MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 234 June,1982

TRIG: AN INTERACTIVE ROBOTIC TEACH SYSTEM

by

James R. McLaughlin

Abstract

Currently, it is difficult for a non-programmer to generate a complex sensor-based robotic program. Most robot programming methods either generate only very simple programs or are such that they are only useful to programmers. This paper presents an interactive teach system that will allow a non-programmer to create a program for a six degree of freedom mechanical robot. In addition to conventional guiding capabilities, the teach system will allow the user to create complex programs containing sensor-based moves (move until touch), loops, and branches.

1. INTRODUCTION

Currently, it is difficult for a non-programmer to generate a complex sensor-based robotic program. Most robot programming methods either generate only very simple programs or are such that they are only useful to programmers. This paper presents an interactive teach system that will allow a non-programmer to create a program for a six degree of freedom mechanical robot. In addition to conventional guiding capabilities, the teach system allows the user to create complex programs containing sensor-based moves (move until touch), loops, and branches.

This thesis introduces TRIG (Teaching Robots Interactively through Graphics), a graphically oriented interactive robotic teach system. TRIG attempts to combine the benefits of the two common robot programming techniques in use today: teach by guiding and "textual" programming. TRIG maintains the simplicity and immediate feedback of the guiding approach while incorporating many of the flexible control structures available to most textual programmers.

1.1. HISTORY OF ROBOTS

In traditional automation, specialized automated devices are designed to perform tasks that are tedious, exhausting, or unsafe for human involvement [Franchetti 78][Kato 79]. These devices are custom built and tailored for each installation. Consequently, they are rather costly and only economically feasible for sufficiently high throughputs [Nevins 75]. These traditional automation devices typically utilize no direct external sensing in their operation. In the automation of an assembly operation, for example, this is a severe limitation. The lack of sensing requires that item to item tolerances be rather small (certainly much smaller than can be handled by a human operator), often to the point of being more precise than the functionality of the assembled object dictates [Nevins 75]. Typically, vibrational tracks are required to properly seperate and orient pieces of the assembly. During

the assembly process, parts must also be precisely positioned by accurate jigging. Overall, this leads to a very costly mechanism to perform the assembly task.

Another major drawback of a specialized device is the excessive quantity of redesign of the machine required to reflect any but the most trivial design changes in the product being assembled. Such costly obstacles heavily restrict the economic viability of these specialized devices [Ihnatowicz 78].

During the last two decades, considerable effort has been expended towards creating general purpose automated manufacturing equipment. In particular, much of this effort has been directed towards computer controlled manipulator arms, commonly referred to as *robots*. Such computer controlled devices have the advantage that they can be programmed to perform any of a broad range of tasks with only minimal engineering and debugging effort. This has made the robot become viable for a broad range of needs [Corwin 75] [Gini 79].

The quality of an automated device can generally be measured by the following attributes [Corwin 75]:

- Technical Capability
- Flexibility
- Reliability

In terms of technical capability and flexibility, a computer controlled robot can outperform a specialized machine. Computer control allows tremendous flexibility in deciding how to respond to any given situation. The robot can modify its actions to reflect changing inputs, which makes such features as error detection possible [Cunningham 79]. Such systems are performing successively more complex tasks with decreasing human involvement [Corwin 75]. Also, the redundancy required to minimize down-time is greatly reduced when identical robots can be used to perform many different tasks. A small number of robots can serve as the redundant

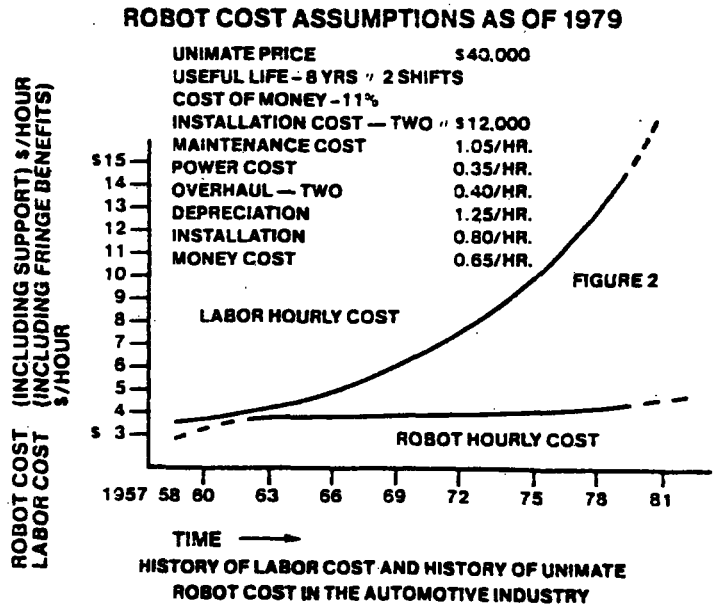


Figure 1. Cost of Labor vs. Cost of Unimate Robot [Engelberger 79]

replacement for a large number of similar robots (.1n:n), rather than the redundancy factor of 1:1 required for different machines to have similar protection.

There is doubt as to whether or not robots are as reliable as their highly specialized counterparts. This is probably due to the relative infancy of most robotic designs. For example, the Mean Time Between Failure (MTBF) for a 2000 series Unimate robot more than doubled during its first three years in the field [Engelberger 74]. Currently, robots have MBTF's of greater than 400 hours, which many feel is high enough to consider the machine "reliable", regardless of the relationship to the reliability of "hard" automation [Engelberger 74].

Recently, economic issues have also played a large role in the increased use of robots in industry, the primary factor being rising labor costs. During a period in which U.S. labor costs rose over 200%, the cost of a Unimate robot rose only 40%[IR

76.2] (see figure 1).

Additional benefits of a robot over a specialized automation device are:

- *Reaction Time.* An MIT study several years ago revealed that specialized machines required an average of 12 months and a range of 7 to 24 months to realize [Engelberger 79]. Robots, on the other hand, are available "off the shelf" and can be used after relatively minimal ancilliary work [Zermeno 79].
- *Debugging Cost.* The overall debugging cost of a specialized device is large, since the system is custom built and everything must be examined. With a robot, the control hardware and software is already debugged - all that needs attention is the user software.
- *Obsolescence.* Some specialized devices reach obsolescence even before they are put into operation, usually due to changing job specifications that the original device can not be readjusted to perform. Robots are much more flexible, allowing many design changes to be accomodated by software modifications.

Some other benefits of a computer controlled robot are:

- Positioning flexibility
- Control flexibility
- Ease of teaching
- System diagnostics
- Ease in interfacing to other equipment

1.2. METHODS OF PROGRAMMING ROBOTS

Today, such industrial robots typically consist of a manipulator with six degrees of freedom. Three major methods of generating robot programs have emerged through the years [Gini 79] [Salmon 78]. They are

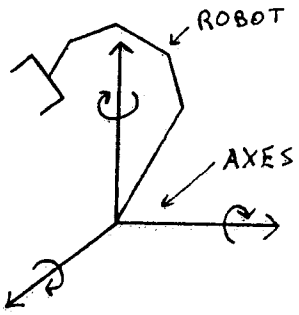
1) teaching by showing (guiding)
2) explicit programming
3) task-level programming

Teaching By Showing

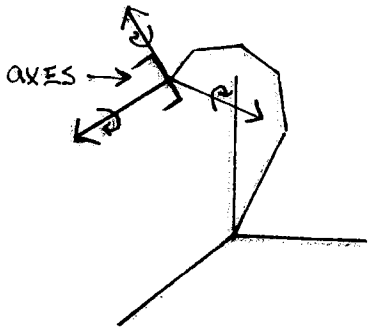
Most robots can usually be programmed by *teaching*. In *teach mode*, a hand held pendant or similar device with a button for each degree of freedom can be used to move the arm through the desired positions using one or more different mathematical translation systems. The most common such systems are world coordinates, tool coordinates, and joint mode.

- In the world coordinate system (figure 2a), the teach buttons control movement of the tool along the x-axis, y-axis, z-axis, and revolutions around each of these axes, where the axes are defined relative to the base of the robot. All axes move as necessary to produce the desired movement of the end effector.
- In the tool coordinate system (figure 2b), the teach box buttons control movement along and relative to axes similar to the world coordinate axes, only the axes are defined relative to the robot tool itself. Thus, movement in the negative z direction will cause the tool to retract from its current position directly backwards with no side-to-side or rotational movements.
- In joint mode (figure 2c), the teach box buttons map directly to each joint of the robot, i.e., depresing the joint 1 button will result in movement of joint 1 with all other joints remaining fixed.

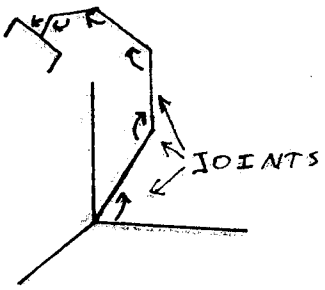
The teach box also contains a "record" button, which will save the information about the current robot position when pressed. To program the robot in teach



(a) World Coordinate System



(b) Hand Coordinate System



(c) Joint Mode

Figure 2. Three Different Teach Box Systems

mode, one moves the robot arm around using the teach box[1] until the robot is in a desirable position and then presses the record button. This saves the current robot location and orientation as the first step in the newly defined program. In a similar manner, one defines as many positions as desired.

Once the positions have been saved, the new "program" can be executed. This merely amounts to a rote playback of the stored positions, like a tape recorder.

Explicit Programming

Explicit programming entails the writing of a textual program, much like conventional computer programming. Programs can be constructed that exploit many different types of control structures (loops, conditionals, subroutines, etc.) Variables are also introduced, which allows the user to monitor performance of the robot and perform various actions based upon the values of these variables. The teach box is usually used by the programmer, but only to define the locations that the program will need - not to define the program.

Many explicit programming languages exist today. VAL [VAL 80] and SIGLA [Park 77][Salmon 78] are two of the more widely used examples of this style of programming.

This method is termed *explicit* programming because the programmer must specify the exact positions to which the robot is to move. The programmer explicitly controls every motion made by the robot. The program cannot simply state abstract tasks like

```
PICK UP THE BOX
```

The program must specify the exact position and rotation for the tool to assume (such positions are usually defined by the teach box). This means, for example, that

[1]Several alternative and original methods of specifying robot locations without using a "teach box" to control the robot have also been explored, but will not be presented in this paper. These methods are still inherently a teach by guiding approach [Seger 73][Kelly 77].

the robot does not employ any sort of collision avoidance scheme in determining its trajectory: the programmer must navigate the arm around any obstacles.

Task-level Language

Rather than specifying exact positions for the robot to move to, task-level languages allow the user to specify goals to be achieved by the program, and the system decides how best to move the manipulator to achieve the desired effect. The language's compiler (or interpreter) decides what forces and torques to exert, what speeds to use, where a part should be grasped, what trajectory to use, etc. This obviously requires that the system have explicit knowledge of the environment in which it is working (locations and specifications of parts, etc.). As noted by Grossman and Taylor [Grossman 75], this effectively shifts the problem of program generation from the procedures (assembly steps) to the declarations (part descriptions). However, this problem is not as large as it may seem, as CAD produced part descriptions can be utilized, simplifying the declarations required of the programmer. Because of the heavy computational demands of such a language, their use is limited to very large computer systems [Salmon 78].

Task level languages are currently still in the research and experimental stages. Two examples of such systems are AUTOPASS and LAMA [Lieberman 77][Lozano-Pérez 76].

1.3. CONTRAST BETWEEN THE METHODS

Teaching by guiding is the most widely used method of program generation used by current industry [Gini 78.2]. It has the obvious benefit that one need not be experienced in robotics or programming to generate the program. Manipulating the robot using the teach box is relatively easy for even a complete novice [Blanding 79]. The user does not write a "textual" program, and thus is not required to be able to relate abstract symbols to actions. This method also provides immediate feedback from the robot to the programmer. However, the associated cost is the very limited

capability of the program: there are no provisions for conditional transfers of control, loops, etc. The lack of text also makes maintainability, documentation, and modifications difficult.

Such programs that merely lead the robot through a series of predefined positions without sensory feedback are not very useful in many modern applications (such as assembly) [Gini 79][IR 76.1], and certainly do not exploit the sensory capabilities of current robots. More sophisticated programs are desired, to allow for larger part-to-part tolerances, recovery from object slippage, etc. Clearly, although the simplicity of the process is exceptionally desirable, teaching by guiding (as it now stands) is not a very powerful method of generating programs.

Currently, explicit programming is used in a few industrial robotic systems. This method overcomes many of the simplistic shortcomings of the teach by guiding approach, but now the robot user is required to be a programmer: the robot can no longer be programmed by a layman. Also, the immediate feedback present in the teach by guiding method has been lost, causing more errors and thus a lengthened debugging process. In order to gain the added flexibility that explicit languages provide, one has lost most of the simplicity of the teach by guiding process.

A task-level language simplifies the programming of the robot because it parallels the way in which we think. The natural way in which humans approach a task is in terms of the desired goals and subgoals of that task. It is natural to think of an assembly operation, for example, in terms of the desired effects on the parts, not in terms of the exact locations through time of a manipulator to carry out the task [Gini 78.2]. For the user, task-level languages could make complex program generation a rather simple process. However, task-level languages have proven to be exceptionally difficult to implement. Most of the problems of such a language are at the planning and representation level [Taylor 76], due to the ambiguity of the task specification. All such languages are still in the research phases, and show little hope of reaching full implementation in the near future.

It seems clear that a system that could retain the simplicity of the guiding process yet incorporate the more complex control structures of explicit programming would be desirable. This is the function that TRIG attempts to perform.

1.4. TWO EXAMPLE PROGRAM GENERATION SYSTEMS

1.4.1. VAL

VAL is the most widely used industrial robotic programming language. It is used in Unimation Inc. industrial robots. In VAL, the robot can be programmed both by guiding and by explicit programming methods. In *teach mode*, the teach box can manipulate the arm in the three coordinate systems presented earlier: *world*, *tool*, and *joint*. The box also has a *free mode*. In free mode, selected joints of the robot go limp (and if one is not careful, will cause the robot to crash to the ground) and can be manually positioned by the user [VAL 80].

As in the example teach mode presented earlier, the defined positions can only be replayed exactly as recorded. There are no provisions for loops or conditional transfers of control.

To overcome these limitations, the Unimate robots can also be programmed explicitly in a BASIC-like language (this method is by far the more common). Constructs for simple loops, integer arithmetic, conditional branches, and position transformations are provided. Subroutines, or subtasks, can be defined and used as necessary by main programs. Facility to communicate with auxilliary devices has also been included via the SIGNAL command.

A simple program to wait for exterior signal number 5 and then pick up a small box and drop it into a bin 10 times might look something like figure 3.

```
1)    SETI COUNT = 10
2)  99 WAIT 5
3)    APPRO PICK, -50
4)    MOVES PICK
5)    CLOSEI
6)    DEPART -50
7)    MOVE DUMPIT
8)    OPENI
9)    SETI COUNT = COUNT - 1
10)   IF COUNT GT 0 THEN 99
11)   RETURN
```

Figure 3. A Sample VAL Program

Lines 1 and 9 provide examples of the integer variable operations provided in VAL, which are of the form

SETI <intvar> = <intvar or int> [<operation> <intvar or int>]

The MOVE[S] command is the basic robot movement motion command. MOVES specifies a straight line trajectory to the specified location, whereas MOVE instructs the robot to move along a joint-interpolated trajectory (which may diverge from the straight trajectory but is computationally simpler and executes faster). APPRO[S] and DEPART[S] move the robot hand to the position that is the specified distance (in cm) backward or forward, respectively, along the current negative z axis of the hand from the specified position.

In the above program, the locations PICK and DUMPIT can be defined by using the teach box or can be numerically specified. All arm movements are specified explicitly by the exact final location and orientation of the manipulator. The robot knows nothing about its global environment, and is only required to interpolate between the destination positions that it is given by the user to determine trajectories.

1.4.2. AUTOPASS

AUTOPASS (AUTOmated Parts ASsembly System) is a world-modelling language developed by IBM. The intent of AUTOPASS was to relieve the user of any need to consider anything more detailed than the basic goals of the required task. In turn, the compiler does all of the dirtywork. The compiler plans such things as the grasping positions, trajectories, velocities, etc. (keeping collision avoidance and part constraints in mind). There are no variables, no loops, no conditional branches, etc., because everything that needs such structures is to be taken care of by the compiler. Instead, AUTOPASS commands deal with such things as tool control, instructions for placement, parts, etc [Park 77].

A typical AUTOPASS command, PLACE, has the following format [Lieberman 77]:

PLACE *parta* ON *partb*

and has four (optional) qualifying phrases:

GRASPING *surfaces*
SUCH THAT *final-condition*
SUBJECT TO *constraints*
THEN HOLD POSITION

Other typical instructions include OPERATE, ATTATCH, VERIFY, SLIDE, TURN, SWITCH, and PUSH. The user program is compiled prior to run time into an explicit programming language. The only requirements upon the user are that he be able to correctly plan the assembly process and not specify impossible steps.

A program segment to obtain a screw from a screw-dispenser might look like figure 4 [Liebernman 77].

```
PLACE screw-driver IN screw-dispenser
SUCH THAT screw-driver-tip CONTACTS dispenser
PLACE screw-driver-tip ON screw
SWITCH screw-driver ON
PUSH screw-driver-tip UNTIL seated
MOVE screw TO task
```

Figure 4. A Sample AUTOPASS Program

Clearly such a language requires an extensive geometric model of the world, which is non-trivial to generate. Only a subset of AUTOPASS was ever implemented by IBM [Lozano-Pérez 79].

2. TRIG

The general approach taken in TRIG is to provide a graphical flowchart-like representation of the program to the user, and to have a simple command syntax that will allow the user to make arbitrary changes to the program in a simple, intuitive manner. Facilities are provided for non-linear control structures in the form of loops and jumps. TRIG also has the sensory capabilities of a *guarded move* (move until touch) with conditional branching.

TRIG is currently running in the Artificial Intelligence Laboratory at the Massachesetts Institute of Technology. It consists of roughly 2000 lines (excluding comments) of code written in ZetaLisp, a dialect of Lisp. The target robotic device is the Purbrick Arm located in NE43-905 (see figure 5). The Purbrick Arm is a six degree of freedom (three translational and three rotational) robot that is directly driven by a PDP 11/45 (connected to the LISP Machine on which TRIG runs). The platter on the table has 2 degrees of freedom (X and Y) and the section mounted on the wall has the remaining four degrees of freedom (Z, ROLL, PITCH, and YAW). Its gripper has two fingers, one of which contains a force sensing device.

2.1. THE DISPLAY

The basic user interaction in TRIG is carried out through a graphical/textual display. The terminal screen is broken up into two displays and a command window, as shown in figure 6. The user gives commands and is queried when necessary in the Program Editor window. The entire user program is represented in the Full Display in miniature form without much detail - it does little more than indicate the existence of each step. The Partial Display window is a detailed listing of the program (as much as will fit in the window). At any given moment, both the Full Display and the Partial Display have a blinker that is located at the *active step* (to be explained later) in the program. Because both blinkers point to the same step in the program, the Full Display can be used to give a global idea of the location of

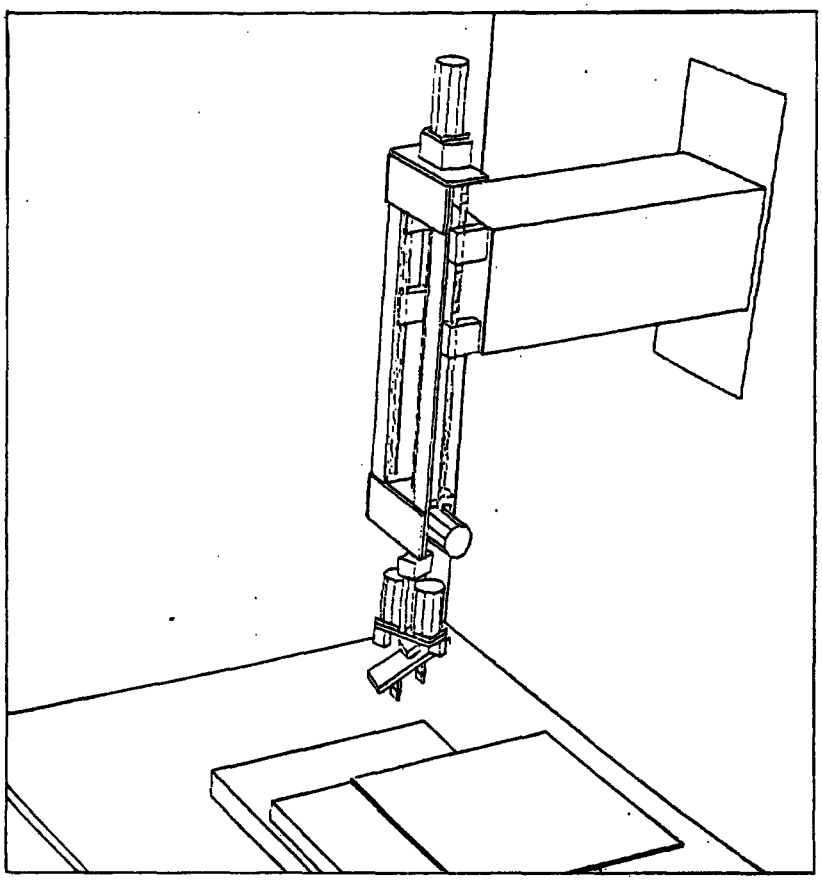


Figure 5. Purbrick Arm

the program section that is displayed in detail in the Partial Display. Any change in program content by a command from the user is reflected immediately in both display windows.

The representation of the program chosen is a 1-2 tree structure in which every node has zero, one, or two sons. The top node in the tree is the first step in the program. Transfer of control following completion of a step proceeds in a downwards fashion (except as redirected by loops). If a node has no sons, then that step is a terminating step in the program. If a node has only one son, then the transfer of control will proceed to that son after the node has completed execution.

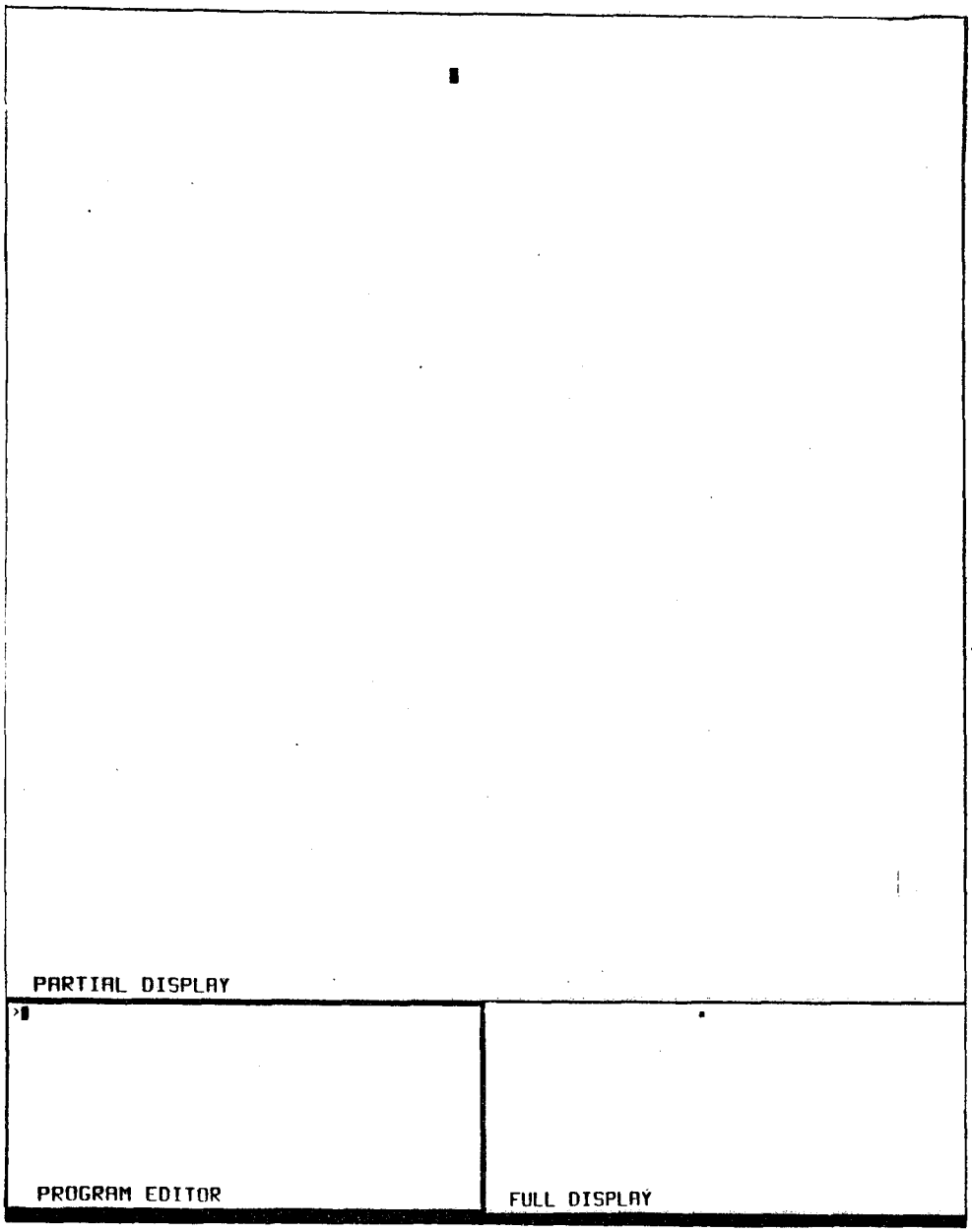


Figure 6. Screen Layout of TRIG

If a node has two sons, then the node is a step that represents a guarded move, and transfer of control will proceed to either the right or left son, dictated by whether or not an excessive force was detected.

2.2. TRIG COMMANDS

In TRIG, the user creates a program by typing simple commands and moving the robotic arm around using the teach box. The available commands are documented in the Appendix. Only a simple subset of the listed commands are usually required to generate a program - many are provided for special capabilities that are not always needed or are present to override defaults.

The commands can be broken up into two subsections: those that communicate with the robot and have no effect upon the program being designed, and those that pertain to the user's program.

The commands that communicate only with the robot are provided so the user can verify stored positions, create new positions, navigate the robot to desirable locations, and verify that certain guarded moves are feasible (test thresholds, etc.).

A Sample Program

Suppose that one wants to create a program that will perform the trivial task outlined in figure 7.

1. move the robot to the center of the workspace
2. pick up block A
3. dump block A in a bucket
4. pick up block B
5. place block B where block A was
6. move the robot back to the center of the workspace

Figure 7. Example Task No. 1

Upon starting up TRIG, both display windows will be empty (except for each window's blinker) and the ">" prompt will be in the program editor window as in figure 5, signalling that TRIG is ready to accept a command.

In the task of figure 7, Step 1 requires a movement to an absolute position (in contrast to a relative movement) in the robot's workspace. So, the first command entered by the user should be

```
>M CENTER
```

This will allow the teach box to control the robot arm, and will name the position that is saved to be CENTER (the choice of the position name CENTER is arbitrary, as are all position names). It will also make the first position in the program be a move to CENTER. The teach box that TRIG currently uses operates the robot in joint mode. When the robot has been positioned in the center of the workspace, the record button is pressed and the ">" prompt will appear.

The partial display window will look as it does in figure 8, indicating that the program is now one step long and contains a single move the the absolute position CENTER. In the display, all program steps (except loops) are represented as two grouped names: the step name and the position name. In figure 8, the default "***" step name indicates that no step name was given to the defined step. (Similarly, the

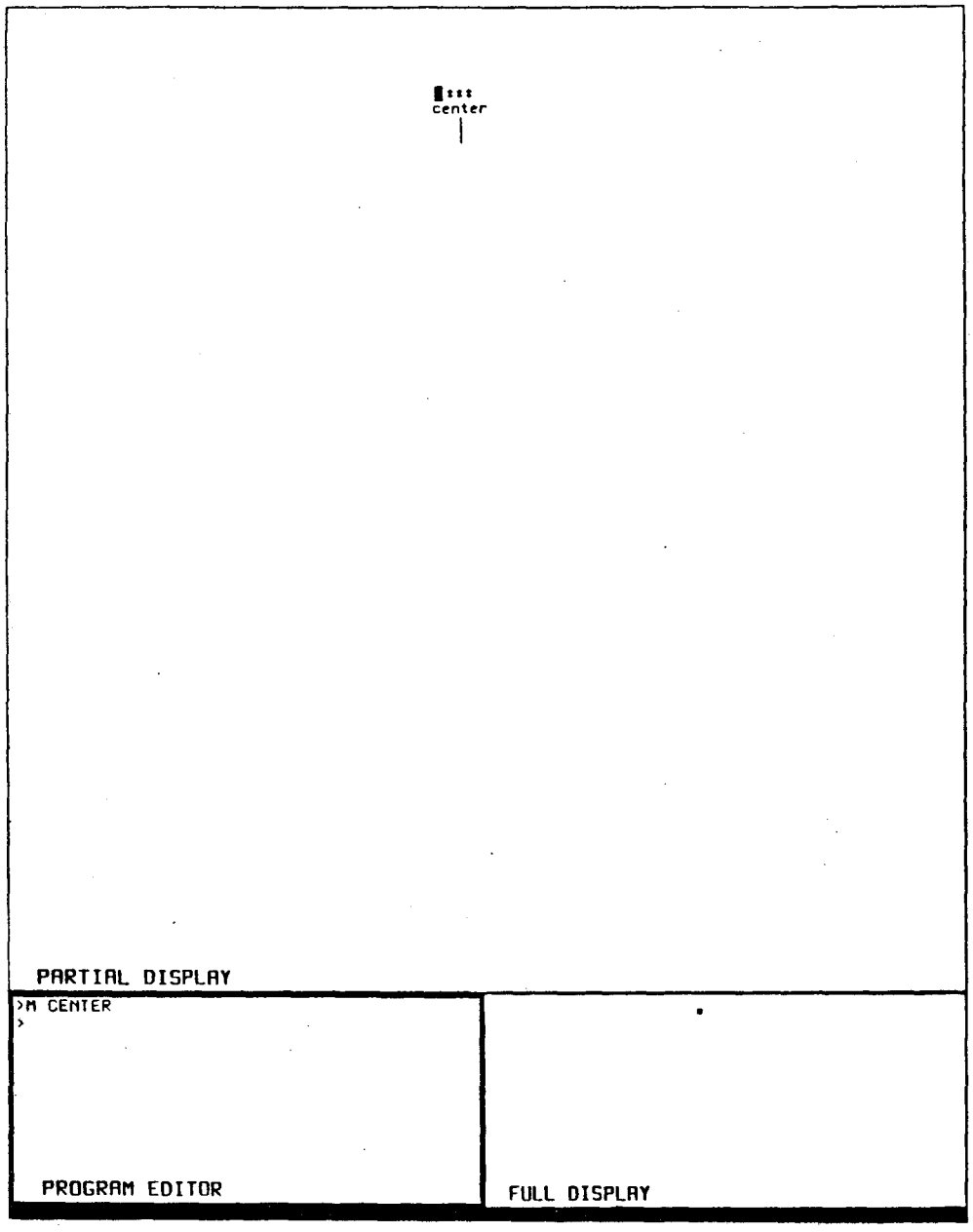


Figure 8. Screen After One Step of Example 1

absence of a position name is indicated by "- - -".) In order to distinguish relative and absolute positions, relative positions are preceeded by a "#" (see figure 14 for examples).

Note that there is a cursor to the left of the step that was just defined (in figure 8). This cursor indicates the current *active step* of the program. All step insertions, step deletions, executions, and cursor movements are defined relative to the current active step.

Step 2 also involves an absolute movement. The position of block A will need to be used again (to set block B there in step 5), so the position ought to be given a name. The next command should be

```
>M BLOCK
```

The robot should be moved with the teach box until its grippers are on either side of block A, and then the position should be saved. (This and all succeeding steps in this example assume that there will be no collision with the block itself or any other object in a linear move from CENTER to BLOCK. If such a collision were possible, an intermediate position directly above BLOCK might be necessary.) The robot still needs to grasp the block, so the user should type

```
>GRASP
```

The GRASP command adds a step to the program that will cause the robot to close its grippers when the step is executed, as well as cause the robot to grasp the box during the teach process.

The next step is simply

```
>M OVER-BUCKET
```

(and the robot should be moved to a position over the bucket and the position should be saved). The robot still needs to let go of the block, so now

```
>UNGRASP
```

Now the first three steps of Example 1 have been programmed. The Partial Display of the program will look as it does in figure 10. The remainder of the program can be found in figure 9:

```
>M CENTER
>M BLOCK
>GRASP
>M OVER-BUCKET
>UNGRASP
>M OTHER-BLOCK
>GRASP
>M BLOCK
Move to the predefined location? Yes.
>UNGRASP
>M CENTER
Move to the predefined location? Yes.
```

Figure 9. Commands to Perform Task of Figure 8

Note that neither of the last `M` commands required the positioning of the robot: the positions named in the command were already known to the editor and thus didn't need to be defined. TRIG will automatically move the robot to the predefined location. For this reason, TRIG will require the user to confirm that he realizes that the position is already defined. The final program in the Partial Display window should look as it does in figure 11. Because the program is relatively small, the entire program can be displayed in the Partial Display.

This rather trivial example does not utilize much of the power of TRIG. In fact, this example could be handled with ease by most current teach systems. As soon as more of the basics are presented, a more taxing example will be analyzed.

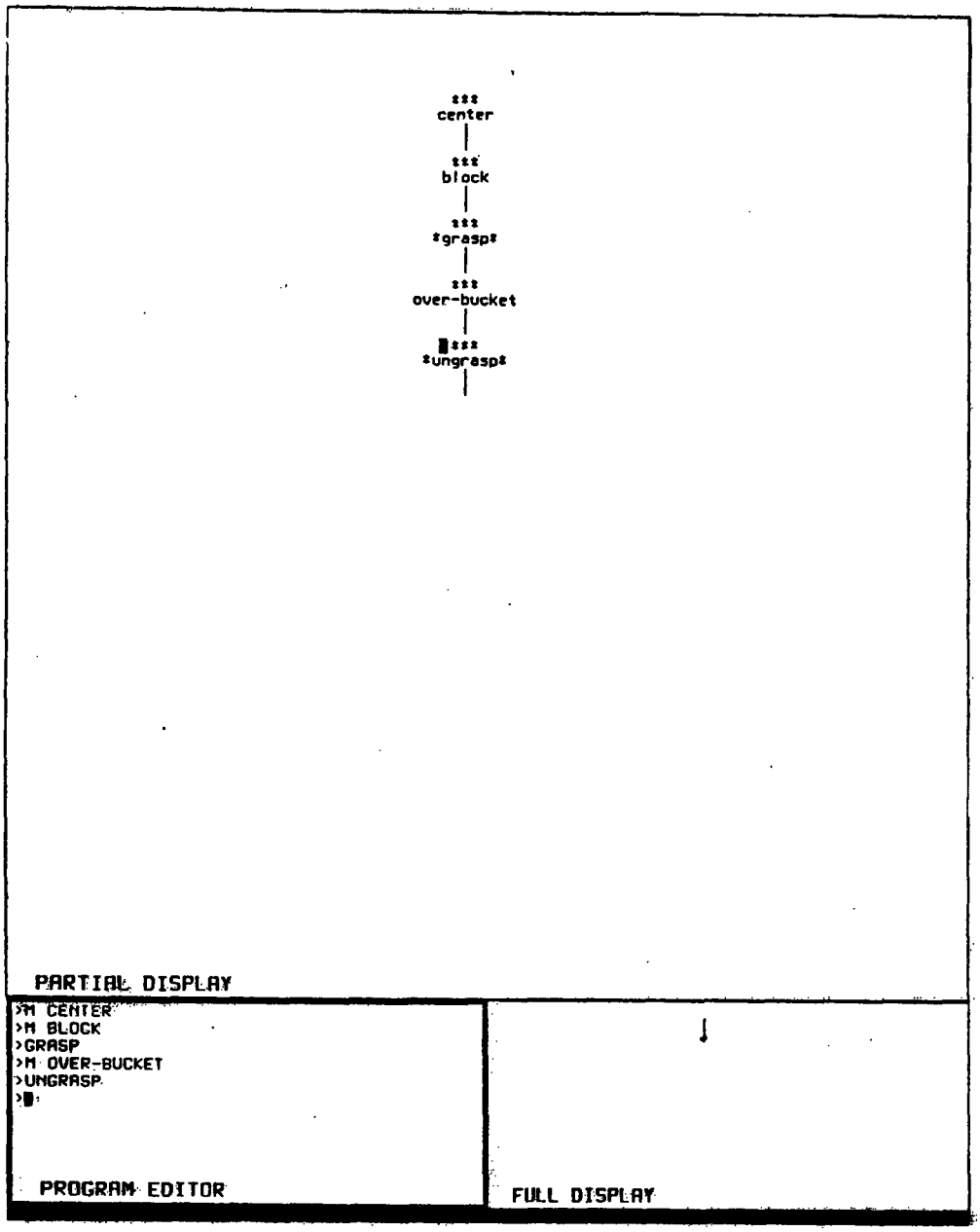


Figure 10. Screen After Three Steps of Example 1

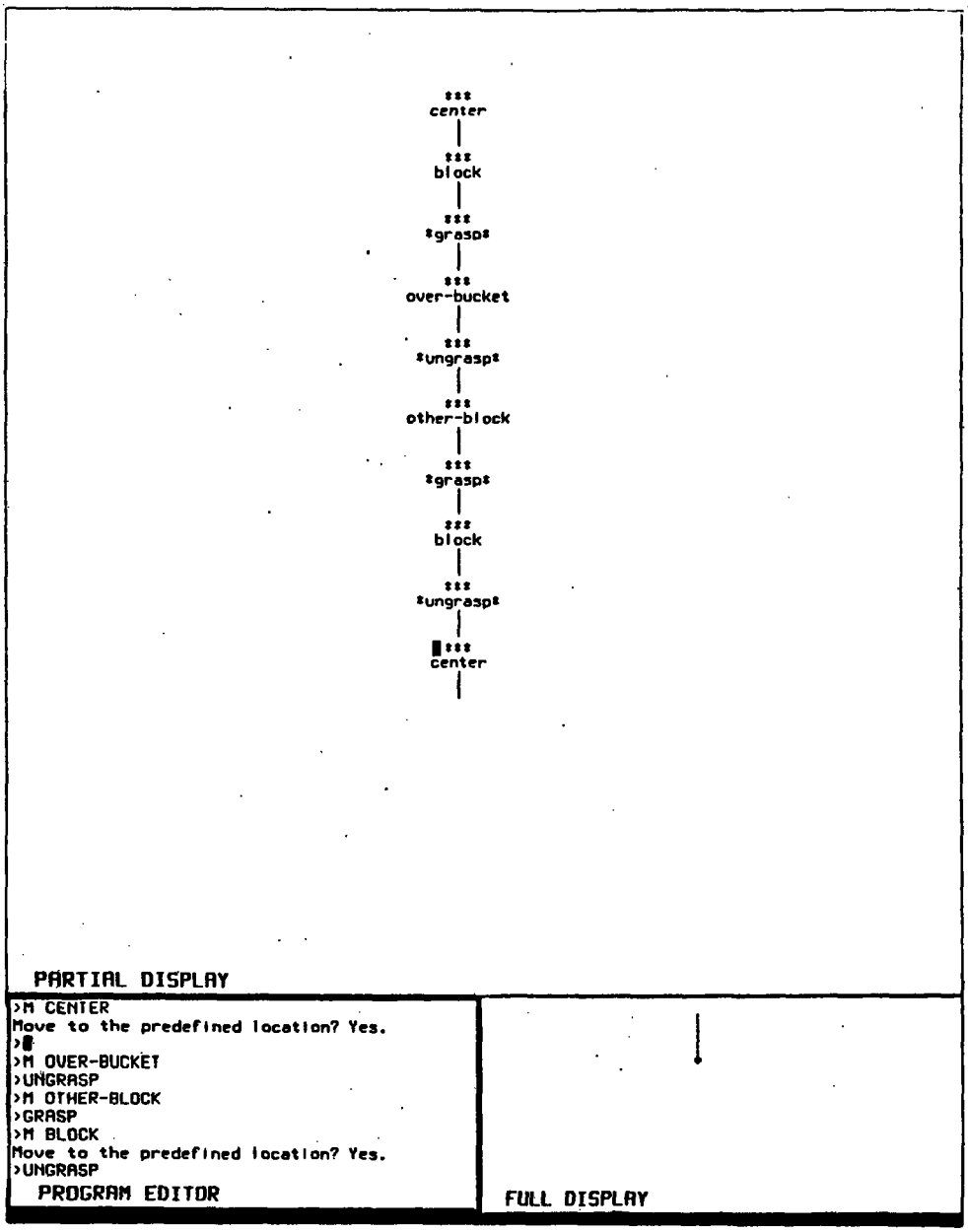


Figure 11. Display After Entering Program for Example 1

At any point in the preceeding teach process, the user can run a section or the entirety of his program that has been defined so far, either forwards or backwards (via the RUN command). In TRIG, running a program backwards will cause an "unwinding" (a traceback) of all steps that preceeded the current step during forward execution. The preceeding steps will be executed in the exact reverse order, including any loops or partial loops that were previously executed.[2]

Because the program is run in a interpretive manner, no pre-compilation is necessary for program execution to take place. This allows the user to instantly verify that any step or steps perform as desired. Should the user want to stop the robot during execution (to avoid disasters or save time once an error has been detected), facility is provided to terminate the program (by striking any key on the terminal or flipping a "kill" switch on the teach box). During program execution, the editing cursor marks the currently executing program step.

Many of the TRIG commands presented in Appendix 1 add steps to the program. M, MA, and MR all add normal move-to-a-position type steps to the program. MR indicates that the movement is to be relative to the current robot position, and similarly MA indicates an absolute movement to a specified position. The generic M command will default to be either a MA or a MR, depending upon where it is used in the program. The default of M can be determined by the rules outlined in figure 12. In the series of commands in figure 11, all of the M commands default to be absolute movements, as no preceeding steps are relative.

[2]It should be pointed out, however, that the effect of a step when executed forward can not always be undone during reverse execution. A step that drops a block into a bucket is one such example.

1. Position name argument not given by user
   OR Position name given is previously undefined:
   -> If a move-to type step exists above the current step,
   then default = type of movement of prior move-to step
   else default = absolute

2. Position name argument is previously defined:
   -> default = type of movement of argument definition

Figure 12. Rules for Default of " M" Command

M, MA, and MR each have an optional position name argument. Without an argument, TRIG will expect the teach box to be used to define a new unnamed position. This is useful when the position is not going to be referenced again in the program, in which case the position name would serve little purpose except documentation. In case the position ever needs to be referenced again, it should be given a name. Specifying a previously undefined position name argument such as "foo" will result in the position defined by the teach box to be known to TRIG as "foo", and can be referenced as such whenever a movement to that position is desired. (This is not to say that unnamed positions cannot be referenced ever again, but the process is rather indirect.) If the argument position name is already defined, repositioning the robot with the teach box is unnecessary: TRIG will insert the step with the predefined position into the program immediately, and move the robot to that location.

The other commands that add steps to the program will be discussed later when they are used in an example.

There are also several commands to manipulate the active step pointer, or editing cursor. Two such commands are N (for Next) and P (for Previous), which each take an optional numerical argument and move the active pointer up or

down the corresponding number of program steps. If the argument is ommitted, it defaults to 1. Whether the movement is up or down, loops are ignored. Moving around a program in this fashion is not unambigious, however: branches exist. TRIG handles branch selection as follows. When moving down the program (using N), TRIG will prompt the user to determine the proper route to take. When moving up in the program (using P), TRIG will move the pointer back through the path of "most recent steps"[3]. In practice, this is almost always the desired movement.

In some cases, moving up and down in the preceeding manner can be very tedious, for which cases the GOTO command can be used. GOTO, which requires a step-name argument, causes the active pointer to move to the specified step-name. Unnamed steps cannot be reached with a GOTO.

Program steps can be deleted in several ways. If only a single program step is to be deleted, the "D" command is used. This command takes an optional step-name argument, which defaults to the active step. A series of steps can be deleted with multiple uses of the "D" command or by using the "DA" command. The "DA" command (who's argument also defaults to the current active step) will delete a step and all dependent sub-steps (any node all of whose parents are decendants of the specified node). In a purely linear program, this would amount to every step beneath the specified step.

Another Example

Figure 13 briefly outlines a palletizing operation.This application will require relative movements, loops, and a guarded move. Figure 14 represents a possible TRIG teach process to perform this task, and the resulting screen setup is shown in figures 16 and 17. Two screens are shown because the program is large enough that the entire program can no longer fit within the Partial Window at one time. When the commands of figure 14 have been entered, the Partial Window will only

[3]The "most recent step" of a step with several possible "fathers" is defined as the father that was active most recently.

contain the latter section of the program (as in figure 16) because the active step cursor is at the end of the program. A command to move the active step cursor into the top section of the program was then typed, which caused the Partial Display to contain the upper section of the program in detail (yeilding figure 17). Figure 15 attempts to show the location of some of the positions used by the commands in figure 14.

---

1. Go to the center of the workspace.
2. Pick up a box from a stack. If the stack is empty, move to done position.
3. Place the box in the pallet.
4. Transfer the remaining boxes from the stack to the pallet. If the pallet is full, move to the done position.
5. Move to done position.

Figure 13. Example Task No. 2

```
1)  >M CENTER        ... define center...(center of the workspace)
2)  >M STACK         ... define stack...(just above the top of the stack)
3)  >CN GET-BLOCK
4)  >GRASP
5)  >MG BOTTOM
    Threshold: .7
    Force Vector: (.3 .3 .3)   ... define bottom...
                                   (at the bottom of the stack)
6)  >M DONE          ... define done...(done position)
    Down the guarded branch? No.
7)  >CN FINISH
8)  >P 3
9)  >RUN 3
10) >MR INCHUP
    Down the guarded branch? Yes.   ... define inchup...
          (inch up the manipulator until it no longer touches the block)
11) >UNGRASP
12) >M DWNBLK        ... define dwnblk...
          (hand down until fingers can pick up block)
13) >GRASP
14) >MA PALLET       ... define pallet...(above corner of pallet)
15) >MR PLACE        ... define place...(down into pallet)
16) >UNGRASP
17) >M -PLACE
    Move to the predefined location? Yes.
18) >M YOVER         ... define yover...(over to next row in pallet)
19) >SAVPOS PALLET
20) >CN RECALC1
21) >LOOP GET-BLOCK 2
22) >M XYOVER        ... define xyover...
          (over one column and back 3 rows in pallet)
23) >CN NEWROW
24) >SAVPOS PALLET
25) >LOOP GET-BLOCK 4
26) >LINK FINISH
```

Figure 14. Sample Palletizing Program (for Example 2)

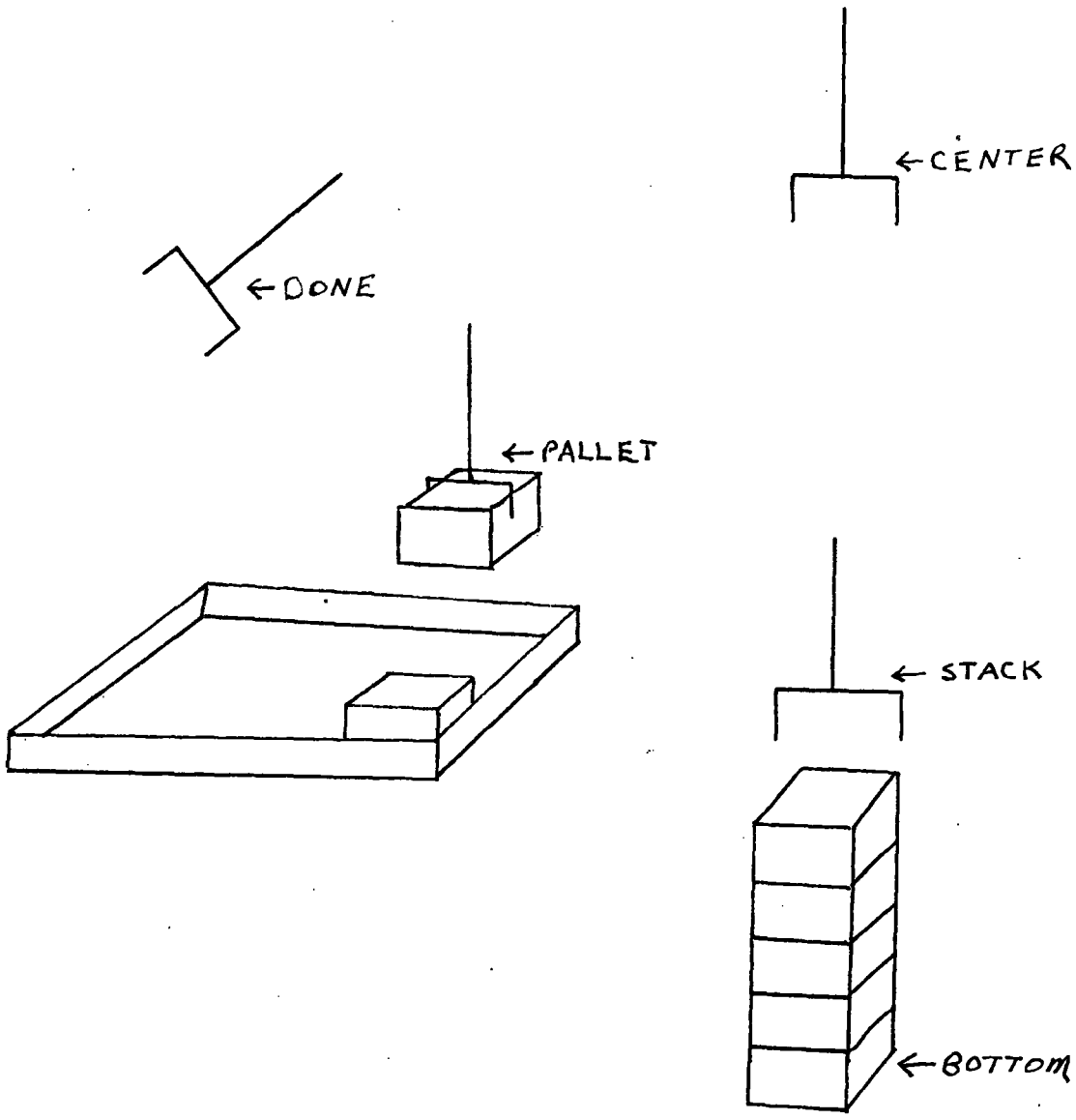


Figure 15. Approximate Manipulator Positions defined during Example 2

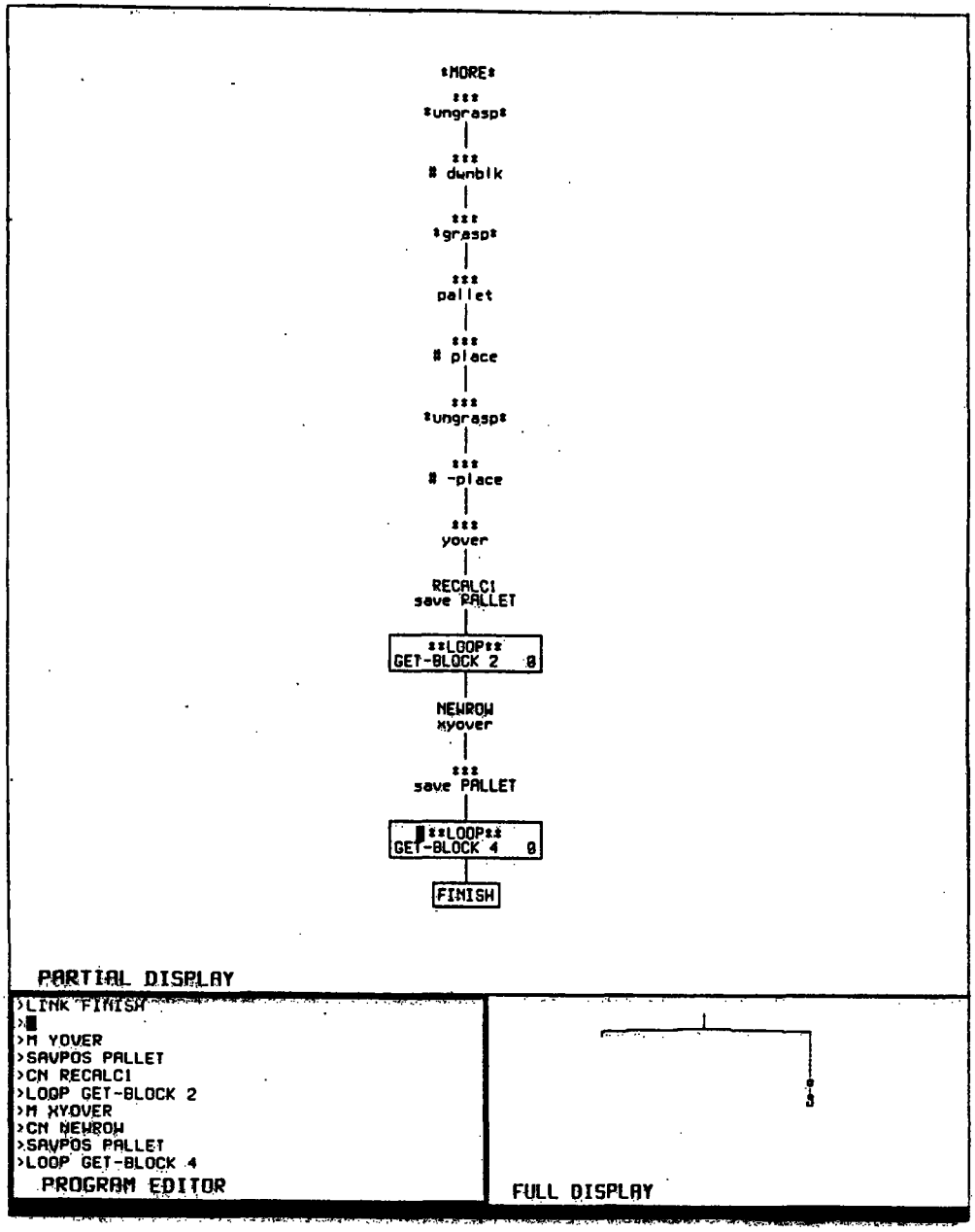


Figure 16. Completed Display for Example 2 (Last Section of Program)

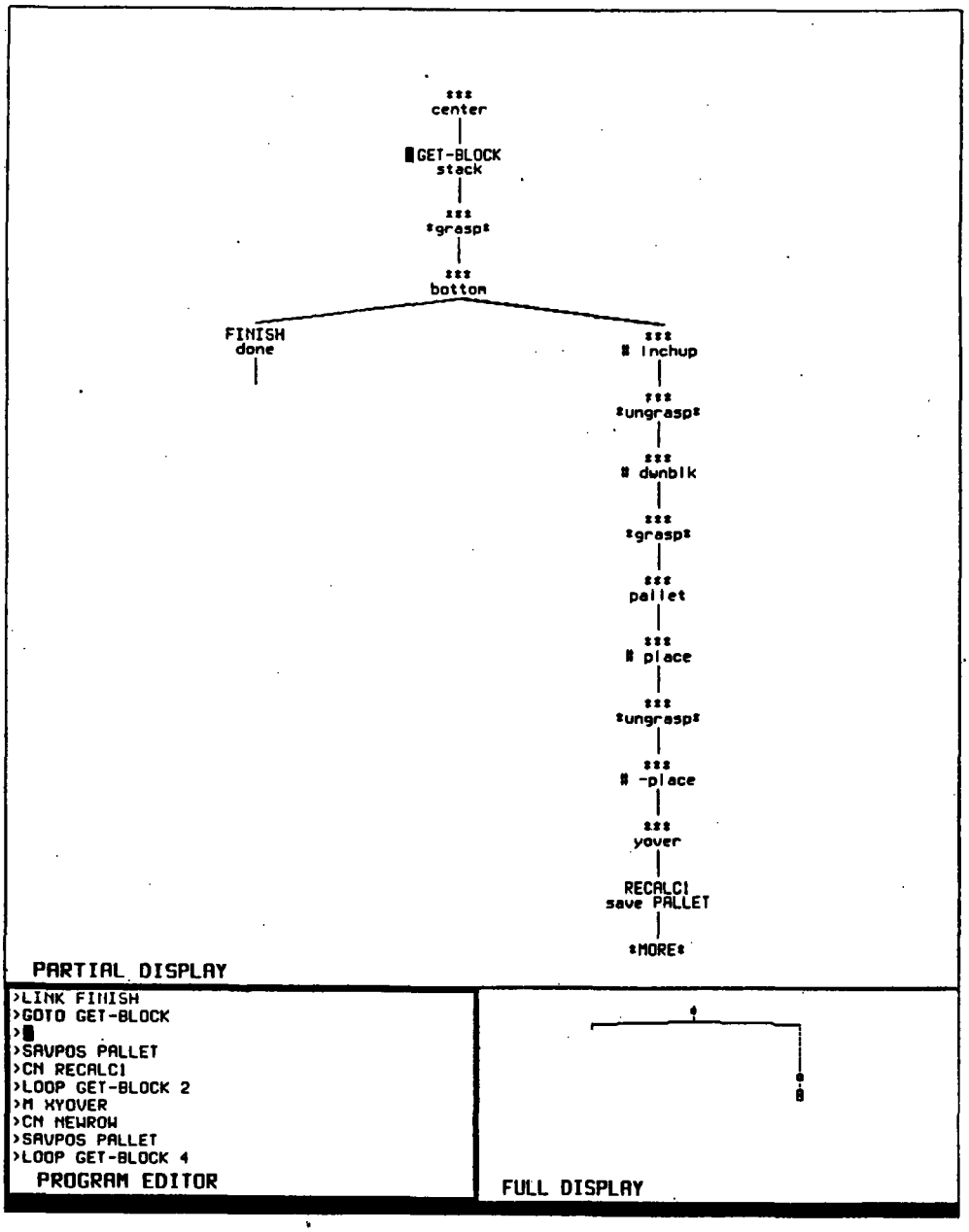


Figure 17. Completed Display for Example 2 (First Section of Program)

The constructed TRIG program utilizes a *guarded move* (move until touch) to tell where the next item on the stack is or if the stack is empty. Nested loops are used to index through the pallet. It should be noted that the program is slightly more complex than necessary, in order to introduce more TRIG commands for the purpose of discussion and clarity. For this same reason, all positions have been given names, although most of these names are unnecessary.

After the robot has been moved to the center of the workspace (line 1 of figure 14), the next subgoal is to find the top block on the stack. This is acheived by executing a guarded move that starts just above the stack (line 2) and has a goal position at the bottom of the stack (line 5). Before the guarded move is made, however, the grippers are closed (line 4) purely for esthetic reasons. (The guarded move could just as easily be performed with the grippers open.)

The command at Line 3 changes the name of the current active step (in which there is a movement to STACK) to GET-BLOCK so that it can be referenced later. The previous move step (to CENTER) is still unnamed, as can be seen in figure 17.

If any blocks are present, the previously mentioned guarded move will cause the manipulator to hit them. The MG set of commands (MG, MAG, and MRG) are analogous to the set of normal move commands described earlier (M, MA, MR), the only difference being that they construct a guarded move step rather than a normal move step. A guarded move has two force-related values associated with it that the user is prompted for: a threshold and a force vector. When a guarded move is made, the decision of when to terminate the trajectory due to incurred forces depends upon these values. The test for *excessive force* is

$$[\text{Force Vector}] \cdot \begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix} > \text{Threshold}$$

where $[f_x\ f_y\ f_z]$ is the vector of actual measured forces. If this test is true, then an excessive force is said to have occurred, and the current robot trajectory is

terminated. Control of execution in the program will pass to the right hand side son (the *guarded son*) in the partial display window. Had the robot reached the goal position without incurring an excessive force, control would have passed to the left son (the *normal son*).

Admittedly, line 5 assumes that the stack is not present during this phase of the teach operation (because the user must move the robot to the bottom of the stack in order to define BOTTOM, which would not be possible if the stack were present). Possible improvements to this scheme are presented in the Extensions section later.

Now the guarded move's two sons need to be defined. If no blocks are found, the manipulator can be moved to the DONE position. If a block is discovered, it should be picked up. The choice of which step to define first is arbitrary, as TRIG allows the user the flexibility to add a step to a program at any location whenever it is desired.

The commands of figure 14 define the normal son first (line 6). As stated earlier, the normal son defines the action to be taken if the manipulator makes it to position BOTTOM without encountering an excessive force. Because this step will be required later (when the pallet is full), it should be given a name. Line 7 names this step FINISH. The lack of excessive force will only occur if there are no more blocks in the stack. In this case, a terminating step of the program has been reached which has no steps that need follow it.

The guarded son (what to do if an excessive force is detected) of the guarded move has yet to be defined. To do this, one must manipulate the editing cursor back to the guarded step so that its other can be defined. A simple P command will not suffice, however, even though it will position the cursor in the desired location. This is because the manipulator is still in the DONE position, a position that it will never be in prior to executing the guarded move to BOTTOM step. The teach process is simplified if the two steps prior to the guarded move are executed. This will get

the manipulator into the position that it will be in when the program reaches the guarded move during normal program execution. Alternatively, the effect of lines 8 and 9 can be produced by:

```
>GOPOS STACK        ;send the robot to position STACK
>P                  ;move the cursor back to the guarded step
>RUN 1              ;execute the guarded move
```

or

```
>RUN -1             ;undo the last step (undoes FINISH)
>RUN 1              ;execute the guarded move
```

or any of several other combinations of commands. (Before the final RUN command is issued, one should make sure that there is a block on the stack for the manipulator to pick up.)

Steps 10 through 17 cause the robot to grasp the block on top of the stack (the block it bumped into) and place it into the pallet. Step 17 also demonstrates a useful variation of the family of move commands. Previously defined relative positions with a preceeding "-" sign can be used as a command argument, indicating a relative movement with the same magnitude as the predefined position and an opposite direction. This saves the user from having to define relative movements that are the mere opposites of predefined movements.

Step 18 moves the manipulator above the next row in the pallet, which is where the next block should be placed. The SAVPOS command in line 19 defines this location to be the new location for the position PALLET. The argument to the SAVPOS command becomes a "variable" position, the value of which can be changed over time (by using SAVPOS commands). In this case, the PALLET position is used to indicate the position for placement of the next block. Line 20 assigns a name to the SAVPOS step, purely for the sake of discussion.

Loops are introduced in line 21. The LOOP command requires two arguments, a step-name and a loop count. The loop step thus created will cause the program

to transfer control from step RECALC1 to step GET-BLOCK twice (and therefore the "body" of the loop will be executed three times) before transfer of control from RECALC1 finally proceeds to NEWROW. The displayed form of the loop step (see figure 17) shows the node to be looped back to, the desired loop count, and the current loop count (which is updated as it changes during run-time).

Steps 22 through 24 move the manipulator back over to the first row in the next column and redefine PALLET. (When a row is filled up, the next block will go in the next column, so PALLET should be reset to that location.) All that remains is to repeat the row-filling process 4 times (step 25) and then execute the FINISH step that was previously defined.

The LINK command of line 26 causes the flow of control in the program to proceed to FINISH when the lower loop to GET-BLOCK finally falls through. Thus, it causes an unconditional branch to the specified step in the program. Such branches are displayed as the target step name, surrounded by a box (see figure 16).

Now that the program is defined, running it will place the first 15 blocks in the stack into the pallet. If it is then run backwards (with a RUN -100 command, for example), the robot will pick up the 15 blocks in the pallet and place them back on the stack.

To briefly demonstrate the editing capabilities of TRIG, assume that the command on line 21 had been ommitted. The correction can be made by:

```
>GOTO RECALC1
>LOOP GET-BLOCK 2
```

and the omission will be corected. Similarly, by manipulating the active step cursor to the proper location, any number of steps can be inserted or deleted.

The program can be saved in a file with the W (for Write) command, and previously saved programs can be read into TRIG using the R (for Read) command. Both of these commands take a filename argument.

3. IMPLEMENTATION

The internal representation of a TRIG program is relatively simple. Each program step is a node in a 1-2 tree. Every node in the tree has forward pointers to one or two sons and backward pointers to at least one father, (see figure 21). Each node has specific information associated with it, as detailed in section 3.1 and figures 18 and 19. The major properties of the node are necessary to properly construct the tree and the program. The minor properties have transient values (that assist in display creation, etc.) and are recalculated frequently during program editing and execution. When a `WRITE` or `READ` command is performed, only the major properties are written or read, respectively. Execution of "the program" merely consists of moving through the tree, executing the intended task of a node, and following various pointers to determine which node to execute next (to be explained in more detail later).

3.1. SIMPLE STEP NODES

The 1-2 tree used by TRIG contains many nodes that only have one son, and represent a particular instruction for robot motion. These nodes will be referred to as *simple* nodes. Every manipulator motion except the guarded move is caused by the execution of one of these nodes.

These nodes can be seperated into two categories: Group 1) those that change the location and orientation of the manipulator and Group 2) those that operate the grippers. A group 1 simple node is created by a `M`, `MA`, or `MR` command. The value of the *position* property of these nodes specifies the name of their corresponding position. Any number of group 1 simple nodes can indicate movement to the same position. All such nodes have a *type* property of *normal*.

1) Type
2) Position
3) Sons
4) Fathers
5) Pstack
6) Loop Information
7) Threshold
8) Force Vector

Figure 18. Major Properties of a Node

1) Value
2) Location
3) Cursor Location
4) Display Number

Figure 19. Minor Properties of a Node

1) Position Data
2) Init Value
3) Relative

Figure 20. Properties of a Position

The positions specified by these nodes also have properties (see figure 20), which distinguish absolute positions from relative positions, etc. When a group 1 simple node is executed, its corresponding position is first examined. Absolute positions are output directly as a manipulator goal. If the position is relative, the current manipulator location is added to the corresponding relative position data

and the sum is treated as an absolute location for manipulator positioning.

The group 2 simple nodes differ from the group 1 nodes structurally in that they require no position information. GRASP and UNGRASP commands create nodes that have the unique types *grasp* and *ungrasp*. This allows such nodes to be identified as pertaining to a specific direction of gripper motion, without any additional position information. This method is acceptable as long as there are only two gripper states. If a variable width of opening of the grippers were allowed, the gripper information would have to be stored and treated in a similar fashion to (or even as a part of) the position information. On many current robots, the gripper information is an integeral part of the position information.

3.2. OTHER STEP NODES

3.2.1. Loops

During program execution, if one temporarily neglects loops, the node to execute following execution of the current node can be determined from the *sons* and *fathers* lists (major properties of a node) associated with the current node. These lists are constructed as would be expected for a doubly-linked list, as shown in figure 21. Every node has at most two sons, and can have any number of fathers (muitiple fathers are caused by the LINK command).

However, in the case of loops, one can no longer trace program execution through the sons and fathers lists. During forward execution, a loop node will either pass control to its son or back to some other node, depending upon the value of the loop count (commonly known as looping back or falling through the loop). The node to loop back to (the *loop target node*), the current loop count, and the target loop count are all stored in the Loop Information property of the loop node.

Further complications arise when the program is run backwards. The Loop Information property of a loop node contains information that serves as a forward pointer only - for the process to be reversible, the loop target node needs to have

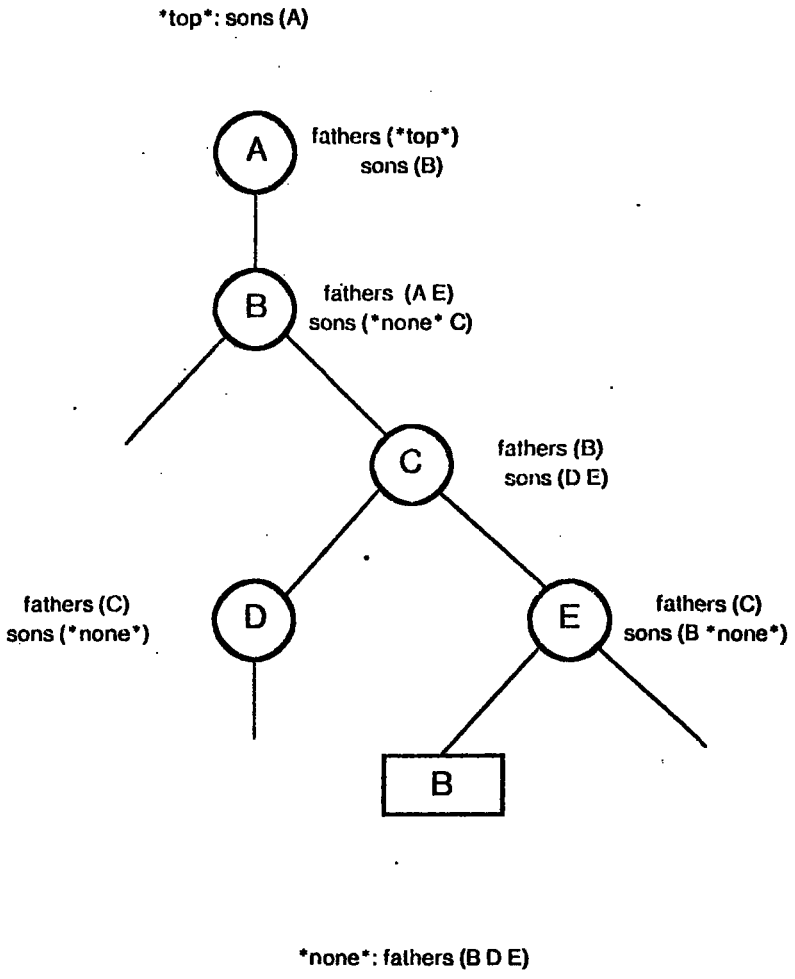


Figure 21. Construction of Father-Son Lists

some kind of a pointer back to the loop node. This is a basic property of any doubly linked list; if node A points to node B, then node B has some form of a pointer to node A, thereby facilitating mobility in both directions.

One alternative is to place a pointer to the loop node in the loop target node. As long as no two loop nodes specify the same loop target node, this method will work acceptably. However, as soon as two or more loop nodes point to the same loop target node (which occurs frequently when nested loops are used), confusion results. When this multiple loop target node is encountered during a backwards execution, it will be difficult to choose the proper loop node to jump back to out of the several that may exist.

TRIG solves this problem by introducing *bloop* (for backward loop) nodes. Bloop nodes and loop nodes occur in pairs - every loop node has a pointer to one and only one bloop node, and every bloop node has a pointer to one and only one loop node. The bloop nodes are placed in the tree immediatly preceeding the loop target node, such that the only father of the loop target node will be the bloop node. When multiple loop nodes have the same loop target node, their corresponding bloop nodes "stack up" on top of that node in the proper nested order.

Figure 22 shows the way in which loop and bloop nodes are inserted into the TRIG tree. The user commands on the right will produce the tree on the left. Each box in the tree on the left represents a node in the internal TRIG tree. Each node in the figure has a name (at the top) and the function the node is to perform (at the bottom). Prior to the first loop command ("LOOP LOC1 2"), the tree only contains the nodes A, LOC1, and D. The loop command causes the nodes C and E to be inserted. Similarly, the second loop command causes the insertion of nodes G and B. During forward execution the bloop nodes are ignored and treated as a "do nothing". The loop nodes, however, will transfer control to their corresponding bloop nodes the specified number of times. When executing in reverse, the loop and bloop nodes swap functions: the loop nodes are ignored and the bloop nodes

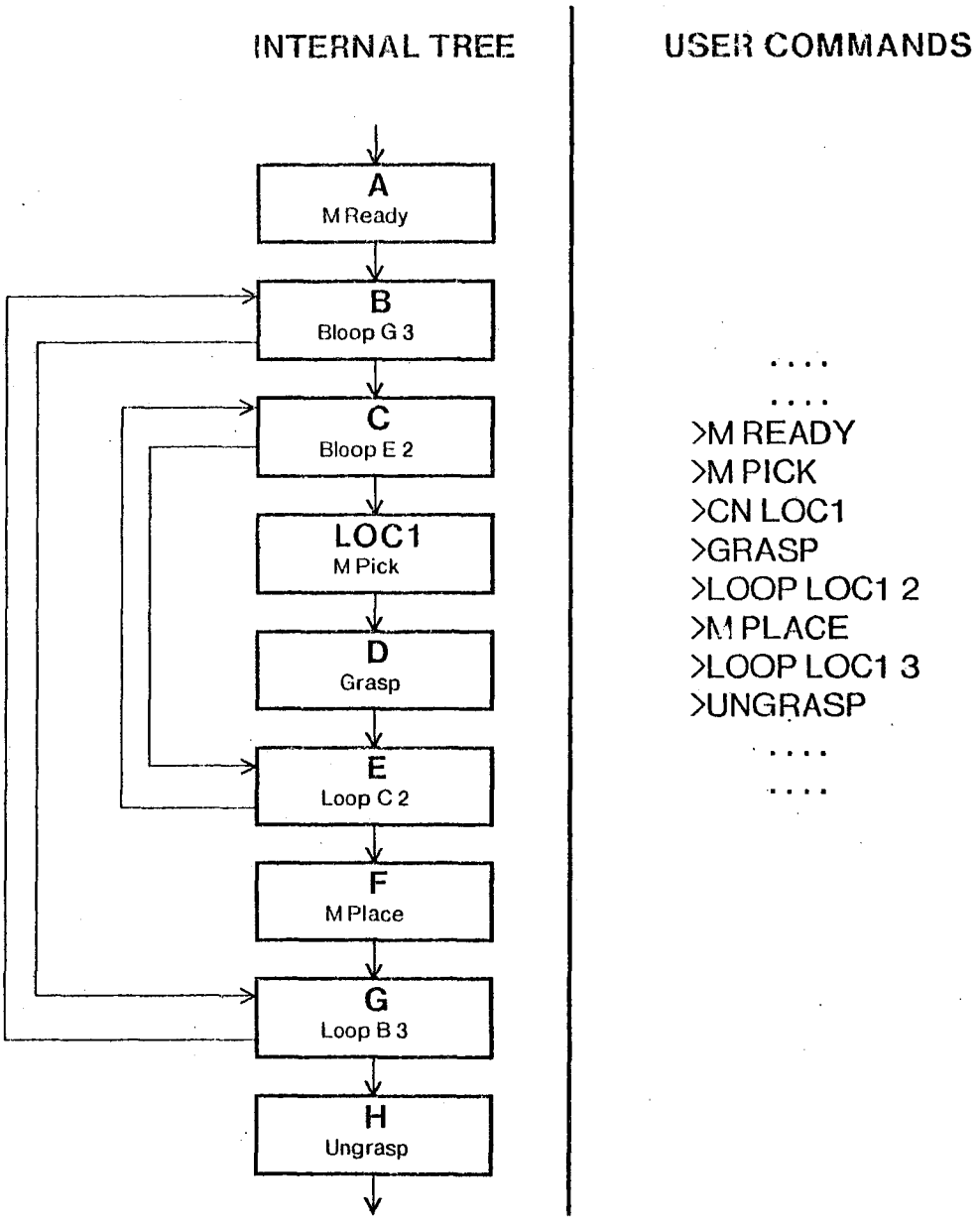


Figure 22. Insertion of Loop-Bloop Nodes

cause the transfers of control.

TRIG also takes care of initializing loops every time they are entered, causing nested loops to behave in the conventional way: an inner loop is iterated its corerresponding number of times for every iteration of an outer loop. In the example of figure 22, LOC1 and D are executed $(2 + 1)(3 + 1) = 12$ times. (Because the loop nodes loop *back*, their bodies are executed one more time than their loop count.

In order to keep this implementation method invisible to the user, bloop nodes do not appear in the program display windows (see figures 16 and 17).

3.2.2. Guarded Moves

Guarded move nodes are the only type of nodes in TRIG that have two sons. They are similar to normal moves in their treatment of relative and absolute position data. Guarded nodes have user-supplied values for the properties *threshold* and *force vector*, which are used in the test for "excessive force".

A seperate trajectory calculation routine had to be written for guarded moves that makes a slow, linear trajectory. The trajectory routine used for normal moves is much too fast to be used for guarded moves. Even if the user specifies a very small threshold for the excessive force calculation, the inertia of the arm will cause it to drive much harder into any object in its path than is desirable. Also, stopping the arm abruptly when it is travelling at high speeds can damage the arm's hardware. The guarded trajectory proceeds slowly enough that inertial effects are minimal and abrupt stops in the trajectory are not detrimental to the manipulator.

The new guarded trajectory routine also performs the necessary force readings and calculations. If the encountered force is great enough, the trajectory is terminated. Because the arm only has a force sensor on one of its fingers, the results are usually better if the guarded moves are made with the gripper open, causing only the "sensitized" finger to strike the object. This is because having the gripper

closed adds to the rigidity of the fingers as a unit, causing the force sensors to detect a smaller force than if the gripper were open.

3.2.3. Savpos Command

The *init value* of a position (see figure 20) is necessary because of the existence of the SAVPOS command. Prior to any program execution, all position data values are set equal to the position's init value. In order to have the user's program run the same way each time it is run, the initial values of all of the positions must be consistent from run to run. For example, the program built by figure 14 requires that the initial position PALLET correspond to the corner of the pallet every time that the program is run from the top, even though the position data of PALLET is modified many times during the course of the program.

3.3. NOTES ON EXECUTION

3.3.1. Special Nodes

There are two nodes that are present in every tree. These nodes are *top* and *none*. *top* is always the first (top) node in the tree and points at the first user-defined step (if one exists, else points at *none*). *none* is the son of any node without a user-defined child; it signals a termination of the program. In short, *top* and *none* are the beginning and end of any and every TRIG program (see figure 21). These nodes allow TRIG to locate the beginning and detect the end of program execution.

3.3.2. The Position Stack

The fact that a TRIG programs contain both absolute and relative movements makes the implementation of running a program backward difficult. For this reason, the *pstack* property of a node was created. The *pstack* (for position stack) of a node contains information relevant only to backwards execution. When a program is being run backwards, the desired effect is to "undo" each command in a backtrace of the recent execution. When there are no relative movements, "undoing" the

effect of a move-to-a-position type step is simple: simply execute the most recent move-to-a-position type step prior to that step. When relative movements are introduced, however, the solution is not as simple. The position that the robot was in prior to any particular movement can be a sum of any number of prior relative movements. Rather than search back, possibly through a lengthly series of loops, to determine the position that the robot was in before a particular step was executed, TRIG takes advantage of the plentiful memory space of a LISP Machine and creates a stack of previous positions for every node. When the program is run forward, the current robot position is pushed onto the stack of a node before the node is executed. This enables the backwards execution feature to be rather simply performed: to undo a move-to type step, simply pop the top position off of that node's stack and move the robot to that position.

4. CONCLUSIONS

4.1. DISCUSSION

The thrust behind the development of TRIG was the need for a non-programmer to be able to generate a complex robot program. TRIG has not been widely tested yet, so it is difficult to determine its success. Certainly, TRIG is not a polished, perfected system. Rather, it is a prototype of a new method of robot program generation. Time will uncover many of its flaws that may not be visible yet. Several of the possible extensions to be presented later will greatly enhance its ability to generate complex programs and create a friendlier user interface. It is hoped that enhanced versions of TRIG may someday be suitable for general use.

As it stands, TRIG has been used to generate numerous programs fairly readily. The author has created many programs with TRIG, most of which are at least as complex as Example 2 presented earlier (figure 14). Such tasks are non-trivial to write in explicit programming languages, yet TRIG seems to handle them rather easily.

TRIG is easier to understand when one is faced with the actual display screen and can run a sample program than it may seem from the preceeding introduction. This is because TRIG relies heavily upon the immediate feedback from the robot and the continuously updated display windows. Because of this interaction, it is difficult to give a proper flavor of TRIG here.

4.2. EXTENSIONS

One possible extension to TRIG that would be valuable is the introduction of subroutines. Such an addition would make many applications programs much simpler and would help add modularity to TRIG programs. It would also reduce the clutter in the display windows when the same series of commands are executed several times in different sections of the program, by replacing the series with

a single *subroutine call* node. TRIG would lend itself easily to this extension during execution, as little more than defining another node type and its associated properties need be done. The bulk of the work will lie in the editing phases of subroutines, as now the instruction set will have to be augmented to allow for the seperate displaying, referencing, and editing of multiple program sections. A possible consequence of subroutines would be a "library", which could contain the most commonly used subroutines and could be loaded into the editor at the beginning of a session.

At some point, TRIG can be used to generate output in an explicit programming language that can in turn be given to a manipulator. This would remove much of the overhead that TRIG imposes and would allow the resulting program to be run on a smaller machine (smaller than a LISP Machine) due to the saved space. TRIG would be the front end of the system, operating as an editor and debugger. Once the correct program is created, it can be "compiled" into the more efficient explicit programming language.

A source of possible problems in the current version of TRIG was pointed out on page 33. Currently, when defining a position for a guarded move to go to, the user must place the robot in that position. Many times, this involves moving away whatever the guarded move was supposed to run into, which is not always convenient or possible. (Say, for example, that the object is fixed!) Alternatively, the user could be allowed to define a vector with the robot which would define a direction for the move (this is the method used in FUNKY [Grossman 77]). The user might also specify a scaling factor for the defined vector to determine the endpoint of the trajectory.

Certainly, TRIG's current graphical interface is somewhat crude. The tree is not as efficiently or as well displayed as might be possible. Also, the addition of a graphical input device with menus on the display screen might be considered, as this would simplify the user interface considerably.

One of the system's major limitations is the inability to comment the programs that are created. This makes it more difficult to modify previously written programs. It is not exactly clear how comments can be incorporated into the existing structure of TRIG without cluttering up the program display screen to a large extent.

ACKNOWLEDGEMENTS

I would like to thank Tomás Lozano-Pérez for the support, guidance, information, and encouragement which he has provided. Conversations with Matthew T. Mason have also proved very helpful.

I thank the General Motors Corporation for supporting much of my undergraduate education with a scholarship.

Special thanks to Jeanine Marie Matouk for the patience, care, and support she has shown throughout this entire process.

5. APPENDIX OF TRIG COMMANDS

The Following Commands Control the Robot Motion

FREE

Allow the robot to be controlled by the teach box until the save button is pressed.

GOGPOS <[-]pos-name>

Move the robot (guarded) to the specified position. The user will be prompted for a threshold and a force vector.

GOGRASP

Close the robot grippers.

GOPOS <[-]pos-name>

Move the robot to the specified position.

GOUNGRASP

Open the robot grippers.

The Following Commands Affect the Robot Program

CALPOS

Begin a position calculation. When the ENDCAL <pos-name> command is typed, <pos-name> will be defined as the relative position difference between the current robot positions when the two commands were typed.

CN <step-name>

Change the name of the current step to be <step-name>.

CP <pos-name>

Change the position of the current step to be <pos-name>. The position specified by <pos-name> must already be defined.

CT <threshold>

Change the force threshold of the current step to be <threshold>. (Only applies to guarded steps.)

CV <force-vect>

Change the force vector of the current step to <force-vect>. (Of the form "(x y z)" - only applies to guarded steps.)

D [<step-name> [{n,g}]]

Delete program step named <step-name>. If program step is guarded, then n or g specifies which subtree to save.

DA [<step-name>]

Delete program step <step-name> and all dependent steps.

DLINK <step-name> [{n,g}]
Make the current step unlink from predefined and multiply used step <step-name>.

ENDCAL <pos-name>
Terminates a position calculation. <pos-name> will be set to the relative position between the robot position when CALPOS was typed and the current robot position.

EX <file spec>
Terminate editor, writing program out to file <file-spec>.

FLUSH
Erase the current program.

GOTO <step-name>
Move editor cursor to step <step-name>.

GRASP
Make the robot grippers close.

HERE <pos-name>
Define the current absolute robot location to be <pos-name>.

L <step-name> [{n,g}]
Make the program link to predefined step <step-name>.

LOOP <step-name> <# of times>
When executing program, loop to step named <step-name> the specified number of times before proceeding on to next node.

M [[-]<pos-name> [{n,g}]]
Move (normal) to postion <pos-name> if it has been pre-defined, else move normal and define user specified position to be <pos-name>. Type of movement (relative or absolute) is determined by use.

MA [<pos-name> [{n,g}]]
Same as M only type of movement is absolute.

MAG [<pos-name> [{n,g}]]
Same as MG only type of movement is absolute.

MG [<[-]pos-name> [{n,g}]]
Move (guarded) to postion <pos-name> if it has been pre-defined, else move guarded and define user specified position to be <pos-name>. Type of movement (relative or absolute) is determined by use.

MR [<[-]pos-name> [{n,g}]]
Same as M only type of movement is relative.

MRG [<[-]pos-name> [{n,g}]]
Same as MG only type of movement is relative.

N [<# of steps>]
Proceed to next (normal or guarded) program step.

P [<# of steps>]
Proceed to previous program step.

Q
Terminate the editor, saving nothing.

R <file spec>
Read in the program saved in file <file-spec>.

RUN [<# of steps>]
Execute the specified number of program steps starting with the current active location. A negative step count indicates to run the program backward. If no argument, executes the entire program.

SA <pos-name>
Save current absolute position as <posname>.

SAVPOS <pos-name> [{n,g}]
Create a program step that will set the location of <pos-name> to the current robot position during program execution.

SR <pos-name>
Save current relative position as <posname>.

UNGRASP
Make the robot grippers open.

W [<file spec>]
Write program out to file <file-spec>.

6. REFERENCES

[Blanding 79]
W. Blanding, "Potential Warehouse Applications of Industrial Robots", *The Industrial Robot*, Vol. 6, No. 3, 125-129.

[Corwin 75]
Merton Corwin, "The Benefits of a Computer Controlled Robot", Proceedings of the Fifth International Symposium of Industrial Robotics, Chicago, Illinios, September 1975, 453-469.

[Cunningham 79]
Carole S. Cunningham, "Robot Flexibility Through Software", Proceedings of the Ninth International Symposium of Industrial Robotics, Washington, D.C., March 1979, 297-307.

[Darringer 75]
John A. Darringer, Michael W. Blasgen, "A High Level Language for Research in Mechanical Assembly", IBM Research Report RC-5606, September 1975.

[Engelberger 74]
J. F. Engelberger, "Three Million Hours of Robot Field Experience", *The Industrial Robot*, Vol. 1, No. 4, June 1974, 164-168.

[Engelberger 79]
J. F. Engelberger, "Robotics in 1984", *The Industrial Robot*, Vol. 6, No. 3, 115-119.

[Finkel 76]
Raphael A. Finkel, "Constructing and Debugging Manipulator Programs", Stanford Artificial Intelligence Project Memo AIM-284, Stanford Computer Science Report STAN-CS-76-567, August 1976.

[Franchetti 78]
I. Franchetti, P. Vicentini, L. De Togni, -. Magarini, "Automation of Forging by Means of Robots", *The Industrial Robot*, Vol. 5, No. 3, 121-122.

[Gini 78.1]
Giuseppina Gini, Maria Gini, "Using a Task Description Language for Assembly. The Generation of World Models.", Proceedings of the Eighth International Symposium of Industrial Robotics and the Fourth International Conference on Industrial Robot Technology, Stuttgart, West Germany, March 1978, 364-372.

[Gini 78.2]
G. Gini, M. Gini, Object Description with a Manipulator", *The Industrial Robot*, Vol. 5, No. 1, March 1978, 32-35.

[Gini 79]
G. Gini, M. Gini, R. Gini, D. Giuse, "Introducing Software Systems in Industrial Robots, Proceedings of the Ninth International Symposium of Industrial Robotics, Washington, D.C., March 1979, 309-319.

[Grossman 75]
David Grossman and Russell Taylor, "Interactive Generation of Object Models with a Manipulator", Stanford Artificial Intelligence Project Memo AIM-274, Stanford Computer Science Report STAN-CS-75-536, December 1975.

[Grossman 77]
David Grossman, "Programming a Computer Controlled Manipulator by Guiding Through the Motions", IBM T. J. Watson Research Center, Report RC 6393, March 1977.

[Ihnatowicz 78]
E. Ihnatowicz, "One Man's View", *The Industrial Robot*, Vol. 5, No. 3, 115-116.

[IR 76.1]
"Computer Controlled Mechanical Assembly", *The Industrial Robot*, Vol. 3, No. 1, March 1976, 7-13.

[IR 76.2]
"European Automotive Industry Most Aggressive Robot User", *The Industrial Robot*, Vol. 3, No. 2, June 1976, 72-74.

[Kato 79]
I. Kato, "Future of Robotics", *The Industrial Robot*, Vol. 6, No. 1, 15-18.

[Kelly 77]
R. B. Kelly, K. C. Silvestro, "V/I-A Visual Instruction Software System for Programming Industrial Robots", *The Industrial Robot*, Vol. 4, No. 2, June 1977, 59-72.

[Lieberman 77]
L. I. Lieberman, M. A. Wesley, "AUTOPASS - an Automatic Programming System for Computer Controlled Mechanical Assembly", IBM Journal of Research and Development, Vol. 21, No. 4, July 1977, 321-333.

[Lozano-Pérez 76]
Tomás Lozano-Pérez, "The Design of a Mechanical Assembly System", MIT Artificial Intelligence Laboratory, Tech Report 397, December 1976.

[Lozano-Pérez 79]
Tomás Lozano-Pérez, Michael Wesley, "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles", Communications of the ACM, Vol. 22, No. 10, October 1979.

[Nevins 75]
J. L. Nevins, D. E. Whitney, "Adaptable-Programmable Assembly Stations: an Information and Control Problem", Proceedings of the 5th International Symposium of Industrial Robotics, Chicago, Illinios, September 1975, 387-406.

[Park 77]
William T. Park, "Minicomputer Software Organization for Control of Industrial Robots", 1977 Joint Automatic Control Conference, San Francisco, CA, 1977.

[Salmon 78]
Mario Salmon, "SIGLA - The Olivetti Sigma Robot Programming Language", Proceedings of the Eighth International Symposium of Industrial Robotics and the Fourth International Conference on Industrial Robot Technology, Stuttgart, West Germany, March 1978, 358-364.

[Seger 73]
B. Seger, "Control Systems for Industrial Robots", *The Industrial Robot*, Vol. 1, No. 1, September 1973, 12-14.

[Tarvin 80]
Ronald L. Tarvin, "Considerations for Off-line Programming a Heavy Duty Industrial Robot", Proceedings of the Tenth International Symposium of Industrial Robotics and the Fifth International Conference on Industrial Robot Technology, Milan, Italy, March 1980, 109-117.

[Taylor 76]
Russell Taylor, "A Synthesis of manipulator control programs from task-level specifiactions", Stanford Artificial Intelligence Project Memo AIM-282, Stanford Computer Science Report STAN-CS-76-560, July 1976.

[VAL 80]
"Users Guide to VAL", Unimation Inc., Version 12, June 1980.

[Zermeno 79]
R. Zermeno, R. Moseley, E. Braun, "The Industrial Use of Robots", *The Industrial Robot*, Vol. 6, No. 3, 141-147.