Massachusetts Institute of Technology
Artificial Intelligence Laboratory

A.I. Memo No. 656 February, 1982

Seeing What Your Programs Are Doing

Henry Lieberman

Abstract

An important skill in programming is being able to *visualize* the operation of procedures, both for constructing programs and debugging them. *Tinker* is a programming environment for Lisp that enables the programmer to "see what the program is doing" while the program is being constructed, by displaying the result of each step in the program on representative examples. To help the reader visualize the operation of Tinker itself, an example is presented of how he or she might use Tinker to construct an alpha-beta tree search program.

Keywords: Visualization, example-based programming, interactive programming environments, computer graphics, debugging, program testing, alpha-beta search, Lisp

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 N00014-80-C-0505.

Seeing What Your Programs Are Doing

Henry Lieberman

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

1. Tinker helps programmers visualize the operation of their programs

Visualization is a powerful tool in programming. Designing a program requires being able to visualize what the program should do. Debugging a program requires localizing bugs to the piece of code responsible, which is often done by visualizing the steps the program goes through and comparing the actual result to the intended behavior. One reason people find programming so difficult is that it taxes their ability to visualize procedures. The enormous amount of detail contained in successive states that programs go through overwhelms most people's ability to keep these details in their heads. Consequently, a programming environment oriented toward helping a user visualize the operation of programs should be very successful in making programming easier.

Tinker is an experimental system which helps a user write Lisp programs, and enables the user to "see what the program is doing" while the program is being constructed. Tinker lets the programmer put together a program step-by-step, and shows the result of each operation as it is performed. Tinker makes programming easier by explicitly displaying information about intermediate states of programs which the programmer would otherwise have to keep in his or her head.

With each piece of code in a program, Tinker associates the value which resulted from that code, to help the programmer in visualizing the effects of that code. When each operation in the program is performed, Tinker displays the output, such as text or graphics, to help the programmer visualize the progress of the program up to that point.

2. Tinker uses specific examples to aid visualization of programs

Programming is the art of teaching procedures to a computer. But conventional programming differs from the way in which people teach each other procedures in at least one important respect: the use of *examples*. People are much more skillful at learning procedures if a teacher presents specific examples than if the teacher presents the abstract algorithm in its most general form. Why is this so?

As each step of the algorithm is presented, the student can follow along, noting the effect of that particular step on the particular situation presented. The teacher points out which features of the situation are important and which are accidental, and the student abstracts the example to learn a procedure for the general case. When a new situation is presented, the student can check each step against his understanding of the example. If no example is present, the student is forced to *imagine* what the effects of each step will be on typical cases. This places a severe burden on the student's short-term memory. Examples help a student learn a procedure by giving the student a tool for visualizing the operation of the procedure. Learning procedures by examples also gives the student the opportunity to start by learning a very simple version of the procedure, then extending the procedure incrementally by considering more complex examples and special cases.

Since the power of examples in learning is so compelling, it seems strange that we should not be able to use examples in teaching a procedure to a computer. Tinker uses examples to make the programming process more natural, closer to the way in which people communicate procedures to each other. With Tinker, a program is written by presenting a specific example, and working out the steps of the procedure on that example. Tinker shows the result of each step as it is given, remembers the sequence of steps, and generalizes a program. More than one example may be shown, and Tinker has the capability to combine several examples to produce a procedure containing a conditional.

A word of caution: the reader should be careful not to confuse Tinker with previous research labelled *programming by example*. This line of research attempted to infer a procedure from the procedure's input-output history, a list of argument-value pairs. The programmer would present example inputs and desired results, without any indication of how the result should be obtained from the input. For instance, the programmer would tell the system that `(REVERSE NIL)` is `NIL` and `(REVERSE '(A B C))` should result in `(C B A)`, and the system should synthesize the usual recursive definition of `REVERSE` in terms of `CONS`. This approach met with some limited success for simple examples, but quickly becomes intractable for larger examples. Imagine

showing a beginner the initial position for chess and checkmate positions, and expecting the beginner to learn chess strategy!

One problem with creating programs from input-output histories is that any given example is generalizable in a potentially infinite number of ways. The system must have some criteria for choosing which generalization to make. Any particular criteria tend to be applicable only in a limited domain, since people might want to take the same example and generalize it in different ways.

Tinker's approach hopes to retain the naturalness of presenting procedures in terms of examples, while using explicit knowledge about the procedure supplied by the programmer to make example-based programming feasible for realistic problems. Often, it is easier for the programmer to begin by working out steps of the procedure, even if he is not sure exactly what steps are necessary, than by specifying the exact form of the answer. The precise appearance of the answer often emerges only after the procedure has been observed in typical situations. Tinker's value lies in showing the programmer the results of all the intermediate steps on examples, making it much easier to detect bugs and understand the program's performance.

3. Tinker lets you write programs and debug them simultaneously

"Seeing what the program is doing" is especially important for debugging. Sometimes, of course, a program is wrong because the programmer has chosen an algorithm that is completely wrong, and the programmer must change some misconceptions and totally rewrite the program. But more often, the programmer's conception of the program is for the most part correct, but some part of the program doesn't implement what the programmer had in mind.

Finding a bug in a program is often a task of *localization* -- trying to find a specific part of the program which is malfunctioning and is responsible for the whole program's misbehavior. Localization of bugs is a matter of *examining successive states* the program goes through, and deciding at each point whether the state of the program conforms to the programmer's expectations. When a state that doesn't meet expectations is encountered, the operation which produced that state can be held responsible for the bug. Most debugging tools (such as tracing and breakpoints) are oriented towards showing the user intermediate states of the program between the start of the program, and its output.

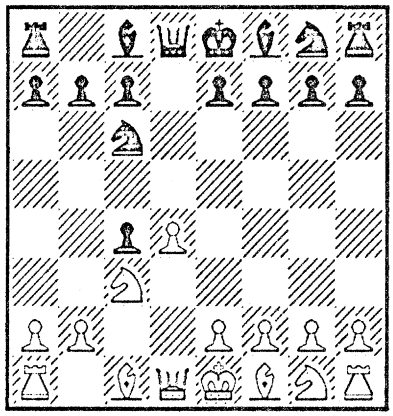
Of course, *preventing* the introduction of bugs into a program is to be preferred to *removing* bugs once they have been introduced into a program. Tinker takes as inspiration the debugging technique of observing intermediate states of a program, and applies this technique to program construction. As a program is constructed with Tinker, the user can confirm that each step satisfies expectations. If an unwanted result is produced, the offending operation can be retracted immediately, before its effects propagate to other parts of the program. This avoids burying the erroneous operation beneath many other, possibly unrelated operations, only to have to fish it out again when some larger program of which it is a part misbehaves.

Conventional programming separates writing a program and debugging a program into two distinct activities. Since a long time passes between the time an operation is written into a program and the time the programmer discovers that the operation is the cause of a bug, it is easy to forget exactly why the operation was put there and the relationship of the operation to the rest of the program. Instead, Tinker interleaves the debugging process with the program writing process, making the introduction of bugs into programs much less likely.

4. An analogy: Tinker provides a chessboard for programming

To illustrate the importance of displaying intermediate states in visualizing procedures, here is an analogy drawn from chess. Below are two representations of a chess game.

White	Black
1 P-Q4	P-Q4
2 P-QB4	PXP
3 N-QB3	N-QB3



When a chess game is represented using a chessboard, it is easy to keep track of what's going on in the game. The chess player looks at the current state of the board, and uses the positions of the pieces to decide what the next move should be. The player can use the current board position to think about the consequences of each of the alternatives for the next move to be made.

When a chess game is represented only as a list of moves, it becomes so difficult to keep track of what's happening in the game that only a few, exceptional *blindfold chess* players are capable of playing in this fashion. The list of moves contains just as much information as the chessboard, yet since the intermediate states are not explicitly represented, the player must try to imagine what the board looks like after a series of moves, a staggering task for any but the most expert.

Conventional programming is a little like playing blindfold chess. When the programmer "makes a move" (writes the next function call or program statement), he must imagine what the result of that move will be on the objects he is manipulating. He must keep the current state in his head, and use the current state to decide what the next operation in the program should be. A common source of bugs is to forget or to misremember some important aspect of the current state of the program, and specify some erroneous operation.

Tinker is like a "programmer's chessboard" in that after each programming operation is specified, the result is shown immediately. Tinker's immediate, graphical feedback makes it much easier to decide what the next operation in the program should be, since it relies to a much lesser extent on the programmer's short term memory. Programining with Tinker should be easier than traditional programming in the same way that playing chess using a chessboard is easier than playing blindfold chess.

5. Examples are especially important for graphics programs

Although Tinker is independent of the subject matter of the program, the advantages of Tinker's programming methodology come through especially clearly in graphics programming. In graphics, the examples are *pictures*. The ability to "see what a program is doing" is essential for graphics programming. It is important to be able to watch pictures appear on the screen as the program is running to assess its performance. The programmer must be able to associate pieces of code with parts of the picture.

While specifications for programs which manipulate text can be given as symbolic descriptions, specifications for graphics programs are pictures. The only way to tell if a graphics program works correctly is to look at the pictures it produces and see if they look right. Thus, formal methods can never completely supplant testing for determining the correctness of graphics programs. Tinker provides an environment for constructing graphics programs where pictures appear on the screen immediately as each graphic operation is introduced into the program. The programmer can immediately see whether the operation specified produced the intended picture.

6. Tinker uses graphics to improve the quality of the programming environment

A goal of Tinker has been to explore how new personal computers with high resolution graphics displays can be used to radically improve the programming process. Most programming environments commonly in use today were originally designed in the days when computers were limited to character-only displays or printing terminals. With high-resolution graphics displays, the screen can be divided into *windows*, rectangular areas of the screen where text and graphics can be displayed independently. Personal computers can have pointing devices like the mouse. Our programming environments need to be restructured to take advantage of these new facilities.

In Tinker, programming happens as much as possible by selecting from a *menu*, where the system display a list of possible choices, and the user picks a choice by pointing, instead of by typing commands. This is better, especially for beginners, since the user doesn't have to remember what choices are available, or remember the syntax of commands, or be proficient at typing.

7. An example problem: Alpha-beta tree search

The best way to visualize the ideas behind Tinker is to watch an example of Tinker in action. Within the limitations of the paper-and-print medium, we will now try to give the reader some feel for what it is like to use Tinker for everyday programming.

The problem we have chosen to present is an *alpha-beta tree search* algorithm [5]. This is a classic problem in Artificial Intelligence, first arising in chess-playing

programs. It has wide application in many problems involving two-person games, planning of actions, and problems requiring search through a space of possible situations. The program must decide what actions to take by searching a tree of possible situations. Each node of the tree represents a situation, each arc an action that can be taken to transform one situation into another. In chess, the situations are board positions, the actions chess moves.

The search proceeds by imagining the effect of each possible move and exploring its consequences. When planning an action, you say "Suppose I make this move...", then turn around and take the point of view of your opponent, imagining "Suppose he then makes this response to my move...", and planning your next response accordingly.

Situations at each node are described by a *static evaluation*, a numerical assessment of the relative advantage for the player at that node. Situations better for you are given higher numbers, those better for your opponent lower numbers. You always choose your best move and your opponent is likely to choose the action best for himself. The value of the top of the tree is determined by the maximum of the values of the nodes immediately below it. The value of the each node at the next level down is determined by maximizing the values of the nodes immediately below it, and so on, alternating minimizing and maximizing steps at each level. This is called the *minimax* search procedure.

Here is a picture of a tree of possible situations, with the leaf nodes of the tree marked with numbers indicating their static evaluations, and nonterminal nodes marked with their minimax values. We show a downward pointing arrow at a node to indicate taking the minimum of the values of branches below that node, and an upward pointing arrow to indicate taking the maximum of values below the node.

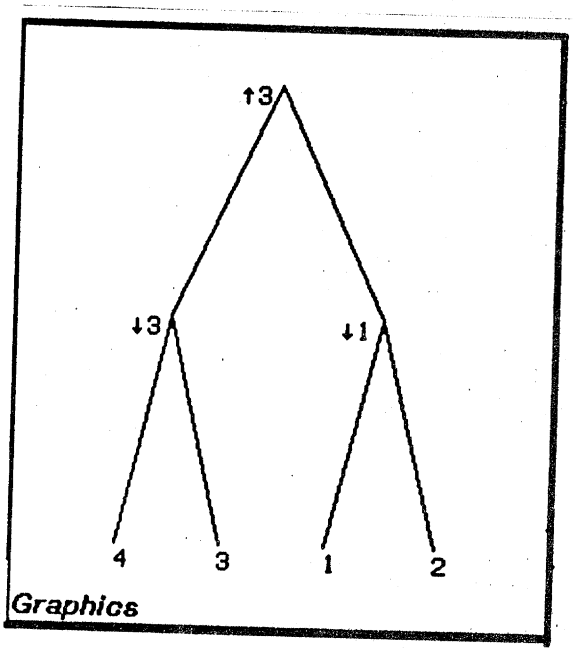


Figure [1]

In certain situations, like the one illustrated above, it's not always necessary to explore the entire tree. The next picture shows the same tree, but captures the process of exploring the tree at a time before every node has been explored.

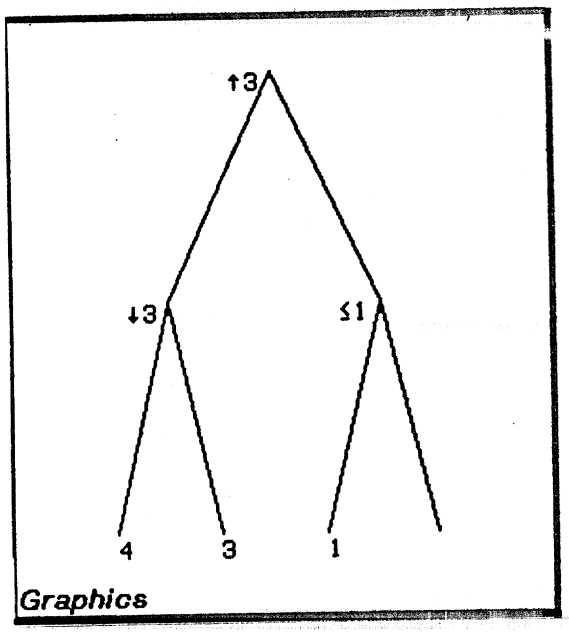


Figure [2]

First we explore the left side, yielding 3, the minimum of 4 and 3. Now, imagine that we've explored the left side of the right branch, yielding 1, but have not yet explored the rightmost branch.

We can immediately conclude the value of the right side of the tree must be "at most 1", since if the number is any higher than 1, 1 would be the minimum of the two. Since the maximum of 3 and "some number which is at most 1" is 3, there's no need to explore the rightmost branch. Thus we can deduce the value of the entire tree without knowledge of every terminal node. This is called the *alpha-beta* heuristic, and it can save a lot of work in tree search problems.

By contrast, on the following tree, the alpha-beta heuristic is not applicable.

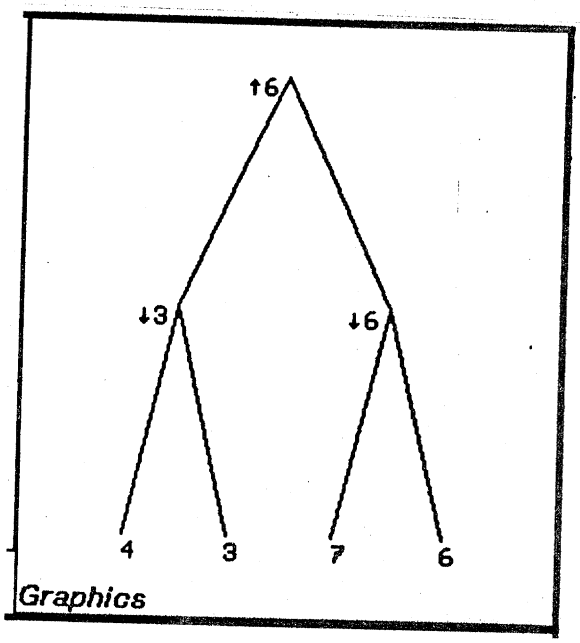


Figure [3]

Since the value 7 of the third branch exceeds the value of the left side of the tree, 3, we are forced to explore the fourth branch. Indeed, the value of the fourth branch, 6 turns out to be the value for the entire tree in this case.

To illustrate the essential ideas clearly, we will restrict ourselves to considering a very simple variety of the alpha-beta search technique. Extensions to more complex versions, such as pruning other branches of the tree, dealing with non-binary trees, etc., can be readily imagined.

We are now going to use Tinker to develop a program to search trees using the alpha-beta heuristic. Just as the two example trees are presented to explain the alpha-beta algorithm to the reader, we will use the same two example trees to show Tinker how to perform the alpha-beta search procedure.

8. A guided tour of the Tinker screen

Before embarking on our project of defining the alpha-beta search procedure, we will take a few moments to explain the mechanics of writing programs with Tinker. This picture shows a typical Tinker display.

Figure [4]

(See next page)

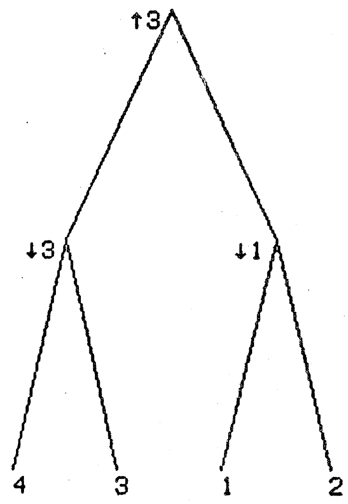
Tinker EDIT menu

- TYPEIN and EVAL
- TYPEIN, but DON'T EVAL
- NEW EXAMPLE for function
- Give something a NAME
- Fill in an ARGUMENT
- EVALUATE something
- Make a CONDITIONAL
- Edit TEXT
- Edit DEFINITION
- Step BACK
- UNFOLD something
- COPY something
- DELETE something
- UNDELETE thing deleted
- UNDO the last command
- LEAVE Tinker
- RETURN a value

```
(DEFUN HISTORY ())

(DEFINE-EXAMPLES (QUOTE HIST !
ORY))
```

ZMACS (LISP Abbrev Electric Shift



Graphics

Defining (HISTORY):

Result: TREE-DISPLAYED, Code: (DISPLAY-TREE-AND-LABEL CUTOFF)

```
; NIL
; NIL
```

Type something to evaluate:

```
(DISPLAY-TREE-AND-LABEL CUTOFF)
```

ZMACS (LISP Abbrev Electric Shift-lock) History Font: A (MEDFNB) *

Type a form. Both the form and its value will appear in the snapshot window

02/17/82 00:21:24 Henry USER: TYI___

Each Tinker operation begins by choosing a menu operation from the *Edit Menu* in the upper left hand corner. In this example, we move the mouse cursor to the operation TYPEIN and EVAL, and press a button on the mouse. The TYPEIN and EVAL operation lets us enter an ordinary piece of Lisp code and have it evaluated.

Tinker then prompts us in the *Typing Window*, at the bottom of the screen, asking Type something to evaluate: and we reply by typing in some Lisp code. Whenever Tinker needs to ask the user a question or print some information, it does so in this window, and the user types all input to Tinker here. The code in this example calls an already-defined function named DISPLAY-TREE-AND-LABEL which draws trees on the screen, telling it to draw a tree stored in the variable named CUTOFF.

The title line of the *Snapshot Window* in the middle of the screen reads: Defining (HISTORY). The code which appears in the snapshot window is always considered to be code which defines the body of some Lisp function. In this case, there's a top level function named HISTORY.

As a result of the TYPEIN and EVAL operation, the text Result: TREE-DISPLAYED, Code: (DISPLAY-TREE-AND-LABEL CUTOFF) appears in the snapshot window. This displays the code entered, along with the value, TREE-DISPLAYED, produced by that piece of code. Whenever some code is evaluated to produce a value, Tinker always remembers and displays the code that was responsible for producing that value. When defining a new function, the Result: ... part of a line in the snapshot window represents the result of performing the function's steps on some particular example, while the Code: ... part represents the general case for the function. In this way, Tinker can display to the user both particular examples and the code for the general case simultaneously. The commands in Tinker's command menu are mostly *editing commands* which edit the objects that appear in the snapshot window.

As a result of executing the code (DISPLAY-TREE-AND-LABEL CUTOFF), in the *Graphics Window* at the top right hand corner of the screen, we see a picture of the tree. The graphics window is used to display drawings which illustrate the behavior of the program.

The *Function Definition Window* at the top center of the screen shows the textual definition of Lisp functions generated by Tinker. Although Tinker has its own representation for programs, it produces ordinary Lisp code, which can be compiled for efficiency.

9. Expressions can be constructed incrementally after viewing their parts

Once Tinker evaluates some code, displaying the code and its result in the snapshot window, the programmer may use both the code and the result as part of some larger expression. The programmer can enter another function call, and specify that something displayed in the snapshot window is to be used as an argument to that function. When the function call is finally evaluated, the specific value of the argument is used to compute the value of the function, and the code which produced the argument becomes part of the expression for the function call. In this way, the programmer can examine the values of small pieces of code to make sure they are correct, before making them part of some larger expression.

Here's a simple example of this. We're going to display another tree on the screen, but this time we'd like to look at the printed representation of the tree *before* constructing the expression to display it.

We use the TYPEIN and EVAL operation, and type in the variable named EXPLORE-FULLY which holds the tree. The snapshot window looks like this:

```
Defining (HISTORY):
Result: #,(A TREE ((4 3) (7 6))), Code: EXPLORE-FULLY
```

This shows us the printed representation of the value of the variable EXPLORE-FULLY In this example, trees are defined to print out the numbers which label the leaf nodes of the tree. The tree EXPLORE-FULLY has a right branch whose leaves are 4 and 3, a left branch with leaves 7 and 6.

Next, we choose the operation TYPEIN, but DON'T EVAL, which puts up a piece of unevaluated code in the snapshot window.

```
Defining (HISTORY):
Result: #,(A TREE ((4 3) (7 6))), Code: EXPLORE-FULLY
Code: (DISPLAY-TREE-AND-LABEL)
```

The line in the snapshot window for the call to the function DISPLAY-TREE-AND-LABEL only has a Code: part, since we haven't evaluated it yet. Now, we choose the

operation Fill in an ARGUMENT. Since DISPLAY-TREE-AND-LABEL is the only function on the screen that needs an argument, and the tree is the only thing that could possibly be the argument, Tinker immediately constructs the function call. Tinker has a policy of automatically selecting the "obvious" choice, when only one object on the screen is plausible to choose as an argument to the current menu operation.

Defining (HISTORY):
Code: (DISPLAY-TREE-AND-LABEL (A TREE ((4 3) (7 6))))

Evaluating this piece of code with the operation EVALUATE something displays the tree on the screen in the graphics window, and changes the snapshot window to:

Defining (HISTORY):
Result: TREE-DISPLAYED, Code: (DISPLAY-TREE-AND-LABEL EXPLORE-FULLY)

Notice that the variable EXPLORE-FULLY which produced the tree becomes part of the code for the function call, rather than just the tree itself (as a constant). Whenever a value is used in further computation, Tinker carries along the code which produced that value. This shows how Tinker can build up complicated expressions one step at a time, while displaying to the programmer the result of each step.

10. We begin with a top-down implementation plan for alpha-beta search

When designing an algorithm, a programmer usually starts with very vague ideas about the problem, and gradually works them out to be more and more specific. In the early stages of working on a problem, it is common to have in mind some examples of how the finished program should behave, without having very definite ideas of what the code should look like. It is also typical to have a rough *implementation plan*, which maps out a strategy for implementing the task, again without committing the programmer to specific details of the code. An implementation plan might involve proposing a few major subroutines and data structures and the communication between them. Decisions made in the implementation plan are often revised in the process of working on the implementation.

In conventional programming, debugging and testing on the machine cannot proceed until a proposed solution becomes specific enough to actually start writing complete pieces of code. Tinker aims to involve the machine at an earlier stage. The programmer should be able to begin working with Tinker as soon as he or she knows some good examples for the problem, and has in mind an implementation plan which is capable of performing the procedure on the examples.

We begin working on the alpha-beta problem with a rough implementation plan. Our plan should include provision for viewing graphically the progress of the alpha-beta search as it explores the tree. Since the program is to be written by presenting specific example trees, we will be able to see dynamically what the program is doing by watching the search procedure move across the nodes of the tree.

We can do this by first displaying the whole tree by drawing only its arcs on the screen. As the search examines each node and decides on a value for that node, we will have our program label the node with its value. This will enable us to see what nodes are being looked at by the program, and in what order the nodes are examined.

Like in most programming situations, we start with a set of already-defined procedures and data structures, and these facilities are available for constructing new programs. We will assume that certain support routines and data have been defined before the start of our session, and we will not present the details of these, to avoid distracting us from the alpha-beta algorithm itself.

First, we will assume that the data structure used to represent trees has already been defined. A tree is either a LEAF node, or it has LEFT and RIGHT branches, each of which is a tree. The functions LEFT-SIDE and RIGHT-SIDE extract the two branches from the tree, and the predicate LEAF? asks whether a tree is a leaf node. Trees may have LABELs at their nodes. We will assume that a set of example trees has been prepared for this session, including the trees CUTOFF and EXPLORE-FULLY.

We will also assume primitive graphics procedures for displaying trees on the screen. The procedure DISPLAY-TREE draws the arcs of the tree on the screen, and DISPLAY-DOTTED-TREE draws them with dotted lines. LEFT-SIDE and RIGHT-SIDE of a tree display the arcs as they traverse them. LABEL-NODE displays the label at a particular node, and DISPLAY-TREE-AND-LABEL displays a tree and labels all its nodes. We could define the tree data structure and display functions using Tinker if we wished.

We will adopt a top-down strategy for implementing the alpha-beta search. We will

start with a top-level function which we will call ALPHA-BETA, which will initialize the display. This will then call a "workhorse" function AB which will compute the alpha-beta value of each node, recursively walking down the tree until leaf nodes are encountered. We will separate the work of AB into two subroutines, AB-LEFT and AB-RIGHT which compute the alpha-beta of the left and right branches of a tree, respectively. The crucial subroutine AB-PRUNE will make the decision whether or not the alpha-beta heuristic is applicable, allowing us to "prune" some branches of the tree.

The process of defining the alpha-beta search with Tinker will require three main examples. We will start by presenting the tree CUTOFF which illustrates the application of the alpha-beta heuristic. This tree will serve as the first example for the alpha-beta function. The search will be defined recursively in terms of a walk down the tree data structure until a leaf node is reached. Computing the alpha-beta value of a leaf node will be the second example, showing how the recursive procedure *bottoms out.* Next, the tree named EXPLORE-FULLY will provide a contrasting example, demonstrating that the alpha-beta heuristic is not applicable in all cases.

11. The first example shows how to apply the alpha-beta heuristic

We are now ready to begin writing the code for the alpha-beta search. The way we start defining a new function in Tinker is to present an *example* function call, showing a typical case in which we will use the function. We work out the steps corresponding to the procedure on the test case.

We construct a call to the ALPHA-BETA function, just as if we had already defined the function. As an example tree, we supply a tree named CUTOFF, the tree we originally used above to illustrate the alpha-beta heuristic. We use the TYPEIN, but DON'T EVAL operation.

Defining (HISTORY):
Code: (ALPHA-BETA CUTOFF)

Now, instead of evaluating that form, we instead tell Tinker that this is a NEW EXAMPLE for function, for the function ALPHA-BETA. Tinker responds by changing

the snapshot window to tell us we're defining an example for ALPHA-BETA, and creates a variable to name the argument to ALPHA-BETA. We name the argument TREE using the Give something a NAME operation.

```
Defining (ALPHA-BETA (A TREE ((4 3) (1 2)))):
Result: #,(A TREE ((4 3) (1 2))), Code: TREE
```

The first action taken by the program should be to initialize the display, drawing the arcs of the tree, but without labelling any of its nodes. We use the function DISPLAY-DOTTED-TREE to display the shape of the tree on the screen, using dotted lines which will be filled in incrementally as the procedure traverses the tree.

```
Defining (ALPHA-BETA (A TREE ((4 3) (1 2)))):
Result: #,(A TREE ((4 3) (1 2))), Code: TREE
Result: TREE-DISPLAYED, Code: (DISPLAY-DOTTED-TREE TREE)
```

In the graphics window, a picture of the example tree appears.

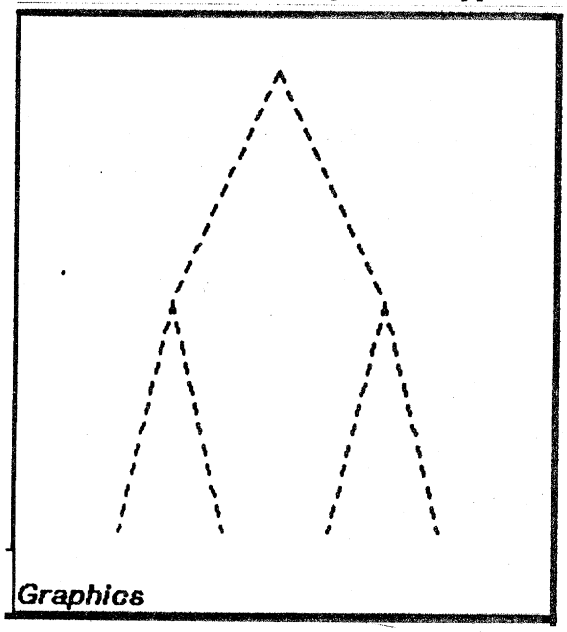


Figure [5]

Now, we pass along the tree to the workhorse function AB.

```
Defining (ALPHA-BETA (A TREE ((4 3) (1 2)))):
Result: #,(A TREE ((4 3) (1 2))), Code: TREE
Result: TREE-DISPLAYED, Code: (DISPLAY-DOTTED-TREE TREE)
Code: (AB (A TREE ((4 3) (1 2))))
```

We choose the command NEW EXAMPLE for function, which recurses, pushing from defining the function ALPHA-BETA to defining the function AB. After we conclude the definition of AB, Tinker will return us to defining ALPHA-BETA.

```
Defining (AB (A TREE ((4 3) (1 2)))):
Result: #,(A TREE ((4 3) (1 2))), Code: TREE
```

Tinker encourages a kind of *top-down debugging*. In traditional, *bottom-up debugging*, subroutines must be defined before their callers can be tested. Tinker allows programming a top-level routine first, then when the need for a subroutine is felt, introducing an example for the subroutine.

Since we intend AB to recurse down the branches of the tree, the first action should be to extract the left branch from the tree.

```
Defining (AB (A TREE ((4 3) (1 2)))):
Result: #,(A TREE ((4 3) (1 2))), Code: TREE
Result: #,(A TREE (4 3)), Code: (LEFT-SIDE TREE)
```

We introduce a new AB-LEFT function, and provide it with the left branch of the tree as an example. We name this branch LEFT-TREE.

```
Defining (AB-LEFT (A TREE (4 3))):
Result: #,(A TREE (4 3)), Code: LEFT-TREE
```

The plan for the AB-LEFT function is to call AB recursively on each of its branches in turn, then compute the minimum value of the branches, and use that value to label the LEFT-TREE. This performs a "min" step of the "minimax" search.

First, we extract the LEFT-SIDE of the tree, since we have to recurse down two levels of the tree at a time.

```
Defining (AB-LEFT (A TREE (4 3))):
Result: #,(A TREE (4 3)), Code: LEFT-TREE
Result: #,(A LEAF (VALUE 4)), Code: (LEFT-SIDE LEFT-TREE)
```

This yields a leaf node in our example. We recursively call AB on the left branch.

12. The alpha-beta function bottoms out when it encounters a leaf node

```
Defining (AB-LEFT (A TREE (4 3))):
Result: #,(A TREE (4 3)), Code: LEFT-TREE
Code: (AB (A LEAF (VALUE 4)))
```

Taking the alpha-beta value of a leaf node is a fundamentally *different* example from computing the alpha-beta of a tree, since we want the ALPHA-BETA function to be recursive in the case of a tree, but to stop when it encounters a leaf node. So, instead of evaluating the call to AB, we tell Tinker this is a NEW EXAMPLE for the function AB.

What action should AB take when it reaches a terminal node of the tree? The AB function should just return the value associated with that node as the alpha-beta value of the node. In addition, it should display the node on the screen, using the predefined function named LABEL-NODE.

```
Defining (AB (A LEAF (VALUE 4))):
Result: #,(A LEAF (VALUE 4)), Code: TREE
Result: 4, Code: (LABEL-NODE TREE)
```

In the graphics window, the value 4 appears at the node. This demonstrates to Tinker that whenever the search procedure reaches a leaf node, it should label that

node with its value, so that we can see what the search is doing. As the search progresses down the branches of the tree, it will replace the dotted lines for arcs of the tree with solid lines.

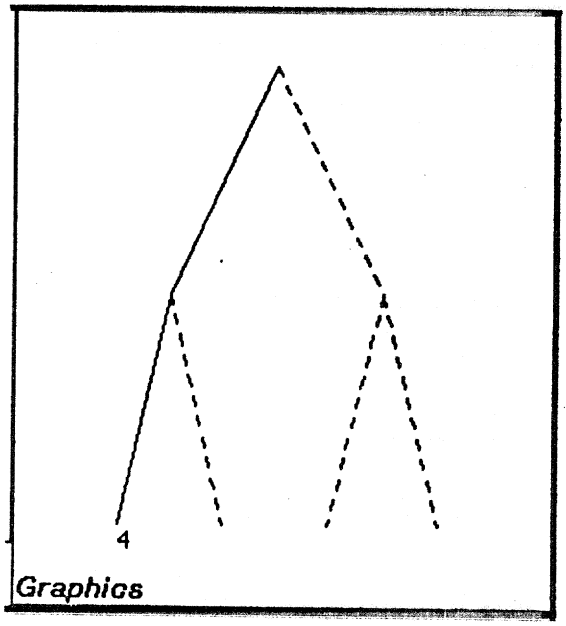


Figure [6]

This is all we want to do to complete the leaf node example for AB, so we choose RETURN a value, returning the value 4. Tinker writes the Lisp code for AB and displays it in the *Function Definition* window.

```
(DEFUN AB (TREE)
       (LABEL-NODE TREE))
```

That definition may look silly, but it is correct for the examples we've shown it so far. Tinker develops functions by a series of *partial definitions.* As each example for a particular function is completed, Tinker produces a definition which is sufficient to make the procedure work as specified on the examples presented so far. When additional examples for an already-existing function are presented, Tinker can integrate the procedure for the old examples with the procedure for the new one. When we complete the example for AB of a full-blown tree, the code for AB will

become more sophisticated. Tinker has the ability to improve the definitions of functions by adding more examples incrementally.

13. The search completes the left branch and proceeds to the right side

Tinker now knows how to perform the AB of a leaf node, so we can apply the definition to the other leaf node on the left branch of the tree. This displays the value 3 on that leaf node.

```
Defining (AB-LEFT (A TREE (4 3))):
Result: #,(A TREE (4 3)), Code: LEFT-TREE
Result: 4, Code: (AB (LEFT-SIDE LEFT-TREE))
Result: 3, Code: (AB (RIGHT-SIDE LEFT-TREE))
```

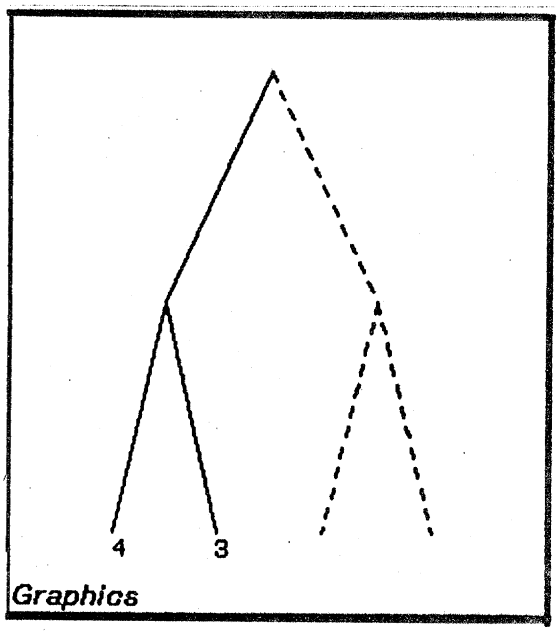


Figure [7]

The next step is to complete AB-LEFT by using the alpha-beta values of the leaves to compute an alpha-beta value for the left side of the tree. The left branch of the tree should be labelled 3 since it should carry the minimum of the two values 4 and 3 on its branches.

```
Defining (AB-LEFT (A TREE (4 3))):
Result: #,(A TREE (4 3)), Code: LEFT-TREE
Result: 3, Code: (MIN (AB (LEFT-SIDE LEFT-TREE)) (AB (RIGHT-SIDE LEFT-TREE)))
```

```
Defining (AB-LEFT (A TREE (4 3))):
Result: #,(A TREE (4 3)), Code: LEFT-TREE
Result: 3, Code: (LABEL-NODE LEFT-TREE "↓" (MIN (AB **) (AB **)))
```

(The double stars "**" indicate places where Tinker elided some details of the code, since the entire code was too large to fit on one line of the screen all at once.)

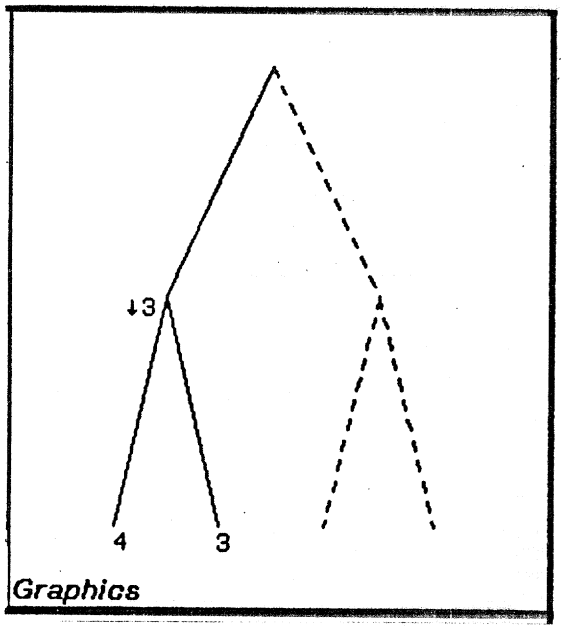


Figure [8]

Seeing that the left side of the tree has been fully labelled, we can be assured that the definition for AB-LEFT has been completed. Tinker's ability to provide visual feedback incrementally during the construction of a program is helpful in "keeping our place" in the developing program. After choosing RETURN a value, Tinker displays the code for AB-LEFT in the function definition window.

```
(DEFUN AB-LEFT (LEFT-TREE)
       (LABEL-NODE LEFT-TREE
                   "↓"
                   (MIN (AB (LEFT-SIDE LEFT-TREE))
                        (AB (RIGHT-SIDE LEFT-TREE)))))
```

After having explored the left half of the tree, the next task is to define the function AB-RIGHT to explore the right half. If we had been doing a standard minimax search, the same subroutine would suffice for both sides of the tree. The search we are going to define is asymmetrical, using the knowledge gleaned during searching the left side of the tree to potentially save work exploring the right side.

The AB-RIGHT function needs to know the value of the left side of the tree, which we'll name LEFT-EXPLORED, as well as the right side of the tree, named RIGHT-TREE. We present a NEW EXAMPLE for AB-RIGHT.

```
Defining (AB-RIGHT 3 (A TREE (1 2))):
Result: 3, Code: LEFT-EXPLORED
Result: #,(A TREE (1 2)), Code: RIGHT-TREE
```

The third branch, the left side of RIGHT-TREE is explored unconditionally whenever we explore a RIGHT-TREE. This again makes use of the definition of AB on a leaf node that we completed earlier.

```
Defining (AB-RIGHT 7 (A TREE (1 2))):
Result: 3, Code: LEFT-EXPLORED
Result: #,(A TREE (1 2)), Code: RIGHT-TREE
Result: 1, Code: (AB (LEFT-SIDE RIGHT-TREE))
```

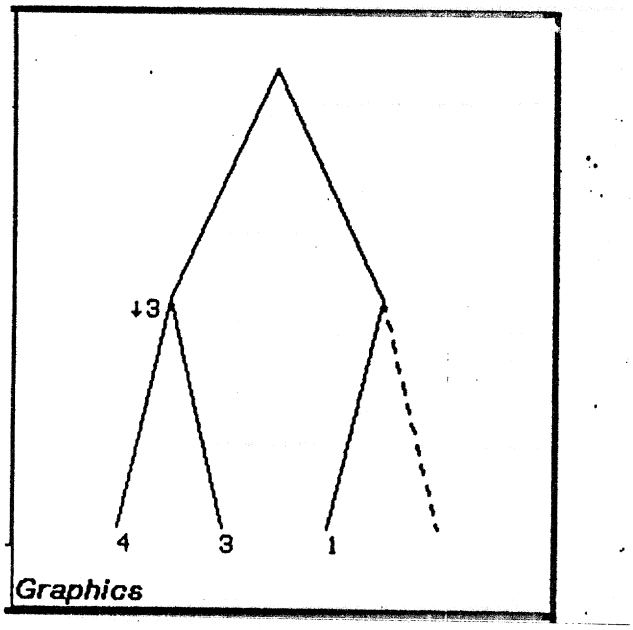


Figure [9]

And now the third of four leaf nodes is marked with its value on the screen. We will introduce the subroutine AB-PRUNE which "prunes" branches of the tree which can be ignored during the search procedure. AB-PRUNE needs the value of the third branch, which we name RIGHT-EXPLORED.

```
Defining (AB-PRUNE 3 1 (A TREE (1 2))):
Result: 3, Code: LEFT-EXPLORED
Result: 1, Code: RIGHT-EXPLORED
Result: #,(A TREE (1 2)), Code: RIGHT-TREE
```

Now, in this case, without exploring the remaining unexplored branch, we can immediately decide that RIGHT-TREE ought to be "at most 1", so we'll put a label on the tree to indicate this.

```
Defining (AB-PRUNE 3 1 (A TREE (1 2))):
Result: 3, Code: LEFT-EXPLORED
Result: 1, Code: RIGHT-EXPLORED
Result: #,(A TREE (1 2)), Code: RIGHT-TREE
Result: 1, Code: (LABEL-NODE RIGHT-TREE "≤" RIGHT-EXPLORED)
```

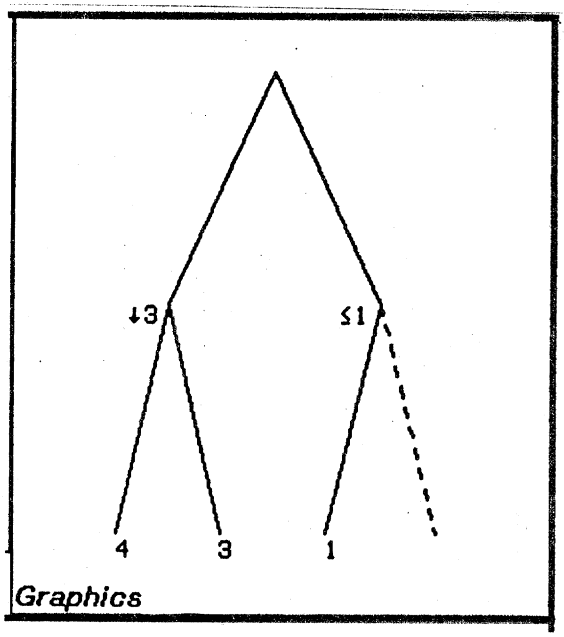


Figure [10]

At the top level of the tree, the maximum of 3 and "at most 1" is 3 regardless of the exact value of the unexplored branch, so we can return 3 as the answer. Completing this yields definitions for AB-PRUNE and AB-RIGHT.

```
Defining (AB-PRUNE 3 1 (A TREE (1 2))):
Result: 3, Code: LEFT-EXPLORED
Result: 1, Code: RIGHT-EXPLORED
Result: #,(A TREE (1 2)), Code: RIGHT-TREE
Result: 1, Code: (LABEL-NODE RIGHT-TREE "≤" RIGHT-EXPLORED)
Result: 3, Code: LEFT-EXPLORED
```

```
(DEFUN AB-PRUNE (LEFT-EXPLORED RIGHT-EXPLORED RIGHT-TREE)
       (LABEL-NODE RIGHT-TREE "≤" RIGHT-EXPLORED)
       LEFT-EXPLORED)
```

```
Defining (AB-RIGHT 3 (A TREE (1 2))):
Result: 3, Code: LEFT-EXPLORED
Result: #,(A TREE (1 2)), Code: RIGHT-TREE
Result: 3, Code: (AB-PRUNE LEFT-EXPLORED (AB **) ...)
```

```
(DEFUN AB-RIGHT (LEFT-EXPLORED RIGHT-TREE)
       (AB-PRUNE LEFT-EXPLORED
                 (AB (LEFT-SIDE RIGHT-TREE))
                 RIGHT-TREE))
```

Returning to the definition of AB on the whole tree, we use the value returned by AB-RIGHT to label the top node.

```
Defining (AB (A TREE ((4 3) (1 2)))):
Result: #,(A TREE ((4 3) (1 2))), Code: TREE
Result: 3, Code: (LABEL-NODE TREE "↑" (AB-RIGHT (AB-LEFT **) **))
```

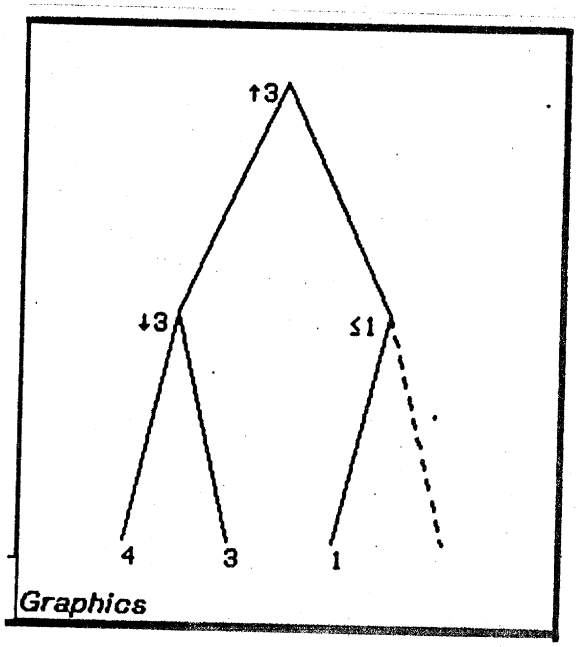


Figure [11]

We have completed the second example for the function AB, showing Tinker how to take AB of a tree, in addition to AB of a leaf node. When Tinker sees two different examples for the same function, it compares the code for the two examples. If the examples differ, Tinker asks us to define a predicate which distinguishes between the two cases. Tinker displays two snapshot windows, one showing the situation when we were defining AB on a leaf, one showing the situation defining AB on a tree. We write code that will appear *simultaneously* in both windows. The object is to define code that will yield *true* in the top window, *false* in the bottom window. This assures that our predicate correctly distinguishes between the two cases. This method of defining conditionals is especially useful in avoiding infinite loop bugs, caused by a predicate continually going down the same branch all the time.

In this case, to distinguish between a leaf node and a full tree, we write a predicate which asks the node whether or not it is a leaf.

Figure [12]

(See next page)

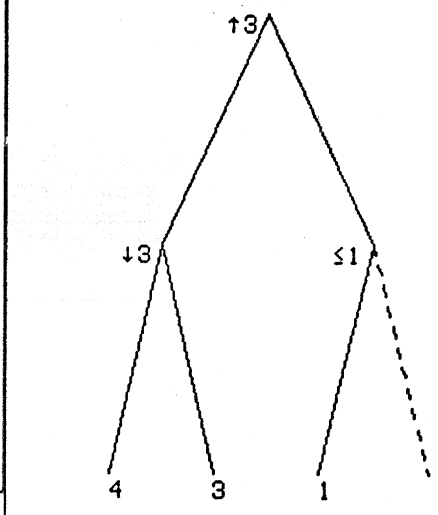
Tinker EDIT menu

TYPEIN and EVAL
TYPEIN, but DON'T EVAL
NEW EXAMPLE for function
Give something a NAME
Fill in an ARGUMENT
EVALUATE something
Make a CONDITIONAL
Edit TEXT
Edit DEFINITION
Step BACK
UNFOLD something
COPY something
DELETE something
UNDELETE thing deleted
UNDO the last command
LEAVE Tinker
RETURN a value

```
                                (AB (RI !
GHT-SIDE LEFT-TREE)))))

(DEFUN AB-PRUNE (LEFT-EXPLOR !
ED RIGHT-EXPLORED RIGHT-TREE !
)
    (LABEL-NODE RIGHT-TREE * !
≤* RIGHT-EXPLORED)
    LEFT-EXPLORED)

(DEFUN AB-RIGHT (LEFT-EXPLOR !
ED RIGHT-TREE)
    (AB-PRUNE LEFT-EXPLORED  !
(AB (LEFT-SIDE RIGHT-TREE))  !
RIGHT-TREE))
```



Graphics

Predicate TRUE for: Result: 4, Code: (LABEL-NODE TREE)

```
Result: #,(A LEAF (VALUE 4)), Code: TREE
Result: T, Code: (LEAF? TREE)
```

Predicate FALSE for: Result: 3, Code: (LABEL-NODE TREE "" ...)

```
 sult: #,(A TREE ((4 3) (1 2))), Code: TREE
..esult: NIL, Code: (LEAF? TREE)
```

```
 (LABEL-NODE TREE "↑" (AB-RIGHT (AB-LEFT (LEFT-SIDE TREE)) (RIGHT-SIDE TREE!
)))?


Type something to evaluate:

(LEAF? TREE)
```

```
ICS (LISP Abbrev Electric Shift-lock) History *
```

Type ? for help. Both the form and its value will appear in the *snapshot* buffer.

02/17/82 18:26:23 Henry USER: TYI___

(In Lisp, NIL represents *false*, and anything other than NIL represents *true*, so the tree in the top snapshot window answered *yes* to the question, the bottom window answered no.)

Tinker now generates a definition of the AB function containing a *conditional.*

```
(DEFUN AB (TREE)
       (IF (LEAF? TREE)
           (LABEL-NODE TREE)
           (LABEL-NODE TREE
                       "↑"
                       (AB-RIGHT (AB-LEFT (LEFT-SIDE TREE))
                                 (RIGHT-SIDE TREE)))))
```

We could also present further examples for AB, and Tinker would create additional conditional clauses separating one case from another. For example, we should probably add to AB another case in which the argument is not any kind of a tree at all, so we can demonstrate a *negative* example as well as a positive one. The action in this case should consist of printing out some sort of error message. This is the way *type checking* can be introduced in Tinker.

This completes also the top-level ALPHA-BETA function.

```
(DEFUN ALPHA-BETA (TREE)
       (DISPLAY-DOTTED-TREE TREE)
       (AB TREE))
```

---

14. Another example shows the alpha-beta heuristic doesn't always work

The program can now perform alpha-beta searches of trees -- but only for examples

where we can apply the alpha-beta heuristic. At this point, Tinker has *overgeneralized* the procedure to conclude that the alpha-beta heuristic works for all trees. This is not always the case for our desired search procedure.

To correct this, we can show Tinker another example, this one representing the class of trees for which it is necessary to explore the whole tree to compute an alpha-beta value. The tree EXPLORE-FULLY has that property.

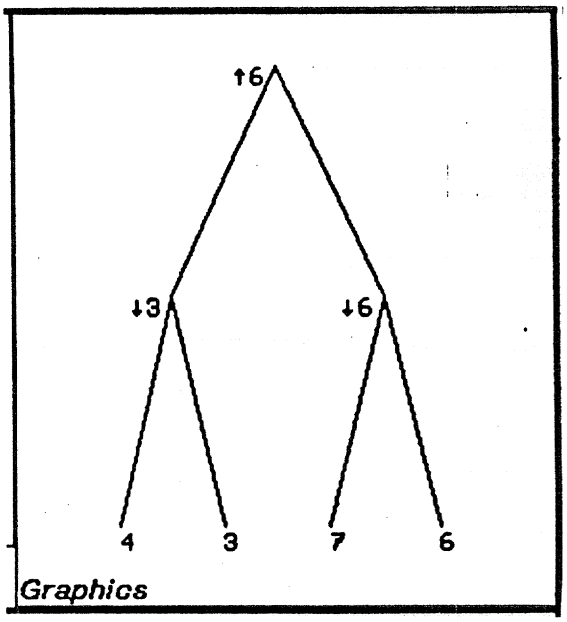


Figure [13]

The only subroutine involved in this change is AB-PRUNE, since AB-PRUNE alone is responsible for exploring the rightmost branch of the tree. As you will recall, AB-PRUNE takes three arguments, the alpha-beta value for the left side of the tree, the value of the third branch and the as yet unexplored rightmost branch of the tree. In the case of the tree EXPLORE-FULLY, LEFT-EXPLORED is 3, RIGHT-EXPLORED is 7, and the RIGHT-TREE has leaves 7 and 6.

```
Defining (AB-PRUNE 3 7 (A TREE (7 6))):
Result: 3, Code: LEFT-EXPLORED
Result: 7, Code: RIGHT-EXPLORED
Result: #,(A TREE (7 6)), Code: RIGHT-TREE
```

We must explore the rightmost branch of the tree, and label the right tree with the minimum of the two leaves on the right side of the tree.

```
Defining (AB-PRUNE 3 7 (A TREE (7 6))):
Result: 3, Code: LEFT-EXPLORED
Result: 7, Code: RIGHT-EXPLORED
Result: #,(A TREE (7 6)), Code: RIGHT-TREE
Result: 6, Code: (AB (RIGHT-SIDE RIGHT-TREE))
```

```
Defining (AB-PRUNE 3 7 (A TREE (7 6))):
Result: 3, Code: LEFT-EXPLORED
Result: 7, Code: RIGHT-EXPLORED
Result: #,(A TREE (7 6)), Code: RIGHT-TREE
Result: 6, Code: (MIN RIGHT-EXPLORED (AB (RIGHT-SIDE RIGHT-TREE)))
```

```
Defining (AB-PRUNE 3 7 (A TREE (7 6))):
Result: 3, Code: LEFT-EXPLORED
Result: 7, Code: RIGHT-EXPLORED
Result: #,(A TREE (7 6)), Code: RIGHT-TREE
Result: 6, Code: (LABEL-NODE RIGHT-TREE "↓" (MIN RIGHT-EXPLORED (AB **)))
```

The value for the top of the tree is the maximum of the values for the two branches. Since the two branches of the trees have values 3 and 6, the maximum is 6.

```
Defining (AB-PRUNE 3 7 (A TREE (7 6))):
Result: 3, Code: LEFT-EXPLORED
Result: 7, Code: RIGHT-EXPLORED
Result: #,(A TREE (7 6)), Code: RIGHT-TREE
Result: 6, Code: (MAX LEFT-EXPLORED (LABEL-NODE RIGHT-TREE ...))
```

This comprises a second example for the function AB-PRUNE. Tinker again creates two

snapshot windows, asking us to distinguish between the two cases, one in which the alpha-beta heuristic is used, one where the tree is explored in its entirety.

The predicate which distinguishes between the two cases tests whether or not the alpha-beta value for the left side of the tree, LEFT-EXPLORED, exceeds the third of the four branches, RIGHT-EXPLORED. In both cases, the left branch evaluated to 3, but in the first case, RIGHT-EXPLORED was 1, which is smaller than 3, but in the second case it was 7, which is greater.

```
Predicate TRUE for: Result: 3, Code: (PROGN (LABEL-NODE **) LEFT-EXPLORED)
Result: 3, Code: LEFT-EXPLORED
Result: 1, Code: RIGHT-EXPLORED
Result: #,(A TREE (1 2)), Code: RIGHT-TREE
Result: T, Code: (> LEFT-EXPLORED RIGHT-EXPLORED)
```

```
Predicate FALSE for: Result: 6, Code: (MAX LEFT-EXPLORED **)
Result: 3, Code: LEFT-EXPLORED
Result: 7, Code: RIGHT-EXPLORED
Result: #,(A TREE (7 6)), Code: RIGHT-TREE
Result: NIL, Code: (> LEFT-EXPLORED RIGHT-EXPLORED)
```

This yields the following code for AB-PRUNE:

```
(DEFUN AB-PRUNE (LEFT-EXPLORED RIGHT-EXPLORED RIGHT-TREE)
    (IF (> LEFT-EXPLORED RIGHT-EXPLORED)
        (THEN (LABEL-NODE RIGHT-TREE "≤" RIGHT-EXPLORED) LEFT-EXPLORED)
        (MAX LEFT-EXPLORED
             (LABEL-NODE RIGHT-TREE
                         "↓"
                         (MIN RIGHT-EXPLORED (AB (RIGHT-SIDE RIGHT-TREE)))))))
```

15. Let's try alpha-beta search on a large tree

Our alpha-beta search procedure is now complete. To illustrate its behavior, we can try it out on a large and complex example which will exercise all of the cases the program knows about. We will try it out on the following tree, called BIG-TREE. Here are successive stages of the alpha-beta program at work.

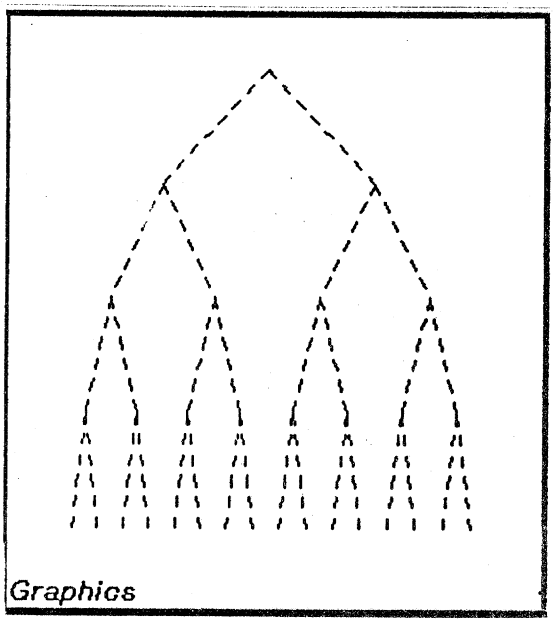
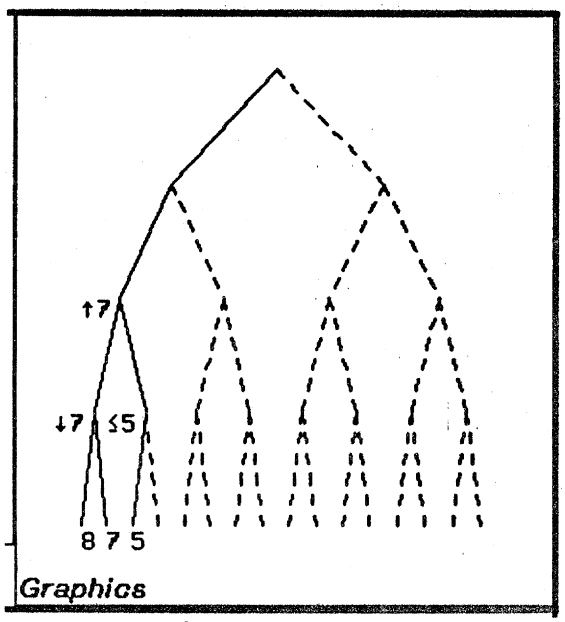


Figure [14]



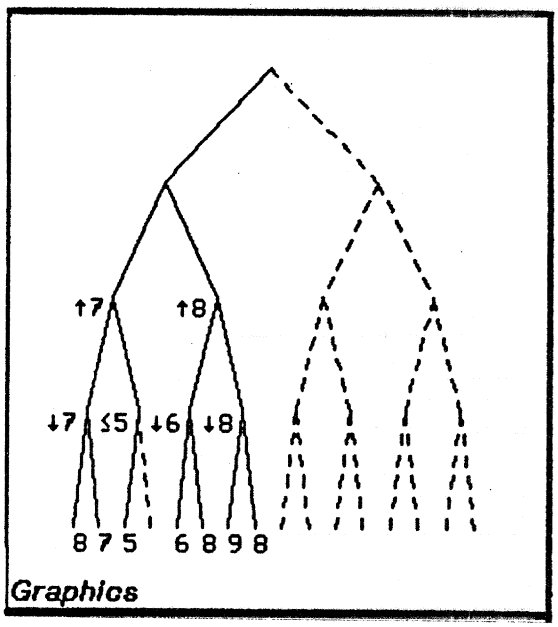
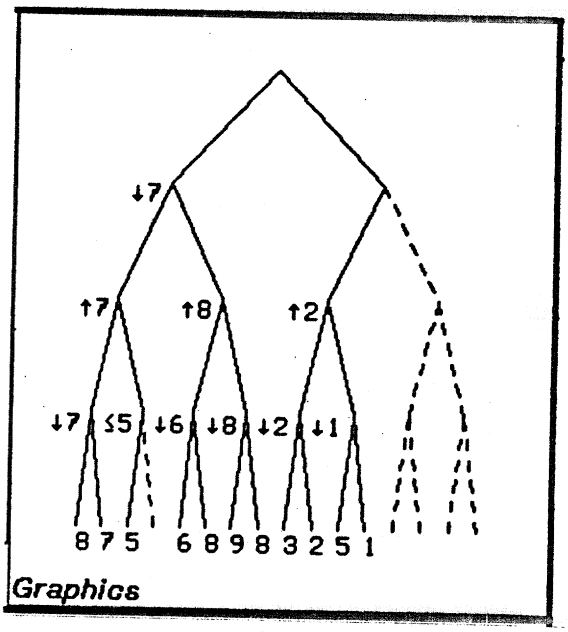


Figure [15]

Figure [16]



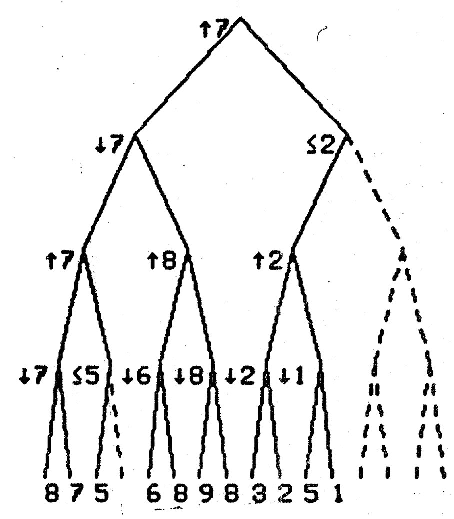
Tinker EDIT menu

TYPEIN and EVAL
TYPEIN, but DON'T EVAL
NEW EXAMPLE for function
Give something a NAME
Fill in an ARGUMENT
EVALUATE something
Make a CONDITIONAL
Edit TEXT
Edit DEFINITION
Step BACK
UNFOLD something
COPY something
DELETE something
UNDELETE thing deleted
UNDO the last command
LEAVE Tinker
RETURN a value

```
(DEFUN HISTORY ())


(DEFUN ALPHA-BETA (TREE)
       (DISPLAY-DOTTED-TREE !
 TREE)
       (AB TREE))

(DEFUN AB-PRUNE (LEFT-EXPLOR !
ED RIGHT-EXPLORED RIGHT-TREE !
)
       (IF (> LEFT-EXPLORED
              RIGHT-EXPLORED !
)
           (THEN
             (LABEL-NODE RIGH !
T-TREE RIGHT-EXPLORED)
             LEFT-EXPLORED)
           (MAX LEFT-EXPLORE !
ZMACS (LISP Abbrev Electric Shift
```



Graphics

Defining (HISTORY):

```
Result: 3, Code: (ALPHA-BETA CUTOFF)
Result: 6, Code: (ALPHA-BETA EXPLORE-FULLY)
Result: 7, Code: (ALPHA-BETA BIG-TREE)
```

Type something to evaluate:

```
(ALPHA-BETA BIG-TREE)
```

ZMACS (LISP Abbrev Electric Shift-lock) History Font: A (MEDFNB) *

Type ... Both the ... and ... value ... appear ... the snapshot ... dow

02/17/82 23:06:42 Henry ER TY

In this example, you can see two distinct alpha-beta cutoffs. The first two nodes looked at were 8 and 7, so their common ancestor is labelled with the minimum, 7. Since 5 for the next leaf node is less than 7, the program did not need to explore the next node.

At the very top of the tree, 7 is computed for the value of the left side of the tree. The left half of the right side yields 2 which is less than 7. This time the program could cut off an entire section of the tree, rather than just the single-node cutoffs we saw previously. This saved almost a quarter of the work involved in examining the entire tree!

We hope this example has successfully illustrated how Tinker uses an example-based programming methodology, incremental program construction, and immediate graphical feedback to make programming easier and more reliable.

Acknowledgements

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

We would like to thank Lisp Machine, Inc. and Symbolics, Inc. for allowing us to use their machines to present a demonstration of Tinker as described in this paper at the Seventh International Joint Conference on Artificial Intelligence.

We would like to thank Giuseppe Attardi, William Buxton, Andy diSessa, Laura Gould, Carl Hewitt, Scott Kim, and Deborah Tatar for helpful comments and suggestions on earlier drafts.

Bibliography

The bibliography in [6] also contains references relevant to the topics discussed in this paper.

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

[2] F. M. Brown, Dynamic Program Building, Software Practice and Experience, August 1981

[3] Gael J. Curry, Programming By Abstract Demonstration, PhD Thesis, University of Washington at Seattle, 1978, Report 78-03-02

[4] Daniel Halbert, An Example of Programming by Example, Master's Thesis, Univeristy of California, Berkeley (Also Xerox PARC Report) 1981

[5] Donald Knuth and Ronald W. Moore, An Analysis of Alpha-Beta Pruning, Artificial Intelligence journal, Vol. 6 No. 4, 1975

[6] Henry Lieberman and Carl Hewitt, A Session With Tinker: Interleaving Program Design With Program Testing, Proceedings of the First Lisp Conference, Stanford University, August 1980

[7] Henry Lieberman, Example-Based Programming for Artificial Intelligence, Seventh International Joint Conference on Artificial Intelligence, August 1981

[8] David Neves, Learning Procedures From Examples, PhD Thesis, Carnegie-Mellon University, 1980

[9] Paul Pangaro, The Animation System EOM, Creative Computing, November 1980

[10] David Canfield Smith, Pygmalion: A Creative Programming Environment, Birkhauser-Verlag (also Stanford PhD Thesis AIM-260), 1975