will contain a conditional that will distinguish between the various cases.

In this example, we have only filled in one value of the three boxes, so as yet we do not have enough information to compute a sum. This situation will change in subsequent examples for ADDER-CONSTRAIN. But for now, the new state after enforcing the constraint is identical to the old state, so we just return the old state as the value for ADDER-CONSTRAIN. We point to the state variable, and indicate that this is the value of the function by using the menu operation RETURN a value.

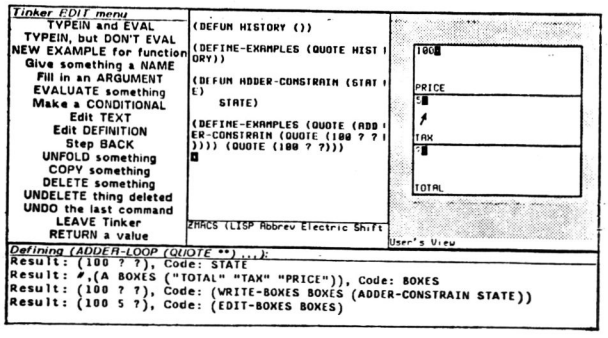
This completes a definition of ADDER-CONSTRAIN, and the code for ADDER-CONSTRAIN appears in the *function definition* window at the top center of the screen. Tinker returns us to the midst of the definition of the function ADDER-LOOP, where we left off.



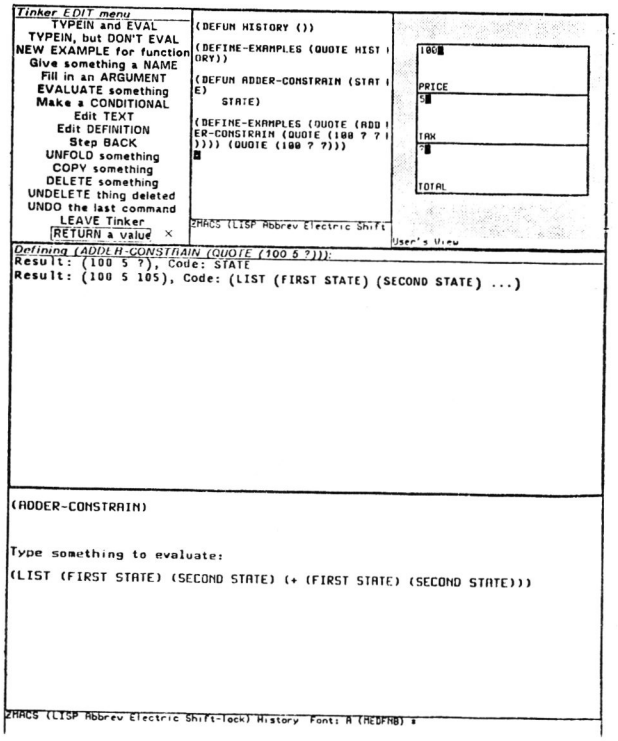
So far, the definition of ADDER-CONSTRAIN is quite trivial, but that situation will soon be rectified as we provide additional examples for that function. The next action that needs to be taken is to write the new state returned by ADDER-CONSTRAIN into the boxes to update the display. This is accomplished by calling the function WRITE-BOXES.

9. Another iteration of the command loop shows computing a sum

After having completed one cycle of the command loop, Tinker must be told to repeat the loop. Again, we use the function EDIT-BOXES to change the contents of one of the boxes. In this case, we choose the middle box, labelled TAX, with the mouse, and change its contents to be 5.



Now, the state returned by EDIT-BOXES is the list (100 5 ?). The values of the first two boxes are now known, but the value of the last box is still unknown. Our task it to compute it automatically to maintain the sum constraint, so our goal should be to have the sum 105 appear in the TOTAL box. We do so by presenting another example for the function ADDER-CONSTRAIN, this time with the list (100 5 ?). The job of ADDER-CONSTRAIN in this case is to compute the sum of the first two values, and return a list with the sum as the third value. We put together the list (100 5 105), which was produced by the code (LIST (FIRST STATE) (SECOND STATE) (+ (FIRST STATE) (SECOND STATE))), and return it as the value for ADDER-CONSTRAIN.

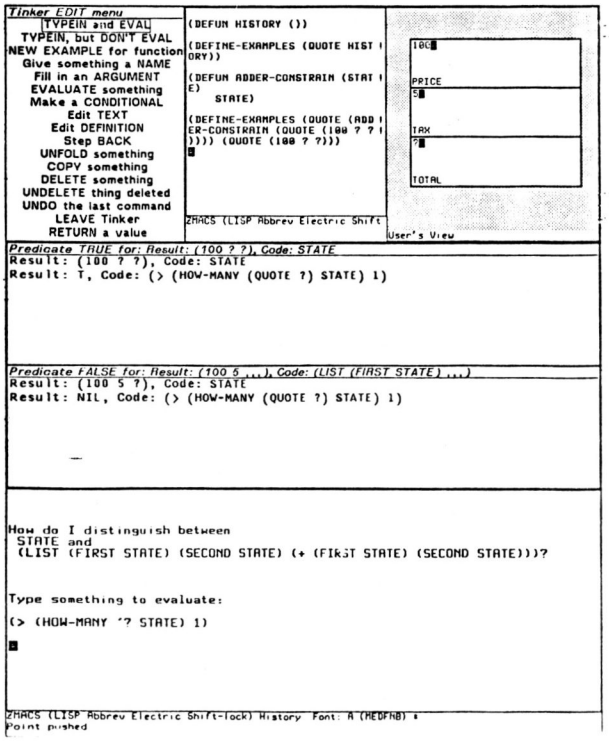


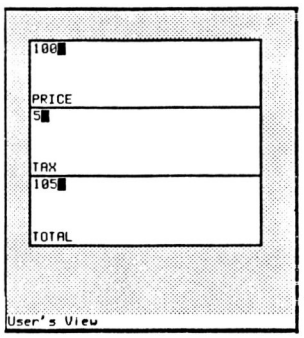
10. Two examples for the same function introduce conditional definitions

Now, we have two examples for the function ADDER-CONSTRAIN. Whenever Tinker has two completed definitions for the same function, it must have some way of *distinguishing* between the two cases. Tinker assumes that since the definitions for the two examples are different, the intention of the programmer was to create a *conditional* definition, with each example representing an equivalence class of arguments to the function. What Tinker now needs to know is a *predicate* that divides the equivalence classes, so the function can decide to which equivalence class a particular set of arguments belongs.

In keeping with Tinker's example-oriented style, the predicate for the conditional is also defined *by example*. Tinker presents to us two examples simultaneously, one showing each situation. Tinker replaces the usual snapshot window in the center with two snapshot windows, the top one in this case displaying the situation where the

STATE is (100 ? ?), the bottom one showing STATE as (100 5 ?). Code constructed using the menu operations and typing will appear in both windows simultaneously, but might evaluate differently in the two windows, due to the difference in the variable environments. Our goal is to produce a predicate that yields *true* in the top window, and *false* in the bottom window. To distinguish between the two cases, we will say that since the top window has more than one unknown value, there is not enough information to compute the sum. We will count the number of question marks indicating unknown values and see if the count is greater than one.





This completes the definition of ADDER-CONSTRAIN, and Tinker writes the following code for us.

```
(DEFUN ADDER-CONSTRAIN (STATE)
       (IF (> (HOW-MANY '? STATE) 1)
           STATE
           (LIST (FIRST STATE)
                 (SECOND STATE)
                 (+ (FIRST STATE) (SECOND STATE)))))
```

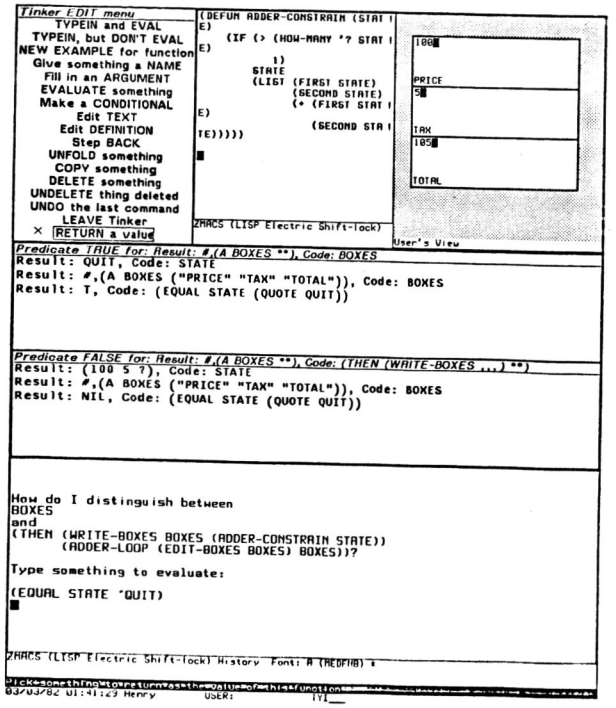
We could augment this definition further by supplying additional examples, such as computing either the PRICE or TAX from the other by subtracting from a known TOTAL. In this way Tinker allows a programmer to incrementally add new expertise to a program by supplying new examples that illustrate use of the new feature. At each point, a partial definition is available which captures all the knowledge put in so far.

11. We must show Tinker how to exit the command loop

We can't forget that there has to be some way of ending the loop.

We can do this by performing another iteration of ADDER-LOOP, but this time instead of editing a box, we will hit a special key which says we want to stop editing. This key causes the EDIT-BOXES function to return the symbol QUIT instead of the usual list of contents of the boxes.

This constitutes another example for the function ADDER-LOOP, in which it just returns, rather than computing a new state and iterating as before. Now the situation resembles the one for ADDER-CONSTRAIN earlier. We have two examples for the function ADDER-LOOP and must supply a predicate to distinguish between them. The predicate sees whether the STATE variable is equal to the symbol QUIT, indicating an exit from the loop.



This yields the definition for ADDER-LOOP below.

```
(DEFUN ADDER-LOOP (STATE BOXES)
       (IF (EQUAL 'QUIT STATE)
           BOXES
           (PROGN
            (WRITE-BOXES BOXES (ADDER-CONSTRAIN STATE))
            (ADDER-LOOP (EDIT-BOXES BOXES) BOXES))))
```

Finally, as a cleanup step, we write code that uses the function DISAPPEAR to cause the boxes to be removed from the screen. READ-BOXES is used to return a list containing the final values of the boxes from the function START-ADDER.

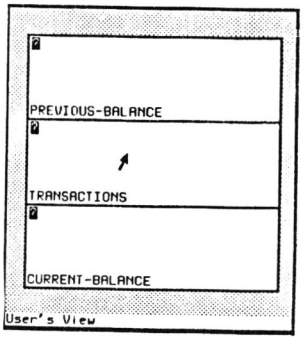
This completes both START-ADDER and ADDER.

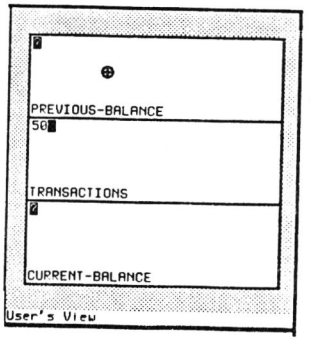
```
(DEFUN ADDER (BOX-NAMES)
       (START-ADDER (CREATE-BOXES BOX-NAMES)))

(DEFUN START-ADDER (BOXES)
       (WRITE-BOXES BOXES '(? ? ?))
       (ADDER-LOOP (EDIT-BOXES BOXES) BOXES)
       (DISAPPEAR BOXES)
       (READ-BOXES BOXES))
```

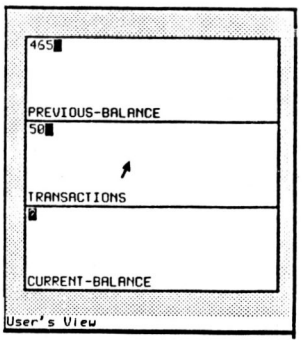
12. Let's try the completed program on another example

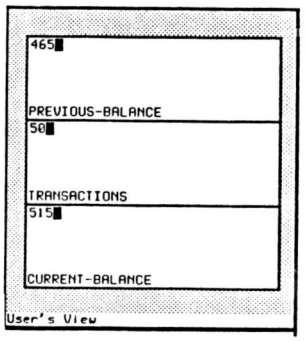
Since Tinker interleaves program testing with program construction, as soon as we've finished defining the ADDER program, we've also completed testing it on a representative example scenario. Thus Tinker should increase the programmer's confidence in the reliability and robustness of programs. But just to make sure, we should test the finished program on another example to verify that it really does work.

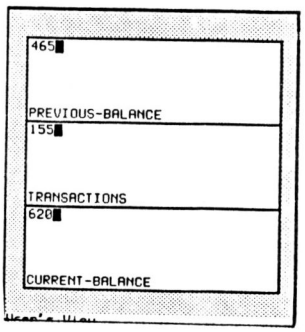
The following pictures show several steps of the completed ADDER program on a set of boxes labelled PREVIOUS-BALANCE, TRANSACTIONS, and CURRENT-BALANCE. Notice the order of editing steps isn't exactly the same as in our first example. After computing the first sum, we can change the value of an already known box, and the sum is recomputed to reflect the change.











Bibliography

[1] Giuseppe Attardi and Maria Simi, The Power of Programming by Example, Workshop on Office Information Systems, Saint-Maximin, France, October 1981

[2] Robert Franskton, VisiCalc: The Visible Calculator, The Non-Expert User, in 73rd Infotech State of the Art Conference, Zurich, 1980

[3] Eugene Ciccarelli, Presentation Based User Interfaces, Memo WP-219, MIT Artificial Intelligence Lab, 1982

[4] Gael J. Curry, Programming By Abstract Demonstration, PhD Thesis, U. of Wash. at Seattle, 1978, Report 78-03-02

[5] Daniel Halbert, An Example of Programming by Example, MS Thesis, U. of Cal., Berkeley 1981

[6] Henry Lieberman and Carl Hewitt, A Session With Tinker: Interleaving Program Design With Program Testing, The First Lisp Conference, Stanford U., Aug. 1980

[7] Henry Lieberman, Example-Based Programming for Artificial Intelligence, 7th International Joint Conference on Artificial Intelligence, Aug. 1981

[8] Henry Lieberman, Seeing What Your Programs Are Doing, AI Memo 656, MIT Artificial Intelligence Lab, 1982

[9] William Newman and Robert Sproull, Principles of Interactive Computer Graphics (2nd edition), McGraw-Hill 1979

[10] David Canfield Smith, Pygmalion: A Creative Programming Environment, Birkhauser-Verlag, 1975

[11] Richard M. Stallman, Emacs: The Extensible, Customizable, Self-Documenting Display Editor, ACM Conference on Text Manipulation, 1981

Acknowledgements

This research has been supported in part by ONR contract N00014-75-C-0522 and in part by ARPA contract N00014-80-C-0505.

I would like to thank Carl Hewitt for encouraging this research William Buxton for a letter with interesting comments on an earlier paper, the Lisp Machine group of the MIT AI Lab for hardware and software support, and Seth Steinberg and Robert Frankston of Software Arts, for discussions about VisiCalc and Tinker.