DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.
Production Systems as a Programming Language

for Artificial Intelligence Applications

Volume III

Michael D. Rychener

December 1976

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa 15213

This report reproduces a dissertation submitted to the Department of Computer Science at Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported in part by the Defense Advanced Research Projects Agency under Contract no. F44620-73-C-0074 and monitored by the Air Force Office of Scientific Research.
Table of Contents

For Volume III

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>V. KPKEG: A Production System for King-Pawn-King Endgames</td>
<td>V-1</td>
</tr>
<tr>
<td>Start of Appendices</td>
<td>V-31</td>
</tr>
<tr>
<td>End of Appendices</td>
<td>V-53</td>
</tr>
<tr>
<td>VI. MillPS/WBlox: A Natural Language Input Toy Blocks Problem Solver</td>
<td>VI-1</td>
</tr>
<tr>
<td>Start of Appendices</td>
<td>VI-69</td>
</tr>
<tr>
<td>End of Appendices</td>
<td>VI-134</td>
</tr>
</tbody>
</table>

Preface to Volume III

This volume contains two chapters, covering work with production systems in the areas of natural language processing and game playing. Chapter V describes a program that plays a simple class of chess endgames, and discusses the possibilities of using production systems for chess in general. Chapter VI describes a system that carries on a dialog about a toy blocks world, and that solves a class of problems in that world similar to the capabilities of Winograd's system. Each chapter has an abstract and a detailed table of contents. It is assumed that the reader has some familiarity with Volume I of this report, which discusses the goals and conclusions of the thesis as a whole, and which introduces the production system language in which the systems in this volume are implemented. The chapters have a similar organization, starting with a general description of the task performed by the system, and proceeding to a description of the system and its behavior. There are sections that discuss issues with respect to the task itself and with respect to the use of production systems.
Chapter V

KPKEG

A Production System for King-Pawn-King Chess Endgames

Abstract. KPKEG is a production system implementation of a program that plays chess endgames, restricted to king and pawn versus king. The program is described and several examples of its operation are discussed. The program's chess knowledge is given, and how this knowledge is expressed as productions is described. Experiments with KPKEG have brought out several features of the principle on which the search is based and the chess knowledge organized, the strategy hierarchy. Features of the productions and how they compare with a Lisp version of a similar program bring out the advantages of this implementation. The productions lend themselves readily to extension to more demanding chess tasks.
## Table of Contents

For Chapter V

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Introduction</td>
<td>V-1</td>
</tr>
<tr>
<td><strong>B</strong> The KPKEG Program in Detail</td>
<td></td>
</tr>
<tr>
<td>B.1 A simple example of program behavior</td>
<td>V-3</td>
</tr>
<tr>
<td>Fig. B.1 Starting and intermediate positions for TEST1; White to move</td>
<td>V-3</td>
</tr>
<tr>
<td>B.2 An overview of the structure of KPKEG</td>
<td>V-4</td>
</tr>
<tr>
<td>Fig. B.2 VAPs for KPKEG</td>
<td>V-5</td>
</tr>
<tr>
<td>B.3 Full detail on selected aspects of KPKEG</td>
<td>V-7</td>
</tr>
<tr>
<td>Fig. B.3 APs for control in the executive</td>
<td></td>
</tr>
<tr>
<td>Fig. B.4 Implementation of terminal position ( f )</td>
<td>V-8</td>
</tr>
<tr>
<td>Fig. B.5 Updating the board for a king move</td>
<td>V-11</td>
</tr>
<tr>
<td>Fig. B.6 Means for moving toward a square</td>
<td>V-12</td>
</tr>
<tr>
<td>Fig. B.7 Summary of chess knowledge in the strategy ( P )</td>
<td>V-13</td>
</tr>
<tr>
<td>Fig. B.8 A typical strategy ( P )</td>
<td>V-13</td>
</tr>
<tr>
<td>B.4 Meanings for KPKEG predicates</td>
<td></td>
</tr>
<tr>
<td><strong>C</strong> Results of Experiments with KPKEG</td>
<td>V-17</td>
</tr>
<tr>
<td>Fig. C.1 Test2 and Test3 initial positions; White to move</td>
<td>V-18</td>
</tr>
<tr>
<td><strong>D</strong> Production-System-Related Features of the Implementation</td>
<td>V-19</td>
</tr>
<tr>
<td><strong>E</strong> A Comparison to a Similar Program in Lisp</td>
<td>V-21</td>
</tr>
<tr>
<td>Fig. E.1 Fragment of Estim function of CP</td>
<td>V-24</td>
</tr>
<tr>
<td><strong>F</strong> Extending KPKEG</td>
<td>V-25</td>
</tr>
<tr>
<td>Fig. F.1 A non-localized K-P-K position</td>
<td>V-26</td>
</tr>
<tr>
<td>Fig. F.2 A task presently beyond KPKEG's capabilities</td>
<td>V-26</td>
</tr>
<tr>
<td>F.1 Acknowledgements</td>
<td>V-28</td>
</tr>
<tr>
<td><strong>G</strong> References</td>
<td>V-29</td>
</tr>
</tbody>
</table>

### APPENDIX

<table>
<thead>
<tr>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong> Program Listing for KPKEG</td>
</tr>
<tr>
<td><strong>B</strong> Cross-Reference of Predicates</td>
</tr>
<tr>
<td><strong>C</strong> Sample Created Net Productions</td>
</tr>
<tr>
<td><strong>D</strong> Detailed Behavior on TEST1</td>
</tr>
<tr>
<td><strong>E</strong> Traces for the Other Tests</td>
</tr>
</tbody>
</table>
A. Introduction

This chapter is concerned with a PS program, called KPKEG (king pawn king endgame), for playing a restricted form of chess, namely, chess endgames in which a king and a lone pawn of one color are opposed by a lone king (hereafter, the subset of chess with king and pawn versus king will be abbreviated K-P-K). Although chess is a specialized area of AI, and is probably a suitable domain only for chess experts (which I am not), it will still be useful for the present thesis for the following reasons. As Berliner (1973) has argued, the classical heuristic search approach to chess has fundamental limitations, which have been observed empirically in performance and theoretically by Berliner on the basis of critical situations in which the search techniques appear to be hopelessly inappropriate. Consequently there has been a shift in emphasis towards bringing to bear more of the kinds of chess knowledge used by human players. Since PSs are being put forth as useful tools in encoding problem-solving knowledge, it is reasonable to do preliminary experiments along the lines here. In addition, even a restricted chess program provides an easy benchmark for comparison with other control structures, since a variety of other programs exist, with a current effort using Lisp on a very similar chess domain.

The central chess concepts behind KPKEG are provided by Fine's (1941) analysis of K-P-K endgames. In this problem area there are a reasonably small number of pieces of knowledge that prove to be adequate for correct analysis. That is, KPKEG relies heavily on the use of patterns of chess pieces and much less on a search of possible move sequences leading to a win or draw. Patterns are used both to direct the program's attention to effective moves and to evaluate positions reached by the search. The search of possible variations of play is conducted under an executive scheme called a strategy hierarchy, developed by Berliner (1975b) as appropriate (at least) to the K-P-K domain. The strategy hierarchy in KPKEG consists of seven levels (to be described in more detail presently), each of which has associated with it goals and move-generation procedures for attempting to achieve the goals. The principle for constructing the hierarchy is that a lower strategy is never attempted in refuting a higher one. On the other hand, a move that attains the goal of a higher strategy is a good refutation of a move aimed at attaining the goal of a lower strategy, since a higher strategy is globally more valuable in the sense that it is more essential to achieving the best game outcome. The way that the hierarchy is used to generate a search tree of moves and replies is that at the top level a player starts by trying to achieve his highest strategy. When that fails, he decreases his strategy level and tries to achieve a success at that level. The opponent, who moves at a lower depth, tries to refute the top strategy by first trying to achieve a strategy at the same level, and then when that fails, by trying moves at higher strategic levels. The search tree is generated as the players alternate in trying to refute plays at higher depths, until a position known to be a win or a draw occurs.

This chapter first presents KPKEG in detail, Section B, and describes several experiments that exhibit its behavior, Section C. Specific issues with respect to PSs are discussed in Section D. In Section E, KPKEG is compared to a similar program implemented in Lisp, with particular attention to the use of PSs to achieve the control structures of the Lisp version. Finally, we consider whether KPKEG can serve as a solid basis for further research, which is important because of the limited aims of the present work.
The objectives of either player in a chess game with only two kings and a pawn are limited. The player with only a king must achieve stalemate, capture the pawn, or block it from its promotion square, in order to obtain a draw. The player with the pawn must promote it safely while avoiding stalemate. To achieve these overall strategic objectives there are a number of lesser considerations, such as controlling the square in front of the pawn, forcing the enemy king in some direction, gaining the opposition (a chess term to be defined below), and advancing the pawn. These objectives have been formulated in KPKEG into seven levels, each assigned a corresponding numerical value.

7. Mate (White) or capture pawn (Black)
6. Queen the pawn or stalemate
5. Advance the pawn or occupy pawn's queening square
4. Control the pawn's path
3. Defend or attack the pawn
2. Restrict (force) the enemy king's move
1. Any other move (essentially away from pawn or enemy king)

The goal for the program then becomes to execute successfully a move at the highest possible strategy level. In this section we first illustrate how such a move is arrived at in a particular example, and then proceed to a more detailed discussion of KPKEG.

B.1. A simple example of program behavior

The position that we will examine is given in Figure B.1, and the complete program behavior trace is given in Appendix D. White starts out trying to achieve its highest strategy, which is to move the pawn onto its queening square; this strategy is at level 6 in KPKEG's hierarchy. The queening square is E8 (using the program's algebraic notation, which is indicated in the figure) and the pawn is at E6, so this fails immediately. White then decreases its level to 5, where the objective is to advance the pawn. Black's level-5 strategy is to intercept the pawn, preventing its advance. White advances the pawn to E7 and black responds C7-D7, at which position the black king is in control of the pawn and its queening square, and the white king is not within striking distance, so that White's advance-pawn strategy can make no further moves. Black's strategy succeeds because White fails to respond, and this success is a refutation of White's top-level move. White has no other ways to implement its level-5 strategy, so advancing the pawn directly is abandoned. White starts over at the initial position with strategy level decreased to 4, whose objective is to control with the king the path of the pawn's advance. Black's corresponding strategy is to occupy the pawn's path to its queening square (which is only slightly different from its level-5 strategy). White now moves its king E4-E5, Black responds C7-D8, White responds E5-D6, and black, D8-E8 (see Figure B.1).

Black's king is now on the pawn's queening square (E8) and it controls the square in front of the pawn (E7), so that White can go no further with its strategy to control the path to E8 from E6. Rather than giving up at this position, White increases its strategy level - Black has succeeded at level 4 but that may not be strong enough to refute some of the higher-level white strategies. White's attempt at level 5, moving the pawn from E6-E7, does in fact lead to a winning position for White, since the black king is forced to move off the queening square, whereupon the white king can move to control it. The way the program actually behaves is that Black's strategies fail to generate any moves (as did
B.1 The KPKEG Program in Detail

White's strategy at level 5 in the first segment of the trace) and E6-E7 thus succeeds. Black's move preceding E6-E7 fails, and the search proceeds by examining alternatives at that point.

This has described about two-thirds of the first column of the behavior trace in Appendix D (up to number 11), which is about one-fourth of the complete search that KPKEG does before deciding that White's E4-E5 is a satisfactory move. The primary characteristic of the program's search has been illustrated: it searches in a very restricted fashion according to a predetermined ordering of strategies, evaluating positions in the light of the strategic objectives currently in effect. That is, move attempts are generated only when they are deemed relevant to achieving success of a strategy, and the determination of what strategy is in effect depends on the strategy behind the previous move or on maximizing the outcome of the position at the top level. We now proceed to give more detail on KPKEG's internal structure.

B.2 An overview of the structure of KPKEG

The Ps of KPKEG are divided into six main groups: the strategy executive (Ps whose names start with S); Ps for updating the internal representation of the board (Q Ps); means for implementing move strategies (M Ps); strategies for White, or more generally the player who has the pawn (W Ps); strategies for Black (B Ps); and the initialization for example problems (X Ps). The strategy executive maintains information pertaining to the state of the tree search and the current strategy level. It also includes a set of Ps that recognize various patterns known to be wins or draws. The executive evokes the White and Black strategies, and uses the moves that they generate to carry the tree search forward. It uses the updating Ps to make the transition from one node in the tree to another. The strategy Ps generate move candidates directly or generate more abstract descriptions of what they intend, which are then converted to move candidates by the means Ps. We turn now to a more detailed look at the set of Ps in which most of the program control is embodied, the strategy executive. Incidental details of the other Ps are brought out, but fuller detail is postponed until the following subsection.

The VAPs (very abstract Ps) of Figure B.2 represent the main features of the strategy executive and of the other Ps as they appear to it. As the reader will recall from Chapter IV, underlining is used in VAPs to denote super-conditions and super-actions, which represent sets of condition or action elements, or the condensed result of many P

---

Figure B.1 Starting and intermediate positions for TEST1; White to move
% SE's: Strategy Executive VAPs; 55 actual Ps %

SE1: findmove –> initialize & select-strategy-move & check-strategy-result;
SE2: check-strategy-result & strategies-exhausted & not levels-exhausted
   –> change-strategy-level & select-strategy-move & check-strategy-result;
SE3: check-strategy-result & strategies-exhausted & levels-exhausted
   –> record-position & succeed-strategy-at-previous-depth;
SE5: check-terminal-position & not terminal-position & not maximum-depth
   –> select-strategy-move & check-strategy-result;
SE6: check-terminal-position & terminal-position-pattern –> terminal-win;
SE7: check-terminal-position & maximum-depth & not terminal-position
   –> static-eval-strategy;
SE8: terminal-win(self) OR succeed-strategy –> refute-strategy-at-previous-depth;
SE9: terminal-win(opponent) –> succeed-strategy-at-previous-depth;
SE10: check-move-result & refuted
SE11: check-move-result & succeed & not depth=1
    –> retract-move & refute-strategy-at-previous-depth;
SE12: check-move-result & succeed & depth=1 –> make-actual-play;
SE13: record-position & position-before-making-successful-move
    –> build-P-to-recognize-as-terminal-position
    & build-P-to-recommend-trying-move-if-position-recurs-at
    & greater-depth-or-at-depth-1;

% UB's: Updating Board for moves; 19 Ps %
UB1: make-move & move-type & location's & controls's –> location's & controls's;
UB2: retract-move & move-type & location's & controls's –> location's & controls's;

% MMC's: Means to Move Candidates; 18 Ps %
MMC1: means-signal & properties-relevant-to-desired-moves –> move-candidate's;

% WBS's: White and Black Strategies; 44 Ps %
WBS1: select-strategy-move & board-pattern –> means-signal's OR move-candidate's;
WBS2: select-strategy-move & board-pattern –> succeed-strategy;
WBS3: static-eval-strategy & board-pattern –> terminal-win;

% TX's: Test Examples; 5 Ps for 3 tests %
TX1: test-signal –> initialize & controls's & location's;

Figure B.2 VAPs for KPKEG

firings. Elements of VAPs that are not underlined correspond to actual program elements, and behave similarly with respect to the way Psnist considers events to be ordered.

Using the VAPs we can follow the example in Section B.1 in enough detail to see the way the program works. At the beginning of TEST1, the user asserts a signal that fires the equivalent of TX1, which sets up the board situation. Then another user signal, "findmove", fires SE1 which initializes the strategy executive and starts the search process.
by asserting "select-strategy-move". As discussed above White starts out trying to achieve its highest-level strategy; the level of strategy being sought is set by the initialization in SE1. The VAPs that respond to "select-strategy-move" are the WBS's, which generate move candidates or recognize success based on board patterns. In the present case none of the WBS's fires, since the level 6 strategy for White is to move its pawn onto the eighth rank. Nothing responds to "select-strategy-move", so that the "check-strategy-result" signal from SE1 is examined, according to the conditions in SE2 and SE3. The situation is that the strategies at level 6 are exhausted but that the other levels haven't been tried yet so that SE2 is true, causing the level to be decremented to 5 and again asserting the "select-strategy-move" signal.

This time the strategy is to advance the pawn, and a move-candidate (E6-E7) is asserted by an instance of WBS1. SE4 represents the selection of a move-candidate from a set of them. UB1 responds to the "make-move" signal from SE4, updating the board according to the nature of the move (i.e., whether a king or pawn is moving, and the direction of the move). The "check-terminal-position" from SE4 evokes the testing of the patterns represented by SE5, SE6, and SE7; those are patterns for the small number of known won or drawn positions for K-P-K endgames. In the present example, SE5 is appropriate, and sets up the strategy selection for Black, who must respond to White's pawn advance. The program goes through the sequence represented by WBS1, SE4, UB1, and SE5. Black has moved its king to D7, and White's advance-pawn and queen-pawn strategies (instances of WBS1) recognize that further moves are no good, so that SE2 fires, and then SE3 becomes true when no response to "select-strategy-move" is made. Notice that only strategies not less than level 5 are considered by White, in accord with the strategy hierarchy principle - trying a weaker strategy to refute a stronger one makes no sense.

SE3 first causes the position before Black's move to D7 to be recorded as a known success (via SE13). Then the strategy at the previous depth, namely the one that proposed the move to D7, is made to succeed. The success is noted by SE11, which uses the "check-move-result" signal asserted by SE4 when the move was selected. SE11 takes back the successful move, evoking UB2 to restore the board, and signals that the move at the previous depth is refuted. SE10 responds to the "refuted" signal, using the "check-move-result" that was asserted when E6-E7 (advancing the pawn) was selected by SE4. Generally, after a move is retracted by SE10 via UB2, other move candidates are tried (SE4), other strategies at the same level are tried (imagine a "select-strategy-move" re-asserted by the super-action of SE10), or SE2 and SE3 take effect. In our example, White abandons the attempt to advance the pawn, its level is decreased to 4, and the search continues in a similar fashion.

We now touch on a few points about the VAPs in Figure B.2 that were not brought out by the above. The treatment of terminal patterns recognized by instances of SE6 is according to one of the two procedures represented by SE8 and SE9. Recall that "check-terminal-position" is examined immediately when a new position is created in the search, so that the "succeed" or "refuted" signal to the previous strategy will occur before any other strategies are attempted at that new position. The terminal positions recognized by SE6 are general, as opposed to successes of particular strategies as represented by WBS2 - the result in either case is similar, though. A different kind of terminal position leading to the "terminal-win" signal in some cases is the maximum depth condition. Presently the
maximum depth is 9, and when the search is 9 plies deep, SE7 fires (if the position is not
terminal in any other sense), asserting "static-eva-strategy". If board conditions are
right, in a rather optimistic evaluation, an instance of WBS3 fires; otherwise nothing
further is done and the strategy at the previous depth succeeds for lack of refutation.
The maximum depth cutoff is intended to be used only rarely, since the domain is rich in
specific knowledge, so the present mechanism is only a stopgap, even though it is
successful in the experiments described below. Finally, the MMC1 VAP represents a set of
Ps that are evoked by WBS's to generate moves of desired classes, for instance moving
toward a square. These means to generating move candidates are used whenever the
desired move candidates cannot be easily constructed directly by the WBS's.

B.3. Full detail on selected aspects of KPKEG

As we have seen in the preceding subsection, KPKEG's six groups of Ps form the
following functional units: the strategy executive, the board-updating operations, the
means to strategies, the strategies themselves (two groups), and initialization. This
subsection will indicate subdivisions of each of these groups, except the last. In most
cases, typical Ps will be given to illustrate how certain kinds of chess knowledge are
represented. Descriptions of all of the chess knowledge in KPKEG will be included. For
the S Ps, we will give more detailed, abstract Ps which bring out issues of control. The
listing for the actual program is in Appendix A, and a cross-reference is in Appendix B.
Section B.4 is essential for decoding the actual Ps.

There are 55 S Ps, subdivided functionally into 9 groups as follows:

- **S0-S1**: initialization; 2 Ps. [SE1]
- **S3-S4, S15-S18**: evocation of strategies, change of strategy levels; 6 Ps.
  [SE2-SE3]
- **S5-S9**: tree mechanics, ascending and descending in search tree; 8 Ps.
  [parts of SE4, SE10, SE11]
- **S21-S210**: selection of a move from the set of candidates; 8 Ps. [SE4]
- **S11-S13, S23-S26N**: checking the results of an attempted move; 7 Ps.
  [SE10-SE12]
- **S30’s, PN’s, PW’s, PV’s (created by S60’s)**: checking for terminal
  positions; 13 or more Ps. [SE5-SE7]
- **S40’s**: controlling actions for terminal positions; 3 Ps [SE8-SE9]
- **S50’s**: printing the board externally; 3 Ps. []
- **S60’s**: recording the winning (terminal) positions as Ps; 9 Ps. [SE13]

The basic control in the executive corresponds to the VAPs SE2-SE4, SE8-SE12, and
the RHS of SE5 (i.e., the second through fifth and the seventh group of S Ps). Figure B.3
gives abstract Ps (APs) that elaborate on those VAPs. Each AP has the VAPs and actual Ps
to which it corresponds. Using the APs, we can get a more detailed picture of the control
flow. The process of finding a move starts when the initialization asserts "select-strategy &
check-other-strategy" (the latter signal is synonymous with "check-strategy-result" in
the VAPs). If a strategy produces move-candidates, S2a will select one by using first a
"max" metric, which takes the distance between two squares to be the maximum of the

---

Square brackets enclose the names of the corresponding VAPs from Figure B.2.
The KPKEG Program in Detail

S0a: [SE2, SE10; S3] check-other-strategy & depth & not select-strategy & not succeed
  & not move-candidate's & not refuted
  -> select-strategy & check-other-strategy;

S0b: [SE3; S4] check-other-strategy & depth & select-strategy-unresponded-to
  -> change-level;

S0c: [SE4; S5-S6] descend(move) & depth & (current-level OR level-from-preceding-depth)
  & current-mover
  -> make-move & check-terminal-position & erase-check-terminal-position
  & increase-depth & establish-level-at-new-depth & mover-is-other-player;

S0d: [SE10, SE11; S7-S9] ascend(move) & depth & current-level & current-mover
  -> erase-strategy-tried's & retract-move & restore-captured-pieces
  & decrease-depth & mover-is-other-player
  & erase-strategy-signals-from-depth-being-ascended-from;

S1a: [SE11, SE12; S11-S13] succeed(move,depth)
  -> ascend(move) & refuted(previous depth) OR make-the-move-if-depth-1;

S1b: [SE2, SE3; S15-S18] change-level & depth & current-level
  -> select-strategy & check-other-strategy
  & current-level(decreased if depth = 1 OR increased if depth > 1)
  OR depth (in case all levels have been tried);

S2a: [SE4; S21] move-candidate & depth & not check-move-result
  & not move-offboard & not move-onto-piece-of-own-color
  & not move-candidate-whose-destination-square-is-closer-to-pawn's-queening-square-by-max-metric-or-same-by-max-metric-and-closer-by-min-metric
  & not move-candidate-equal-by-previous-test-and-with-destination-square-lexically-less-or-destination-square-lexically-less
  -> descend & check-move-result;

S2b: [SE3, SE11; S23, S26-S27] check-move-result & not refuted & depth(one deeper)
  & not move-candidate-at-depth-one-deeper
  & not other-strategies-to-be-checked-at-depth-one-deeper
  -> erase-move-candidates & record-win & succeed;

S2c: [SE8, SE11; S24] succeed-strategy -> refuted(previous depth);

S2d: [SE10; S25] check-move-result(depth) & refuted & depth(one deeper)
  -> ascend(refuted move);

S3a: [SE5; S38] erase-check-terminal-position -> select-strategy & check-other-strategy;

S4a: [SE8; S41] terminal-win(for mover) & depth
  -> refuted(previous depth) & not erase-check-terminal-position;

S4b: [SE9; S42-S43] terminal-win(for opponent) & depth
  -> check-other-strategy & all-levels-have-been-tried;

Figure B.3 APs for control in the executive

absolute values of the differences between their corresponding numerical coordinates; for equals by the "max" metric, S2a applies a "min" metric, which is similar except that the minimum is taken into account; when there are still contending candidates after those tests, lexical order is used.
The KPKEG Program in Detail

SOc then carries out the bookkeeping involved in descending a ply, and evokes the Q Ps via "make-move" to update the board for the move selected by S2a. In descending, the usual action is that the mover at the new ply inherits the strategy level from the preceding move that it makes in the current search variation, that is, from two plies back. The level from one ply back is used in going from depth 1 to depth 2. This inheritance of levels injects some continuity into the search, since a player first tries to continue what he was trying on his preceding move. After the board is updated for the selected move, control returns from the Q's to examine the "check-terminal-position" signal asserted by SOc. Terminal positions are recognized by a set of Ps not shown (discussed below), and if nothing is recognized, S3a fires and the strategy selection is started at the new depth, as before.

There are three ways for the descent in the search tree to stop: the recognition of a terminal position, the recognition of the success of a strategy, and the exhaustion of all possibilities, which is a failure of a strategy. Terminal positions (including maximum search depth, which is terminal in a weak sense only) are checked in response to the "check-terminal-position" signal, asserted only when a new position is first entered from a lesser depth (closer to the root of the tree) - not when a position is re-instated from a greater depth (descendent node). If a terminal position or explicit success occurs, "terminal-win" is asserted and processed by S4a and S4b. S4a specifically refutes the strategy at the previous depth; the "refuted" signal is processed by S2d. S4b sets up an exhaustion condition so that S2b will get control, resulting in a success at the previous depth. S2b recognizes a failure of one strategy, implying the success of another, by noting that a move has not been refuted by the strategy at the descendent node (that strategy has tried all its possibilities with no success). The implied success is signalled by "succeed", which is picked up by S1a.

SOd carries out the bookkeeping of the actual ascent to the parent node in the search tree, evoking Q's with "retract-move" to update the board. After the ascent, control falls back to one of two places: to S2d if "refuted" is present (from S21a), which continues to propagate results back one more ply; or to S2a or SOa if there was a success at the descendant node that refuted the move made at the present depth (S2d). S2a selects from any move-candidates that are still available, but if none are there, SOa fires and strategy selection is evoked again.

Strategy selection is driven by SOa and SOb. SOa evokes a strategy (to generate move-candidates) via "select-strategy" and at the same time asserts a signal to which SOa or SOb respond. A strategy consumes the "select-strategy" signal and also asserts a "strategy-tried" signal (not shown except that SOd erases all such during ascent) so that no duplication can occur. Some strategies do respond with move-candidates in several sets, iterating through SOa, but when no further response is possible, SOb fires and the strategy level is changed, via S1b. Levels are changed in two ways depending on depth: at depth 1, which is the depth of the player trying to make an actual external move, the level starts out at the maximum (highest aspiration) and decreases when things don't work; at other depths, the level starts out at the level inherited from the ancestral (parent) node as explained above (SOc) and increases up to the maximum (in accord with the strategy hierarchy principle). When the maximum is reached, the action represented by the second half of S1b's RHS (after the "OR") is taken, and the "depth" signal is picked up by S2b. When the minimum is reached (depth 1 only), the program has failed to find a move to make.
To summarize the discussion so far, there are two aspects of the strategy executive: chess and PS control. There are several points with regard to the former. The executive does a fairly standard tree search, but it uses success or failure of strategies to evaluate positions rather than a more conventional material criterion. Strategy levels are inherited from parent or ancestral nodes, so that some unity of play over various depths in the tree is evident. Strategy levels start high and decrease at depth one, and start at the inherited value and increase at the other depths. Recognition of terminal positions occurs when a position is examined for the first time.

PS control is primarily of a fall-back nature: a move is made, for instance, and a Working Memory instance records it; when it has been processed, control falls back to examining that record and proceeding accordingly. In addition to checking move results, this occurs when strategy levels are exhausted ("depth" from S1b), during ascent ("depth" from S0d - note that S0a and S2a have explicit exclusion conditions to determine the appropriate action), when strategies are tried ("check-other-strategy"), and when terminal positions are recognized ("erase-check-terminal-position", S3a). Another kind of control is used for generating move candidates and for selecting strategies: When a strategy fires, it asserts a signal that inhibits future firings in the same context. Move-candidates exist as a set in Working Memory, so that when S2a is examined, a new one from the set is found (and erased). When there are no more candidates, control falls back to S0a and S0b.

Continuing now with more details of KPKEG, the Ps corresponding to VAP SE6 (Figure B.2), the S30's, encode conditions for recognizing ten terminal positions as follows:

a. A pawn on the eighth rank that cannot be captured by the enemy king; conditions i. and j. below are excluded; this is defined to be a win for White. (S31)
b. No pawn on the board (it has been captured); this is a draw (which is considered a failure for White). (S32)
c. The black king stalemate. (S33)
d. Checkmate. (S34)
e. The black king with the opposition and the white king not directly in front of the pawn; condition h. is an exception. (S35)
f. The white king on the same file as the pawn, two or more squares in front of it, and the black king not closer to the pawn than the white king. (S36)
g. The white king on the square in front of the pawn, with the opposition (to be defined below). (S360)
h. The white king on the sixth rank, in front of the pawn somewhere and fairly close, and the black king not closer to the pawn. (S36R)
i. A special stalemate condition with the pawn just promoted at C8, black king at A7, and white king controlling B6. (S37L)
j. Similar to i., but reflected to the right side of the board (F8, H7, G6). (S37R)

These may not be correct or powerful enough from the chess standpoint (see Fine, 1941) but they suffice for present first-approximation purposes. "Opposition" is an endgame term that is defined narrowly as a situation in which the kings are on the same file with one intervening square; the player not on the move has the opposition. This set of Ps is augmented by specific patterns added as Ps, which recognize specific board situations that have been determined during search to have a known eventual result.
An example of how one of these conditions is expressed is in Figure B.4. Refer to Section B.4 for predicate meanings.

- "WK FRONT2": CHECK:TERM(D,P) & NOT SATISFIES(D,D EQ 1) & KP:HAS(C1)
- bind the kings and the pawn to variables:
  & ISKING(A1) & HASCOLOR(A1,C1) & ISKING(A2) & VNEQ(A2,A1) & ISPAWN(A3)
- establish rank and file for locations of white king and pawn; both on same file:
  & LOC(A3,S1) & RF(S1,R1,F1) & LOC(A1,S2) & RF(S2,R2,F1)
- white king two or more in front of pawn:
  & SATISFIES2(R1,R2,R2 & GREAT R1+1)
- location, rank, and file of black king:
  & LOC(A2,S3) & RF(S3,R3,F3)
- black king not closer to the pawn than the white king:
  & NOT SATISFIES3(R2,R3,F3,MAX(ABS(R3,R1),ABS(F3,F1)) & LESS R2-R1)

\[ \Rightarrow \text{TERM-WIN(C1,1,36) & NEGATE(C1) & NOT ERSCHECKTERM(D,P);} \]

Figure B.4 Implementation of terminal position f.

The Ps corresponding to the UB (updating board) VAPs (Figure B.2), the Q's, are grouped as follows:
- Q0-Q0c: print a move trace; 2 Ps.
- Q1-Q2: initiate move retractions; 2 Ps.
- Q3-Q4: move pawn forward and backward; 2 Ps.
- Q7: detect illegal king move, i.e., into check; 1 P.
- Q8-Q9: bookkeeping for any capture move; 3 Ps.
- Q10-Q19: king moves; 9 Ps.

A move is given as an origin square and a destination square. The Q's print a trace, detect the type of piece to be moved, determine the direction of the move, change the location of the piece, detect captures, save information about a captured piece so that it can be restored, and update the squares controlled by a piece as it moves.

Of the two types of moves in KPKEG, the king move is more typical of the majority of chess moves than is the pawn move. The actual move is done in two steps (P firings), one to do the part common to all directions for the move and the other (one of 8 Ps) to do direction-specific updating. The split into two is largely for reasons of economy of expression. All eight directions are distinct because of the board representation, which uses a different predicate to show the relation of a square to each adjacent square. Figure B.5 gives the common P and one of the directional Ps.

The M Ps, corresponding to VAP MMC1, are divided into five groups:
- M1-M8: generate move candidates to move toward a square; 8 Ps.
- M9-M14: special cases for moving toward; 3 Ps.
- M11-M14: handle the delayed assertion of move-toward candidates; 5 Ps.
- M16: generate candidate to move to a square; 1 P.
- M17: special case for moving to; 1 P.

The means to move candidates for strategies are quite important to reducing the number of necessary strategy Ps, since for moving in eight directions, different sets of move candidates are appropriate. There are three move candidates in the set for moving toward one square from another: one to a square approximately in the same direction as the target, and two that are adjacent to the first. Figure B.6 gives a typical means P.
The KPKEG Program in Detail

B.3

Q11: "K COMMON" :: MAKE-MOVE(S1,S2) & LOCA(S1,S1) & ISKING(S1) & NOT OFFBOARD(S2)
test that move isn't onto a controlled square:
  & NOT EXISTS(C1,C2,AS2) & CONTROLS(A,S2) & HASCOLOR(A,C1)
  & HASCOLOR(A2,C2) & VNEQ(C1,C2) & NOT RETRACTING(S1,S2) )
check that the move isn't onto a piece of the same color:
  & NOT EXISTS(A2,C) & LOC(A2,S2) & HASCOLOR(A1,C) & HASCOLOR(A2,C) )
make sure that the square is reachable:
  & CONTROLS(A1,S2)
signal for capture check and direction-specific components:
  => CHECK-CAP(A1,S2) & MAKE-MOVE-K(A1,S1,S2)
and do the updating common to all king moves:
  & LOC(A1,S2) & CONTROLS(A1,S1) & NEGATE(1,2,7) & NOT RETRACTING(S1,S2);
moves the king diagonally left-forward:
Q16: "K DIALGF" :: MAKE-MOVE-K(A1,S1,S2) & DIALGF(S1,S2)
establish the squares whose control will change:
  & DIAFR(S1,S3) & CONTROLS(A1,S3) & DIAFR(S1,S4)
  & CONTROLS(A1,S4) & RANKR(S1,S5) & CONTROLS(A1,S5) & FILEF(S1,S6)
  & CONTROLS(A1,S6) & DIALGF(S1,S7) & CONTROLS(A1,S7)
  & DIAFR(S2,S8) & DIALGF(S2,S9) & DIALGF(S2,S10) & FILEF(S2,S11)
  & RANKL(S2,S12)
make the changes (2 controlled squares stay controlled):
  => CONTROLS(A1,S8) & CONTROLS(A1,S9) & CONTROLS(A1,S10) & CONTROLS(A1,S11)
  & CONTROLS(A1,S12) & NEGATE(1,4,6,8,10,12);

Figure B.5 Updating the board for a king move

M1: "MOVE TW DBR" :: MOVE-TOWARD(D0,A,S2) & NOT CONTROLS(A,S2) & LOCA(S1)
determine that the direction is diagonally right-backward, using rank and file coordinates:
  & RF(S1,R1,F1) & RF(S2,R2,F2) & SATISFIES(F1,F2,F1 LESS F2) & SATISFIES(R1,R2,R1 GREAT R2)
locate the appropriate three squares and set up the moves:
  & RANKR(S1,S3) & FILEF(S1,S4) & DIAFR(S1,S5)
  => MOVE-HOLD(D0,S1,S3) & MOVE-HOLD(D0,S1,S4) & MOVE-HOLD(D0,S1,S5) & NEGATE(1);

Figure B.6 Means for moving toward a square

The bulk of the chess knowledge in KPKEG is in the strategy Ps, the W's and B's, corresponding to the WBS VAPs in Figure B.2. As indicated in the VAPs, the knowledge is represented three ways: one for move-candidate generation, one for recognizing immediate success, and one for making a maximum-depth static evaluation. Since the three are somewhat similar, we consider details for the first only, in Figure B.7. Again, as for the terminal-position chess knowledge, no claims are made for correctness of these strategies in general. But they are adequate as a first approximation, and from the present limited success, we conclude that Ps are adequate for encoding whatever the correct knowledge is. The relation between the last two columns in Figure B.7 is that at the same level the strategies are (intended to be) opposites: success of one refutes the other. The levels are (intended to be) such that success of a strategy at a higher level refutes a move from a lower level, but not vice versa. A typical strategy P is given in Figure B.8.
KPKEG

The KPKEG Program in Detail

<table>
<thead>
<tr>
<th>Level</th>
<th>P group</th>
<th>White</th>
<th>Black</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>B1</td>
<td>Checkmate (impossible in K-P-K)</td>
<td>Capture pawn</td>
</tr>
<tr>
<td>6</td>
<td>W2, B2</td>
<td>Queen the pawn, move to 8th rank</td>
<td>Stalemate</td>
</tr>
<tr>
<td>5</td>
<td>W3, B3</td>
<td>Advance pawn, move king off square in front of pawn</td>
<td>Intercept pawn by moving toward pawn’s queening square</td>
</tr>
<tr>
<td>4</td>
<td>W4, B4</td>
<td>Control path of pawn by moving king toward the square two in front of the pawn</td>
<td>Block pawn by moving toward any square in the pawn’s path</td>
</tr>
<tr>
<td>3</td>
<td>W5, B5</td>
<td>Defend the pawn by moving toward it</td>
<td>Attack the pawn by moving toward it</td>
</tr>
<tr>
<td>2</td>
<td>W6, B6</td>
<td>Move toward the enemy king, to restrict its movement; always fails at depth 2; try to gain the opposition</td>
<td>Same as for White</td>
</tr>
<tr>
<td>1</td>
<td>both W7</td>
<td>Any move not toward the enemy king and not toward the pawn; always fails at depth 2</td>
<td>Same as for White</td>
</tr>
</tbody>
</table>

Figure B.7 Summary of chess knowledge in the strategy P's

W4, "CONTR P" = SELECT STRAT(DP) & KP:HASP(P) & CUR:LEVEL(DL) & SATISFIES(LL EQ 4) & NOT( EXISTS(X) & STRAT:TRIED(X,LD) & SATISFIES(XC EQ W4) )
bind pawn and white king:
  & ISPAWN(A1) & ISKING(A2) & HASCOLOR(A1,C) & HASCOLOR(A2,C)
find the square two in front of the pawn:
  & LOC(A1,S1) & FILEF(S1,S2) & FILEF(S2,S3) & NOT CONTROLS(A2,S3)
evoke means and indicate the strategy has been tried:
  => MOVE: TOWARD(D,A2,S3) & STRAT:TRIED(W4,LD) & NEGATE(1).

Figure B.8 A typical strategy P

B.4. Meanings for KPKEG predicates

Two sets of KPKEG predicates are central to the program and to the representation of the game, and are given here to provide an index into the following alphabetical list:


Board representation: CONTROLS, DIAGLB, DIAGLF, DIAGRB, DIAGRF, FILEB, FILEF, LOC, OFFBOARD, RANKL, RANKR, RF.
The following are the types for the arguments of predicates in the description below:

- **a**: actor, i.e., particular piece
- **c**: color
- **d**: depth
- **f**: file
- **g**: player
- **i**: level (of strategy)
- **p**: rank
- **q**: square
- **r**: signal that a move is ready
- **S**: strategy
- **W**: potential moves
- **X**: pawn
- **Y**: king
- **Z**: board

- **ASCEND(a1,a2)**: second to a lower ply by retracting the move from a1 to a2. (S)
- **CAPTURED(g,s,d)**: at d, g was captured and removed from s. (S, Q)
- **CHANGELEVEL(d)**: change the strategy level at d. (S)
- **CHECKCAP(p,d)**: check if there are any captures by p moving onto s. (Q)
- **CHECKMOVE:RESULT(d,s1,a2)**: check the result of the move made from s1 to a2 at d. (S)
- **CHECKOTHERSTRAT(d,p)**: check for other strategies for p at d, after at least one strategy has been tried. (S)
- **CHECKTERM(d,p)**: check if the current position (at d) is a terminal one; p2 is to move. (S, PN)
- **CONTROLLED(g,a1,a2)**: g controlled a (see CONTROLS) before it was captured in the search at d. (S, Q)
- **CONTROLKL(p)**: set up the CONTROLS instances for king p. (X)
- **CONTROLLP(p)**: set up the CONTROLS instances for pawn p. (X)
- **CURLEVEL(d)**: i is the current strategy level at d. (S, Q, W, B, PN)
- **DEPTH(d)**: d is the current search depth. (S, Q)
- **DESCEND(a1,a2)**: move one ply deeper by moving a1 to a2. (S)
- **DIAGLB(a1,s1,**s2): a2 is diagonally left and forward from a1. (Q, M, W, X)
- **DIAGLB(a1,s1,**s2): a2 is diagonally right and back from a1. (Q, M, W, X)
- **DIAGRF(a1,s1,**s2): a2 is diagonally left and forward from a1. (Q, M, X)
- **DIAGRF(a1,s1,**s2): a2 is diagonally right and forward from a1. (Q, M, X)
- **ERS:CHECKTERM(d,p)**: erase the corresponding CHECKTERM; this signals completion of the check. (S, PN)
- **ERS:MOVES(d)**: erase examined MOVECAND's at d. (S)
- **ERS:STRAT:TRIED(d)**: erase STRAT:TRIED's at d. (S)
- **FILEB(a1,a2)**: a2 is directly back along the file of a1. (all but PN)
- **FILEF(a1,a2)**: a2 is directly forward along the file of a1. (all but PN)
- **FINOMOVE(p)**: find a move for p; typed by user. (S)
- **HASCOLOR(c)**: c has color c (B or W). (all but M)
- **ISKING(p)**: p is a king. (all but M)
- **ISPAWN(p)**: p is a pawn. (all but M)
- **KP:HASPP** (p): this is a K-P-K game; p has the pawn. (S, W, B, PN)
- **KP:KINIT(p)**: initialize for a K-P-K game; x is a dummy. (S, X)
- **LASTPN(p)**: production x is the last one added to the position-net module. (S)
- **LOC(p)**: x is located on x. (all)
- **MAKEMOVE(a1,a2)**: make the move from a1 to a2. (Q)
- **MAKEMOVE:K(p,a1,a2)**: update the board (CONTROLS) for the king move of p from a1 to a2. (Q)
- **MAKEMOVE:T(a1,a2)**: print the trace message for the move from a1 to a2, then signal MAKEMOVE. (Q, S)
- **MAXDEPTH(d)**: d is the maximum depth for the search. (S)
- **MAXLEVEL(p,i)**: is the maximum strategy level for p. (S)
- **MEANS:EXAM(d)**: signal that MOVECAND's are not to be generated by a means (M Pa) at d, but rather the potential moves are to be held for examination (MOVE:EXAM). (Q,W)
- **MEANS:HOHD(d)**: hold the emission of MOVECAND's at d from a means (M Pa) until all possibilities are ready. (M, W, B)
- **MEANS:REL(d)**: release the moves held back by MEANS:HOHD at d. (M)
- **MINLEVEL(p,i)**: i is the minimum strategy level for p. (S)
- **MOVECAND(a1,a2)**: the move from a1 to a2 is a candidate at d. (S, M, W, B)

---

* The initials appearing at this place refer to P groups to which a predicate is relevant.
* PN stands for Pa in the position net generated by the RECORD:BLO process.
MOVE-EXAM(d, s1, s2) the move from s1 to s2 is ready for examination by a strategy (see MEANS-EXAM). (W, M)
MOVE:HIST(x) x is a list of the moves made in descending to the current depth, used for external display only. (S)
MOVE:HOLO(d, s1, s2) s1 to s2 is a potential MOVE-CARD at d, generated by a means. (M)
MOVE:TO(d, s) generate moves to get g to s at d. (O4, W, B)
MOVE:TOWARD(d, s) generate moves toward s from g's present location. (O4, W, B)
MOVE:PR(p) p is the color to move in the current position. (B)
RECORD(p, S1, S2) for external display, p is moving g from s1 to s2 as a real game move. (S)
NODE-COUNT(x) x is a count of the number of nodes searched, for external display. (Q, S)
OFFBOARO(s) s is off the board; it exists as a dummy location to simplify the board patterns. (S, Q, M)
PLAYER(p) p is a player, either B or W. (S)
PRINT:BOARD(x) print the board externally; x is a dummy. (S)
PRINTED:BOARD(x) the board has been printed; x is a dummy; this is used to prevent the board display twice with no intervening changes. (S, Q)
RANK(d, s1, s2) s2 is directly to the left of s1, same rank. (Q, M, W, X)
RANKR(d, s1, s2) s2 is directly to the right of s1, same rank. (Q, M, W, X)
RECORD-BLD(d, s1, s2, x) record the key move toward a terminal position, which recognizes that s1 to s2 is the key move; d and i are the depth and level at which the importance of the position was determined and x is a list that is the common part of the LHS of the set of Ps. (S)
RECORD:DONE(d, x) the P whose tag (P1, PV, or PW) is x has been recorded at d in the RECORD-BLD process; this prevents duplication. (S)
RECORD:FIN(d) the main part of the RECORD-BLD process is finished at d. (S)
RECORD:FIN2(d) finish the RECORD-BLD process by erasing various intermediate data. (S)
RECORD:PRE(d, s1, s2, s3, s4, s5, p) at d and i, s1 to s2 is the key move leading to a terminal position (see RECORD-BLD); p is to move, and s3, s4, and s5 give the positions of the pawns, white king, and black king. (S)
RECORD:WIN(d, s1, s2, s) record the terminal position, see RECORD-BLD, at d, key move s1 to s2. (S)
REFUSED(d) the strategy at d is refused, at least with respect to a particular move (CHECK-MOVE-RESULT). (S, Q)
RESTORE:CAP(d) restore the captured piece removed by a capture move (CAPTURED), at d. (S)
RESTORE:CON(d) restore the CONTROLS removed by a capture move (CONTROLLED); g was captured at d. (S)
RETRACT:HOLO(d, s1, s2) hold the retraction (RETRACT:MOVE) of the move s1 to s2, since it was never made due to illegality. (Q)
RETRACT:MOVE(s1, s2) retract the move from s1 to s2, restoring the board state to its previous condition; the reverse of MAKE-MOVE. (Q, S)
RETRACTING(s1, s2) s1 to s2 is being retracted; this suppresses certain legality checks for king moves. (Q)
RFD(x, f) x has rank r and file f, both numbers. (S, M, W, B)
SAVE:STATIC(d, p) save the CONTROLS of p at d, as CONTROLLED. (Q)
SELECT:STATIC(d, p) at d, do a static strategy estimation for p. (W, B, S)
SELECT:STRAT(d, p) at d, do a (dynamic) strategy selection, which generates move candidates, for p. (W, B, S)
STATIC:EVAl(d, p) signal that STATIC:EVAl is appropriate in the current position; this affects the direction of processing after CHECK:TERM. (S)
STRAT:TRIED(d, x) strategy x (the name of a P) has been tried at d and L (S, W, B)
SUCC:STRAT(d, p, x) strategy x (the name of a P) has succeeded for p at d and L, the success is known statically, without further search. (S, W, B)
SUCCEEDED(d, s1, s2) at d, the move s1 to s2 has succeeded, in the strategic sense. (S)
TERM:WIN(p, d) p has a terminal win (for White, a check win, for Black, a draw), indicated by P x; at maximum depth this evaluation is static and not as strictly a win (see SELECT:STATIC). (S, W, B, PN)
TESTkn(x) initiate the test problem n = 1, 2, 3, typed externally by the user. (Q)
TRACING(s) a dummy predicate used to show the printing of an external trace. (S, Q)
The KPKEG Program in Detail

\[ \text{WBW:RES:EXAM(d,g)} \] examine the results of the means evoked by \( P \) \( WBG \) (moving \( g \) at \( d \)) using \( \text{MEANS:EXAM} \). \( WBW \) is a static estimator (\text{SELECT:STATIC}) that uses simply the existence of one of a class of moves. (W)

\[ \text{W7:RES:ERS(d)} \] erase the results of the \( W7:RES:EXAM \) process. (W)

\[ \text{W7:RES:EXAM(d,g)} \] at \( d \), examine the results of the means evoked by \( P \) \( W7 \) using \( \text{MEANS:EXAM} \). \( W7 \) desires moves that are not generated by the means. (W)

\[ \text{W7W:RES:EXAM(d)} \] similar to \( WBW:RES:EXAM \). (W)

\[ \text{WINCAND(d,s1,s2)} \] at \( d \), \( s1 \) to \( s2 \) is a candidate that has led to a win in an identical situation at a different depth; see \text{RECORD:BLD}. (S, PN)
C. Results of Experiments with KPKEG

KPKEG has been tested on three simple problems, called Test1, Test2, and Test3. These are not intended to be representative of the class of all K-P-K positions, but KPKEG's behavior does demonstrate that it is an adequate basis for a more complete program. Test1 is discussed in detail in Section B, and is exhibited in Figure B.1. Appendix D examines KPKEG's behavior on Test1 in detail, exhibiting: the program trace, showing search behavior in the tree of chess moves; the state of Working Memory after the run, which includes the internal board representation; the trace of P firings corresponding to the program trace, broken into distinguishable corresponding segments; and a control flow summary trace, which breaks P firings into groups. Appendix E contains four more program traces, two for some experimental options on Test1, and one each for Test2 and Test3.

Test1 is a good test because it requires more searching than the typical K-P-K position. This searching exercises KPKEG's executive Ps and results in the evaluation of a variety of terminal positions. It also allows meaningful comparison of effects of various options on the search. KPKEG's behavior on Test1 has been described in some detail in Section B.1. The traces on Test1 in Appendix E are of primary interest here. Two search options explored with Test1: (1) The procedure of decrementing strategy levels from the maximum at depth 1, but passing down strategy levels to other depths, and incrementing from those to the maximum. (2) The storing of winning positions for future use in the search. The standard version of KPKEG, with both of these options turned on, finds a move for Test1 by searching 40 nodes. A version with the strategy level changed to decrements at all levels searches 80 nodes (the first trace in Appendix E), and a version without the position storing searches 60 nodes (the second trace in Appendix E). The combination with both options in their non-standard setting was not tried. This is good evidence, at least as far as a single test position can provide, that the standard version has the proper options.

The most significant change in KPKEG's behavior on Test1 results from an experiment not shown: if the carrying down of strategy levels from two plies back is not done (Ps S5 and S6 become S5 modified to work at all depths so that the level from one ply back is used), the search goes on for hundreds of nodes and fails to find a satisfactory move for White. One critical point is the situation at node 35 (please refer to Appendix D), where Black is at level 5 but White responds at level 4, as in the sequence leading to node 23 (which happens to be caught by the position net, PN-5); in the alternate version, White is forced to be at level 5, and only tries to advance the pawn, failing to refute Black's move and eventually failing at depth 1 with the move E4-E5. This demonstrates that the alternative is detrimental to the evaluative effectiveness of the program.

Test2 and Test3 (Figure C.1) are rather similar as starting positions, but have some interesting differences in their search. Their shared traits are more important than their differences. Both tests show the application of the kind of specific knowledge that is typically applied in K-P-K positions. In particular, White searches very few nodes, four or less, in finding a winning move. Black, on the other hand, searches many more in its futile attempts (it would probably be more reasonable for the program to resign in situations
whose value is known). For illustrative purposes, the search is useful because it exercises some more of the terminal-position recognizers, and makes use of strategy Ps (W's and B's) for the lower levels (Test 1 only got as low as level 4).
D. Production-System-Related Features of the Implementation

KPKEG's organization makes programming by gradually adding Ps easy. There is a clear division into the strategy executive, the strategies, the means, the terminal patterns, and the board-updating process. KPKEG was built up by leaving strategies and terminal patterns unspecified until the executive was in good shape. The action of the unspecified parts was easily filled in by manual intervention at pre-arranged points. The executive developed from an initial approximation by adding Ps to represent new cases of necessary action and by modifying the existing Ps to be more discriminating. For instance, there are many ways that a move can be refuted or allowed to succeed (APs S1a, S1b, S2b, S2c, S2d, S4a, and S4b in Figure 8.3), and these ways developed gradually as tests were tried. When the executive was in good shape, strategy Ps and terminal patterns were added, resulting in more executive modifications as still more was found out in doing tree search over a wider range of positions. The options for the executive discussed in Section C were not tried until all the gaps were filled in. Two features of this mode of programming are very dependent on using Ps: each P does a relatively small manipulation to a global Working Memory (half a dozen or fewer changes), and the action of the unspecified modules is usually the firing of just one P, even in their final form.

Several kinds of control are exhibited in KPKEG: iterating through sets of things to be tried, evoking some process and at the same time asserting a second signal to which control will fall back, and factoring a complex selection or decision process into a cascade of P firings. The executive iterates over strategies by repeatedly evoking the strategy Ps to get move candidates. The strategy Ps each assert a Working Memory item that prevents repetitions at the same depth, amounting to a simple way of keeping the context of the generation. Within each strategy the order in which Ps fire is indeterminate, but there could easily be more control, with nothing added to the executive. Another form of iteration through a generated set is in using the move candidates asserted by a single strategy. Each time control falls back to P S21 (AP S2a), it selects (with one firing) one of the candidates, and erases it so it won't be considered again.

The RHS of AP S0c illustrates the way control can be arranged to fall back to process signals stacked up in Psnlst's :SMPX. First, it evokes the Q's with a "make-move" signal, and control falls back to the stacked "check-terminal-position" signal (the second conjunct in S0c's RHS). This results in evoking the terminal position patterns (S3o's) and if none fires, control flows to a P that responds to "erase-check-terminal-position", also asserted by S0c (see AP S3a). When an exhaustion of strategies occurs (all levels tried at some depth), control falls back to the appropriate place by re-asserting the DEPTH instance (AP S1b). The "new" DEPTH is then examined in connection with instances at the previous depth that recorded what was being tried when the descent occurred, in order to check the results. A new DEPTH is also responded to when an ascent occurs (a more specific signal is not used), and the response varies according to whether move candidates and untried strategies exist (APs S0d, S0a, S0b, S2a) - here the response is selected from a range of possibilities, illustrating the potential for openness of control.

Control through the factoring out of cases is evident in two places as a result of the board representation, which distinguishes eight inter-square relations. The king-move Ps
(Q10's) consist of one P that fires for all king moves plus a set of eight, one of which fires to finish the move. The means Ps for moving toward some square are also eight in number. A strategy P decides what is to move toward and a means P fires to produce that actual move candidate. This cascading of selections from among sets of Ps is the essence of PS control: action sequences alternate with complex selections of what is to occur next, which allow potentially the application of large amounts of knowledge. As more knowledge is applied in directing control flow, more intelligence will result in the overall process.

Most of the chess knowledge in KPKEG is encoded in the S30's, the W's, and the B's, whose content was discussed in Section B.3. The knowledge is exclusively in the form of patterns for recognition (LHS), with relatively simple actions (RHS). The patterns consist of testing: locations and controlled squares for the chess pieces, inter-square relations, numerical rank and file properties, inter-piece distance, and relations of pieces with each other and with the edges of the board. The actions are "terminal-win" signals, move candidates, or signals to evoke means to move candidates. This simplicity is due to the simplicity of chess knowledge (at least in K-P-K), the condition-action nature of PSs, and the organization of KPKEG into executive, strategies, etc. Note that even though the Ps representing chess knowledge are not independent of the containing strategic control, and thus include control signals, the control is minimal and uniform over functionally similar Ps.

The design philosophy is to establish a flexible matrix into which specialized knowledge is added. It is not necessary to limit added knowledge to single-P packages, as is illustrated in several places (e.g. the W7 Ps). The general properties of KPKEG allow easy encoding of chess knowledge, but the syntactic features could stand improvement, as we will discuss in the next paragraph.

Several features of the PS architecture are especially awkward or inefficient for the chess task in particular. (1) The primary inefficiency in KPKEG is in finding one match among a set of Ps that are constructed such that only one match (or perhaps a small number) in a given situation is likely. This is the case for most of the chess knowledge, i.e., the strategies and the terminal-position patterns. The opportunity for savings is that failing one match from the set might be used to reject some set of Ps from consideration. A simple and effective remedy is to store (and perhaps represent externally) the Ps as a tree of tests, where rejecting some branch in the tree amounts to rejecting the set of Ps whose RHSs correspond to that subtree's terminal nodes. (2) A related problem is a certain repetitiveness of bindings in the patterns. For instance, many of the patterns start out by binding variables to the locations of the kings and the pawn. This problem can be remedied in the same way as the preceding one. (3) The Working Memory for the board representation predicates is heavily loaded, probably resulting in high costs for patterns that access a number of board relations. Since, at present, the instances of each predicate are implemented simply as a list, there is room for improvement. The match routine could be modified to evoke functions to compute relations, perhaps resulting in a significant cost saving over the present access of a long list. (4) There are probably a number of recurring pattern expressions of a chess-specific nature that could be made more easily expressible by syntactic conventions. These could be obtained by detailed study of existing Ps and by analysis of chess knowledge. Further detail on this is beyond the present scope, since it appears applicable only to chess tasks.
E. A Comparison to a Similar Program in Lisp

KPKEG can be compared in detail to a similar program in Lisp, developed by Perdue (1975). Perdue’s program, CP, can presently do tasks similar to KPKEG’s, but is intended to develop into a much broader class of chess endgames. This section will first compare the overall organization in the two programs. Differences in chess knowledge content and in approach to the problem give rise to behavior differences, to be discussed second. Considering superficial aspects, such as conciseness and efficiency, also gives rise to contrasts, discussed third. Differences in the details of representations and processing will be discussed last.

The control organizations of KPKEG and CP are quite similar, ignoring for the moment that the means for implementing control are radically different. The main function in CP is Findamv (find-a-move), which controls the tree search, and calls other procedures to recognize terminal positions, to try making moves, and to do tree bookkeeping. Findamv is an iterative (as opposed to recursive) alpha-beta minimax procedure, looping over a body of code that either descends or ascends in the tree according to results of subordinate function calls. This corresponds roughly to the control parts of the S P group (i.e., excluding the S30’s), which in effect loop by re-examining the “check-other-strategies” signal. The tree-bookkeeping functions correspond to S5-S7, and the functions called by Findamv to recognize terminal patterns correspond to the S30’s. The major action of Findamv is to call the function Tryamv (try-a-move), which results in a new board position. Tryamv calls several functions in turn, the most important of which are More!Moves and Move2. Move2 actually executes chess moves and corresponds to the Q Ps. More!Moves has a producer-consumer relationship to the strategy function RG (recognize), and calls Genmvs (generate-moves) with the results of RG. It is “producer-consumer” because More!Moves calls RG repeatedly, each time obtaining something new, in much the same way as the S Ps repeatedly evoke the W’s and B’s. RG and its subordinate functions examine the board and propose strategies in correspondence with KPKEG’s W and B Ps, except that RG produces an instantiated strategy descriptor rather than actual move candidates. More!Moves takes RG’s output and passes it to Genmvs, which executes (Evals) the instantiated strategy descriptor to produce actual move candidates. Genmvs thus corresponds to the move-candidate assertion by the W and B Ps, and also to the M Ps.

In summary, the overall form of control organization is quite similar in the two programs. KPKEG maintains its control with explicit Working Memory items and by responding to new items in Working Memory, whereas CP uses the conventional Lisp control stack. But Ps in KPKEG group naturally into sets that functionally correspond to Lisp functions in CP.

KPKEG and CP differ markedly in behavior, even though the control organization can be put into the above correspondence. CP is not strongly based on the strategy hierarchy principle, but rather does a mini-max alpha-beta search using more conventional evaluation procedures. Because of this and because of differences in the chess knowledge (e.g. the

• As far as I know, the organization of the two programs was developed independently.
patterns tested in CP's RG don't correspond exactly to KPKEG's W Ps). CP's search is shorter, covering around 10 nodes on KPKEG's TestI as opposed to 40. CP is designed so that strategies tend to generate very few moves at each node, whereas KPKEG aims to make the strategies generate all conceivable moves that might lead to the strategic objectives at the particular strategy levels. In addition, CP doesn't search through alternatives when backing up, but returns all the way to the initial starting position and try new move sequences from there. Even though these differences give rise to different behavior, I maintain that they are non-essential, in the sense that they could easily be brought into line without changing the characteristics of the two programs on which the following comparisons are based.

There are a number of differences between KPKEG and CP that are primarily attributable to differences between Psnlst and Lisp, and secondarily perhaps to the difference in programmers. KPKEG has 140 Ps, with a listing of about 900 lines, whereas CP has about 270 functions with a listing of about 2640 lines. By these (very crude) measures, KPKEG is much more concise, a factor of 2 in elementary program units and a factor of 3 in size of program listing. In run-time efficiency, KPKEG is somewhat worse than CP, using 20 seconds per node (which turns out to be 20 P firings) as opposed to about 6 seconds. Section D contains a discussion of some possible causes for inefficiency in the PS, and suggests some modifications. In addition, it should be pointed out that the present PS is done by interpretation, rather than by compiling the Ps into some kind of optimal network, which would have the potential of speeding up the recognize-act cycle by avoiding duplication in condition testing (see Chapter VII).

The most marked contrasts between KPKEG and CP are in the relatively low-level details of how things are represented and processed. Where KPKEG uses Working Memory relations to represent the chess board, CP uses a two-dimensional array, accessing squares by their coordinates. The KPKEG representation is actually dual: one way expresses the eight intersquare relations (e.g., C3 to D2 is the DIAGRB direction), and the second way associates coordinates to the square names (e.g., RF(F4, 4, 6)). The dual representation is in part forced by a peculiarity of Psnlst, which doesn't allow constants to be expressed directly in the LHS match; using the coordinates as constants indirectly would force a search through 64 pairs of variable bindings. This becomes intolerable when one is testing for two squares' having some relation between them, requiring a search through 64 X 64 binding pairs to find the right set satisfying, say, some arithmetic predicate. (Even without the peculiar limitation, convenience in programming and readability of Ps might recommend the dual representation.)

A related feature is KPKEG's use of Working Memory for CONTROLS relations, where CP recomputes them each time they're necessary. CONTROLS is used to indicate that a piece can move directly onto a square, and is involved in testing, e.g., whether the pawn is safe on some square. For the king, for instance, CP tests control of a square by testing whether the king is on one of the eight adjacent squares, and that in turn is tested by simple arithmetic on the square's co-ordinates. To do this test by co-ordinates in KPKEG would not be combinatorial as mentioned above, but would be cumbersome, requiring testing of eight numerical predicates between the king's coordinates and the square's. In Lisp the cumbersomeness can be packaged into one function, but to do this "subroutining" in PSs would force breaking a single match into three, one to set up the test, one to do the test (one of eight Ps might fire), and one to finish matching the condition that included
the test. Some clumsiness is still inherent in the PS implementation of CONTROLS, as is illustrated by the king-move Q Ps. There, eight Ps are required to do the CONTROLS updating when a king move is made, one P for each potential king-move direction. Note that these eight are coded once, for each chess piece, so that there need be no concern along these lines in dynamic augmentation situations. But the use of extensive Working Memory relations like CONTROLS (as opposed to intensive recomputed relations) is a mechanism that is essential when relations become more complex, as they certainly do in chess, and the mechanism is provided by PSs as an essential architectural feature.

Both programs represent the board as a global structure that is updated and down-dated as moves in the search are made and retracted. CP records necessary contextual information for the board at each depth in a stack that is correspondingly pushed and popped, whereas KPKEG uses a depth argument that is attached to predicates that store essential information such as captured piece locations.

CP keeps its strategies and move candidates in a similar structure, a context list whose head (Car) is a list of untried ones and whose tail (Cdr) is the list of old, tried ones. KPKEG's Working Memory only stores, for move candidates, the untried ones, and for strategies, the ones that have been tried (STRAT:TRIED). Each strategy P includes a condition to ensure that no STRAT:TRIED exists for it, to avoid duplication, whereas move-candidates are simply erased on use (this doesn't guarantee that different strategies or different Ps of the same strategy don't generate the same moves, which are then tried). For each entry in CP's board-context stack, there is that pair of lists, where KPKEG marks the elements with a depth argument. The way CP handles generation of candidates for these lists is to generate a full list and then test whether the elements of that list are on the appropriate context list. Under this regime, for instance, in the producer-consumer iteration between More!Moves and RG, a list might be produced, only to discover that all its elements had already been added to the context list. In practice, for the sizes of lists encountered in CP, this is apparently not prohibitively costly.

Finally, we examine the parts of CP where PS-like patterns are tested. CP uses uniform database procedures constructed for storing properties, whereas KPKEG uses the existing Working Memory. CP has two functions, FORALL and EX (Exists), which perform iteration over lists and selections from lists according to specifiable Lisp COND's, operations that are included in the PS match. CP's patterns also make more use of function calls to test various conditions than do the Ps in KPKEG. In CP, all of the pattern testing is under strict control and is embedded in variable-binding contexts that establish the data for the patterns. This is less true of KPKEG, although sets of Ps are under control of explicit Working Memory items asserted at specific points in the control flow. Figure E.1 gives a pattern roughly comparable to Figure B.4, illustrating the function-calling style of the Lisp patterns.

In PSs, control of which matches are done is potentially more flexible and efficient: In KPKEG, selection is from an unordered set of P conditions, whereas a Lisp function containing a set of tests is executed in a fixed pre-determined order. The order of testing of P conditions could thus be rearranged dynamically as different Working Memory states

- Some subroutining in the PS is used, however, to handle what is common to the eight, for program conciseness.
A Comparison to a Similar Program in Lisp

**Figure E.1** Fragment of Estim function of CP

It is conceivable to code a Lisp pattern matcher that has desirable efficiency properties as long as patterns to be matched are not allowed to become too arbitrary. Efficiency could also be maintained in more arbitrary patterns by including heuristic information in patterns, to guide the matcher. This would make adding patterns more difficult, however. The PS approach is to adopt specific and perhaps stringent conventions which allow a general procedure to compute an optimal matching strategy. This is not to say that such a procedure has been developed yet, but there is some indication that the problem is tractable.
F. Extending KPKEG

This section will consider the foreseeable problems in extending KPKEG to a more complete chess program. First, we consider some topics having to do with the executive and with the strategy hierarchy principle. Then, we consider how KPKEG might be extended to more complex domains. These will require a number of extensions to KPKEG's representational capabilities, such as more complex inter-piece relations and descriptions of dynamic situations. In the following, the emphasis will not be on details of such extensions, but on their demands on the capabilities of PSs.

In the course of the preliminary experiments with KPKEG already described, several features of the strategy hierarchy principle and the executive have come to light. In a past try of Test1 in which KPKEG arbitrarily chose to try E4-D5 as its first move at level 4, KPKEG didn't see an opportunity to take the opposition and achieve its strategic objectives because its strategy level was too high, above the level for the opposition strategy. In general, it seems to be the case that two things are not quite right: the present ordering in the hierarchy may not be correct, requiring experiments with alternative orderings; and the whole level-oriented focus may be too narrow, requiring opening it up somehow to allow strategies to take over that look much closer to being successful, rather than sticking to a strategy that requires more search and whose success is not strongly indicated in the present situation. With respect to re-ordering the strategy hierarchy, it would be easy to change the appropriate Ps to different levels by substituting a different level constant. But attention must also be given to whether the principle is itself unattainable with the fine distinctions between levels at present. Perhaps fewer than seven levels is more appropriate for K-P-K, or perhaps no ordering is correct in all situations.

With respect to the narrowness of focus, perhaps the most promising approach would be to set up a few specialized patterns that would match and redirect the program's attention when the board is changed, before the ordinary strategies are evoked. For instance, it might be useful to recognize situations where king moves result in having the black king move out of the square so that the pawn is clear to advance; or situations where the pawn is left open to attack in the course of some other strategic maneuvers. A more radical change to KPKEG would be to reorganize the strategies to be much more bottom-up, analyzing the board in terms of what looks possible, rather than top-down as at present, setting up goals to try particular things in a predefined order. This would probably require much better descriptive capabilities as described below.

Finally, with respect to the strategy hierarchy, on the tests tried there appears to be no need for the standard alpha-beta minimax procedure; i.e., the search always stays in the region above "alpha", converging on the best available move from above. A proof or refutation of this property may emerge as the principle is exercised on chess tasks that aren't as limited as K-P-K.

None of the difference of 40 nodes searched versus 10 nodes for CP are due to alpha-beta considerations.

V-25
There are simple variations on the present task domain that introduce new complexities and that may force major changes in the basic descriptive elements that the Ps. work with. The tests used for KPKEG deal with relatively localized situations, as opposed to ones requiring many moves to bring the pieces together for the localized rules to apply. Such a situation is illustrated in Figure F.1.

```
... ... ...
  BK ... ...
  ... ... ...
  ... ... ...
  ... WP ...
  ... ... ...
  ... BK ...

Figure F.1 A non-localized K-P-K position
```

This class of situation requires at least the use of special strategies that generate fewer alternative move candidates, and candidates that are more specifically directed toward particular distant squares, than the present move-towards means. It also requires that the maximum search depth be increased (from its present setting of 9) or allowed to be changed as the situation demands. (Perhaps maximum depth is the wrong approach, but there will probably remain the idea that at some point the situation requires a static evaluation such as the one done now when the maximum depth is reached.)

Tasks with more than one pawn introduce considerable complexity. A typical situation is given in Figure F.2. Some salient features of such tasks are: the necessity for the white king, as well as the black, to broaden its strategy to stop enemy pawn advances; the necessity to divide the board into two or more sectors of activity that are to some extent independent of other such sectors; and the necessity to describe relations between such sectors, with particular attention to the ways in which individual pieces can perform functions in more than one sector.

```
  ... BP ... 
  ... BK ... 
  ... WP ...
  ... BP... 
  UP ... ... 
  WPK ... ...

Figure F.2 A task presently beyond KPKEG's capabilities
```

As we have pointed out several times above, advances in KPKEG's power depend on enriching its board representation. Three levels of descriptive organization can be
distinguished: relations, which are computed directly from the board, for instance, CONTROLS in KPKEG; chunks, which combine several relations, usually labelling commonly recurring or important combinations; and board sectors, which are the semi-independent units of analysis described above in connection with more complex endgame tasks. For KPKEG, which already has relations to a limited extent as Working Memory items, it is feasible to have relations, chunks, and even sector divisions computed when the board is updated, by Ps that recognize conditions that make or break the descriptive units. These Ps would not need to be specifically evoked, but would work in a bottom-up fashion (the considerations of efficiency discussed in Section D would apply here). Note that in already having some relations, and in the proposed updating capability, KPKEG is superior to CP, where additional ad hoc procedures and calling conventions would be required. CP and other similar program structures would probably find it difficult to direct their activity in a recognition-oriented bottom-up mode, since the structure lends itself so easily to the contrary top-down mode. It is envisioned that having better descriptive capabilities would prove advantageous in expressing strategy Ps and similar patterns, in changing KPKEG to be more bottom-up as just described, and in allowing patterns such as those constructed by KPKEG itself to recognize terminal position classes instead of specific positions.

Several specific features of KPKEG are troublesome with respect to more ambitious applications to chess. One is the problem of using the present Ps for a game in which Black has the pawn. The Ps do not mention Black or White, using a Working Memory instance (KP:KAS:ASP) to determine which color the pawn is. But Ps that test board configurations rely heavily on the orientation of the board: “forward” is always towards White’s eighth rank (Black’s home row). A solution might be to transform the entire board representation so that it would be reversed with respect to the external game but would internally match the white-pawn assumption. Another feature of KPKEG is the repetitiveness of the search. The specific strategies may be at fault for generating duplicate moves; the strategy hierarchy, or its implementation as seven levels, may be at fault; or it may simply be necessary to implement a more general mechanism to prune duplicates. The general mechanism might consist of Ps that would record the results of specific moves in specific situations so that all future searches could take advantage of past effort. This, of course, has benefits beyond simply preventing duplicates. It also raises an issue that is pertinent even to the present, limited P-building scheme. That is, how can the number of Ps added be ultimately controlled, so that the set of Ps converges to a more-or-less stable size, or at least somehow avoids all possible board placements for each pattern? Perhaps the convergence will occur when more powerful descriptive devices are used, e.g., the chunks mentioned above. Using more abstract descriptors of the board in this way would result in Ps with greater generality, and in fewer distinct Ps overall. An alternative is a scheme of generalization that might collapse a set of existing Ps into one according to a general procedure. At present, only indications of the need for further research can be put forward.

Finally, we briefly consider some requirements for improved chess programs as put forward by Berliner (1973, 1975a). The basis for the improvements to be considered is the idea of a causality facility, whose purpose is to determine why a search fell short of aspirations. It must differentiate between failure for superficial reasons (a particular move, for instance) or for deep ones (inherent features of the situation). The first specific improvement comes from the idea of building a refutation description as a result of a search that failed strategically. The refutation description includes features of the position.
and of the search that the causality facility proposes as essential to the failure. It is used by move generators that try to counter those features, thus giving the program a way of restricting available move choices. For K-P-K, the implementation of this idea would result in searches with fewer branches than in KPKEG, but with the option of generating specific extra branches to meet specific demands. Since move-generators are Ps in KPKEG, the immediate approach to try would be to build specific Ps sensitive to elements of a refutation description in Working Memory. The second improvement comes from the idea of lemmas. Lemmas are the followup of a causality analysis, functioning to reject lines of play on the basis of a description of a difficulty that is known to be fatal to all such lines of play. The PS approach to this involves building a P to act as a "demon" to recognize such situations and immediately refute moves that don't surmount the difficulty.

We can now review the progress KPKEG has made toward its aim of establishing PSs as a viable architecture for chess programming, especially in comparison with Lisp and other conventional architectures. The standard variety of search in a tree of moves has been readily implemented, using knowledge in Ps to significantly reduce the amount of search. Modular sets of Ps cooperate smoothly to achieve an overall organization similar to a subroutine hierarchy, but with more flexibility and openness than subroutines. PSs are a concise and easily augmentable way of representing strategic knowledge in chess. PSs are also appropriate for complex selections and behavior that frequently requires complex choices. The present implementation has been useful as a pilot study of the K-P-K task, lending itself to explorations of various options and to development of control knowledge incrementally. Explorations of options take place usually by simple modifications in RHSs of Ps and by splitting an existing P into two or more finer discriminations, for action alternatives. The PS approach shows significant promise for bottom-up action, i.e., action intimately connected to the immediate problem-solving situation, which seems desirable in comparison to top-down hierarchically-controlled direction of action. There is the possibility of syntactic modifications to improve efficiency and smoothness of expression of chess patterns. Finally, approaches to more complex chess tasks are well within current PS capabilities, with natural and immediate application to several proposed mechanisms for improving the state of chess programming technology.

F.1. Acknowledgements

The author gratefully acknowledges help from Hans Berliner and Crispin Perdue in developing the concepts on which KPKEG is based, and in reading and commenting on a preliminary draft of this chapter.
G. References


END.
[Partial content not legible due to poor image quality]
TEST: ORDINARY VERSION WITH P MULTIN

LEVEL = 5 W
1 MOVING UP FROM ES TO E7 LEVEL S
2 MOVING BK FROM C7 TO DB LEVEL S
CAN'T MOVE C7 DB
5 MOVING BK FROM C7 TO DB LEVEL S
LEVEL + 9 M
LEVEL + FAIL DEPTH B M
SUCCEEDED C7 DB = 523

(MOVING UP FROM ES TO E7 LEVEL S)
ADDED BK-3 DEPTH 2 LEVEL S E7 D7
RETRACTING C7 D7
RETRACTING ES E7
LEVEL = 4 W
5 MOVING BK FROM C7 TO DB LEVEL 4
5 MOVING BK FROM C7 TO DB LEVEL 4
7 MOVING BK FROM DB TO DB LEVEL 4
8 MOVING BK FROM DB TO DB LEVEL 4
CAN'T MOVE DB DB
9 MOVING BK FROM DB TO DB LEVEL 4
CAN'T MOVE DB DB
LEVEL = 5 W
10 MOVING BK FROM ES TO E7 LEVEL 4
LEVEL = 5 B
LEVEL = 6 B
LEVEL = 7 B
LEVEL + FAIL DEPTH 6 B
SUCCEEDED ES E7 = 523

(MOVING UP FROM ES TO E7 LEVEL S)
ADDED BK-5 DEPTH 6 LEVEL 6 E7
RETRACTING ES E7
RETRACTING DB DB
11 MOVING BK FROM DB TO DB LEVEL 4
CAN'T MOVE DB DB
LEVEL = 5 B
12 MOVING BK FROM DB TO DB LEVEL 4
TERMINAL WIN FOR H = PH-2

(MOVING UP FROM ES TO E7 LEVEL S)
ADDED BK-2 DEPTH 5 LEVEL 5 E7
RETRACTING ES E7
RETRACTING DB DB
13 MOVING BK FROM DB TO DB LEVEL 4
CAN'T MOVE DB DB
LEVEL = 5 B
14 MOVING BK FROM DB TO DB LEVEL 4
LEVEL + 9 M
LEVEL + FAIL DEPTH 9 M
SUCCEEDED ES E7 = 523
ADDED BK-3 DEPTH 3 LEVEL 3 E7 D8
RETRACTING ES D8
RETRACTING C7 DB
15 MOVING BK FROM C7 TO D7 LEVEL 4

(CAN'T MOVE C7 D7)
16 MOVING BK FROM C7 TO DB LEVEL 4
17 MOVING BK FROM ES TO DB LEVEL 4
18 MOVING BK FROM ES TO DB LEVEL 4
TERMINAL WIN FOR H = PH-8

(MOVING UP FROM ES TO E7 LEVEL S)
ADDED BK-3 DEPTH 2 LEVEL S E7 D7
RETRACTING C7 D7
RETRACTING ES E7
LEVEL = 5 M
19 MOVING BK FROM C7 TO C7 LEVEL 5
20 MOVING BK FROM C7 TO C7 LEVEL 5
21 MOVING BK FROM C7 TO C7 LEVEL 5
22 MOVING BK FROM C7 TO C7 LEVEL 5
CAN'T MOVE C7 DB
23 MOVING BK FROM C7 TO C7 LEVEL 5
LEVEL = 5 B
24 MOVING BK FROM C7 TO C7 LEVEL 5
LEVEL + 9 M
LEVEL + FAIL DEPTH 6 B
SUCCEEDED FB FF = 523

(MOVING UP FROM ES TO E7 LEVEL S)
ADDED BK-5 DEPTH 5 LEVEL 5 E7
RETRACTING FF FF
RETRACTING DB DB
25 MOVING BK FROM C7 TO DB LEVEL 4
CAN'T MOVE DB DB
26 MOVING BK FROM C7 TO DB LEVEL 4
27 MOVING BK FROM C7 TO C7 LEVEL 4
28 MOVING BK FROM C7 TO C7 LEVEL 4
CAN'T MOVE C7 DB
29 MOVING BK FROM C7 TO C7 LEVEL 4
30 MOVING BK FROM C7 TO C7 LEVEL 4
LEVEL = 5 W
31 MOVING BK FROM C7 TO DB LEVEL 4
LEVEL + 9 M
LEVEL + FAIL DEPTH 6 B
SUCCEEDED ES E7 = 523
ADDED BK-6 DEPTH 7 LEVEL S E7 E8
RETRACTING E7 E8
RETRACTING C7 DB
32 MOVING BK FROM C7 TO DB LEVEL 4
CAN'T MOVE C7 DB
LEVEL = 5 B
33 MOVING BK FROM C7 TO DB LEVEL 4
CAN'T MOVE C7 DB
34 MOVING BK FROM C7 TO C7 LEVEL 4
CAN'T MOVE C7 D7
35 MOVING BK FROM C7 TO C7 LEVEL 4
TERMINAL WIN FOR H = PH-8

RETRACTING C7 DB
RETRACTING ES E7
RETRACTING DB DB
TERMINAL WIN FOR H = PH-8

RETRACTING C7 DB
TERMINAL WIN FOR H = PH-8
### Table: Percentages of Firings of Each Type, Out of Total 842

<table>
<thead>
<tr>
<th>Type</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1.0</td>
</tr>
<tr>
<td>Y</td>
<td>27.0</td>
</tr>
<tr>
<td>Z</td>
<td>1.0</td>
</tr>
<tr>
<td>P</td>
<td>10.0</td>
</tr>
</tbody>
</table>

### Appendix E: Tracks for the Other Tests

**Test 1:** Version of Spies with All Depths Reorienting from B5 to Level 6

- SK
- MP
- ME
- WP
- XR
- SF
- TM

**Level - 6 N**
- 1 Moving up from ES to E7 Level 7
  - Level - 6 B
  - Level - 5 B
  - CAN'T MOVE E7 08
  - 2 Moving up from C7 to DB Level 6
  - Level - 6 W
  - Level - FAIL DEPTH 6 8
  - Succeed E8 06 = 523

**Level - 5 W**
- 3 Moving up from ES to E7 Level 7

**Level - 6 W**
- 4 Moving up from ES to E7 Level 7

**Level - 5 W**
- 5 Moving up from ES to E7 Level 7

**Level - 4 W**
- 6 Moving up from ES to E7 Level 7

**Level - 3 W**
- 7 Moving up from ES to DB Level 7
  - Level - 6 B
  - Level - 5 B
  - Level - 6 B
  - Level - FAIL DEPTH 6 8
  - Succeed ES 06 = 523

**Level - 2 W**
- 8 Moving up from ES to DB Level 7
  - Level - 6 B
  - Level - 6 B
  - Level - 6 B
  - Level - FAIL DEPTH 6 8
  - Succeed ES 06 = 523

**Level - 1 W**
- 9 Moving up from ES to DB Level 7
  - Level - 6 B
  - Level - 6 B
  - Level - 6 B
  - Level - FAIL DEPTH 6 8
  - Succeed ES 06 = 523

**Level - 0 W**
- 10 Moving up from ES to DB Level 7
  - Level - 6 B
  - Level - 6 B
  - Level - 6 B
  - Level - FAIL DEPTH 6 8
  - Succeed ES 06 = 523

**Level - 5 H**
- 11 Moving up from ES to DB Level 7
  - Level - 6 B
  - Level - 6 B
  - Level - 6 B
  - Level - FAIL DEPTH 6 8
  - Succeed ES 06 = 523

**Level - 4 H**
- 12 Moving up from ES to DB Level 7
  - Level - 6 B
  - Level - 6 B
  - Level - 6 B
  - Level - FAIL DEPTH 6 8
  - Succeed ES 06 = 523

**Level - 3 H**
- 13 Moving up from ES to DB Level 7
  - Level - 6 B
  - Level - 6 B
  - Level - 6 B
  - Level - FAIL DEPTH 6 8
  - Succeed ES 06 = 523

**Level - 2 H**
- 14 Moving up from ES to DB Level 7
  - Level - 6 B
  - Level - 6 B
  - Level - 6 B
  - Level - FAIL DEPTH 6 8
  - Succeed ES 06 = 523

**Level - 1 H**
- 15 Moving up from ES to DB Level 7
  - Level - 6 B
  - Level - 6 B
  - Level - 6 B
  - Level - FAIL DEPTH 6 8
  - Succeed ES 06 = 523

**Level - 0 H**
- 16 Moving up from ES to DB Level 7
  - Level - 6 B
  - Level - 6 B
  - Level - 6 B
  - Level - FAIL DEPTH 6 8
  - Succeed ES 06 = 523
<table>
<thead>
<tr>
<th>TIME</th>
<th>FIRE</th>
<th>HITS</th>
<th>WEPCT</th>
<th>E/F</th>
<th>E/T</th>
<th>1/F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>10.5</td>
<td>7.00</td>
<td>2470</td>
<td>7.13</td>
<td>2.31</td>
<td>3.22</td>
</tr>
<tr>
<td>0.113</td>
<td>0.251</td>
<td>0.840</td>
<td>0.194</td>
<td>0.319</td>
<td>0.319</td>
<td>0.242</td>
</tr>
</tbody>
</table>

1787 INSERTS 1628 DELETES 445 MAPPINGS 8 NEW OBJECTS
MAX = 314 LENGTH 118
CORE (FREE/FULL) 16518 / 21790 USED 16518 / 5821
FIRED 97 OUT OF 143 PRODS
Chapter VI

MiliPS/WBlox

A Natural Language Input Toy Blocks Problem Solver

Abstract. The MiliPS/WBlox production system is a combination of two major systems, one for processing a simple subset of natural language and the other for solving problems in a simple toy blocks domain. The emphasis of the natural language part is to study some problems of ambiguity and to illustrate a direct, non-syntactic-parsing approach to understanding natural language. The blocks problem solver deals with simple blocks manipulations, but deals with them in a general way. It features a simple goal-subgoal mechanism and conventions that allow choicepoints for a backtracking search. The blocks manipulations are a close imitation of Winograd's Planner system.
# Table of Contents

For Chapter VI

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>VI-1</td>
</tr>
<tr>
<td>B</td>
<td>VI-5</td>
</tr>
<tr>
<td>C</td>
<td>VI-21</td>
</tr>
<tr>
<td>D</td>
<td>VI-27</td>
</tr>
<tr>
<td>E</td>
<td>VI-49</td>
</tr>
<tr>
<td>F</td>
<td>VI-63</td>
</tr>
<tr>
<td>G</td>
<td>VI-67</td>
</tr>
</tbody>
</table>

## APPENDIX

<table>
<thead>
<tr>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>VI-70</td>
</tr>
<tr>
<td>VI-74</td>
</tr>
<tr>
<td>VI-77</td>
</tr>
</tbody>
</table>
Table of Contents

MiliPS/WBlox

D  Detailed Trace for MiliPS Test2 ........................................ VI-83
E  Augmentations to MiliPS for WBllox .................................. VI-91
F  WBllox Program Listing ................................................... VI-95
G  Cross-Reference of WBllox Predicates ................................ VI-102
H  Traces for WBllox Tests .................................................... VI-108
I  Detailed Trace for WBllox Test1 ......................................... VI-131
MiliPS/WBlox

A. Introduction

MiliPS is a production system (PS)\textsuperscript{1} implementation of an extension of MILISY (mini-linguistic system), a mini-program used to illustrate natural language processing in the CMU AI course. MILISY takes in facts about a toy blocks scene in restricted natural language, builds up a database of those facts, and answers queries about them. This chapter presents MiliPS in two versions. The first version, consisting of MiliPS alone, augments the language-processing aspects of MILISY, while the second, consisting of a further augmentation of MiliPS plus another system, WBlox (W for Winograd), emphasizes block-manipulation problem-solving aspects.

MiliPS aims to make the language processing more complete than MILISY, in being able to give information on and query more features of the blocks scene. The language that MiliPS understands is composed of descriptive attribute values (adjectives), nouns, main sentence function words, prepositional phrases representing relations between objects, and subordinate clauses that can be used to further refine descriptions of objects. This language can be expressed as an ambiguous context-free grammar, but MiliPS does not proceed by extracting the grammatical structure of its input as a parse tree. Ambiguities are resolved by flexible use of features of the scene, essentially as soon in the process of scanning the input as is logically possible.

The blocks manipulations that constitute WBlox are based closely on the problem-solving part of Winograd's SHRDLU program (Winograd, 1972). That subsystem of SHRDLU was coded in Micro-Planner (Sussman and Winograd, 1970; henceforth, referred to as Planner), a language specifically designed to make certain heuristic search operations automatic. WBlox moves single objects (rectangular blocks and pyramids) between locations in the scene without spatial rotation, finds locations to put them, builds stacks of them, and packs them compactly into a space if necessary. WBlox uses a hierarchical goal-subgoal structure to break big operations down into more primitive ones, with a set of indivisible primitives consisting of moving the hand to specific locations, grasping objects, and letting go of objects. At certain key points in the problem solution process, arbitrary choices are made, requiring WBlox to record its choice and the context, so that corrections are possible later in response to unforeseen difficulties. The particular approach to the search through the space of choices in WBlox is intended to imitate the Planner approach, not to represent the best scheme for PSs, which it certainly isn't.

The toy blocks domain has features that are abstractions of a much more general domain of discourse. It is composed of objects that have certain non-changeable attributes, and that enter into relations with other objects. This certainly models (abstractly) the physical world in which humans move, but it also goes much further, representing important aspects of human sociocultural organization, of economic systems, and of numerous more abstract formal (or informal) disciplines such as computer programming. (A piece of a computer program has attributes, e.g. what it is intended to do, and relations, e.g. dependence on other code for its inputs; there can be several pieces of code competing for the same space within a "block" of computer storage, etc.) Of

\textsuperscript{1} PS will abbreviate production system, plural PSs; P will abbreviate production, plural Ps.
course, how relationships and attributes are structured in real domains does not correspond to how they are treated in toy blocks, but it is to be hoped that some of the more general techniques that work with a blocks domain might carry over, requiring only modification of the detailed semantics of specific relations and attributes.

That correspondence to more important problems provides some motivation for pursuing the present study. More motivation comes from the desire to develop a flexible PS-based approach to natural language processing, and to test its feasibility on a significant and classical AI task. WBlox also provides the opportunity to compare a PS program to a functionally similar one written in Planner. It may also provide future comparisons to other AI programming systems and proposals, and act as a benchmark.

For those familiar with Winograd's (1972) program, I will summarize the primary differences between the MiliPS/WBlox system and SHRDLU. The blocks part of SHRDLU has direct analogs in WBlox, except that WBlox doesn't do quite all of the bookkeeping and memory functions (such as remembering all the steps of a plan so they can be "executed" at the end of planning). This only means that MiliPS can't answer questions about why it did various steps in performing a particular command, and when it did them. The language understanding part of SHRDLU is much more capable than MiliPS. The internal representation is not as rich in MiliPS, especially in semantic attributes, e.g. "manipulable", and the language doesn't give full access to features of the representation that it does have, like size and location. It recognizes only the imperative form of verbs, and can't deal with other more descriptive references to the commands that it can do. It doesn't interact to resolve ambiguities as SHRDLU did, but simply gives an error message and waits for a corrected version of the sentence. It is unable to dynamically define new words as SHRDLU was apparently able to do. Finally, there is very little in the way of language generation. Its replies are mostly fixed, and the ones that aren't fixed are descriptive, giving (stupidly) all the attributes' values for an object or all the relations it has with other objects, in order to tell the user about the object. On the other hand, it is quite capable of handling most of the ambiguities and reference problems that SHRDLU did, except references to objects in other sentences of a conversation, using, e.g., pronouns. It has captured many desirable features that go with a problem-solving system such as WBlox, and is a satisfactory first approximation.

The approach here has been in a way opposite to Winograd's. MiliPS started out as a comparison of PSs to MILISY, a program with very modest aspirations and serious deficiencies in dealing with its model of the blocks scene. MiliPS first overcame those deficiencies, and went rather far beyond any conceivable extensions of MILISY within its own control structure, which was a more traditional phrase-structure transformational one. Any comparison of PSs with that structure is not possible now because it would require either large extrapolations in MILISY's abilities or actually trying to extend the implementation to compare with MiliPS. After MiliPS had supposedly been refined to a stable version, the blocks manipulation task came along, and the urge to use MiliPS as an interface to a blocks problem solver was irresistible. But only a minimal sort of extension to MiliPS could be justified since the blocks manipulations were more central to the goals of investigating the properties of PSs. Thus the language is only a convenience in the final MiliPS/WBlox system. Winograd on the other hand concentrated on linguistic issues, and tacked on the blocks program as an easy means toward illustrating the power of his linguistic understanding system.
The structure of this chapter reflects the dual history and forced juxtaposition of two lines of research. Section B and Section C are devoted solely to describing MiLiPS: its overall structure, the input language, how the language deals with describing all the desired features of the limited blocks scene, and the system it uses to disambiguate complex descriptions. The latter section gives more complete details of the actual PS structure. Section D and Section E do corresponding things for the WBlox system, touching only in passing the nature of the extensions to MiLiPS that were required. Near the beginnings of both descriptions, some typical sentences and behaviors are discussed.
B. Overview of MiliPS

This section gives a general overview of MiliPS, postponing details until the next section. Section B.1 first discusses a few of the tests given to the program, with only vague descriptions of the processing done. It then gives a precise description of the task domain, including a grammar for the input language and a systematic presentation of semantic capabilities. Section B.2 uses very abstract Ps to describe the way the program works and outlines the processing of an input. Several levels of semantic processing are distinguished. Section B.3 discusses PS control and organization, low-level PS features, representation, and the expected extensibility of the present approach to syntax and semantics.

B.1. Features of the task

MiliPS has been tested on a set of 25 sentences, forming a continuous conversation about a single growing scene. The full dialog is given in Appendix C, along with trace information that will be explained in Section C. The following sentences will give the reader some idea of its capabilities.

MiliPS starts out with no initial scene, building up everything from descriptions of a scene by the user.

INPUT 1: (A LARGE GREEN BLOCK IS ON A TABLE)
In response to the first part of 1, MiliPS creates a block, adds "size large" and "color green" to its internal representation. It creates a table after scanning the rest of the input, and adds "color red" to its representation. Finally, it notes the relation "on" between the two new objects.

REPLY 1: (OKAY)
MiliPS indicates with the first reply that it has used everything in the input and hasn't noticed any unresolved ambiguities, inconsistencies, etc.

In three test sentences (not shown) MiliPS has been told about a ball on the block, and is able to determine that the description in 5 refers to that particular ball.

INPUT 5: (THE BALL ON THE BLOCK IS SMALL)
The relation "on the block" is necessary because there is a second ball in the scene. The effect here is to add "size small" to the internal representation for the ball.

REPLY 5: (OKAY)

The first five inputs describe a scene, and the next five primarily ask questions on that scene.

INPUT 7: (WHAT IS BLUE)
The query asks for all objects that have the color blue. MiliPS processes "what" by forming a set of all the objects in the scene; "what" is essentially a very ambiguous noun phrase. Then it applies any further predicates in the sentence as restrictions to that set, and if anything is left when the end of the sentence is reached, it describes it as its answer.

REPLY 7: (THE BLUE BALL) (THE SMALL BLUE BALL)
In describing objects, it uses whatever attributes it knows about that object, which happen
to be size and color, taken in that fixed order. Note that its two descriptions are not necessarily unambiguous, and in this case would be insufficient as references in an input. That is, in order to refer to the first ball, an input would have to include some relation that didn't also hold for the second (which relation may in fact not exist).

INPUT 10: (IS THE BOX ON THE TABLE NEAR THE BLOCK)

MiIPS's scene is sufficient to determine that after "box" the question is about a particular object, the only box in the scene. The relation "on the table" is already true of the box, so it is redundant; if the question ended after "table", MiIPS would answer "yes". MiIPS notes the redundancy and continues on, willing to abandon that answer if something negative comes along. The second relation, "near the block", is in fact inconsistent with both preceding objects, i.e., it can't be referring to either the table or the box. Inconsistency can mean that the system has definite information to the contrary, or it can mean, as in this case, that no information exists one way or the other.

REPLY 10: (NO INFORMATION ON RELATION NEAR)

It really means "on the relation near between those two objects". Note that it can do no deduction on other information that it has about the objects. For instance, it might reasonably deduce that nearness held if the block were in the box.

Once again, some declarative inputs will be skipped, to get to a sentence with new features.

INPUT 22: (WHERE IS THE BALL IN THE BOX ON THE RED FLOOR THAT IS RED)

"The ball" is ambiguous to start with, as is "the box". A unique box is determined because the floor is unique as described. When the floor is found, the system knows that there is an unused relation, "on", and backs up in a list of the current objects to resolve the box ambiguity. The same process applies to the "in", but the ball remains ambiguous. The scan through the sentence continues, and "that is red" is found to be redundant with respect to the floor (the program only checks semantic redundancy, not the superficial redundancy that "red" has already been used to describe the floor). The redundancy leads the program to look back in the list of current objects for something that redness can apply to, and finds the main subject, the ball. The end of the sentence is reached, so a reply is constructed.

REPLY 22: (THE LARGE GREEN BALL IS NEAR IT) (THE SMALL RED BALL IS IN THE UN-RED BOX)

A "where" sentence prompts MiIPS to give the relations that an object has with others, and also the relations that other objects have with it. In the first reply above, "it" refers to the small red ball (the program doesn't keep track of the proper order of its replies, though it easily could). The "un- red box" is one that MiIPS has only been informed of as being not red. Making the reply use a subordinate clause was not considered important enough to warrant the further necessary Ps, so the "un-" form was adopted.

A final query exercises the ability to extract questions and use relations that are separated from the objects to which they refer.

INPUT 25: (IS THE BALL NEAR THE GREEN BALL IN THE BOX THAT IS NOT ON THE RED TABLE BLACK)

Here the box is not disambiguated until the end of the clause that follows it, and the subject ball is not disambiguated until the box is. The "in the box" relation restricts the subject ball, and "near the green ball" stands by itself and also restricts the subject ball. (It was somewhat troublesome to construct such a test.)

REPLY 25: (NO INFORMATION ON COLOR BLACK)
The final word in 25 expresses the question. MiliPS knows the ball is red, but cannot deduce that it is thus not black, and instead says it doesn't have positive or negative information.

The tests given to MiliPS are all expressed in a language with fairly rigid form, which can be described with a context-free grammar. Since grammar was not deemed of primary importance, a simple form with adequate power for the task at hand was preferred. The language is adequate in the sense of being able to express descriptions of objects, their relations, and their attributes, and it is sufficiently ambiguous to offer significant problems of referent determination. As others have pointed out, a strictly grammatical approach to processing natural language cannot suffice to explain or understand ordinary language use by humans, so the actual approach taken on the given grammar is one that perhaps will work in a situation where the language's apparent grammar is much more complex, but where grammar is largely disregarded and understanding is driven by semantics and pragmatics. MiliPS puts each word scanned into a word class, and simply checks the word class of the preceding word to see if the grammar would allow such an adjacency. No more global context (phrase structure or parse tree) is used in this simple error checking, except that in a couple of cases the main sentence type is used to help determine the exact word class. Almost complete reliance for detecting anomalies is thus on the semantic phase of the analysis. For more detail on the structure built to represent the input, and to verify that it isn't a parse tree, see Section B.2.

The input language for MiliPS is given in Figure B.1. There are six major types of sentences (<S>'s), which are given in the first line of the syntax. <SD> is a simple declarative sentence, <SE> tells MiliPS of the existence of a new object, <SQD> is a query about a definite object, <SQE> is a query for the existence of some object as described, <SQW> is a query that seeks an object (or all such) satisfying a description, and <SQWR> asks the relative location of an object.

The two main subcomponents of the grammar are object descriptors, <OBJ-DESCR>, and predicates or relations of objects, <REL-PRED>. "Predicates" are attributes inherent in an object, while "relations" place the object in the toy scene, giving adjacency, containment, etc. relations. A glance at the last few lines of the syntax gives a good idea of the limitations of the domain of discourse.

Needless to say, this grammar is highly ambiguous, in particular with regard to the referent of a <RELPHR> or <RELCL>. The universe of discourse consists of a "scene" with five kinds of objects, which have attributes size or color, and which can be in certain relations to one another. Any object can usually be described fully using the appropriate combination of attributes and relations. Exceptions can easily be generated by describing duplicates of some objects, but these are ambiguous in this context anyway. MILISY doesn't have the property that an object with a unique description can be described in its input language (it doesn't have subordinate clauses or the ability to conjoin relational phrases). MiliPS corrects this defect, while introducing possibility for ambiguity.

Ambiguities are resolved in a "natural" way. A phrase applies to the object immediately preceding it, unless it is inconsistent with it, in which case it applies to the
Overview of MiliPS

Abbreviations in grammar names: S - sentence; D - declarative; E - existential; Q - query; W - what; WR - where; OBJ - object; DESCR - descriptor; REL - relation or relative; PRED - predicate; CL - clause; OPT - optional; COP - copula; DEF - definite; INDEF - indefinite; MOD - modifier; SEQ - sequence; AV - attribute-value (value of an attribute, i.e. of size or color); PHR - phrase; N - noun; PRON - pronoun.

Figure B.1 The input language for MiliPS

preceding object, and so on. This "backup" occurs only past objects whose referents have been uniquely determined. Also, a phrase that is consistent with an already-uniquely determined object is said to be redundant, and may be used to restrict the referents of a previous object (more precisely, the most recent one that satisfies the following condition), if the phrase is consistent with it and if that previous object is not uniquely determined. Ambiguities for referents in <SQW> and <SQD> are handled somewhat differently, since an inconsistency might be the purpose of the query, that is, to determine if some property or relation holds. These will be discussed in detail below. Note that several consecutive prepositional phrases or subordinate clauses can apply to the same object, without a separating "and" where it would ordinarily occur in human communication.

The database consists of a simple record of properties and relations of objects described in input sentences. It is stored as a particular set of Working Memory predicate instances, which set is left intact across sentences. In declarative sentences, <SD> and <SE>, using the indefinite "a" determiner causes creation of new objects. No attempt is

- This is not an inconsistency in the database, which would be analogous to logical inconsistency in theorem-proving systems, but rather a disagreement between an interpretation of an input and the database.

B.1 VI-8
made to keep the database consistent, and no inference is done to answer queries; only a
simple lookup of the facts specified is done in this case, and also in the case of the
processing of relations and properties for ambiguity resolution. In particular, negations of
any sort are recognized only if explicit (following MILISY conventions here). There is no
inherent reason why a more sophisticated data-base regime could not be implemented, but
the focus of the current work is on certain of the language-processing aspects.

MiliPS's first reaction to an input is to scan across it, left to right, noting word
classes and, near the beginning, assigning a type to the sentence. The sentence types,
which correspond directly to the main grammatical classes descendent from <S>, are used
in minor ways to guide the classification of words. In particular, how "a" is treated
depends on sentence type: in a declarative sentence, it is indefinite, and results in creating
a new object to which it then refers; in a <SQE> query, "a" really means "any", and is
treated as if it were "the", which turns out to be the right way. Sentence type is used in
a more significant way in treating unusual semantic occurrences, namely, inconsistencies,
redundancies, unresolved ambiguities, and phrases that have no referents.

For declarative sentences, of type <SD> and <SE>, the response to the whole input
is to add to the subject of the sentence the relation or attribute-value that follows the "is"
or "is not". For these, it is known that at some point new (and thus inconsistent)
information is to appear, so it doesn't treat it as an error. The presence of the
Inconsistency actually is a helpful cue to the processing, allowing it to be done bottom-up,
rather than doing a more directed, top-down search for something new. If there is no
Inconsistency, there is either a redundancy, which is accepted without comment, or an
ambiguity, which is an error.

Queries of type <SQD> and <SQE> ask definite questions, namely specific relations or
attributes of a particular object. For these, inconsistency becomes a definite "no" or "no
information", and can sometimes be detected before the end of the sentence is reached.
Redundancy can be turned into either a positive or negative answer, depending on
whether the redundancy holds with respect to the subject or with respect to a lesser
object and is at the same time inconsistent with the subject. Ambiguities or null referents
in these are errors.

For <SQWR>, which asks "where?", MiliPS simply outputs a list of of all the relations
that pertain to the subject. No "unusual" occurrences are allowed. A sentence of type
<SQW> desires ambiguities or null references, since it asks for which set of objects in the
scene satisfy some description. It starts by assuming the full set of scene objects, when it
recognizes "what"; and as each relation or attribute-value in the sentence applies, the set
is narrowed down. If the result of the restrictions is the empty set, "nothing" is answered.
Otherwise, the object or objects in the set are "described" by adding the full list of known
attribute-values to a corresponding noun.

There seem to be six kinds of completeness that are desirable in a system like
MiliPS: completeness of reference, completeness of description, completeness of query
logic, complete ability to manipulate the model, and complete symmetry of input-output
behavior. Completeness of reference means that any object that is describable uniquely
using the attributes and relations given, can be described in the language. MiliPS has this
kind of completeness, although the particular set of relations it has could be augmented so
that scenes that are presently relationally equivalent could be further distinguished. MiliPS also lacks certain kinds of reference to which humans are accustomed, such as being able to refer to the time recency of an object, as in "the third ball" or "the block mentioned before the red one". Completeness of description means the ability to describe a new object sufficiently so that it will be unique with respect to later attempts at describing it, i.e., so that it can be the unique referent of some phrase. MiliPS has this kind of completeness also - it allows descriptive relational phrases to be strung together indefinitely, e.g., in <SD> type sentences.

Completeness of query logic can best be described in terms of possible arrangements of definite and indefinite items in an abstract notation as follows: having an object \( x \) related to object \( y \) by relation \( R \) will be denoted \( xRy \); similarly, \( x \) has a value \( v \) for attribute \( A \) is represented \( xAv \). A query logically can have a "?" in one or more of the three positions of either the \( xRy \) or \( xAv \) triples, plus the forms \( xRy? \) and \( xAv? \) are allowed, to give a total of eight possibilities for each form of triple. For the \( xRy \) form, they are (using \( x \) and \( y \) as definite objects, and "on" as a particular typical relation): \( xR? \) (what is \( x \) on top of), \( x?y \) (how is \( x \) related to \( y \)), \( ?Ry \) (what is on \( y \)), \( ??y \) (what has any relation to \( y \)), \( ?R? \) (what is on anything), \( ??x \) (where is \( x \)), \( ??? \) (what relations do you know), and \( xRy? \) (is \( x \) on \( y \)). For the \( xAv \) form, they are (using color as a typical \( A \), red as a typical \( v \), and \( x \) as a typical object): \( xA? \) (what is \( x \)'s color), \( xv? \) (what of \( x \) is red), \( ?Av \) (what has color red), \(?vv \) (what is red), \( ?A\) (what has color), \( ?? \) (what are the properties of \( x \)), \( ??? \) (what does everything look like), and \( xAv? \) (does \( x \) have color red). For the present, we ignore the further complications of numerical and other forms of quantification, keeping the logic within a propositional system.

MiliPS does not have all of those forms of query completeness, but some are included in more general cases, as the following enumerates. The forms \( xRy? \) and \( xAv? \) are gotten with \(<\text{SQD}>\) or \(<\text{SQE}>\); note that here and in most cases, if a "v" is given, the "A" is implicit, for instance, "is \( x \) red" rather than "does \( x \) have color red". Thus \(<\text{SQD}>\) and \(<\text{SQE}>\) include \( x?v \). The \(<\text{SQW}>\) sentence type gets queries of the forms \( ?Ry \) and \( ?Av \), and also, because of the 1-1 mapping between \( v \)'s and \( A \)'s, \( ?v \). \(<\text{SQWR}>\) answers the relational variety of \( x? \), and includes, but gives much more than is required, for \( x?? \) (for \( Av \), \( xR? \), \( xAv? \), and \( ??y \). MiliPS has \( ??? \) for \( xAv \) variety, by giving it "what is" (not allowed, by the strict grammar above, but the program accepts without specific modification), and this also answers but gives extra, for \( ?A? \). \( ??? \) for \( xRy \) and \( ?R? \) can be obtained by asking "what is" and then "where is \( x \)" for each thing that it gives as its reply; this gives a lot more information than is desired by the exact query. Thus, a user of MiliPS can find out everything about the scene, but only in sometimes cumbersome ways, and only if he or she does the computing necessary to reduce voluminous answers.

For MiliPS, completeness of manipulation involves being able to make changes to blocks configurations after they have been described. This would include being able to undo the effects of mistaken inputs, e.g., to remove a newly created object. MiliPS doesn't have manipulation capability at all. Completeness in symmetry of input-output behavior means being able to describe things in the same way that things can be recognized in inputs. This also is beyond MiliPS. It has internal representational features, such as color and size, that can't be used explicitly in inputs (e.g., "what color is the ball?"). Finally, completeness of definability and augmentation, which deal with defining new words and otherwise adding to a program's language capability, is lacking in MiliPS. The
completeness scheme just presented has not been discussed or applied elsewhere, to the best of my knowledge, so at the moment it is difficult to say precisely how MiliPS compares to other systems.

B.2. The organization and components of MiliPS

MiliPS processing is driven by a left-to-right scan across an input. At each scan position, a word is given a lexical class, adjacencies are checked to insure local grammaticality, and appropriate semantic processing, in a hierarchy of several possible levels, is done. The processing is thus bottom-up, with the number of levels above the lexical level that do processing dependent on particular conditions. Each level recognizes its applicability and acts accordingly, and its output may result in fulfilling the conditions of the next higher level. At each scan point, the maximum that can be known about the intention of the input is actually known (how this is useful is discussed in Section B.3). The following paragraphs give general information about the processing and organization, filling in details on each of the levels.

The main components are represented as very abstract Ps (VAPs)* in Figure B.2. In order to define and clarify those components, we will abstractly follow through the processing of Test 2, for which a detailed trace appears in Appendix D. Test 2 is "A BLUE BALL IS ON THE TABLE". The test is started by a "scanned" signal on the left end of the input string, a marker position to the left of "A". VAP SN then acts to cause "A" to be scanned. The "scan" signal is processed by an instance of VAP GR1, which in this case notes the initial "A" as signaling a sentence of type <SD>. "A" is classified as an indefinite determiner (its "word-class"). Next an instance of GR3 fires, verifying correct grammar for the word - in this case, "A" signals a noun phrase is starting, so that the grammar check is for correctness of a noun phrase at this point. A noun phrase is considered grammatical if it is preceded by: the word "THERE" if this sentence is type <SQE>; a relation word, i.e., a preposition; a copula ("IS" or "IS NOT"); or the left end of the sentence. When the determiner is processed, initialization is done for a new noun phrase (VAP NP). At this point nothing further can be done, and the scan resumes because of the "scanned" signal previously asserted by SN1 and stacked according to Psnlst's event order mechanism.

"BLUE" is tagged as an attribute-value word by an instance of VAP TG. This leads to the grammar check for attribute-value, which is a set of cases similar to the ones listed above for noun phrase. This particular case of attribute-value, because an indefinite determiner has preceded it, is not processed as in FR2, but is stored as a future restrictor on the new scene object to be created when the noun of the phrase is scanned. The scan continues, reaching "BALL", which is tagged as a noun by an instance of TG. The grammar is all right because it is preceded by an attribute-value. Specific noun processing is now done (VAP NP3), influenced in this case by the indefinite determiner. A new object, BALL-1, is added to the scene, and the remembered attribute-value "BLUE" is added as its color.

Once again, the scan continues, on to "IS". The word is tagged as a copula, is checked for grammaticality, and its action signalled (NP2). A noun-phrase boundary necessitates checks that all referents are determined for current objects (VAP BR8), since

* See Chapter IV for a description of the VAP notation.
8.2 Overview of MiliPS

SN: scanned(previous) & next-position -> scan(next) & scanned(next); [4 Ps]
TG: scan & particular-word -> word-class; [22 Ps]
ER: error-at-position -> collect-input-up-to-error-for-reply; [4 Ps]
ET: interesting-event -> print-external-trace-message; [9 Ps]

GR1: scan & particular-initial-word -> word-class & sentence-type; [7 Ps]
GR2: scan & particular-word & sentence-type -> word-class; [4 Ps]
GR3: word-class & lexical-adjacency & context -> word-class-action; [27 Ps],
where word-class-action = (determiner, copula, attribute-value, predicate, noun,
new-relation-open)

NP1: determiner -> initialize-new-noun-phrase; [4 Ps]
NP2: copula -> noun-phrase-boundary; [2 Ps]
NP3: noun -> create-new-scene-object OR restrict-referents; [7 Ps]

FR1: question-word OR definite-determiner
-> setup-possibilities-from-all-scene-objects; [4 Ps]
FR2: attribute-value -> restrict-referents; [2 Ps]
FR3: restrict-referents & single-matching-possibility -> refers; [1 P]
FR4: restrict-referents -> delete-non-matching-possibilities; [8 Ps]
FR5: predicate -> check-predicate-restriction; [1 P]

BR1: refers(new) & new-relation-open -> check-relation-restriction; [2 Ps]
BR2: check-relation(or predicate)-restriction & new-object -> add-relation(predicate); [2 Ps]
BR3: check-relation(or predicate)-restriction & feasible-to-restrict
-> restrict-referents; [6 Ps]
BR4: check-relation(or predicate)-restriction & relation(predicate)-is-redundant
-> backup-redundant-relation(predicate); [2 Ps]
BR5: check-relation(or predicate)-restriction & relation(predicate)-is-inconsistent
-> backup-inconsistent-relation(predicate); [4 Ps]

BR6: backup-redundant-relation(or predicate)
& some-previous-object-ambiguous-and-feasible-to-restrict
-> restrict-referents; [10 Ps]
BR7: backup-inconsistent-relation(or predicate) & preceding-object
-> check-relation(predicate)-restriction; [3 Ps]
BR8: noun-phrase-boundary
-> ensure-all-referents-found & update-current-current-object-pointers; [5 Ps]

MS: inconsistent(or redundant)-relation(or predicate) & sentence-type
-> add-relation(or predicate) OR answer-question OR error; [8 Ps]
VR: sentence-boundary & sentence-type -> reply OR describe-object; [23 Ps]
DO: describe-object & attribute's & relation's -> reply; [15 Ps]

Figure B.2 VAPs for MiliPS
restricting phrases are not allowed to restrict things across copulas, except in one case determined by special sentence type (<GSQW>). Because of this completion nature of a noun-phrase boundary, the only current object that is really current is the main noun of the sentence, so BR8 also includes the action of making other nouns non-current (there are no such others in the present example; they occur, for instance, in case there are relation phrases in the sentence). If there were some definite noun for which a referent had not been determined, an error would be noted at this point, keyed by the noun-phrase boundary.

The description of the remainder of the sentence, "ON THE TABLE", will be abbreviated somewhat, hitting only the new points exemplified. The relation "ON" is noted as referring in part to the current object, which is the main noun in the sentence, and also in part to an unscanned object, so it is left open (to be caught later by VAP BRI). The determiner "THE" is definite, causing the process of referent-determination to be initialized (FR1) by collecting a set of all the scene objects as possible candidates. Then "TABLE" is scanned, noted as a noun, and used to restrict the set of referents for the current object (VAPs NP3, FR4). In this particular scene, there is only one table, so that all objects except the table are ruled out by the noun "TABLE". This triggers FR3, which leads to BRI, and now the relation ON is completed, making it (BALL-1 ON TABLE-1). This in turn triggers the check for relation restriction, and VAP BR2 is applicable as a special case of restriction, simply adding the relation to the new object BALL-1. In most cases, it really would be a restriction, since it would be the case that the preceding noun would still be ambiguous, with a set of possible referents, and the new relation would serve to narrow down those possibilities. After the new relation is added, the scan continues to the end of the sentence, and a sentence boundary is signalled. This first acts as a noun-phrase boundary (BR8), making the subject noun the only one current. It then triggers the main sentence actions according to cases of VAP VR, which in this case causes the formation of the standard reply, "OKAY".

There are several aspects of the components of MiiIPS as outlined in the VAPs that have not been touched on by the above example. First, a "predicate" is recognized as an attribute-value preceded by copula, and is so tagged by the grammar check (GR3). It is further processed as a restriction similar to the restriction done when a new relation is formed as in the example above (FR5). That is, a predicate is an attribute-value that is placed after the noun that it restricts. The relative pronoun that precedes the copula (as in "which is" or "that is") is not used in this predicate detection, but its own grammar adjacencies must be correct, i.e., it must follow a noun or another predicate.

Second, the VAP MS represents what is done as a fairly high-level semantics process, namely it processes redundancies or inconsistencies as recognized by other semantic Ps according to sentence type. Some sentence types, as sketched in Section B.1, actually thrive on such anomalies. Third, the action of the BR VAPs has only been briefly touched upon, so we now turn to more detail on that.

As we mentioned at the beginning of this subsection, the semantics can be seen as a hierarchy of levels. These levels are reflected in the organization of the VAPs: the FR VAPs treat ambiguities of reference of noun phrases; the BR VAPs treat the assigning of relations and predicates to their proper objects, so that the best use of their information content is made in resolving ambiguities that couldn't be done previously by the FR's; MS...
is a last resort for handling inconsistencies and redundancies that can't be applied to ambiguities by the BR's; and VR and DO do the generation of replies based on the outcome of the other levels. As mentioned before, the main data structure used by the BR's to represent ambiguity is a set of possible referents for an object (noun phrase). The BR's use a structure composed of such sets: a linked list with the most recently-scanned object as the current one.

In finding a place to apply a new relation or predicate, the BR's always use the current object. If it is already unambiguously determined, an attempt is made to apply the relation or predicate to a previous object in the linked list. If the relation or predicate is redundant, a check is made before going ahead and trying to apply it to a previous object (BR6). That is, a check is made for the proper sort of unresolved ambiguity at some previous point in the list of objects. The check prevents irreparable damage being done on the basis of a feature whose resolution is not very urgent. If it is inconsistent, the application of it to some object is more urgent, so the backup to a previous object is tried regardless of what the result might be (VAP BR7). When such a backup is done, the linked list of objects is updated, making the preceding object the current one, and discarding the former current one forever (no later relations or predicates will be able to refer to it — to allow that would allow a strange sort of cross-over of reference, rather than the more ordinary nested reference, where a phrase refers to a close object, a later phrase refers to an object more towards the beginning of the input, and so on). Finally, the reader will notice that there is always a feasibility check before the actual restriction of the set of possible referents is done (VAP BR3). This is because the restriction process is irreversible, and maintaining that irreversibility seems desirable, the alternative being some kind of backtracking mechanism. If the restriction process were allowed to go unchecked, it might apply a restriction such that the entire set of possibilities would be thrown out, rather than recognizing a genuine inconsistency and acting accordingly. It seems reasonable to try to anticipate such conditions than to let them happen and then try to recover.

As support for the claim that no parse tree is formed, I now summarize the information that is kept as the scan proceeds across an input, and emphasize how that information is used to avoid referring back to the actual text after it has been seen once. The type of the sentence is kept (<SD>, <SE>, etc.), providing guidance for a few grammar decisions, but for the most part being used to make main semantic decisions. When an indefinite noun phrase is being scanned, the unused attribute-values are kept until the noun is reached, at which point they are added to it. When a relation is scanned, it is remembered until the noun phrase that follows it has been completed, at which point a full relation is formed (the noun phrase providing its second argument, in effect). The definiteness or indefiniteness of a noun phrase is incorporated into the representation and processing of the noun phrase immediately, even though the noun phrase is at that point quite incomplete. That is, the determiner sets up a group of noun-phrase anticipations. Question words and noun phrases are converted into sets of possible referents, discarding the lexical forms without further ado. For objects (representing noun phrases), the linked list records order of occurrence in the input, but objects are really semantic entities, no longer attached to lexical forms as would be the case in a traditional parse tree. This structure of semantic entities is the sole source of elements that are processed in making use of inconsistencies and redundancies. At no time does the scan back up and re-scan some portion of the input in order to try to assign to it a different interpretation, as is done in more conventional parsing programs (e.g. Winograd, 1972).
B.3. Production system and natural language task issues

This subsection discusses two independent sets of issues. The first set pertains to implementing various control and organization structures in PSs, to representational features, and to how the PS implementation compares to MILISY. The second set bears on the task and on more general processing of natural language: the use of adjacency checks instead of a full grammar, the determination of referents, and the need for a more sophisticated database.

The main control mechanism is the left-to-right scan across an input. At each scan point, the processing is bottom-up, based on successive recognitions of specific P conditions. This leads naturally to a vertical organization, in the sense that at each point, the maximum is known: all levels (lexical, grammar, semantics, pragmatics) have a chance to react as fully as possible. This allows the surface structure of the sentence to be discarded. Such vertical organization is less likely in systems where syntax and semantics are more sharply separated, and is of course ideally suited to the recognition-driven nature of PSs. There is a potential for top-down operation, since Ps could set up anticipations that might affect future recognitions.

Ps can be grouped conceptually in modules that treat similar features of the internal representation. The modules correspond to levels in the hierarchy (lexical, grammar, etc.) and to reasonable units within those levels. Generally a module acts by firing a single P, so that a module tends to represent with Ps the cases that elaborate the knowledge in the module.

At a somewhat lower level in organization, the scan uses the Panist :SMPX event-stacking mechanism to maintain control. It emits at the same time both a "scan" signal and a "scanned" signal, the latter being stacked until the former is examined ("scan" enables the lexical classification Ps). When "scanned" is examined, it moves the scan pointer forward or signals an error in case the "scan" signal has not been consumed.

There is another issue with respect to the initial left-to-right scan, namely, the way that a large number of Ps have the "scan" signal as a condition element. This gives a strong top-down flavor, or at least makes the Ps look like a big subroutine, rather than having them driven on more bottom-up specific recognitions. This may have an efficiency cost, but that is less important than the inflexible subroutine style. A more accurate model of language processing by humans, and a more suitable one for PSs, might be to have the input string encoded in some way such that only one element at a time would become available to the lexical Ps. Note that this is enhanced by the vertical organization discussed above, since that organization distributes the computation roughly evenly over the words. These elements would be quite specific and would presumably have very few occurrences in LHSs of Ps. (This would also work fine as a model of lower-level processes, where parts of words (phonemes or whatever) would be recognized to form a symbol representing the whole word, or the best guess at what the whole word is.) A further alternative might be to break the lexical processing into a hierarchy, with fewer Ps responding to "scan" and lexical classes of items, and with other Ps responding to the outputs from those lowest levels.

The tests for grammatical adjacency are carried out in similar fashion for all of the
classes in MiiliPS: there is a set of Ps that recognize correct adjacencies, plus a single P whose condition is the negation of all of the correct conditions, which thus recognizes an error condition. This is quite clumsy if the grammar is extended, because a new P must be complemented by an extra condition in the error P. One alternative is to use sequenced control signals as is done for the control of the scan, where the second signal would be deleted by each correct adjacency P, but would otherwise be recognized as an error. A second alternative is to implicitly order the Ps by special case, that is, a P that is a special case of another is before the other in examination order. Then the error P could be one with a single condition, keyed to the signal that initiates the grammar check; it would always be more general than the specific adjacency tests because they would include a test on the initiating signal plus the actual adjacency conditions.

Two peculiarities of Psnlist are used to advantage in MiiliPS. First, the F Ps (FR VAPs) in some cases fire "simultaneously" a number of times, both in generating possibilities for referents and in erasing those possibilities after further restrictions have been found. Without the automatic multiple-firing mechanism, some further control would be necessary to ensure iteration through all such firings. Second, the D Ps (DO VAPs) for describing objects are such that a set of objects can be described in "parallel" by having the Ps at each step fire a multiple number of times, one for each element in the set. This is similar to the multiple firing of the F Ps, except here there is a succession of such P firings by different Ps, whereas in the former case only a single P fired multiply. Here also, some explicit iteration control would otherwise be necessary. This kind of behavior is evident in those tests in Appendix C that involve describing several objects.

The primary representational issue in MiiliPS is the choice of representing things as Ps or as Working Memory structures. In particular, the way MiiliPS keeps the scene representation in Working Memory violates the principle that long-term items be stored as Ps. As it is, MiiliPS erases its entire Working Memory between inputs, except for the instances of a few select predicates which are its database and which stay around for the duration of a conversation (e.g., for the full set of 25 inputs on which MiiliPS was tested). To best represent the scene as Ps, some kind of discrimination network seems appropriate. This would necessitate radical changes to the present process of referent determination, since the present one forms a set of all objects in the scene, stored in Working Memory so easily accessible, and restricts the possibilities as more information comes in. The opposite method would be used if the scene were stored as Ps. As the input were scanned, a description would be formed, and as soon as the description became specific enough to evoke a scene object, a P would fire and supply a name to the description, thereby giving the system access to further information about it, to be confirmed or rejected by further inputs. The case of having evoked more than one such object would have to be considered, and some means of matching the objects in order to further discriminate them would have to be supplied. It seems that having conflicts between objects with respect to partial descriptions arise in this form and be treated according to a general matching discriminator is more satisfactory than the present Working Memory database from the standpoint of adding further contextual cues to the discrimination, e.g., time of creation and scene configuration dynamics. It seems more satisfactory in part because of apparent problems in getting hold of a large set of objects in Working Memory and examining them in such a way as to find descriptions that are indistinguishable and to

---

*Cf. the canonization of objects in GPSR, Chapter IV.*
find how partial descriptions of them might conflict. Storing them as Ps makes the conflicts fall out more naturally in the course of normal task processing, and sets forth a process whereby such conflicts are resolved incrementally. Some of the apparent difficulty with using Working Memory may be due to the nature of PS architectures or of Psnlst. Since discrimination nets are usually built to use a minimal number of tests to distinguish objects, it is likely that the P storage would use less computer memory overall, especially if there is some way of avoiding duplication of conditions in Ps by sharing the overlapping parts. The problem of how to store long-term information is of minor importance for the present study, which focuses more on natural language processing, so the present stopgap seems acceptable; other chapters of this thesis do focus on such storage problems.

Three other representational and low-level PS issues can be mentioned. Words are represented two different ways in Working Memory, as a consequence of limitations in efficient match power in Psnlst, namely limitations in the way constants are used in LHSs (see Section C.2). Also, many very similar Ps in the lexical recognition process could be reorganized into a set of Ps that simply recognize an element as a member of the set, plus a single P, keyed to membership in the set, that does the more complex actions now done in each P in the set. Augmentation would then be extension to that set rather than addition of a P. Some of the Ps in the description process (DO VAPs) could perhaps be more optimal by combining their actions into a single P with more actions and conditions. This is an instance of the general operation by which frequently-recurring P firing sequences are collapsed into a single firing that removes the necessity for intermediate communication signals, but that is more special-purpose. The specific case at hand is that two P firings are required to get a size-color attribute-value description constructed, where one would suffice. (At present, I am restricting such collapsings to Ps within the same module, but an automatic collapsing process might detect others.) Finally, the use of a near-total erasure of Working Memory between each input sentence has avoided the problem of inter-sentence confusion of data. Otherwise, special erasure Ps that would embody specific assumptions of what needs to be erased (and that would consume more run time) would be needed. The massive erasure is however unattractive from the standpoint of modelling a memory that fades over time, which is probably of concern to psychologists.

Several differences between the original MILISY and MiliPS are worth noting. MiliPS employs a single uniform mechanism to implement processing that was done by MILISY in two distinct phases: a syntactic parser and a set of semantic transformation rules. The use of PSs for both functions (although the functions have been radically redefined) indicates their flexibility and power over the particular special-purpose mechanisms in MILISY. MILISY constructs a phrase-structured tree representation of an input (or several, in case of ambiguity) and processes it semantically by rewrite operations capable of doing certain tests on the tree structure. It is not apparent whether its rules could be augmented to perform the semantic disambiguation that MiliPS performs, or not; the fact that MILISY might generate many possible parses before finding an appropriate one makes it more cumbersome at best. MiliPS makes significant extensions in MILISY’s behavior, especially in its ability to disambiguate, to handle subordinate clauses and phrases, and to answer "where" questions. MiliPS is about five times slower than MILISY (16 seconds versus 3 to 4), but MILISY would undoubtedly worsen in its performance on the more complex MiliPS tests. MiliPS is run by a PS interpreter, and compiling the Ps is expected to more than compensate for such speed factors. MiliPS has a listing about 2 to 3 times as
long as MILISY's. But both of these comparison measures are less than satisfactory because the two programs have diverged functionally.

Several issues can be raised in connection with the language task, which don't bear directly on the implementation as a PS. The local-adjacency nature of the syntactic checking in Milips may work only because the task is suitably restricted. Certainly, the present language doesn't contain all the basic components that unrestricted language does, but if the abstract toy blocks world does represent a significant portion of what natural language is about (objects, their relations, their attributes) then there might be some justification for trying to extend the approach to more demanding tasks. It is hard to envision a syntax system that requires less effort to carry out, except none at all. The weak syntax checking done here is justified as being a source of redundancy, preventing the system from taking action on too little input or on input not adequately structured, avoiding the possibility of irreversible undesirable actions on its environment. There are alternative approaches to doing the same kind of adjacency tests, which might turn out to be more suitable for other grammars, especially larger ones. One is to have Ps that reject bad adjacencies, rather than requiring a positive approval action. Another is to have more expectations set up, mixing top-down and bottom-up, rather than the pure bottom-up here. The possibilities for the kind of word following some word may be fewer than the possibilities for word classes preceding some word, and a mixture of the forward and backward strategies might minimize the number of required tests.

With respect to the process of referent determination, the present process forms a set of possibilities as soon as it sees a determiner-function word, whereas waiting for slightly more input would allow the process to start with an initially much smaller list. For example, the phrase "the" might refer to many more objects than "the blue". This strategy seems to be quite easy to implement as an extension to the present process. (This is a consideration regardless of whether the scene is in Working Memory or stored as Ps as discussed above.) The overall conceptual structure of ambiguity, inconsistency, and redundancy developed here, with the idea of keeping a linked list of current objects, seems general and natural, and thus worth pursuing in more demanding tasks. There are some choice points within that process that are currently not necessary, but might become so later. In particular, Milips makes use of redundant information to restrict wherever it can, but that restriction might turn out to be invalid after more input is scanned. This possibility doesn't arise in any of the present tests, and may be very rare in general. Also, the possibility of mutual disambiguation is not considered here, though it probably is necessary in general. By this, I mean for instance that two objects that are related to each other in some way might be ambiguous unless in both cases the relationship is considered. Another kind of disambiguation that is not handled arises when an unresolved ambiguity can nevertheless be used to resolve a previous ambiguity, such as might be the case in the phrase "the block on the table", where there are several tables but only one block on any table.

● But see Hays (1964) for a scheme with similar emphasis, proposed by a theoretical linguist.
● Pratt (1975) gives efficiency as a reason for using syntax; i.e., syntax is applied to ease some of the burden on semantics and pragmatics; such a consideration is not evident here because all of the ambiguities are among syntactically correct forms.
The specific organization of how redundancy and inconsistency are treated can probably be streamlined and made more flexible, now that the tests given to MiIPS have brought out a number of cases that were not envisioned in the original structure. For instance, having action depend on sentence type might be replaced by a more general component dependence, where components are present over a large set of sentences, i.e., where sentences can be classified more parsimoniously by using component features than by assigning each a distinct type. The present task is certainly restricted in that each lexical word can be interpreted in only one sense, whereas in general discourse, words must be disambiguated by lexical context or even more global considerations. Finally, the present system of disambiguation and referent determination assumes sentences are self-contained, for instance, with no pronouns or other (elliptical) references to phrases in immediately preceding ones. It is possible that most intra-sentence processing would stay intact in the face of that bigger demand, with only the need for “epicycles” to handle larger units of text. Certainly it is not hard to imagine that structures could be left open or with changeable default values, in the expectation that later inputs might fill them in. The present philosophy at the lower semantic level might be successful at larger levels: all input is converted to some internal form (for instance, surface structure of a string is not used after it has been passed in the scan), and any revision in initial expectations has to be done on that internal form without recourse to the raw external form. That is, a faithful internal representation should be amenable to mapping or restructuring in emergency situations. A form of such mapping is exemplified in the flexible way that MiIPS resolves inconsistencies using only its semantic representation.

The database inferencing capabilities in MiIPS have been intentionally kept very weak, partially because they were weak in MILISY and partially because of the emphasis on other aspects. Class exclusions on values of attributes, and relations between relations are not used. For instance, knowing an object is red doesn’t give the system the ability to use that it isn’t blue - “not blue” is only known if there is explicit information. The set of relations between objects might just as well be nonsense syllables, since they don’t interact and are not intended to be adequate in terms of representing all spatial properties.
C. Details on MiliPS

C.1. The tasks given to MiliPS

The entire list of sentences given to MiliPS is given in Appendix C. Included is the input text, a program trace that tells major events in processing the text, and the state of the database portion of the Working Memory, from which it can be deduced what the lasting effects of the text were. In this subsection, we first examine the program trace to make that appendix comprehensible. Then we point out other appendices that the reader might find to be of interest. Finally, the full set of sentences is described briefly in terms of what features are illustrated by various subsets of sentences.

Figure C.1 gives a segment of Appendix C. First, a display of the database is given. From it, we see that there are two objects, indicated by ISA, namely, BLOCK-1, a block, and TABLE-1, a table. The attributes of BLOCK-1 are color green and size large, given by HASAV, and similarly the table has color red. The next line, HASREL, tells that BLOCK-1 is on TABLE-1.

The next segment in the figure gives the trace that the program emits as it scans the sentence. The first two trace lines, starting with "ADDING" show what the program does when it scans the phrase "A BLUE BALL", namely, it creates an object BALL-1 (the second ADDING) and makes its color blue (the first ADDING). The next event of note happens when it gets to "TABLE", which it knows refers to TABLE-1, the third program trace line. After that, it finishes up processing the "ON", which was left hanging until the object following it was scanned. It notes that it adds the relation (BLOCK-1 ON TABLE-1) with the last ADDING line. Finally, its standard reply is made.
The database after the run is given, showing that it has added an instance to each of ISA, HASAV, and HASREL.

Figure C.2 gives the program trace only, for a more complicated example, to show a few other features of what the program emits. The first line after the input text shows the status as of the second word, which has been tagged internally as B2-1 (decoded: the second word, which starts with B, the first token for such a word). The phrase "THE BALL" has also been named OBJ-1, and the main point of the message is that OBJ-1 is ambiguous, referring at least to BALL-1 and BALL-2 (in this case, those are the only referent possibilities, but in general, more would exist, with the same message printed). Continuing, the next trace message says that OBJ-2, the name given to the second noun phrase "THE BLOCK", has a unique referent, BLOCK-1. This means that the ON relation left hanging can be completed, noted by the "RELRESTR" line. After the restriction has been done, the ambiguity for OBJ-1 has been resolved, making it refer to BALL-2. The scan continues, reaching the predicate "SMALL". It notes that this is inconsistent with the subject BALL-2 (referred to as OBJ-1), in the line starting with "PREDINCON". In that line, S7-1 refers to the seventh word in the text string, which starts with S, namely "SMALL". Since this is a declarative sentence, the inconsistency is taken in stride, that is, it is added to the subject as a new attribute-value, signalled by the ADDING line.

Appendix D gives a rather complete trace of the behavior of the PS on Input 2, including each P firing and the changes it made to the Working Memory. The reader should be able to follow it by using the description of that test given in Section B.2. At the end is a full display of the Working Memory. To understand the meanings of predicates, consult Section C.2; the program itself and a cross-reference are given in Appendix A and Appendix B. As mentioned above, Appendix C gives the program's behavior for the full set of tests. In addition, the portion after the third segment, tests 11-15, gives a summary of the control flow between groups of Ps (according to the first letter of the P name) for that test segment.

The full set of sentences is divided into five segments, for ease of debugging and presentation. The tests are given to the program via the X Ps, given at the end of Appendix A. The first segment, tests 1-5, consists entirely of declarative sentences, describing an initial scene. The second segment is four queries and one declarative
The queries illustrate some of the simpler descriptive capabilities of the system. The third segment has as its main new feature the use of "NOT", both in declarative and interrogative sentences. It should be clear from these tests that the way the program encodes and uses negation is rather primitive. The last two segments are similar. They illustrate the processing of much more complex sentences, with numerous ambiguities, inconsistencies, and redundancies to be resolved.

C.2. Meanings for the predicates in MiIPS

The descriptions in this subsection are given alphabetically by predicate. The predicates for the residual database are ISA, HASAV, and HASREL. Lexical classifications start with the letters "IS". Sentence types start with "GS". See the beginning of Appendix D for a sample of how an input sentence is represented internally. The trace itself in that appendix and the display of the entire Working Memory after the program finishes on that test should provide some clues as to which predicates might be of interest.

Predicate arguments in the following descriptions are typed according to the conventions:

- **a**: attribute: COLOR, SIZE
- **o**: object: BALL-1, BLOCK-3, etc.
- **p**: position in string: T1-1, B5-1, etc.
- **r**: relation: IN, ON, UNDER, and NEAR.
- **s**: sign: POS or NEG
- **t**: temporary object token: OBJ-1, OBJ-2, etc.
- **v**: value: LARGE, RED, etc.
- **w, x, y, z**: arbitrary.

1. **ADDAV(a,t)** add new attribute values for t to new object a.
2. **ANSPRED(a,v,s)** answer a question according to the result of testing whether the predicate (a,v,s) is true of o. (V, M)
3. **ANSPREDIN(a,v,s)** the predicate represented by (a,v,s) is the final word of a sentence. (V, A)
4. **ANSPREDRED(a,v,s)** a potential ANSPRED is redundant. (V, M)
5. **ANSREL(a,r,o2,a)** answer the question according to whether (o1,r,o2,a) is a true relation. (V)
6. **ANSRELINC(a,r,o2,a)** the relation (o1,r,o2,a) is inconsistent, so answer accordingly (depending on sentence type). (V, M)
7. **ANSRELRED(a,r,o2,a)** a potential ANSREL is redundant. (V, M)
8. **AVRESTRI(t,p,a,x,s)** restrict the possibilities for t by applying the restriction that it be (a, v, s). (F, A)
9. **COStGNA(a)** is the sign of the most recent copula (R, G)
10. **CUROBJ(t1,t2)** t1 is the current object, and t2 is the previous current one. t1 and t2 may be also o1 and o2 by type (A, R, N, F, B, M, V, G)
11. **CUROBJP(t1,t2)** t1 and t2 are previous CUROBJ pairs (B, M, V, G, N)
12. **DLFDET(t)** a definite determiner is at p (N, G)
13. **DETFIND(t,p)** find possible referents (INDOSS) for t, at p (F, N)
14. **DESCRAV(a,v,s)** describe o by attaching to the list x the value for s of the attribute a, if any. (D)
15. **DESCRIBED(o)** describe o by finding and concatenating all of the (a, v, s) properties for o. (D, V)
16. **DESCRIBEDRED(o,v,s)** o has been (partially) described using (a, v, s). (D)
17. **DESCRNK(a,s)** a2 follows a1 in the predetermined order of describing the attributes of an object (DESCRIBE) (D)

* Letters in parentheses after a definition are initials of P groups in which the predicate is used.
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRPHRASE(p,v,d)</td>
<td>is the final output phrase describing (DESCRIBE) a. (V, D)</td>
</tr>
<tr>
<td>DETSEEEN(p)</td>
<td>at p there is a determiner, either definite or indefinite. (A, N)</td>
</tr>
<tr>
<td>ENDMARK(p)</td>
<td>p marks the left or right end of the input string. (S, T, E, A, N)</td>
</tr>
<tr>
<td>EQx=xx(p)</td>
<td>the word at p is equal to xxx. (T, G)</td>
</tr>
<tr>
<td>ERROR(p,v)</td>
<td>an error has occurred at p; x is a list to be added to the reply. (E, S, A, R, P, N, F, B, M, V)</td>
</tr>
<tr>
<td>ERRORS(p,v)</td>
<td>error scan from right to left is at p, collecting a list x. (E)</td>
</tr>
<tr>
<td>ERRREF(t,p)</td>
<td>for reference in case of error, t is at p. (E, B, N, G)</td>
</tr>
<tr>
<td>FINDAMBIGP(t1,p,a,v,s,t2)</td>
<td>link backwards by CUROBJP relations to find a place with remaining ambiguities to attach a redundant (a, v, s). t2 is where the search started, t1 is the current place in the search, and p is the location of the (a, v, s). (B)</td>
</tr>
<tr>
<td>FINDAMBIGR(t1,p,a,v,s,t2)</td>
<td>like FINDAMBIGP, but for a relation (r, o, s). (B)</td>
</tr>
<tr>
<td>FINDPOSS(t,o)</td>
<td>o is a possible referent for t. (F, B, V, M)</td>
</tr>
<tr>
<td>GSDx(z)</td>
<td>z is a sentence of type SD, a declarative sentence. (N, M, V, G)</td>
</tr>
<tr>
<td>GSE(z)</td>
<td>z is a sentence of type SE, declarative starting with &quot;there&quot;. (M, V, G)</td>
</tr>
<tr>
<td>GSOD(z)</td>
<td>z is a sentence of type OD, the question form of a D type of declarative (GSD). (A, M, G)</td>
</tr>
<tr>
<td>GSQEC(z)</td>
<td>z is a sentence of type SQE, the question form of the E type of declarative (GSE). (G, N, F, M, V)</td>
</tr>
<tr>
<td>GSQW(z)</td>
<td>z is a sentence of type SQW, the question form starting with &quot;what&quot;. (N, F, B, M, V, G)</td>
</tr>
<tr>
<td>GSQWR(z)</td>
<td>z is a sentence of type SQWR, a question starting with &quot;where&quot;. (M, V, G)</td>
</tr>
<tr>
<td>GSYBED(z)</td>
<td>z has been typed according to GSN, GSE, etc. (G)</td>
</tr>
<tr>
<td>HASAV(o,a,v,s)</td>
<td>o has value v for attribute a, sign s. (E, F, B, V, D, M, N)</td>
</tr>
<tr>
<td>HASREL(r1,r2,s)</td>
<td>r1 has the relation r to r2, sign s. (E, F, B, V, M)</td>
</tr>
<tr>
<td>HASRELNC(r,t,a,s)</td>
<td>t has the relation r, sign s, to some object yet to be seen in the input. (B, R)</td>
</tr>
<tr>
<td>INDEFDET(p)</td>
<td>an indefinite determiner is at p. (N, G)</td>
</tr>
<tr>
<td>ISA(o,w)</td>
<td>o is an object of the class w. (E, F, D, N)</td>
</tr>
<tr>
<td>ISAV(p,a,v,s)</td>
<td>the attribute value (a, v, s) at p checks out grammatically; continue to process it as such. (A, N, F)</td>
</tr>
<tr>
<td>ISAVW(p,a,v)</td>
<td>the word at p is an attribute value (a,v); this signals the need for a grammar check. (A, T)</td>
</tr>
<tr>
<td>ISCOPE(p,a)</td>
<td>the word at p is a copula, sign s. (G, A, R, N, T)</td>
</tr>
<tr>
<td>ISDEF(t)</td>
<td>t is known to be a definite object by its determiner. (A, N)</td>
</tr>
<tr>
<td>ISIINDEF(t)</td>
<td>t is modified by an indefinite determiner. (A, N)</td>
</tr>
<tr>
<td>ISNOUN(p,w)</td>
<td>the noun at p, word w, is grammatically all right; initiate further processing on it. (A, R, P, N, G)</td>
</tr>
<tr>
<td>ISQOUNW(p,w)</td>
<td>the word at p is a noun, w, this signals the need for a grammar check. (G, N, T)</td>
</tr>
<tr>
<td>ISPRED(p)</td>
<td>the AV at p is (see ISAV) is a predicate, which means it follows a copula. (A, R, P, F)</td>
</tr>
<tr>
<td>ISREL(p,w)</td>
<td>the relation word w at p is all right grammatically; continue to process it. (R, N)</td>
</tr>
<tr>
<td>ISRELPRON(p)</td>
<td>the relative pronoun at p is grammatically all right; initiate the normal processing for it. (P, N)</td>
</tr>
<tr>
<td>ISRELPRONW(p)</td>
<td>the word at p is a relative pronoun; proceed by checking whether it is grammatically all right. (P, T)</td>
</tr>
<tr>
<td>ISRELW(p,r)</td>
<td>the word at p is r; this signals the need for a grammar check. (R, T)</td>
</tr>
<tr>
<td>LEFTOF(p1,p2)</td>
<td>p1 is to the left of p2 in the input string. (S, T, E, G, A, R, P, N)</td>
</tr>
<tr>
<td>MAKISA(p,w,t1,t2)</td>
<td>make t1 at p into an ISA; its word is w, the previous object is t2. (N)</td>
</tr>
<tr>
<td>NEWAV(t,a,v,s)</td>
<td>record (a, v, s) so it can be attached to the actual object that t represents, when it becomes determined. (N, A)</td>
</tr>
<tr>
<td>NEWOS(o,a)</td>
<td>o is a new object (new ISA). (F, B, N)</td>
</tr>
<tr>
<td>NPBOUND(p)</td>
<td>a noun-phrase boundary is at p. (B, S, N)</td>
</tr>
<tr>
<td>NPOUOND(p)</td>
<td>delete the NPBOUND signal for p. (B, N)</td>
</tr>
<tr>
<td>NPOCHK(p)</td>
<td>check that it is grammatically correct to start a noun phrase at p. (N)</td>
</tr>
<tr>
<td>NRESTR(p,w)</td>
<td>restrict the possibilities for t at p to be nouns of class w. (F, N)</td>
</tr>
<tr>
<td>NULLREF(t,p)</td>
<td>the set of references for t at p is empty. (F, V)</td>
</tr>
<tr>
<td>OCHR(p)</td>
<td>check if the possible referents for t have been restricted to a unique or null set. (F)</td>
</tr>
<tr>
<td>OLOAV(p)</td>
<td>the AV at p is old, ISA has been responded to. (A, F)</td>
</tr>
</tbody>
</table>
OLDREF(t) the REFERs for t has been examined. (B)
OLDREL(p) the relation at p has been processed; ISREL has been responded to. (R)
PREDINCON(t,p,a,v) the predicate (a, v, a) is inconsistent with t at p. (B, M, E)
PREDINCON(t,p,a,v,s) print a trace for and assert the corresponding PREDINCON. (E, B)
PREDREDUN(t,p,a,v) the predicate (a, v, a) is redundant for t at p. (B, M, E)
PREDREDUN(t,p,a,v,s) print a trace for and assert the corresponding PREDREDUN. (E, B)
PREDRESTR(t,p,a,v) restrict the possible references for t at p according to whether (a, v, a) is true. (F, E)
PREDRESTR(t,p,a,v,s) print a trace for and assert the corresponding PREDRESTR. (E, B)
PREDRESTRCHK(t,p,a,v) check whether the corresponding PREDRESTR should be applied. (B, F)
QOUN(p) the noun at p is a question noun. (G, T)
QWFIND(t,p) find possible referents (FINDPOSS) for t at p. (F, G)
QWRDESCR(t,x,w) initiate the second step in the reply generation process for QWR sentences (see GSQWR and QWRREPLY). (V)
QWRREPLY1(o,x,w) the current phrase in building the first part of the QWR answer (see QWRREPLY1) for object o is x, with word w used to separate further additions to x from the present x. (D)
QWRREPLY2(o,x,w) like QWRREPLY1, but for the second part of the QWR answer (QWRREPLY2). (D)
QWRREPLY3(o) generate the third kind of reply for a QWR sentence, which covers the case where o has no relations to other objects. (D, V)
REFERENCES(o) t refers to o, t may also be of type o. (F, B, M, V, N)
RELINCON(t,p,r,o,a) the relation (r, o, a) is inconsistent with t at p. (B, M, E)
RELINCON(t,p,r,o,a,s) print a trace for and assert the corresponding RELINCON. (E, B)
RELREDUN(t,p,r,o,a) the relation (r, o, a) is redundant for t at p. (B, M, E)
RELREDUN(t,p,r,o,a,s) print a trace for and assert the corresponding RELREDUN. (E, B)
RELRSTR(t,p,r,o,a) restrict the possible references for t at p according to the relation (r, o, a). (F, E)
RELRSTR(t,p,r,o,a,s) print a trace for and assert the corresponding RELRESTR. (E, B)
RELRSTRCHK(t,p,r,o,a) check whether the corresponding RELRESTR should be applied. (B)
REPLY1(x) x is a list of words constituting an external reply. (V, E, D)
SCAN(p) the scan is positioned at p. (S, T, G)
SCANFIN(p) the scan is finished at p. (S, V)
SENTBOUND(z) the sentence boundary has been reached for sentence z. (V, S)
SENTENCE(z) z is the current input sentence. (S, G, N, F, B, M)
TEXT(x) x is the list of words in the input string. (S)
TRACING(x) an indicator that a program trace is being printed. x is a dummy. (S, E, F)
WORDEQ(p,i) the word at p is equal to xi. (T, G, E, N)
WBlox is a PS that solves blocks manipulation problems, taking commands from an augmentation of MiliPS and performing actions on the scene in order to fulfill the commands. This section and the next give an overview of the WBlox part of the system and then more details, respectively. Section D.1 presents a few examples of the problems solved by the system. Section D.2 sketches the changes made to MiliPS in order to handle the expanded task domain. Section D.3 discusses the goal-subgoal mechanism used to solve problems, and describes the way backtracking works, allowing choices to be tried, undone, revised, and tried again. Section D.4 through Section D.6 discuss issues with respect to the particular PS implementation, and with respect to implementation-independent features of the task domain that were elucidated by the present work. Section D.7 compares PSs with the original Micro-Planner implementation.

D.1. A few examples of WBlox tasks

WBlox starts with a toy blocks scene identical to that used by Winograd (1972), namely, a tabletop with a box and a variety of rectangular blocks and rectangular-based pyramids. The test sentences given to the MiliPS/WBlox system were designed to test the blocks problem-solving capabilities and exercise as many of the Ps as possible. This contrasts with Winograd's apparent preference for exercising only the natural language capabilities (though not necessarily exhausting all of them) and only using those parts of the blocks program that were evoked as a result of that. Thus what is presented here and more fully in the next section and Appendix H is a more complete demonstration of the blocks problem-solver designed by Winograd than was given by him.

The first input sentence is a simple command to put one object on another.

**INPUT 1:** (PUT THE SMALL RED BLOCK ON THE BLUE BLOCK)
The MiliPS part of the whole system recognizes that the small red block is not already on the blue block, i.e., that there is a serious inconsistency in the sentence. Because it involves a relation that can be associated with the PUT command, that inconsistency becomes the intent of the sentence, and is given to the problem-solving part of the system. In the initial scene, the small red block has a pyramid on top of it, so that the first problematic part of this command is to find another place to put the pyramid. This evokes the goal to GETRIDOF the pyramid. GETRIDOF in general first searches on the table for an empty place, then looks at blocks in the scene to see if space is available there. In the present case, it has no trouble finding space on the table, and proceeds to move its hand to the pyramid, grasp it, lift it to some random location within the clear region on the table that it selected, and let go of it. Now the pyramid is out of the way, so the program looks for space on top of the blue block. The blue block is all clear, and is big enough to accommodate the red one, so the program goes through a sequence of grasping, lifting, and so on, similar to that for the pyramid, to put the block in that clear space.

**REPLY 1:** (1 (OKAY))
The MiliPS subsystem responds OKAY after checking that what was commanded has actually been accomplished by the WBlox PS. Outputs are tagged with integers ("I" here) in case there is a set of replies, to provide a sequencing for them.
0.1 Overview of W8lox

We now skip over two inputs, one asking a question and the other commanding that a green block be put in the box.

INPUT 4: (PUT THE GREEN BLOCK ON THE BLOCK IN THE BOX)

Looking at this superficially, it is ambiguous in a couple of ways. At the command level, it appears ambiguous because the system knows two ways to PUT, namely IN or ON, so that the input may be requesting a PUT... IN or PUT... ON action. This ambiguity is resolved by normal processing of the sentence: the IN phrase is needed to resolve the reference to "THE BLOCK", so that only ON remains as a candidate for the main command action. The superiority of the bottom-up approach over a top-down one is evident here, and the difference between the two can be accentuated further by adding more relations. The second ambiguity is presented by "THE GREEN BLOCK". There are two green blocks in the scene, but fortunately, both are referred to in this sentence: one is in the box, so it is the second block, which forces the ambiguity of the first one to be resolved in favor of the other one. This other green block is not on the first one, the one in the box, so that the inconsistency is taken as the intention of the command, and the W8lox part of the system can work on the specific problem posed. This problem is solved directly by moves similar to those used in the first INPUT above, since no other objects are in interfering locations. The program's reply is the same as in the preceding example.

For the next example, we skip a few inputs that had no effects of concern to us at present.

INPUT 12: (PUT A SMALL PYRAMID AND A SMALL PYRAMID AND A GREEN BLOCK AND THE SMALL RED BLOCK ON THE LARGE RED BLOCK)

Several things of note occur in the input. The use of "A" in a command causes the system to choose from among a set of existing objects that match the given description, rather than creating a new object as was the case in MilPS alone. In fact, in this case it chooses two pyramids, taking care to make the choices distinct. The use of "AND" means that all conjoined objects are the main ones for the command, that is, the command works with a set of objects. The command is to put the set on the large red block, since the final phrase, starting with "ON", is inconsistent with the scene.

From the point of view of the problem-solving system, this command presents difficulties because all of the specified objects will not fit on the large red block unless some of them are piled on top of each other in some way. W8lox does not recognize ahead of time that the area isn't sufficient, but rather, attempts to put them on, trying a couple of variations in arrangement (which exhausts the possibilities in this case), before deciding to try the necessary packing operation. When working with a set of objects, W8lox tries to place the largest first, then the next-largest, etc. In this case, after placing three of the four objects, the space is filled, so it backs up and tries to put the third object in a different location. This fails because the third object filled up the only available space. It then backs up further and tries to put the second object in a different location. Now the second and third objects used up a rectangular region on the large red block, each filling up half of it, and the program always tries to pack objects closely together when it is putting a set of them somewhere, so that there is really no alternative place to put the second object either - packing implies using the lower left-hand corner of the region. (The program doesn't reason in this way, exactly, but tries to locate space and finds only the point already seen.) So it backs up to the first object, and can find no alternative place for it either, for similar reasons. Thus it has backed up to its starting place, and now it pursues an alternative strategy, called the PACK strategy, which says
place an object, then try to put one other object on top of it, then place the next object, and so on. It puts the first object on the large red block, then puts the second object, a pyramid, onto the first object, then puts the third object onto the large red block, and the fourth on top of the third.

REPLY 12: (1 (FAILED TO PUT PYRAMID-3 ON)) (1 (FAILED TO PUT PYRAMID-1 ON)) (2 (OKAY))

The program replies that the two pyramids aren't strictly on the large red block as it had expected, and then says OKAY anyway, because some of the things it expected were fulfilled. (The first two replies are tagged identically because they were noticed "simultaneously"). The two pyramids were in fact placed on the two blocks that were placed on the large red block (pyramids being preferred by PACK for placement on top of just-placed blocks, since nothing can be put on a pyramid).

This time inputs not shown have had the system put some more things in the box, and had it add some new black blocks to the scene. It has just picked up one of the black blocks.

INPUT 18: (PUT IT IN THE BOX)

"IT" always refers to the object in the hand of the model, by convention. There is no trouble understanding the input, but severe problems in carrying it out. The program fails to find enough clear space in the box to put the block that it's holding, so it tries a drastic strategy: clearing out all the things in the box, and putting them back in in PACK mode, placing them all as closely together as possible. As above, the PACK operation includes putting every other object on top of one just placed rather than on the box proper. It succeeds, after about 65 subgoals and 70 primitive grasp, lift, and let-go actions (about ten times more than required for INPUT 1 above). The program responds simply OKAY as above.

The final example we consider here consists of building a stack of objects.

INPUT 19: (STACK UP A LARGE RED BLOCK AND A SMALL BLOCK AND IT AND A SMALL PYRAMID AND A BLACK BLOCK AND A LARGE GREEN BLOCK AND A SMALL PYRAMID)

In stacking up a set of objects, the program first chooses the largest block as the base of the stack and places it on the table. As its next step, which is repeated until all the blocks have been placed, it selects the largest block that hasn't been placed and puts it on the top of the stack (the block in the set of things to be stacked that has nothing on top of it). In this step, if the largest block that hasn't been placed is too big, it is left out, and the next one selected instead. Also, if there are two or more blocks that are the next-largest, and if one of them is already in the right place, it is left there and the process continues to the next (the program also notices if the base of the stack is already on the table when it starts). After all blocks are placed, the program selects the biggest pyramid from the set that will fit, if any, and places it. Any other pyramids must be left out.

REPLY 19: (1 (LEFT OUT PYRAMID-3))

The program checks for completion of the command by checking an internal representational set that records stacks of objects. This stack record is kept for all object movements: whenever one object is put on a block (table and box are excluded as stack members, by this definition) it becomes a member of the block's stack, or if the block wasn't in a stack, a new stack is created with both objects in it. For this reply, the program noted that one of the pyramids is not in the same stack as all the other objects that it was to stack up. This is right, because the command was not completely fulfillable,
given that pyramids can't support other objects. MiliPS could in principle recognize such ill-formed commands, but it doesn't.

D.2. Changes to MiliPS for the WBlox task

Appendix E gives the portions of MiliPS that changed in converting it to translate the external language into inputs for WBlox. This subsection describes the changes, following roughly the order of their appearance in that appendix. Most of the changes, 70%, were additions of Ps, and the rest were minor changes to existing Ps, usually changing one condition or action element. No Ps were deleted. There are three main kinds of changes: lexical and grammatical changes, which are rather minor; changes to how relations are handled, adding two new varieties of relations, indirect ones and computable ones; and changes to main sentence semantics in order to interface to the blocks problem-solving Ps. After describing the changes, the varieties of blocks commands are described, along with details on main sentence semantics for them. Finally, the changes in internal representation of the scene are sketched.

In the tagging Ps (T Ps) are all of the changes that effect modifications to the acceptable language. The system now knows about PYRAMID where it used to treat BALL. To make that change, only two Ps were changed, one a T and one an N, the N that handles creation of new scene objects. The word IT is recognized as a noun phrase, and is taken always to refer to the object in the model's hand. This requires only a single P, which does all the actions necessary to make the system believe that a noun phrase just went by. This approach was taken as the easiest way to ensure that objects in the hand could be referred to uniquely, the problem being that such objects don't have the same relations to other objects that other objects do. It was easier than adding the code necessary to make use of phrases like "in the hand" or "that you are holding". IN and ON are now tagged as indirect relations, to be discussed below, and TO THE LEFT OF, TO THE RIGHT OF, BEHIND, IN FRONT OF, ABOVE, and BELOW are recognized as computable relations, also discussed below. The new prepositions UP and DOWN are also recognized, but they are only lexically treated as relations, and are otherwise just complementary modifiers for command words.

The G Ps have a number of changes relating to main grammar types. These changes also carry over into N Ps and B Ps, some of which are discussed here, others later. First, blocks commands are a new type of sentence, the imperative, or <Sl>, called GSI internally. In such imperatives, "A" is taken as meaning a choice is to be made, as opposed to the old action of creating a new scene object. The actual choice is made by B Ps. The imperatives start with a particular set of command words, PICK, GRASP, STACK, and PUT; G Ps recognize these and assign the imperative type to the sentence at hand. At the same time, these words set up expectations of complementary modifiers, for instance, PICK expects UP somewhere, PUT may be followed by DOWN, etc. "AND" is recognized as a noun-phrase boundary and is used to conjoin only main sentence objects in imperative sentences. The grammatical-adjacency tests for noun phrase were rewritten to make control cleaner and augmentation easier - augmentation now requires only the addition of Ps, not also the addition of negated conditions in a P that recognizes bad conditions. Similar changes could have been made to other such Ps, but one illustration is sufficient, and the others didn't require modification anyway.
In the F Ps, the relation restriction process, by which relations are used to restrict possible referents, is split into two stages to handle a peculiar kind of ambiguity in imperatives. The command PUT expects some kind of inconsistency to occur, so that it can turn that into a command to be fulfilled, but this can interfere with the determination of referents when there is a relation that might be interpreted as both a valid restriction and an inconsistency. That is, a relation might be true of one possibility, while another possibility exists for which the relation is not true. Given the two distinct interpretations, the process assumes the relation is to be used as a normal restriction, but saves the other possibility as something that can be used in case no other inconsistency can be found. Test sentence 16 illustrates this kind of "backup".

The way that the new classes of relations are handled shows up in changes to F Ps and B Ps. Computable relations are the ones that depend on exact locations in the scene, for instance, IN FRONT OF (that locations are now exact is discussed below along with other representational changes). When these relations are completed, that is, have definite objects to which the relations are to be applied, a B P evokes a set of F Ps that assert temporary relations into Working Memory that represent specific computable relations. For instance, when "TO THE RIGHT OF THE LARGE PYRAMID" is scanned, assuming only one large pyramid, a computation is made to determine all objects to its right, and temporary representations of all of the resulting TORIGHTOF relations are asserted. These relations are used to restrict other referents in a way similar to ordinary relations and to indirect relations, to be discussed now.

Recall that the "check-relation-restriction" process (see Figure B.2), which is B Ps, checks to make sure a relation restriction is applicable before going ahead with it. In that process, when a relation that is tagged as indirect is encountered, Ps are evoked to compute temporary indirect relations from the specific relation that is the subject of the check. Indirect relations are the transitive closure of a relation, and are computed by the B10 Ps. For instance, given "IN THE BOX", a transitive closure is computed using ON, by asserting indirectly-IN for all objects ON objects in the box, and for all objects indirectly-IN, and so on. The relation ON is also given the same treatment, propagating indirectly-ON's. The actual referent-restricting Ps (F Ps) are augmented by a set of Ps that use these indirect relations in a way similar to the way the restrictions for normal relations were used before. The indirect relations are erased from Working Memory after each input sentence is finished (along with everything else except the representation of the scene). An alternative that would have required fewer added Ps would have been to assert normal relations and some record that certain normal-looking relations are really temporary, so that they could be explicitly erased at sentence boundaries. These temporary relations would then enter perhaps into blocks manipulation updating operations and into the process that describes the scene and its objects - it is not clear that this is desirable.

Now that there is provision for such indirect relations, any further classes of relations that are to be treated as temporary need not require further Ps to be handled properly. The present program has an example of this, in that computable relations are kept in the same form as are indirect ones, and don't require mechanisms beyond the initial assertion. Ultimately, if the scene should be represented as a more long-term entity in the Ps themselves, all Working Memory relations would be temporary, so that further decisions would have to be made as to differential treatment of types of temporary relations.
0.2 Overview of WBlox

The M Ps have two types of changes, reflecting new main semantic action. The new <SP> imperative sentence class occurs in several P conditions that want to restrict the class of sentences to which they apply. The M60-M8Os are specific Ps added to process <SP>-specific information and issue commands to the blocks problem-solving Ps. Within these, redundancies and inconsistencies are treated according to the new conventions required for imperatives, to be discussed further below.

The V Ps also have a couple of modifications and augmentations. There is a set of Ps that handles reply generation for imperatives, which includes checking that commands were actually carried out. Replies themselves are now numbered, so that textually identical descriptions can be distinguished, for instance the two "LARGE GREEN BLOCK's" in the reply to the sixth test sentence. The count of replies is initialized at the beginning of the scan by a TP.

There are four commands that are extracted from input sentences and issued to the WBlox Ps. The PICKUP command is obtained from sentences of the form, "PICK ... UP ... ", where either "...
" may be empty in particular cases. For this form, referents of objects must be exact. The program checks that it is not already holding in the hand the main object in the sentence. This form will not take compound phrases, since the hand can only hold one thing at a time.

The PUTDOWN command is obtained from sentences of the form, "PUT ... DOWN ... ", where either "...
" may be empty. As for PICKUP, referents must be exact, and further, the object referred to must be in the hand. Actually, all such forms can simply be expressed as "PUT IT DOWN".

The PUTON command comes from forms "PUT ... ". The PUT can be matched to either ON or IN (the latter only goes with the BOX, and becomes a PUTON that is processed specially in some cases). This form may take compound main nouns. The system processes all such as a set, applying a single relation to them all. The specific relation to be applied to the main noun or nouns is obtained from an inconsistency in the sentence. At present, this is restricted to IN and ON, but in principle it should apply to any relation, with the intent of the command to make that relation true (the restriction is inherited from Winograd's program). The explicitness of inconsistency considerations here makes that kind of extension quite feasible, whereas it is not clear that such a general mechanism would arise naturally from Winograd's treatment (whatever it was in this case). If an input contains a redundancy but no inconsistency, or if it contains neither, it is a redundant command and requires no action; the program in the latter case will complain, but in the former will say OKAY.

The STACKUP command comes from sentences like the PICKUP one, with STACK instead of PICK. These forms must have compound main nouns, and the referents must be exact.

Finally, we sketch the representational changes necessitated by the addition of manipulations to the scene, done by WBlox. The primary change is that objects have specific spatial locations and sizes, according to a standard three-dimensional coordinate system. As in Winograd's system, an object can't be rotated, and is always rectangular and aligned with the coordinate axes. The location of an object is the location of its
Overview of WBlox

D.2

lower-left-hand corner (minimum x, y, and z values). There is now a hand in the scene, represented as a point with neither size nor attributes nor relations to other objects, except that it can be grasping or empty. All relations are now assumed to be positive (POS), where in MiliPS, distinction was made between POS and NEG. To have negatives would be to allow a certain vagueness that doesn't fit with exact locations (although ultimately it might be desirable, for a fully general system), e.g., "NOT IN THE BOX" would have an object seemingly floating freely at any location not on the box's surface. (This is, I believe, independent of whether "NOT" can be handled in inputs, which it now cannot be.) There is a new structure that is kept track of in the scene: the stack. A simple stack is just a set of objects, one on top of another up to some height. The generalized notion of stack here is that an object is in a stack whenever it is on top of an object in a stack. A stack is created whenever an object is placed on top of another that is not already in a stack (except the table and the box). Thus stacks really include tree-like structures of blocks – all blocks in such a structure are in the same stack.

D.3. The main components of WBlox

For the most part, the WBlox Ps work independently, as a submodule, of the MiliPS system. The language produces a single command or a set of instances representing a command on a set of objects, which evokes specific WBlox top-level Ps, which in turn evoke the full problem-solving system. When the problem solving is finished, the top-level goal succeeds and control falls back to some checking signals, left around when the WBlox Ps are evoked, which evoke a process that checks the results and forms a reply.

There are four top-level operators that are evoked from outside the WBlox system: PICKUP, which commands a specific block to be picked up; PUTDOWN, which commands a specific block to be put down on the table or wherever there is space available; PUTON, which commands that an object or a set of them be put on some other specified object (PUTON includes putting things in the box); and STACKUP, which commands that a set be stacked, one on top of another.

There are eight subordinate operators that are used by the top-level ones and by each other as subgoals to accomplish particular action sequences. PUTONI puts a single object on another object; PACK puts a set of objects onto an object, under the constraint that they are to be packed as closely as possible. GETRIDOF involves finding some unused space to put an object and going through the actions that put it there. CLEAROFF uses GETRIDOF iteratively to clear everything off some object. PUT takes an object and places it at a specific location. GRASP attaches the hand to an object, sometimes necessitating a CLEAROFF so that it can do so, as well as an occasional GETRIDOF for what the hand is already holding. RAISEHAND computes a location above where the hand is, and moves it there. MAKESPACE tries to clear away just enough objects from a surface to free up space to fit a particular object.

The preceding set of operators all make use, ultimately, of a small set of primitive operators, which do the actual changes to the scene model and which do not further evoke other actions. MOVEHAND moves the hand from one location to another, doing all the necessary updating to object locations, to IN and ON relations, and to stack structures. MOVEHAND fails to do the motion if the location moved to is not clear to the extent
required for the object that the hand is grasping. UNGRASP causes the hand to let go of
an object it's holding. The converse of UNGRASP is to assert that the hand is grasping, an
action that is a subpart of the GRASP action and not separated as a named primitive. The
most complex primitive in the system is FINDSPACE, which is sometimes entered at one of
its subordinate steps, LOCATESPACE. FINDSPACE scans the surface of a specific object to
find an open region suitable for placing another object. It is the only primitive that fails
explicitly with a signal that is then processed in specific ways by the evoking process.
Further levels of primitiveness can be imagined, but they weren't implemented here or in
the original system being imitated. For instance, MOVEHAND could involve computing actual
trajectories for the motions, so that no collisions with other objects occur. These
considerations are simply assumed to be always solvable and not touched on further here,
although it is conceivable, for instance, that the trajectory computation might not be
possible without further rearrangements of blocks.

Figure D.1 gives an outline of how the blocks commands interact. The components
of the outline structure in the figure are the operators. Arguments for the operators are
given in parentheses, and comments are given in square brackets. In form, the structure is
an AND-OR graph, with connections of nodes to other nodes in the graph indicated by
comments "above" and "iterates". This connection notation is modified to mean a copy of
the structure with modifications, when such modifications are also given in the comment,
e.g., "without MAKESPACE" is such a modificational comment. In numbered sequences, AND
is implicit between steps, e.g. 1 AND 2 AND 3 under PUT. OR is given explicitly and means
the step in question has alternatives, if the OR is between two steps with the same
number, or if means the sequence of steps preceding the OR has the steps following it as
an alternative, if sequence numbers differ directly before and after the OR. One ambiguity
with this definition of OR is under PUTIN, where 1 is to be alternated with 1 AND 2
following the OR, not 1 AND 2 OR second 1 AND 2. The comment "primitive" indicates
primitives in the above categorization of operators. The comment "iterates" means that
the iteration is to be through the set in the immediate vicinity, until the set is exhausted.
Details on how the various selections and primitives work, and on how sequencing is done
in particular cases will be presented in Section E. The remainder of this subsection makes
general comments on organization.

Most of the components given in Figure D.1 work within a set of conventions that
make up a goal-subgoal mechanism. The top-level goals are commands from the input
language via Milips. Subgoals arise as the components or operators encounter difficulties
in being immediately applicable. Specific problems that can arise are encoded as Ps that
recognize difficulties, and that then construct the appropriate subgoals. Sequencing of
both the AND and OR types is by using a couple of specific goal-related signals, one of
which (the predicate NEXT) specifies what to do if a subgoal succeeds (AND), and the other
(the predicate NEXTF), what to do if a subgoal fails (OR). If neither NEXT nor NEXTF is
given, the goal that evoked the subgoal succeeds. There is a small executive (5 Ps) that
processes success and failure signals according to these conventions. The primitive
operators in the system are not treated within these goal conventions because their
operation is immediate, so that sequencing can be done with ad hoc evoking-process-
specific signals. The same executive-avoiding mechanics are used for steps within goals
that don't cause difficulties otherwise.

The justification for including the executive and goal-sequencing conventions is that
in all but the simplest problem situations goals of the same type are evoked recursively, though there are intervening levels of goal structure between the recursive calls. That is, goals do not directly evoke themselves as subgoals, but most situations give rise to recursive nesting in some way. If in these nesting situations, a particular goal process
Overview of WBlox

relied on ad hoc signals for sequencing, there would be more than one instance of some signals, causing confusion between the two processes. Thus, goal status for separate invocations of the same goal are distinguished with an extra argument that names the goal. Also, the NEXT and NEXTF sequencing predicates contain within them inactive versions of signals that are to be asserted, so those signals are effectively hidden and can’t interact with information from active goals. If the Psnlst interpreter distinguished between matches to a P on the basis of recency of data being used in a match, and fired the P only using the most recent data (saving others until they eventually become the most recent), then the goal executive mechanism would not be necessary. (This architectural variation has been seriously considered as an interesting PS alternative.) But Psnlst, given a P with any match at all that it has come to consider for recency reasons, fires all the instantiations it can find, old and new alike. The recursively-nested structure of Planner control isolates separate goal contexts effectively, although it hides them much moreopaquely (making access to other contexts impossible) than is the case in the present PS implementation.

It is fruitful to briefly compare the present solution of goal–subgoal management to that found in the more general situation, namely in GPS (the General Problem Solver, a version of which is described in Chapter IV of this thesis). The present system is very specialized, with Ps that recognize specific differences, obstacles to success with a goal, and that construct and evoke specific appropriate subgoals to treat those differences. Thus a single P firing combines the workings of the GPS match and the table of connections, between differences found and operators that might reduce them. In all cases, a difference has a unique operator that is effective. Differences are local features of the scene, so that there is no need for GPS’s general match, which would want to work on two different versions of the scene (actual and desired). The closest analogue in GPS would be the performance of matches to a described, abstract object, which contains only a few features of the scene that are relevant to the main goal. But with the present high degree of specialization goes a loss of flexibility in applying operators and in using methods. The operators are very specific, and are encoded to include their own fixed subgoal sequencing. The lack of general treatment of goals and methods means that the executive doesn’t evaluate progress and shift problem-solving efforts accordingly. There is also no provision for recognizing infinite loops of goals. Certainly, looping in blocks problems is possible in general, but it may be that the present restricted operator structure can not give rise to loops, although it would if it persisted in a reasonable way in trying to attain a goal.

One detail in the dynamic behavior of the system that is hinted at in Figure D.1 by the comment “choicepoint” is the management of alternative selections within operators. Winograd’s original implementation made use of Planner language primitives to ensure that all such alternatives would eventually be explored, according to a strictly depth-first search organization. That is, whenever at certain goal points alternatives existed, information as to the nature of those alternatives was recorded, and if some failure occurred at some later time, the system would back up, undoing all effects in between the

Example: if an object, A, is to be put on object B, but has object C on top of it (i.e., C is on A), and if the only available space to put C to GETRIDOF it is on the targeted space for A on B, and if the only available space to put C is back on A when the program attempts to MAKESPACE on B to put A, then there is potential infinite oscillation.
failure and the most recent goal with alternatives, and would choose another alternative on which to base forward action. PSs have no such mechanism built into the architecture, so it has been necessary to adopt conventions for setting up necessary information so that alternatives can be explored in a similar way, and to code those explicitly wherever necessary. On analysis of the structure of the task, it was decided to designate only a very few locations in the search as such choicepoints. The reason why this required analysis is that the Planner code for the blocks problem solver makes very frequent use of the particular primitive that achieves this mechanism (THGOAL), but only a few uses of it are actually necessary to ensure proper backtracking, the others being used to provide other functions of THGOAL. Section D.7 will go into more detail on how the final search behavior differs.

The primary function of choicepoints in WBlox is to record the current state of goals with alternatives, and to record which alternatives have already been tried. The only choicepoints in WBlox involve locations where objects are placed. If there seem to be other meaningful alternatives in terms of the task, they have here been reduced to location choicepoints. Further, the only part of the system's actions that is recorded so that it can be undone in the act of backtracking, is the sequence of primitive actions performed, along with, for some goals involving a set of objects to be iterated through, a record of the state of the iteration (i.e., which things in the set have been tried). All other goal information, for instance the goal-subgoal structure and what has succeeded or failed, is irrelevant to the backtracking and is simply disregarded in backtracking. That is, for the most part when the system backtracks, it simply reverses the sequence of hand motions and grasping and ungrasping actions that it has done since the most recent choicepoint. Whenever one of the primitives is performed, it records an event time, an integer that is incremented each time such an event occurs, and when a choicepoint occurs, the current event time is associated with it so that the backtracking can reverse the right actions. Each primitive action is also responsible for asserting an element that says what its opposite is, so that the action can be undone. The action reversal goes through the same mechanism that is used in the forward direction, e.g. the MOVEHAND primitive is evoked, so that all the proper bookkeeping is done automatically (invisible to the backup controller).

Further details on the implementation of choicepoints will be given in Section E. Even though choicepoints have been fairly easy to implement, reducing backtracking to manageable proportions, the strict depth-first variety of backtracking used here and in the original program is not considered the best way to proceed, either in this task or in general. The particular position that the PS philosophy implies on this issue is discussed further in Section D.4.

D.4. Production system issues

The next three subsections consider the issues that arose in WBlox with respect to PSs, with respect to the language used to converse about blocks, and with respect to the problem-solving operators. Included in the first is a discussion of the suitability of backtracking as a method within a PS implementation, and what an alternative problem-solving structure might look like. Also included are features of control and organization, and a discussion of some time and space efficiency characteristics of the system. Then (Section D.5) we go on to consider in detail the extensions that would be necessary to
Overview of WBloc

bring the system up to the level of competence of Winograd's system on the natural language side. Finally (Section D.6), there is a discussion of some details of the blocks problem-solver, independent of the implementation as a PS, which suggest difficulties and possible significant improvements in its abilities.

The most important issue with respect to PSs is the suitability of the backtracking method inherited from the Planner version of the problem-solver. Backtracking implies that there is provision to ultimately try all possible variations in sequences of problem operators in attempting to solve a problem, if that should be necessary. These alternative sequences are tried in depth-first order, and in Planner there is little program control over which alternatives at any point are tried first. In the toy blocks domain, this has proved to be no strain on the control capabilities of PSs, although analysis has simplified somewhat the amount of backtracking that is really necessary, and, further, certain features of PSs as a language, to be discussed in Section D.7, remove some of the control needs that backtracking is used for in Planner programs.

Nevertheless, for this domain it seems feasible to adopt a strategy that requires no backtracking or backup of any kind. Such a system would always work forward from its present situation, adjusting to problematic situations by applying problem-solving methods that attack those problems directly, after analyzing to find the real causes of the problems. For instance, instead of doing backtracking within GETRIDOF, which searches among alternative locations for putting an object in an out-of-the-way place, problem operators could be applied to do direct blocks rearrangements to alleviate shortages of available space. In such a scheme, the history of the choices made in attempting to solve a problem becomes global, and is no longer associated with particular choicepoints in the goal structure. For instance, all operations that have been performed on an object, and in particular where it has been placed, would be available for examination by GETRIDOF in the process of finding somewhere else to put it. Such a strategy might produce plans for actions that are non-optimal in the sense that the same object is handled several times, each shifting it to a new location, but it is judged easier to analyze such plans after the main goal has been achieved, to smooth out such (rare) rough edges. I don't know of any real exploration of the consequences of such a strategy, although the approach is similar to the kind of information-gathering discussed by Newell and Simon (1972, chapter 12) in connection with human problem-solving behavior in playing chess. Such a scheme is not foreign to the constructs included in the Conniver programming language (Sussman and McDermott, 1973). A primary component of such a strategy is a fuller system for analyzing and describing what is problematic about a situation, and for linking such a description with available methods.

Further analysis of how things are tried in the present backtracking structure could improve WBloc's problem-solving ability, or at least efficiency, and perhaps eliminate or minimize the amount of backtracking necessary. WBloc includes all of the selection that was used by Winograd to improve the search behavior, with perhaps minor improvements in a few places (to be discussed in Section E). For example, it orders sets of objects so that the largest object is considered first, in placing them somewhere. But further orderings could improve the process even more, for instance, allowing GETRIDOF to always make best use of available space by using the smallest space large enough to accommodate an object. More details on where this is possible will be given in Section D.6. PSs are advantageous in this kind of improvement due to the power of selection inherent in LHSs of Ps.
The present implementation of choicepoints (see Section D.3) illustrates how PSs might be applied to problem-solving situations in which backtracking is necessary, either because not enough analysis has been done to allow more intelligence to be built into the problem solver as discussed above, or because genuine choices do exist. In such a general case, the PS architecture allows several variations on the scheme, according to task demands. One is to use P Memory instead of Working Memory to record the choicepoints, to save Working Memory space (and matching overhead) and perhaps to avoid interference between similar information at different choicepoints. In recording choicepoints, there is always a choice between storing what has been tried and what remains to be tried, which in WBlox was resolved in favor of storing what has been tried. When the task requires much more of the choicepoint mechanism, namely keeping track of entire memory contexts to return to, as in Conniver (Sussman and McDermott, 1973), PSs offer at least two alternatives also. Presumably, it is not best in such cases to use Working Memory to store the alternative data contexts. Ps can be used to store entire states as RHSs or sets of RHSs, to be made current by the proper evocation into Working Memory. Ps can also be used to store update information, so that going from one state to another previously-stored one is done by a sequence of P firings, each making incremental updates to the current Working Memory state. For both of these, some method of storing path information, or other evocation cues, must be adopted, so that states can be accessed. For this, in principle, either Ps or Working Memory could be used.

The overall control organization of components of WBlox is as a hierarchy, along the lines given in Figure D.1. The processing is directed by explicit goals in Working Memory, and intra-goal sequencing is done by specific ad hoc control signals. In terms of modules of Ps, which conceptually means Ps that share common knowledge assumptions, the entire system is divided roughly according to the first letter of Ps' names, but in the WBlox part, modules are larger than is warranted by conceptual organization: all of the higher-level goal parts are in the W module, and the primitive operators are in the Q module. But given that, it is still the case that generally, the action of a module consists of firing very few Ps (one, two, or three, usually), which perform some actions and pass control to another module's Ps. This is true of most of the modules in the MilIPS part, and is at least partially true in the WBlox part. In WBlox, on the average, one WP fires, then about three QPs fire, then control goes back to a WP. This is based on figures given in the control flow summary trace in Appendix H, after the first program trace segment. This supports the claim that PSs lend themselves easily to a modular organization of knowledge, and are the right level of conciseness to express incremental applications of such knowledge modules.

PSs are used to advantage to do a variety of complex selections within single LHSs. Several processes order a set of objects by size by using an LHS that performs a match on the set and selects the largest for its next action. Some of these make the selection under the constraint that the object will fit on top of some other object. (Details on which ones make such selections will appear in Section E.) The MAKESPACE process selects an object that is the smallest one large enough to accommodate another object. FINDSPACE uses single-P selections to find greatest lower bounds on a region along X and Y dimensions, and to find least upper bounds on the two dimensions. That is, given a point in a clear region, it selects the object that forms the closest boundary of the region in a particular direction. It also uses such selections to shift its attention from a point that is obstructed by an object to the nearest point on its boundary, which may adjoin on a clear region suitable for further examination. (FINDSPACE will also be discussed further below.)
All of these selections would be clearer to express if Psnlst had an additional simple match primitive (see Chapter VII). As it is, the expression of such selections is sometimes awkward and repetitious. But at a higher program-organization level, it might be better to have a selection module or goal, rather than having each problem operator do its own selections. Having the separate selection would be warranted if it were to become more complex, e.g., based on history or on considerations other than simple local ones or on interactions with other goals.

A variety of control sequencing devices are used in WBlox. Iterations in PUTON, STACK, and PACK are controlled by signals that record the processed elements in the sets that the operators are working on. Simple match conditions exclude these tried elements from being considered in the selections involved in these processes, and the signals are noted in the same way as primitive hand actions, so that backup can take them into account. FINDSPACE uses modifiable defaults in computing boundaries of a region, which means that as a first attempt at a boundary a default value is used, and then Ps may or may not fire according to conditions, to update those default values. Later Ps make use of the existing values without then having to be concerned with where they came from. Double signals for controlling steps in a process are used in several places: in FINDSPACE, in some grammar adjacency checks, and in checking the results of the whole blocks process. That is, a P evokes one step of a process and at the same time asserts a signal that at the proper time (when it pops out of the examination stack for events, SMPX) evokes a P that asserts a signal that starts the next step of the process. This device avoids having the next step evoked prematurely from intermediate results from the preceding step. A disadvantage is that the control signal must be included in the Ps of the second process that may accidentally suffer from premature firing, or usually all of them, to avoid having to know too much in advance. In the cases at hand, this is not a serious problem, since the second step is one P or a small number of Ps.

The generation of the transitive closures of the IN and ON relations takes advantage of Psnlst's ability to fire a P on several sets of data "simultaneously". In this case, a set of Ps amounts to a breadth-first assertion of the indirect relations in the scene, since at each iteration of the set, all the existing indirect relations are extended by another link in the chain or network. This process simply continues until no more new relations are asserted, at which point control falls back to another signal and processing continues (see the B10 Ps).

Bookkeeping after hand moves is done under the control of specific signals. When the hand moves holding an object, relations that the object had are no longer correct, and new relations may now hold, so that checking is done in two distinct steps. Without specific control of these two steps, for instance, newly added relations would be deleted by the step that deletes the existing relations in preparation for any new ones. The program actually started out without specific controls, and was found defective.

As was the case for MillPS alone, everything in the Working Memory is deleted between input sentences, except for instances of special (by convention) database predicates. This removes the need for more careful updating and erasing of unnecessary elements, preventing interference between sentences (which wouldn't necessarily occur), but is unsatisfactory in being rather arbitrary. More reasonable schemes such as having elements automatically deleted after being unused for some number of recognition cycles...
are recommended by this as well as other PSs implemented so far, but cannot be explored in practice within the present scope. Another ad hoc mechanism in WBox is the PSMacro MAKEINSTL (see P WO), which converts the value bound to a variable in an LHS match to be an assertion at the top level in the RHS. This circumvents a deficiency in the Penist language (not allowing variables in predicate position, and not allowing matching of nested structures), but is justified in two ways: it is used sparingly, and it is very convenient in converting data that would otherwise require a set of specific Ps, one for each type of conversion done, according to the particular predicate in the assertion.

Over the 24 tasks given to WBox, run times range from about two minutes up to about 40, with all but one actually under about 10 minutes, and with the average at 4.5 minutes. (There is good reason to believe that the 40 minute figure may be inflated by computer system characteristics at the particular time the run was made, by as much as a factor of 2, based on average run time per P firing, which is ordinarily about 1 second, but in that case close to 2.) The PS uses a total of about 48K words of Lisp cells, and one of the longer tests (19) uses about 5.5K for its dynamic Working Memory, of which about 2K is taken up by the residual database portion. Of the 48K in program, 27K is for the MiliPS part, 21 for WBox. The full PS has 408 Ps, including 3 test Ps, of which 278 are in MiliPS and 130, in WBox. Since the old version of MiliPS has 193 Ps, including 5 test Ps, 85 Ps were added to bring MiliPS up to handling the richer input language. Test 19 has a Working Memory of slightly over 400 instances, of which the database is about 100 items. In that test, even though the total number of items is large, no single predicate has a large number of instances, the most heavily loaded (with about 40) being UNEVENT and NEXT, which are concerned with backup information and goal sequencing, respectively, and which could easily be stored as Ps if it were necessary to reduce the size of Working Memory.

D.5. Extending the language system

There are a number of specific features that could be added to the present system, if it were desirable to bring it to the level of competence of Winograd's original system. In fact, many of the features discussed here go beyond the original, but seem to be within reach of the PS. MiliPS is much weaker than the original in its ability to generate interesting replies. MiliPS has no capabilities to answer "why" questions, which involve knowledge of the problem-solving history that has preceded the question. Some related aspects are being able to use past tenses, being able to deal with queries about actions, and being able to use relative time descriptions such as "the first thing you touched after stacking up the red blocks". MiliPS doesn't know certain verb forms that bear on relations that it has, e.g., "what does the box contain". It also needs to be able to understand some variants on relational phrases, for example, "the block that the pyramid is on", and to be able to deal with the converse of being "in" or "on", namely the support and containment concepts. MiliPS has very little in the way of treatment of pronouns or references that depend on the history of the conversation. MiliPS doesn't handle "and" in a general way, restricting its use to conjoining subjects of commands. The present language can't deal with certain aspects of the internal representation: sizes, locations, and stacks.

MiliPS lacks an ability to handle numbers, as in "stack up three blocks" or "supported by three boxes", and it can't answer "how many" queries. This involves being able to recognize plural forms of nouns, to enforce agreement between nouns and verbs,
and to recognize more general uses of conjunction, which at present is limited to the main nouns of the input. MIIPS would have to be extended to handle negation, which in particular involves some extra Ps in the referent-determination process, that would restrict the set of possible referents in an opposite fashion to the present positive restrictions. This suggestion assumes that it is more reasonable and general to assume that all database attributes and relations have a positive sign, as was assumed here, rather than allowing both signs as in the original MiIPS. If general propositional logic is expressible in natural language, to process it in the present framework would require manipulation of sets of possibilities and their complements, and possibly saving partial results for use in restoring previous interpretations on the basis of new input. For example, in "on the block or to the right of the block", the first candidate relation might make the set of possibilities empty, so that the second alternative would have to be tried with the set that existed before the first phrase was seen.

MIIPS is less interactive than SHRU, specifically lacking the ability to lay out choices in an ambiguity situation and allow the user to specify in a simple way which one was intended. It can't augment its language ability as could SHRU. SHRU was able to attach proper names, e.g., Superblock to objects, and it could converse about a previously-unseen concept like "ownership" or a new structure of blocks like "steeple".

MIIPS lacks an ability in many cases to rule out interpretations purely on the basis of semantics, as opposed to pragmatics, as was used in the original blocks system to rule out having the table try to pick up blocks, for instance. An exhaustive examination of the possibilities of occurrences of various kinds of relations in commands, namely whether a particular phrase is used as a restriction of possible ambiguity, as a redundancy, or as an inconsistency to be applied elsewhere, leads to some cases that weren't judged to be common enough to warrant attention in MIIPS, but that might be desired in a fuller system. One case contains phrases that are all inconsistent with the main noun, but that are at varying levels of specificity with respect to being turned into the command relations to be fulfilled by the system. For instance, in "put the pyramid in the box on the red block", suppose the scene contains no pyramid in the box, and that there is a red block in the box. In this example, both relations are inconsistent with the main noun, and both could be commands, but the second is more specific and consistent as a command with the first, and should thus be preferred. A second case involves a redundancy that might be inconsistency with the main noun, but is subsequently superceded by a real inconsistency. Thus bindings of relations to be command relations has to be tentative in some cases, with possible updating after more of the input is seen.

How feasible is it to make these extensions? Adding to the grammar of the language accepted is relatively easy, involving just adding grammatical classes and figuring out the appropriate adjacencies to be checked. Eventually, under pressure from complex languages, it might be better to systematize and generalize to the extent of using some kind of case-based structure for grammar expectations, analogous to the current way that a "pick" command expects to contain an "up" somewhere. Also as structures get more complex, the variety of sentence types might be systematized so that processing depends not on those types but on classes of types or on attributes of types, e.g., sentence types in which an indefinite determiner should be taken as a choice, as in present imperatives. The plausibility of being able to extend the present system is supported by the completeness assertions in Section B.1, and also by the relatively clean system of treating things as
ambiguities, redundancies, or inconsistencies. The number of Ps estimated to be required for such an extension is in the vicinity of 200-300.

D.6. Blocks problem-solving issues

The present blocks operators closely parallel Winograd’s, but it is useful to discuss them with a view toward extension, and for the purpose of raising more general problem-solving issues. One feature that was discovered in the course of testing was the possibility of interference between goals. The particular instance of this phenomenon occurs in a few situations where the program finds space to put an object, then evokes a subgoal to grasp the object, and in the process of grasping it, manages to place another object in the target location. This occurs in the problem-solving connected with inputs 18.0 and 24 (five times in the latter), which will be discussed in Section E, and it occurs only within a CLEAROFF operation, which has GETRIDOF as a subgoal, which in turn evokes PUT which evokes GRASP, which may evoke another GETRIDOF to place some object that is in the way of the GRASP goal. Apparently no other locations in the goal-subgoal structure have such a combination where interference can occur. The trap is that the FINDSPACE is done before it is certain that all other objects are in a proper location for the follow-up operation. This problem was corrected accidentally by the program itself without specific modification, due to the iterative structure of CLEAROFF: it checks the existing situation on the object being cleared off each time it iterates, essentially double-checking previous attempts, and not assuming that those previous attempts were successful. MOVEHAND checks the target location for clear space for an object being grasped, and does nothing if the location is occupied. In the original program, if such a thing occurred, the failure to PUT the object in the space would have caused a failure, with backtracking to try to do something (blindly) to correct the error. Even though in the specific goal structure here the problem is not serious, it is the case in general that some provision should be made for such interfering goals, at least providing for some communication of intentions. In the particular space problem here, one solution, used by Sussman (1973, Section 4, pp. 88-90), is to establish “ghost” objects that occupy space but can’t be manipulated as ordinary objects. There is one other approach in the present case, a trivial change that rearranges the sequence of operators so that the FINDSPACE is done after the GRASP is finished, which is the subject of an experimental patch to the WBlox system, discussed in Section E.3. But the general problem of goal interference deserves further attention.

As discussed above, backtracking is considered not the best approach, especially for PEs, where it is possible to add as much guidance as desired. For the toy blocks domain in particular there are improvements that might eliminate the need for it altogether. A couple of things should be investigated as improvements along this line. Both considerations deal with the placement of objects in empty spaces, which process grows as the factorial of the number of objects to be placed, under the backtracking strategy used in the original blocks system. Several processes presently choose to work first with the largest object in the set of objects that they’re working with, but the way that “largest” is determined is by taking the sum of their length and width, which is the metric used in the original. This might be improved by using area, by using the larger dimension, or by some measure dependent on context (for instance, when putting objects in a space narrow in width, width would be a more important consideration). Choosing the right largest object is important because such routines as PACK assume that using the largest
object first will guarantee being able to fill the space, if any arrangement at all satisfies that goal.

The second consideration to eliminate backtracking is probably more important, namely, using available empty spaces, particularly on the table, more effectively. This assumes a more global view in FINDSPACE, which will be discussed below. One trick is to use a space for an object that is just large enough to accommodate the object, but that minimizes the extra space that is wasted because the object doesn't fill it completely. Some care must be taken here with shapes of spaces, since in the present system, spatial orientations of blocks can't be changed (for instance, they can't be rotated 90 degrees). Care is necessary because two spaces might be equivalent for one object, but for another object, only one of the spaces is right due to its shape. Another consideration is that before spaces are filled in some processes, a better idea must be obtained on what objects in the scene will ultimately have to be moved to allow the main goal to be attained. In some cases, this requires a rather exhaustive pre-examination. For instance, in STACKUP, it may be necessary to move only small objects off of blocks that are to go near the base of the stack, but later it may be necessary to get rid of a larger object that is presently on top of one of the blocks to go near the top of the stack. Along the lines of allocating space optimally, there are conceivably a number of heuristics, applicable in special situations, which could help guarantee a minimum of backtracking, for instance, taking account of specific sizes and shapes to fill odd clear regions. In some cases, it might be possible to anticipate the need for PACK, rather than trying the ordinary PUTON first, such as when a set of objects has too much area to fit on a surface without it. Note that in the present task, there are no esthetic considerations, nor are there practical constraints such as putting tall blocks toward the rear of the scene so that they're less likely to get knocked over in moving the arm around. These constraints might be applied to distinguish apparently equivalent locations under the criteria above.

Two things about choicepoints in WBlox deserve mention. First, they are not exactly the same as the ones that are logically present in the original program (by my examination of the Planner programs; it is difficult to tell exactly because the THGOAL primitive is used in many places that aren't choicepoints in the sense used here). In two places in the original, a set of objects was processed using the backtracking mechanism, rather than sorting the set by size as was used in other places in that program, and which corresponds to the selections used in WBlox. That is, an object would be picked at random, say from all those on top of some block, and if later processing based on that choice failed, backup would come back and cause another to be picked, and so on. Also, for the goal interference problem discussed above, the original would have failed some subgoal, causing backtracking, rather than letting the iterative nature of an operator do the double-checking as in WBlox. These differences will be discussed in more detail in Section D.7. The choicepoint mechanisms in WBlox are presently distributed in specific form in several places, rather than having a general mechanism used by the various operators that need choicepoints. The same approach is used to record specific primitive events that are backed up (undone) when a failure occurs. If there were a common process used by all choicepoints, perhaps some of the work now done in various places that requires things to be expressed with several Ps could be expressed more concisely, particularly things that have to do with evaluating whether to go ahead with a particular choice or whether to reject it, say, because it duplicates a previous one or because a numerical limit has been exceeded for such attempts.
The present FINDSPACE process returns the first suitable region found after a random selection from the points on a surface have been examined. At each such random point, a process applies to try to find the largest clear region surrounding the point. Although details appear below (Section E.2), it suffices here to point out that such a random basis leads to a program that is hard to debug because behavior is rarely reproduced reliably. It is based on the FINDSPACE in the original program, but in the course of development, several minor improvements have been made, and some major possibilities for further improvement are now evident. FINDSPACE could function best by searching a grid of points in the region, where the grid need not be any finer than the size of the object that is to be placed. For the smallest block in the present scene, the grid for the table would be 100 points, ranging down to less than 20 for a majority of the blocks. Most of the grid points would be rejected immediately due to being located inside an object already on the table. More would be included in regions already found, so that the actual work of examining the space around a point would probably be required for fewer than the maximum of 10 random points that are now examined. The process would then be guaranteed to find space if it existed, rather than the present arbitrary cutoff after 10 points (which are generally not in 10 distinct clear regions). The most sensible strategy would be to find the clear space once (especially for the table and the box, which usually have a lot of space and are used frequently as locations for other objects), and to keep the list of regions globally available and updated when objects are moved. Alternatively, rather than updating, a new invocation of FINDSPACE could first check grid points in regions that existed at the previous invocation.

D.7. Comparison of WBlox to the original Planner version

The two programs are apparently quite similar in behavior, although there are a few minor differences that arose to keep mechanisms within WBlox similar in design philosophy. There is one major qualification to comparisons of this sort: detailed behavior traces are not available for the original program, especially on the kinds of tests that are used here to verify that everything in the program is in good working order. Also in at least one case the program code was too obscure to attempt to duplicate its actions too closely, so an informed guess was made as to its function.

One behavior difference has to do with where choicepoints occur in the program. In the original, as mentioned before, when MOVEHAND failed because the movement caused one object to overlap the space of another, a failure resulting in backtracking occurred, whereas WBlox recovers by iterating the main goal that gave rise to the MOVEHAND command. (Actually this would apply to PUT in the original, which duplicated the overlap check in MOVEHAND, but not in WBlox.) The failure in the original could thus result in retrying some choicepoints before getting back to finding another place to put the object. The CLEAROFF operation in WBlox applies a selection by size to the objects on top of an object that need to be cleared off, whereas the original simply had a loop that selected at random, subject to backtracking choices. Thus WBlox has no choicepoint in CLEAROFF, where the original did. Similarly, in MAKESPACE, WBlox uses a selection by size, where the original relied on backtracking to correct any stupid choices.

The PUTON operation in the original program, when working to put a set of objects on another object, simply tried once to put the set on, in some arbitrary order, and on
failure proceeded to try to PACK them on. WBlox selects items from the set by size, largest first, and when PUTONI fails, tries to find alternative locations if possible before giving up and using PACK.

There are two differences in the hierarchical structure of the blocks operators, between the two versions. GRASP in WBlox does GETRIDOF, for an object in hand, before doing CLEAROFF of the object to be grasped; the two operations were done in the opposite order in the original. (WBlox follows Winograd's book here, which disagrees with the available Planner code, Card, et al., 1972, which was used to obtain details.) UNGRASP in WBlox includes support checks that were part of PUT in the original. UNGRASP refuses to let go of an object if it is unsupported, whereas the original would refuse to PUT it at an unsupported place. It turns out that UNGRASP never fails anyway, in WBlox, since other operators are sufficiently careful where they try to put things. As mentioned above parenthetically, WBlox has no check for object overlap in PUT, but only in MOVEHAND, whereas the original had it (redundantly) in both places. One minor difference between the two is that when the original does select objects from a set according to size, it sorts the whole set once, and uses the sorted list result thereafter, where WBlox simply selects the largest remaining object each time it examines the set.

The basic strategy in programming the present version was to take advantage of the selective power of the PS rather than to rely on a weak and inevitably stupid process such as backtracking to arrive at an appropriate sequence of actions. It is probably true that PSs are more suitable to situations where specific knowledge can be applied to help the program make appropriate selections, than to situations where the only available method is a weak exhaustive search.

Superficially, the two versions have some similarities. The lengths of the listings of the two programs are almost identical, both around 950 lines, although the PS listing looks more densely packed onto the page. The original program consisted of about 105 Planner theorems and Lisp functions, whereas WBlox has 130 Ps. But in the computer, WBlox uses 21K words, where the Planner version used 8.8K. One of the larger scenes for WBlox used about 2K words, where the original used 1.3K, but for a slightly smaller scene, so the two are similar in scene storage. A major contrast is run time, since the original ran in 5 to 20 seconds, as compared to about 60 times that for the PS. This is distorted in Winograd's favor by several problems given to WBlox that were intended to cause considerable problem-solving, perhaps a factor of 5 to 10 times more than any of the original ones. Thus the adjusted efficiency difference is within the order-of-magnitude improvement that is expected to result from efforts to compile Ps.

On a statement-by-statement basis, the main conclusion reached by comparing the contents of Planner theorems and Ps is that a Planner theorem, with several conditional accesses to its database, and with backtracking ultimately trying all the possible paths of execution through such a procedure, corresponds to several Ps, with each one representing one of the conditional steps in the Planner theorem. (To explain why the numbers above are so close, it needs to be pointed out that there are not many Planner theorems that convert to several Ps.) Figure D.2 gives a direct contrast between the two modes of expression. Alternatively, if actual conditional cases are few, a set of Ps can represent all the conditions and actions for all the possible execution paths through the theorem. For this alternative, some cases can usually be logically excluded, because some
combinations of conditions, corresponding to paths, are not meaningful. Also, some of the Planner backtracking search is invisible at the surface level in P LHSs, hidden within the PS match.

```
theorem TC-Cleartop(consequent Cleartop(x));
    begin; local variable y;
        if not Support(x,? then assert(Cleartop(x)) also succeed(theorem);
    Loop;
        if goal(Support(x, ←y)) then goal(Getridof(y),use(TC-Getridof)) also go Loop
        else assert(Cleartop(x)) also succeed(theorem);
    end;

W3: clearoff(g,x) & supports(x,y) & not supports(x,object-bigger-than-y)
    & not supports(x,object-same-as-y-and-lexically-greater-than-y)
    -> newgoal(g1) & getridof(g1,y) & next(g1, "clearoff(g,x)");
W6: clearoff(g,x) & cleartop(x) -> succeed(g);
    % cleartop is asserted automatically by MOVEHAND %
```

Figure D.2  CLEAROFF expressed in simplified form as a Planner theorem and as Ps

The Planner goal primitive, THGOAL, serves three functions. The first corresponds to a condition within an LHS, i.e., an access of Working Memory, so that a Planner user is sometimes evoking an explicit primitive where a PS user need not do so. Note that this puts failures to match the database in Planner into the backtracking mechanism, where in PSs it is simply a failure to match a P. The latter seems to have some advantages in clarity of expression, since it ties condition elements together into coherent units rather than having an unbroken string of them. The second function of THGOAL in Planner corresponds to evoking subordinate problem operators by RHS actions in Ps, except that Planner generally uses explicit references to appropriate theorems, where the selection is done by recognition in PSs (recognition of a signal or a goal). This can include iterating through a variety of methods (which is different from choicepoints within a method). The third function corresponds to setting up choicepoints in PSs. The PS expression of this is more complex than for Planner, but it has much more flexibility and selectivity. For these three functions, PSs thus provide means that are more direct, more flexible, and more explicit with regard to intent. That relatively little explicit mechanism in PSs was necessary to duplicate the problem-solving search built into the Planner language indicates that the Planner approach is not precisely suited to the domain at hand, and even lends itself to using blind search where slight additional knowledge (selectivity in making actions) can be quite effective in producing adequate problem-solving behavior.
This section presents enough details to give the reader a fuller picture of the inner workings of WBlock and to allow the reader to understand the corresponding complete detail in the appendices. First, a segment of program trace is explained, so that details of the program’s behavior (Appendix H) can be followed. Section E.2 gives details on each of the problem operators. Section E.3 discusses the particular aspects of tasks, and describes a peculiarity of the backtracking mechanism along with an experiment that modifies the behavior to be less strange. Section E.4 gives details on WBlock’s predicates, which are important for reading the actual Ps in Appendix F.

E.1. An example in more detail

Figure E.1 gives the program trace for test sentence 1. The first six lines give a trace of the processing of the input, similar to that for the old MillPS program. The main thing to notice is that there remains an inconsistency at the end of that processing, and that it then becomes the intention of the command. The top goal for the problem-solving system is on the "STARTING" line, which says it is to put BLOCK-1 onto BLOCK-5. The part of the scene that is pertinent to this command is that on BLOCK-1 there is a small pyramid, PYRAMID-1, and that BLOCK-5 has nothing on top of it. The first action taken to achieve the PUTON goal is to establish the subgoal G-1, to CLEAROFF BLOCK-1 - objects with other things on top of them can never be moved, in this model of toy blocks. The line after the G-1 line is indented, to indicate that the goal established there is a subgoal of the previous one. Goal G-2 is to GETRIDOF PYRAMID-1, which at the start was on top of BLOCK-1.

The next five lines give the trace of FINDSPACE working. It selects several points at random on the table, to try to find space to put PYRAMID-1, finally settling on the region on the table with lower left-hand corner at point (600 0 0) (using standard X-Y-Z Cartesian coordinates) and with upper right-hand corner at (1200 600 0), as indicated by the line starting "FOUND REGION". To go through that more slowly, "REJECTING" indicates that the given point is within some object already on the table, so it can't be considered, but FINDSPACE uses that point to shift to the point on the boundary of the obstructing object that is closest to the first, and then looks for a clear region at that boundary point, as indicated by the "LOOKING AT" line (when it follows a "REJECTING" line). In this case, attention shifts from (780 721 0) to (780 600 0), where the first happened to be inside the box, and the second is on its lower boundary. Considering the boundary point doesn't help, because the clear region found according to FINDSPACE's limited capabilities is too small to fit the pyramid, as noted by the "REGION AT" line. The next attempt with a new random point on the table is successful, finding the large region with lower left-hand corner at (600 0 0). Using the FINDSPACE result, GETRIDOF establishes a new subgoal, G-3, to PUT PYRAMID-1 at a random point in that clear space.

PUT has GRASP as a subgoal, and GRASP in turn wants to CLEAROFF the pyramid before it grasps it; the CLEAROFF goal succeeds immediately, since the pyramid has nothing on top of it (the program does not make use of the fact that pyramids never have things on them). The line starting with (0) is the first primitive hand movement, which
1 INPUT TEXT IS "PUT THE SMALL RED BLOCK ON THE BLUE BLOCK"
OBJ-1 AMBIG 53-1 BLOCK-1 PYRAMID-1...
OBJ-1 AMBIG R4-1 BLOCK-1 PYRAMID-3...
OBJ-1 REFERS BLOCK-1
OBJ-2 AMBIG 88-1 BLOCK-5 PYRAMID-2...
OBJ-2 REFERS BLOCK-5
RELINCON OBJ-1 85-1 ON BLOCK-5 POS
STARTING GT PUT BLOCK-1 ONTO BLOCK-5
GOAL G-1 CLEAR OFF BLOCK-1
  GOAL G-2 GET OFF PYRAMID-1
  REJECTING (789 721 0)
  LOOKING AT (780 690 0)
  REGION AT (689 689 0) TOO SMALL
  LOOKING AT (789 9 0)
  FOUND REGION (688 0 0) TO (1200 600 0)
    GOAL G-3 PUT PYRAMID-1 (808 451 0)
      GOAL G-4 GRASP PYRAMID-1
        GOAL G-5 CLEAR OFF PYRAMID-1
          G-5 SUCCEEDS
            (8) MOVING HAND FROM (0 100 400) TO (150 150 200)
            (1) GRASPING PYRAMID-1
          G-6 SUCCEEDS
            (2) LIFTING PYRAMID-1 FROM (100 100 100) TO (988 451 0)
            TAKING PYRAMID-1 FROM STACK-3
            STACK-3 DISMANTLED
            (3) LETTING GO OF PYRAMID-1
            ADDING PYRAMID-1 ON TABLE-1 (POS)
          G-3 SUCCEEDS
          G-2 SUCCEEDS
          G-1 SUCCEEDS
          FOUND REGION CLEAR TOP BLOCK-5
          GOAL G-7 GRASP BLOCK-1
            GOAL G-8 CLEAR OFF BLOCK-1
              G-8 SUCCEEDS
                (4) MOVING HAND FROM (958 501 100) TO (150 150 100)
                (5) GRASPING BLOCK-1
              G-7 SUCCEEDS
                (6) LIFTING BLOCK-1 FROM (100 100 0) TO (400 940 400)
                (7) LETTING GO OF BLOCK-1
                ADDING BLOCK-1 ON BLOCK-5 (POS)
                MAKING STACK STACK-4 BLOCK-1 BLOCK-5
              G-6 SUCCEEDS
              GT SUCCEEDS
            REPY (1 (OKAY))

Figure E.1 Program trace for WBlox input sentence 1

moves it from its starting location to the center of the top of the pyramid, which point is computed from the location of the pyramid (100 100 100) and its size, also (100 100 100). The next line, starting with (1) to indicate another primitive hand movement, shows the hand actually grasping the pyramid. The numbering of the hand movements reflects the internal bookkeeping (the actual value is called EVENTTIME) that is being done in case
backtracking is required: only the hand movements and some assertions that keep track of what's been tried in connection with commands that have multiple inputs (PUTONSET, STACKUP, and PACK) are recorded in this way and subsequently undone in case backtracking occurs (the latter do not appear in the program trace, so there will appear to be gaps at times). When backtracking is going on, the program trace prints again those hand movements, but reversed to show their undoing, with the same numbers attached. That backtracking is occurring is thus evident by the descending numbers for those movements. Only a few of the tests given to WBlox require backtracking, as will be discussed in Section E.3.

After the grasping movement, the GRASP goal, G-5, succeeds, and control returns to the parent goal, the PUT goal G-3. The six lines in the trace up to "G-3 SUCCEEDS" show the completion of the PUT operation, with a hand movement lifting the pyramid to the target location, and with a further hand movement to let go of it. The other lines show the bookkeeping that is done as a side effect of the movements. First, when the pyramid is moved, it is no longer on BLOCK-1, so that the stack composed of the pyramid and the block, STACK-3, is no longer a stack. Second, when the pyramid is let go, the program notes that it is now on the surface of the table, and records that fact internally.

The remainder of the trace shows little that is new, as the program proceeds to put BLOCK-1 on top of BLOCK-5. In this case FINDSPACE doesn't need to go through the process of looking at random points because the target block is all clear. When BLOCK-1 is finally placed on BLOCK-5, a new stack is created, and both blocks are added to the stack, STACK-4. If any other blocks are added to an existing stack, i.e., are put on top of a block in an existing stack, the attendant operation consists of just noting the addition. This trace has illustrated most of the variety that the reader will encounter in looking over the program traces in Appendix H.

Other features of the material displayed in the appendices include run statistics, production-firing traces, displays of the residual Working Memory instances which compose the program's database, and diagrams of the scenes. All of these except the last should be familiar from the descriptions given of the old MiliPS program. An example of a diagram of a scene is given in Figure E.2.

The diagram shows only the horizontal plane of the scene, with the Y dimension somewhat compressed. Scattered throughout, at points approximately corresponding to actual locations of lower left-hand corners of objects, are markers for the scene objects. The object markers are systematic abbreviations of the objects' names and attributes as follows. Each marker is four characters long. The first character is the first letter of the size attribute-value for the object, if any, e.g., L for LARGE, or just the character ".". The second character is the first letter of the color attribute-value, e.g. R for RED, or "+" if it has no color. The third character is the first letter of the kind of object, e.g., B for BLOCK. The fourth character is the number of the object, i.e., the thing following the "-" in the object's name, e.g., 5 for BLOCK-5. Two exceptions to the above rules are observed: "X" is used for BOX, so as not to conflict with BLOCK, and no string is given for the table, whose location is (0 0 0). A full example is "SRP3", standing for "small red pyramid, PYRAMID-3". As to the spatial location of these four-character markers, two things need to be explained. When two objects are at the same X-Y plane location, but one is above the other (Z dimension), this is indicated by placing the marker for the higher one above the
the marker for the lower one, in the diagram. But having one marker above the other can also indicate that two objects are adjacent on the same plane, so when such ambiguities arise, the display of the database must be consulted, in particular the LOCAT predicate. Also, when two objects are too close together, i.e., would be displayed at the same place in the diagram, the second one is shifted to the right until the first open space occurs, and is placed there.

The system's behavior on Test 1 is displayed in complete detail in Appendix I, including details of each P firing and a display of Working Memory after the sentence has been processed.

**E.2. Details on components**

This subsection will give details on the components of WBlox that do a significant part of the problem-solving. The primary concern is to present information on the parts of the program that use selections (analogous to sorting), iterations through sets, and choicepoints. For more detail, the reader should consult the listing of the actual P's, Appendix F, in conjunction with the information given in Section E.4. Examples of where most of the capabilities are exercised will be discussed in Section E.3.

CLEAROFF is a simple iteration of the GETRIDOF operation. CLEAROFF has two components, one to select the largest object on top of the object to be cleared off, and one to recognize success and end the iteration. The selection of the largest object is on the basis of the sum of the length and width of the object, and ties are broken arbitrarily.
by using lexicographic order on the objects' names. The selection results in establishing a subgoal to GETRIDOF the selected object.

GETRIDOF makes three attempts within the WBlox choicepoint mechanism to find a suitable location on the table at which to place the object to be gotten rid of, and failing that, attempts to place it on some other object that may have enough space. Whether a location is suitable or not depends on whether the whole process backtracks to the particular choice of location or not. GETRIDOF uses FINDSPACE to locate clear spaces of the required size, and for the table has to allow the possibility that FINDSPACE will return a location that has already been tried. Such a duplication is counted as one of the three attempts because it is possible that only one suitable location on the table exists. If FINDSPACE fails to find a region on the table, three attempts are considered done, and GETRIDOF goes immediately to the consideration of other objects. GETRIDOF chooses the non-table objects on which to try to find space arbitrarily (lexicographically on the name), from the set of objects (except boxes and pyramids) that are large enough to accommodate the object to be disposed of. When all the available choices fail to survive later actions, GETRIDOF causes a backup to the previous choicepoint, if any.

FINDSPACE is driven by randomly selected points on the surface on which it is to find space. The only exception to that is when the surface is completely clear. The random point is not chosen from the entire surface, but from a surface whose upper and right-hand boundaries have been trimmed by a fraction (presently two thirds) of the size of the object to be placed (clearly most points in this edge space are unsuitable because the object if placed there would protrude over the edge of the space, but some part of the space must be included so that random points near the edge are considered). In attempting to find space, ten random points are tried, and then the procedure fails. FINDSPACE works solely with the length and width dimensions, due to a restriction on the task environment, namely that an object on top of another must have its entire bottom surface in contact with the supporting object. This restriction guarantees that the space directly above any clear region on an object is clear. A random point is first examined to determine whether it is inside some object, and if so, it is replaced by the point that is closest to the random point on the boundary of the obstructing object.

Using the given point, FINDSPACE then establishes lower boundaries on the clear region around the point by finding the closest object in both the X and Y dimensions independently. This suffices for the present task but is not the best imaginable procedure because the result is a point that may be adjacent to clear space in one direction or the other, so that the region found might be expandable either way with possible conformation in the other dimension. A more exact procedure would take into account interactions between the X and Y bounds rather than considering them independently. (The code for doing this in the original program was rather obscure, so I tried to imitate the best guess as to what it did.) After establishing the lower bounds, the region at the lower boundary point is examined to see if it is big enough. This is done by an easily-expressed PS pattern that tests whether any object overlaps the space defined by the point augmented by a region of the desired size. If a fit is possible, upper bounds for the region are found by again testing X and Y coordinates independently, locating the closest objects in back of and to the right of the given random point. The final augmented region is used to determine the location returned by FINDSPACE, by taking the lower left-hand corner in it, by taking a random point that will still allow the object of the desired size to be placed, or
by computing the point such that the object will be centered within the space. These options are chosen according to whether the space is to be packed, is on the table, or is otherwise on a block, respectively.

PUTON can come from the MiliPS part of the system as a single assertion or as a set of similar assertions. In the former case, it is immediately converted to a PUTON1 goal. In the latter case, a set is formed of the objects to be PUTON and the goal becomes a PUTONSET goal, to put that set on the target object. Before starting, PUTONSET sets up a choicepoint, so that in case there is a failure to put on the whole set of objects (resulting in backtracking to the choicepoint), an alternative strategy can be tried, to CLEAROFF the target object and to then PACK the set of objects onto it. PUTONSET iterates over the set, establishing a PUTON1 goal for each object selected. The selection is by the size of the object and, within sets of equivalent objects by size, arbitrarily by lexicographic order on the name. Each object selected is recorded so that future selections don't use the same object. That record is subject to backup, so that there is also recorded something allowing that record to be undone, similar to the undoing of a primitive hand action. PUTONSET is also used to do the first part of a PUTIN goal, but for PUTIN the action taken when there is a failure to put all the objects on is to add objects that are already in the box to the set of objects, to CLEAROFF the box, and then to PACK the set onto the box.

PUTON1 includes no selections of objects from sets, but does involve setting up a choicepoint, recording the location of the clear region found by FINDSPACE. When backtracking occurs, PUTON1 retries FINDSPACE to see if there are any alternative locations. It will try up to three times to find locations, and then will fail. In case PUTON1 fails and is not a subgoal of the PUTONSET procedure, it evokes MAKESPACE, which tries to remove objects from the target object until space can be found. MAKESPACE takes off objects according to size, preferring the smallest object larger than the desired space, but if none of those exists, removing the largest object and iterating that removal until space does exist. PUTON1 has one other variation, namely, it checks to be sure that the target object is in fact larger than the object to be placed, and if it isn't, fails.

STACKUP, like PUTONSET, takes a set of objects (blocks or pyramids), selects from the set according to size, records the selection so that it can be undone in backing up, and uses PUTON1 as a subgoal. But in addition to the size criterion, STACKUP must first use blocks, and if any pyramids are to be stacked up, one is selected to be put on the very top of the stack. STACKUP uses a match pattern to decide where the present top of the stack is, and always checks, when it is making the selection for the next object, whether some object that hasn't been tried yet is already on top of the block at the top of the stack. If the object that is accidentally in place already is not smaller than any of the other untried objects, it is left in place and recorded as an attempt as if it had been moved to that position. If the PUTON1 operation fails for one of the blocks in the set, the process goes on anyway, and that fact is duly reported in the program's reply. Note that this strategy does not always lead to the maximal stack, since the program's size metric is based on the sum of length and width, and since a very "large" object may have such a strange shape (e.g. very long and narrow) that no further objects can be put on it. No size metric used by itself can be suitable for building stacks. A more successful procedure would have to study the specific blocks' sizes in order to avoid this difficulty.

PACK is very much like STACKUP and PUTONSET in its basic operation, except that it
doesn't use PUTONI. It wants to put a set of objects on top of another object in the most space-economical way possible. It evokes FINDSPACE and records the results as choicepoints, as PUTONI does, and tries three locations, including duplicates, before falling back to the previous choicepoint. For each block so placed, there is an additional step that attempts to put something from the set on top of that. This secondary stacking is only one layer high, and after something is placed by that step, the process returns to putting things on the original target object. In making the secondary layer, pyramids are preferred to blocks, because blocks are more valuable in making the primary layer since they can be put upon. Also, the selection for the secondary layer is based on placing the largest (by the usual metric) object that will fit. If the secondary placement attempt fails, the process continues with the basic step.

E3. Features illustrated by the tasks

The tasks given to the MillIPS/WBlox system are divided into eight segments, each consisting of three or four tests, which were so divided to allow easy testing of the program. The tests are stored as RHSs of Ps that are evoked by user commands, displayed at the end of Appendix F. Program behavior is given in Appendix H, and there is a very detailed trace segment in Appendix I. This subsection will go through the features of each segment.

The first test in the first segment has been discussed at length in Section E.1. The second test is a query that the system answers by describing a number of objects. Some of the objects are identically described by the system, but the practice of numbering the replies allows them to be distinguished to some extent by the user. The list of objects in the reply may be surprising in that the system uses comparisons of objects' lower left-hand corners to determine whether one is to the right of another, sometimes going against standard usage. The third test is a simple command similar to the first test, involving the box instead of a block, and using one of the new computable relations ("to the right of") in specifying the object to be moved.

The second segment has four tests, 4 through 7, of a similar nature to those in the first. Test 4 shows the system successfully handling a superficially ambiguous sentence. Tests 5 and 6 are straightforward queries. Test 7 shows a command involving a compound construction for the main object of the command, namely to put two objects somewhere.

The third segment also contains four tests, 8 through 11, three of which are queries that divulge no important information. The command Test 9 was originally intended to try to make the box too full to fit in further objects, but it fails to put the program into any unusual behavior.

The fourth segment has three tests, 12 through 14. Test 12 commands the program to put four objects on top of a large block. Two of the objects are specified by the identical phrase "a small pyramid", which the program correctly interprets by making two distinct choices of objects. In the course of carrying out the command, the program is forced to do backup in the PUTONSET procedure, back to the beginning of the process. It goes forward again using PACK this time, putting the objects on in two layers, with the pyramids not directly on the target object. This extra stacking causes the reply to seem
The fifth and sixth segments, Tests 15 through 18.5 (six tests altogether) are mostly concerned with trying to fill up the box so that the program has to resort to clearing it out and packing the contents in more carefully. Test 15, which isn't directly involved with that strategy, puts a block on a block that is already full, forcing the program to use MAKESPACE to be able to fit it on. The rest of the tests deal directly with putting things in the box. Test 16 has an interesting form of ambiguity, where the program makes one choice for the referent of a phrase and then has to "back up" and take an alternative choice when it discovers that that is necessary in order to have an inconsistency in the sentence that can be turned into a command. The program doesn't really back up though, since it only records the alternative when it makes the first choice, so that it can easily be switched if necessary; this was discussed more fully in Section D.2. Test 18.0 achieves the goal of forcing the program to clear off the box and pack things in more carefully. Tests 18.5 and a repetition of 18.0 were included in the test sequence just in case the first presentation of 18.0 failed to do it (18.0 uses "it" to refer to what is in the hand, so that it really does something new when it is repeated). The tests were not presented interactively, but in an unconditional "batch" mode, so that 18.5 and the second 18.0 were done even though 18.0 alone would have been sufficient in the particular test run - recall that the "randomness" of FINDSPACE makes it difficult to repeat particular behavior.

The seventh and eighth segments, Tests 19 through 24, are designed to force the table to be too crowded, so that the backtracking within GETRIDOF could be demonstrated. Test 19 exercises the STACKUP procedure and stacks up a number of blocks so that they can be out of the way while the table is cluttered up with other things. The set of things to b- stacked included two pyramids, which the program refused to try to do, with the proper warnings. A dump of Working Memory appears after Test 19, to illustrate the kind of information that is stored to record progress within the system of choicepoints, and to illustrate goal-sequencing information. Tests 20, 21, and 22 put objects on the table, and so does 23 except that it turned out not to be necessary in the test sequence in order to produce the backtracking behavior in Test 24.

The backtracking behavior that resulted from Test 24 is rather strange: in trying to pick up the bottom block of the big stack built by Test 19, it gets rid of almost all of the things on top of the bottom block, but then fails to get rid of a particularly large block, and thus has to back up. But the backup takes it immediately all the way back to getting rid of the top of the stack, rather than the more natural-seeming operation of first trying to get rid of the lower objects in different ways, and then working back up to the top if those don't work out. This is due not to the explicit choicepoints in the PS but to the structure of the GETRIDOF process: it finds a place to get rid of the object, then tries to
grasp, which in turn triggers a GETRIDOF when the object being disposed of turns out to have something on top of it. The problem with this is that the choicepoint occurs before the subgoal is evoked so that when backtracking occurs, all of the choicepoints occur before all of the hand movements, resulting in going back to the point where the stack hasn't yet been touched as described above. The behavior exhibited on Test 24 in the eighth segment is, I believe, identical to what would have been done by the original Planner version (it wouldn't have survived in that form if it had been properly tested, I speculate). (This belief is based on "hand" simulation of Planner, and would only be contradicted if Planner's implementation of handling choicepoints is contrary to what seems to me to be the natural order of things; I could not find in the available documentation anything describing that scheme in detail - there is only vague informal description of Planner primitives' semantics). The remedy is to modify the subgoal structure of GETRIDOF, so that it does a GRASP before it does the FINDSPACE. One alternative that might be easily implemented in the PS version, but quite impossible in Planner, is to have backup return to the choicepoint with the most recent primitive hand action, as opposed to the one with the most recent creation. That is, backup would undo things between two specified choicepoints, rather than treating choicepoints as a stack and undoing things from the top only. For the purposes of demonstrating the correctness of my diagnosis, I modified WBlox (with in-core edits that aren't reflected in the main program listing) and ran Tests 22 and 24 again, labelling the reruns to be the ninth segment in Appendix H. The changes to get it to work involved interposing a GRASP subgoal in the RHSs of W11, W13, and W15, and two other modifications that might also be considered fixes of bugs in the GETRIDOF choicepoint bookkeeping, although they don't interfere with the standard GETRIDOF (because the standard version in its backup throws away all of the GETRIDOF goal structure and essentially starts from the beginning again): the NEGATE in W16 has to be (ALL,-6,-9), leaving the HASLEVEL attached to the GETRIDOF goal so that it can be retried, and an extra conjunct in the RHS of W17 is necessary, GETRIDCHOICE(K+1,G,1,02,0,0,0,0), a dummy to make GETRIDOF really act as if it has tried three times on the table when it fails to find space on it in the first attempt. The behavior exhibited in the ninth segment shows a reasonable backup order, although there are redundant GRASPs because only a minimum amount of patching was done to get the desired behavior.

E.4. Meanings of predicates for WbIox

This subsection explains the predicates that are used in the additions made to MiiPS to handle inputs for WbIox, and in WBlox itself. In a few cases, old predicates have been modified slightly as noted.

Many of the new predicates in the MiiPS part start with "IMP" (imperative) or "CHECK". The following are used in goal sequencing and bookkeeping: HASLEVEL, FAIL, NEXT, NEXTF, SUCCEED. To keep track of events and do backtracking, these are used: BACKUP, CHOICECOUNT, CHOICETIME, EVENTTIME, UNEVENT.

Arguments to the predicates are typed according to the following conventions:

- attribute: COLOR, SIZE
- set or stack
- goal
- hand

VI-57
n number
o object: BALL-1, BLOCK-3, etc.
p position in string: T1-1, B5-1, etc.
r relation: IN, ON, UNDER, and NEAR.
s sign: POS or NEG
sx, sy, sz size along the three dimensions
t temporary object token: OBJ-1, OBJ-2, etc.
v value: LARGE, RED, etc.
w arbitrary
x, y, z values of the three spatial dimensions.

ADDINSET(r,o,c) add to set c objects that are related by r to o. (W)
BACKUP(n) back up in the processing, undoing actions until the choicepoint n is reached. (W)
CHAINREL(r,o1,o2) add HASINDRELs asserting r1 of o1 for things that are r2 of o2, forming the transitive closure of r1 of o1. (B)
CHECKFAILFIT(N,o,x1,y1,x2,y2,x3,y3,sx,sy,sz) if this signal is examined, the GROWTOFIT process has failed, since it deletes this when it succeeds; failure means another iteration of FINDLOWPAIR is necessary; arguments as for FINDLOWPAIR. (Q)
CHECKPICKUP(o) initiate the CHECKPICKUP2 check. (V, M)
CHECKPICKUP2(o) do the actual check that the PICKUP command on a succeeded. (V)
CHECKPUTDOWN(o,x,y,z) initiate the CHECKPUTDOWN2 check. (V, M)
CHECKPUTDOWN2(o,x,y) check that o is new put down, i.e., on something, with a location different from (x, y, z). (V)
CHECKPUTON(o1,r,o2) initiate the CHECKPUTON2 check. (V, M)
CHECKPUTON2(o1,r,o2) check that o1 has been put on or in, according to r, o2. (V)
CHECKSTACKUP(o) initiate the CHECKSTACKUP2 check. (V, M)
CHECKSTACKUP2(o) check that o has been stacked up according to the STACKUP command. (V)
CHOOSECOUNT(n) the most recent choicepoint is the n'th. (W, M)
CHICETIME(n1,n2) the n1'th choicepoint is at EVENTIME n2. (W)
CLEAROFF(g,o) clear off the top of o. (W, Q)
CLEARTOP(o) o has a clear top, with no other objects on it. (Q, W)
CONJBOUND(w) a noun-phrase boundary at a conjunction (AND) has been reached in sentence w. (B, G)
CONVIND(r,o) compute and convert computable relations r of o to explicit HASINDRELs. (F, B)
ERSFINDNEARPAIR(n,o) erase all FINDNEARPAIR instances with corresponding n and o arguments. (Q)
ERSFINDPOSS(n) erase the FINDPOSS instances for n. (B)
ERSGETRIDCHOICES(n) erase the corresponding GETRIDCHOICE instances. (W)
ERSPACKCHOICES(n) erase the corresponding PACKCHOICE instances. (W)
ERSPUTONCHOICES(n) erase the corresponding PUTONCHOICE instances. (W)
ERSREMOVEDREL(o1,r,o2) erase the corresponding REMOVEDREL. (Q)
ERSTRIEDPACK(o,c) erase the corresponding TRIEDPACK instances. (W)
ERSTRIEDPUT(o,c) erase the corresponding TRIEDPUT. (W)
ERSTRIEDSTACK(o,c) erase the corresponding TRIEDSTACK. (W)
ERSUNEVENT(n1,n2) erase UNEVENT for n1 while backing up (BACKUP) to choicepoint n2. (W)
EVENTIME(n) the current event is the n'th (all events take one unit of "time"). (Q, W, M)
EXPECTMOD(w1,w2) sentence w1 has the expectation that a modifier (UP, DOWN, etc.) w2 will occur. (T, F, M, G)
FAIL(q) q has failed. (W)
FAILLOCATE(o) space could not be located for o (which is o2 in FINDSPACE). (Q, W)
FAILPACKUP(g,o1,o2) the second major step of PACK failed, namely trying to put o1 on an object just placed on o2; goal g is the main PACK goal of set c onto o2. (W)
FAILPUTON(g,o1) the goal to PUTON o1 onto o2 fails. (W)
FAILPUTONSET(g,o) the goal to put one of the objects in set c on o has failed. (W)

* Letters in parentheses after a definition are initial P groups in which the predicate is used.
the object ignored in that process.

$\text{PUTON}ZCNOICE(n_1, l_2, o_1, o_2, s, y, z)$ the $n_2$'th choice

$\text{LOCATERESULT}(o_1, y_1, z_2, y_2, z_2)$ the region found

$\text{MAKESPACE}(g, o_1, o_2, s, x_1, y_1, z_1)$ makes space on $o_1$;

$\text{MAKESPACE}2(g, o_1, o_2, s, x_1, y_1, z_1)$ the second step in

$\text{MAKESPACE3}(g, o_1, o_2, s, x_1, y_1, z_1)$ the final step in

$\text{MOVEHAND}(o_1, o_2)$ move hand $o_1$ onto $o_2$;

$\text{NEWLOCAT}(o)$ a new location; remove any old relations

$\text{NEWLOCAT2}(o)$ a new location; add any new relations that hold;

$\text{NEXT}(g, w)$ when $g$ succeeds, assert $r$, (W, Q)

$\text{NEXT}(g, w)$ when $g$ fails, assert $r$. (W)

$\text{NUCLEAR}(g)$ the present PUTON process involves a set, so inhibit clearing away objects that seem to be in the way. (Q, W)

$\text{NPCGCHK1}(p)$ check for noun-phrase grammar in the actual checking noun-phrase grammar. (W)

$\text{NPCGCHK2}(p)$ a delayed initiation of the second step in checking noun-phrase grammar. (W)

$\text{NPCGCHK3}(p)$ perform the second step of the noun-phrase grammar check at $p$, which is to signal error if appropriate. (W)

$\text{NREPLY}(n)$ the number of replies so far is $n$. (V, S)

$\text{PACK}(g, c)$ pack the objects in set $c$ onto $o$. (W)

$\text{PACKCHOICE}(g_1, g_2, o_1, o_2, c, x, y, z)$ the $n_2$'th choice at choicepoint $n_1$, trying $g_1$ is to PACK $o_1$ on $o_2$ at $(x, y, z)$. (W)

$\text{PACKPUT}(g, c, o_1, o_2)$ the PUT step of PACK goal $g$ onto set $c$ onto $o_2$ is to place object $o_1$. (W)

$\text{PACKUPON}(g, c, o_1, o_2)$ the second major step of $g$, PACKING $c$ onto $o_2$, is to try to put something from $c$ onto $o_1$. (W)

$\text{PACKUP}(g, o)$ pick up $o$. (W, M)

$\text{PACKUP2}(g, h)$ the finishing step in the PICKUP process is to be done, i.e., raising $h$. (W)

$\text{PUT}(g, o, x, y, z)$ put $o$ at $(x, y, z)$. (Q, W)

$\text{PUTDOWN}(g, o)$ put down on the table or wherever there is space. (W, M)

$\text{PUTM}OEVE(g, o, x, y, z)$ do the actual movement of the hand associated with a PUT. (Q)

$\text{PUTON}(g, c, o_1, o_2)$ put $o_1$ on $o_2$: there may be a set of instances with the same $o_2$ argument (see PUTONSET). (W, M)

$\text{PUTO}N1(g, c, o_1, o_2)$ put the single object $o_1$ on $o_2$, as opposed to PUTON, which might become PUTONSET. (W)

$\text{PUTONICHOICE}(g_1, g_2, o_1, o_2, x, y, z)$ the $n_2$'th choice at $g_1$, a PUTON1 goal of $o_1$ onto $o_2$, choicepoint $n_1$, is the location $(x, y, z)$. (W)

$\text{PUTONPUT}(g, o_1, o_2)$ start the actual PUT step of a PUTON1 goal (W)

$\text{PUTONSET}(g, c, o_1, o_2)$ put objects in set $c$ on $o$ by choosing and using PUTON1 iteratively. (W)

$\text{PUTONSETCHOICE}(g, c, o_1)$ the choicepoint $n$ for PUTONSET involves putting set $c$ on $o$. (W)

$\text{RAISEHAND}(n)$ raise the hand by moving it straight up. (Q, W)

$\text{RELRESTRI}(g_1, p, c, o_2)$ perform the first step in the relation-restriction process, which is to check for possible need for IMPREST, q.v. (F)

$\text{RELRESTRI2}(t_1, p, r, o_2, a)$ the second step in the relation-restriction process, which is to check $r$ of $o$ for possible referents for $t$, to restrict the set. (F)
RELRESTRCK{t,p,r,s} the former RELRESTRCK is now RELRESTRCK2; this now signals a preliminary step to the check-relation-restriction process, which first checks whether the relation at hand is an indirect or computable one. (B)

RELRESTRCK2{t,p,r,s} check whether the corresponding RELREST should be applied. (B)

REMOHASREL{a1,r,a2,s} the relation (a1,r,a2) has been removed; update INSTACK relations affected. (Q)

REMOINSTACK{a,c} e has been removed from c; check if anything remains of c except the bottom block. (Q)

REPLY(n,w) the n’th reply (in order of generation) is w. (V)

REPLY(o) w is a new reply, yet to be counted (see REPLY). (V, E, M, D)

RETRY(g) g is to be retried, i.e., restarted after a BACKUP, with a new choice made. (W)

STACKSET{c} collect the objects from STACKUP instances into set c. (W)

STACKUP{g,o} stack up o as part of a set of such instances. (W, M)

STACKUPSET{g,c} stack up the objects in set c. (W)

SUCCEED(g) g has succeeded; continue appropriately. (W, Q)

TRACEPUTIN(w) print a trace message for the PUTIN command; w is a dummy. (M)

TRIEDPACK{o,c} in PACKing the set c onto somewhere, object o has now been tried. (W)

TRIEDPUT{o,c} in putting c on some object, o has been tried. (W)

TRIEDSTACK{o,c} in stacking up objects in set c, o has been tried. (W)

UNEVENT(n,w) the way to undo the event at EVENTTIME n is w. (W, Q)

UNGRASP{o} let go of o, from the hand. (Q)

USERESULT(a1,a2,a,x,y,w) use the open region found by LOCATESPACE process, which should be of size (ax, ay) on the horizontal plane, in the way specified by w, which is one of (PACK, RANDOM, CENTER). (Q, W)

WBPINIT{g} initialize for starting up the WBox Pa, top level goal g. (M)
F. Summary and Discussion

MillIPS represents a successful implementation in PSs of a language system with some general features, namely, objects with relations and attributes, main sentence forms that describe a scene, imperative forms, and a variety of queries. Inputs are processed without recourse to conventional syntactic parsing, and no tree-structured representation of them is formed. Text is converted immediately on being scanned to an internal form, which is quite sufficient for further manipulations, but which doesn't preserve the surface structure at all. At each point in the left-to-right scan of an input, as much as possible is known and inferred from what has been scanned. Five forms of completeness have been discussed, and MillIPS's capabilities were delineated with respect to those, providing a measure of its potential performance beyond the 50 test sentences exhibited. Linguistic anomalies are systematized into the categories ambiguity, redundancy, and inconsistency, and the main reaction of the system to inputs is based on the interaction of sentence type and the presence of those anomalies. Augmenting an early version to handle the blocks manipulation task was carried out by major additions to the set of Ps with few changes to existing Ps and with no deletions.

WBlox is a specialized problem-solver for blocks manipulations of a simple sort. Its organization is hierarchical, it features operating on a model and carrying out updating procedures as a result of operations, and is capable of backtracking in a search space to find a feasible plan of action. The system's goals are explicit and are sequenced to result in search behavior representable as an and-or graph. A less prominent backtracking mechanism is needed here than in the original Planner implementation of a similar blocks problem solver. Analysis of the problem domain allowed some decisions made formerly by backtracking to become more precise ordering decisions, taking advantage of selectivity in the LHSs of Ps. The remaining decisions requiring potential backtracking were formulated explicitly as choicepoints and associated with a stream of undoable primitive operations, rather than having mechanisms of questionable flexibility built into the underlying architecture as in Planner and other recent AI languages. A set of tests were devised to fully exercise the capabilities of the problem-solving system.

The question of whether the present system, and more generally PSs, could be used for further research can be approached along the lines of the language system and the problem-solving system independently. The completeness considerations in Section B.1 support a wide task domain coverage for the present system and indicate a framework for making additions to the system to rationally order the priorities for augmentation. The precise formulation of semantic cases, discussed in Section B.2, Section B.3, and Section D.4 raise further issues for augmentation and indicate how minor some of the omitted considerations in the present system are. Nevertheless, analysis of the existing cases, explicitly given as Ps and thus in a usable form, might be fruitful for cleaning up the structure and giving it more inherent generality and flexibility.

To use the present techniques in a new task domain would first require a new lexicon, which simply involves changing the tagging process (T Ps), which are independently modifiable. It would probably be necessary to augment the grammatical adjacency tests for new word classes, but this doesn't present obvious difficulties either. The semantics of blocks relations and how relations interact in the understanding of inputs
might be the area requiring the most new problem-solving. There is already present a system of dividing relations into direct ones, indirect ones, and computable ones, and that scheme and its processing conventions might carry over intact (cf. the discussion in Section D.2). The actual use of relations in referent determination would probably be along the lines of the present F Ps, but considerations there would probably not interact with the closely related set of B Ps, given that many interactions have already been worked out in response to the demands of the augmentation included in the present work.

Some further work has already been done by others within the basic blocks problem-solving domain. In particular, Fahlman (1974) describes a reworking and extension of the blocks task, which in retrospect might have served as a better vehicle for comparisons than the original one used here. Of the nature of the blocks tasks that he focussed on, it suffices to say that they involved building more complex structures than in WBlox, sometimes using auxiliary structures, allowing rotations of objects, enabling intercommunication between goals, and modelling the mechanics of contact and balance of objects more carefully. Fahlman developed a flexible control structure within the Conniver framework (Sussman and McDermott, 1972), and asserted its superiority over Planner and similar languages, and also specifically over PSs. Fahlman emphasized the importance of being able to: set up explicit goals; test hypotheses; switch back and forth between alternate promising approaches to a goal; and give up on an approach with specific difficulties communicated back to higher goals. Of those four features, only the last is something that hasn't yet been explicitly demonstrated in PSs, although keeping major alternative approaches for relatively large models also deserves further research in the PS framework. I will now discuss some of Fahlman's points in more detail, and argue that the ability of PSs to grapple with the difficulties of the task domain is promising, if not already demonstrated.

Fahlman developed a "choice-gripe" control structure, in which each choicepoint is explicit and sets up a gripe handler so that failures of subgoals following the choice, when those failures include specific gripes on why they occurred, can be processed appropriately. A gripe handler reacts more flexibly than choicepoint recovery in WBlox, in that it can involve taking better preparatory steps and then retrying the subgoal, or redefining the subgoal in some way and then retrying it, or taking other similar corrective actions. It seems clear that the present choicepoints in WBlox could easily be extended to behave in these more flexible ways, according to task demands, since the recovery is handled by specific Ps.

In trying alternative paths, Fahlman made use of Conniver's multiple data contexts, in which context tags are used to point to complete context alternatives, allowing them to be examined, resumed, or suspended. Such a facility, if the task really required it (as opposed to using it as a convenience because it's there), would be an explicit mechanism in PSs, perhaps storing alternative contexts as Ps and having them selectively evokable for examination or resumption. But a PS approach might be found to avoid that by coding, instead, methods for patching up difficulties or revising an ever-current state to make it look as if something different had been done. Based on the limited evidence on human behavior, e.g. in Newell and Simon (1972), humans seem to make use of mistakes without having to return completely to a state on some other branch of a search tree, and perhaps have better diagnostic and recovery methods because of limitations along the same lines as would be the case in a PS implementation. Rather than storing entire states, the
alternative might be to keep path information so that a previously-seen knowledge state could be recomputed (perhaps laboriously) if necessary.

Several minor topics raised by Fahlman can now be discussed. His system made use of a distinction between primary data and secondary data, which can be re-computed if necessary from the primary data, but which is kept around anyway, subject to erasure if storage becomes scarce. This might correspond to having a fading Working Memory, where items not accessed for some period of time simply disappear. Such a scheme has not been implemented, but it has been indicated as useful in several places in the present work. Fahlman additionally proposes that memory fade be based on the difficulty of recomputation and on some estimate of expected usefulness. Fahlman comments on the overall loose style of his system, which allows it to step back from local jam-ups and try to get around them. This is just as much an attribute of PSs, given the appropriate memory representation of what constitutes a jam-up. He says his program is prone to get into infinite loops, and proposes that a more sophisticated system would record states and occasionally check back to make sure there isn't serious repetition. Such a solution should be equally feasible in PSs, although perhaps not as necessary because PS architectures have some built-in safeguards, e.g. not firing a P on the same data twice unless some of it has been re-asserted. The topic of loops needs further research, certainly. Finally, the goal intercommunication in Fahlman's system, which includes both protection of goals' results from interference and dissemination of useful information to others, should be quite feasible in PSs due to the open, global nature of the Working Memory.
G. References


VI-67
FINAL RUN WITH MODIFIED TRACE

1 INPUT TEXT IS "A LARGE GREEN BLOCK IS ON A RED TABLE"
   ADDING SIZE LARGE (POS) TO BLOCK-1
   ADDING COLOR GREEN (POS) TO BLOCK-1
   ADDING BLOCK-1 ON TABLE-1 (POS)
   REPLY (GO Away)

ISA (BLOCK-1) (TABLE-1) (TABLE-1)
HAGAN (BLOCK-1) (TABLE-1) COLOR GREEN (POS) (TABLE-1 COLOR RED POS)
HAGAN (BLOCK-1) ON TABLE-1 (POS)

2 INPUT TEXT IS "A BLUE BALL IS ON THE TABLE"
   ADDING COLOR BLUE (POS) TO BALL-1
   ADDING BALL-1 ON TABLE-1 (POS)
   REPLY (GO Away)

ISA (BALL-1) (BLOCK-1) (TABLE-1) (TABLE-1)
HAGAN (BALL-1) (COLOR BLUE POS) (BLOCK-1 SIZE LARGE POS) (BLOCK-1 COLOR GREEN POS)
(TABLE-1) COLOR RED POS)
HAGAN (BALL-1) ON TABLE-1 (POS) (BALL-1 ON TABLE-1 (POS)

3 INPUT TEXT IS "THE BALL IS NEAR THE BLOCK"
   OBJ-1 REFERS BALL-1
   RELATION OBJ-1 B2-1 TO NEAR BLOCK-1 POS
   ADDING BALL-1 NEAR BLOCK-1 (POS)
   REPLY (GO Away)

ISA (BALL-1) (BALL-1) (BLOCK-1) (TABLE-1) (TABLE-1)
HAGAN (BALL-1) (COLOR BLUE POS) (BLOCK-1 SIZE LARGE POS) (BLOCK-1 COLOR GREEN POS)
(TABLE-1) COLOR RED POS)
HAGAN (BALL-1) ON TABLE-1 (POS) (BALL-1 NEAR BLOCK-1) (POS)

4 INPUT TEXT IS "A BLUE BALL IS ON THE BLOCK"
   ADDING COLOR BLUE (POS) TO BALL-2
   ADDING BALL BALL-2
   OBJ-2 REFERS BLOCK-1
   ADDING BALL-2 ON BLOCK-1 (POS)
   REPLY (GO Away)

ISA (BALL-1) (BALL-1) (BALL-2) (BLOCK-1) (TABLE-1) (TABLE-1)
HAGAN (BALL-1) (COLOR BLUE POS) (BALL-2 COLOR BLUE POS) (BLOCK-1 SIZE LARGE POS)
(BLOCK-1) COLOR GREEN POS) (TABLE-1) COLOR RED POS)
HAGAN (BALL-1) ON TABLE-1) POS) (BALL-1 NEAR BLOCK-1) POS) (BALL-2 ON BLOCK-1) POS)

5 INPUT TEXT IS "THE BALL ON THE BLOCK IS SMALL"
   OBJ-1 REFER B2-1 BALL-2
   OBJ-2 REFER BLOCK-1
   RELATION OBJ-1 B2-1 TO ON BLOCK-1 POS
   OBJ-1 REFER BALL-2
   PROPERTIES OBJ-1 5 SI-SIZE SMALL POS
   ADDING SIZE SMALL (POS) TO BALL-2
   REPLY (GO Away)

ISA (BALL-1) (BALL-1) (BALL-2) (BLOCK-1) (TABLE-1) (TABLE-1)
HAGAN (BALL-1) (COLOR BLUE POS) (BALL-2 COLOR BLUE POS) (BALL-1 SIZE SMALL POS)
(BLOCK-1) SIZE LARGE POS) (BLOCK-1 COLOR GREEN POS) (TABLE-1) COLOR RED POS)
HAGAN (BALL-1) ON TABLE-1) POS) (BALL-1 NEAR BLOCK-1) POS) (BALL-2 ON BLOCK-1) POS)

6 RUN TIME 1 MIN. 19.8 SEC

EXAM TRY FIRE IMPACT E/A E/FT 1/FT
2412 496 290 412 9.21 4.88 1.32
0.038 0.191 0.300 0.044 0.036

7 INPUT TEXT IS "WHAT IS ON THE BLOCK"
   OBJ-2 REFERENCES BLOCK-1
   REPUTATION OBJ-1 B1-1 ON BLOCK-1 POS
   OBJ-1 REFER BALL-1
   REPLY (THE SMALL BLUE BALL)

ISA (BALL-1) (BALL-1) (BALL-2) (BLOCK-1) (TABLE-1) (TABLE-1)
HAGAN (BALL-1) (COLOR BLUE POS) (BALL-2 COLOR BLUE POS) (BALL-1 SIZE SMALL POS)
(BLOCK-1) SIZE LARGE POS) (BLOCK-1 COLOR GREEN POS) (TABLE-1) COLOR RED POS)
HAGAN (BALL-1) ON TABLE-1) POS) (BALL-1 NEAR BLOCK-1) POS) (BALL-2 ON BLOCK-1) POS)

8 INPUT TEXT IS "WHAT IS BLUE"
   PROPERTIES OBJ-1 B1-1 COLOR BLUE POS
   OBJ-1 REFER B2-1 BALL-1
   REPLY (THE BLUE BALL) (THE SMALL BLUE BALL)

ISA (BALL-1) (BALL-1) (BALL-2) (BLOCK-1) (TABLE-1) (TABLE-1)
HAGAN (BALL-1) (COLOR BLUE POS) (BALL-2 COLOR BLUE POS) (BALL-1 SIZE SMALL POS)
(BLOCK-1) SIZE LARGE POS) (BLOCK-1 COLOR GREEN POS) (TABLE-1) COLOR RED POS)
HAGAN (BALL-1) ON TABLE-1) POS) (BALL-1 NEAR BLOCK-1) POS) (BALL-2 ON BLOCK-1) POS)

9 INPUT TEXT IS "THERE IS A BOX ON THE TABLE"
   ADDING BOX BOX-1
   OBJ-2 REFERENCES TABLE-1
   ADDING BOX-1 ON TABLE-1 (POS)
   REPLY (GO Away)

10
### Table for Helipad Tests

<table>
<thead>
<tr>
<th>A2</th>
<th>A3</th>
<th>Z2</th>
</tr>
</thead>
<tbody>
<tr>
<td>M11</td>
<td>M</td>
<td>0</td>
</tr>
<tr>
<td>M12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M15</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M17</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M18</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M19</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M21</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M22</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M23</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M24</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M26</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M27</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M28</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M29</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M30</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M31</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M33</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M34</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M35</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M36</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M37</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M38</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M39</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M40</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M41</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M42</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M43</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M44</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M45</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M46</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M47</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M48</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M49</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M51</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M52</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M53</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M54</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M55</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M56</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M57</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M58</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M59</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M60</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M61</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M62</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M63</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M64</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M65</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M66</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M67</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M68</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M69</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M70</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M71</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M72</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M73</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M74</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M75</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M76</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M77</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M78</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M79</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M80</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M81</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M82</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M83</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M84</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M85</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M86</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M87</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M88</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M89</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M90</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M91</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M92</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M93</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M94</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M95</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M96</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M97</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M98</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M99</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Fourth Section

16. INPUT TEST IS "WHAT THAT IS NOT RED IS ON THE FLOOR."

**Predicate:** Obj-1 RS-1 Color RED NEG

**Obj-2** refers Box-2

**Relator** Obj-1 Wi-1 on Floor-1 Pos

**Reply:** (The Un-Red Box)

15A (Ball-1 Ball) (Ball-2 Ball) (Block-1 Block) (Box-1 Box) (Box-2 Box)

**Floor-1 Floor** (Table-1 Table)

High: Ball-1 Color Blue Pos (Ball-2 Color Blue Pos) (Ball-2 Size Small Pos)

(Block-1 Size Large Pos) (Block-1 Color Green Pos) (Box-2 Color Red Neg)

**Floor-2 Color Pos** (Table-2 Color Pos)

Manual (Ball-1 on Table-1 Pos) (Ball-1 Near Block-1 Pos)

(Ball-2 on Block-1 Pos) (Box-1 on Table-1 Pos) (Box-2 on Floor-1 Pos)

**Floor-2 on Table-1 Neg**

17. INPUT TEST IS "WHAT IS."

**Reply:** (The Blue Ball) (The Box) (The Un-Red Box) (The Red Floor)

10A (Ball-1 Ball) (Ball-2 Ball) (Block-1 Block) (Box-1 Box) (Box-2 Box)

**Floor-1 Floor** (Table-1 Table)

High: Ball-1 Color Blue Pos (Ball-2 Color Blue Pos) (Ball-2 Size Small Pos)

(Block-1 Size Large Pos) (Block-1 Color Green Pos) (Box-2 Color Red Neg)

**Floor-2 Color Pos** (Table-2 Color Pos)

Manual (Ball-1 on Table-1 Pos) (Ball-1 Near Block-1 Pos) (Ball-2 on Block-1 Pos)

(Ball-2 on Floor-1 Pos) (Box-1 on Table-1 Pos) (Box-2 on Floor-1 Pos)

**Floor-2 on Table-1 Neg**

18. INPUT TEST IS "A SMALL RED BALL IS IN THE BOX ON THE RED FLOOR.

**Adding** Size Small (Pos) to Ball-3

**Adding** Color Red (Pos) to Ball-3

**Adding** Ball Ball-3

**Adding** Color Blue Pos to Ball-3

**Adding** Obj-2 Airing Obj-1 Box-1 Box-2 ...

**Adding** Obj-3 Airing R1-1 Floor-1 Table-1 ...

**Adding** Obj-3 Refers Floor-1

**Adding** Obj-2 Box-1 on Floor-1 Pos

**Adding** Obj-2 Refers Box-2

**Adding** Ball-3 in Box-2 (Pos)

**Reply:** (Obj-1)

19A (Ball-1 Ball) (Ball-2 Ball) (Ball-3 Ball) (Block-1 Block) (Box-1 Box)

**Floor-2 Box** (Floor-1 Floor) (Table-1 Table)

High: Ball-1 Color Blue Pos (Ball-2 Color Blue Pos) (Ball-2 Size Small Pos)

(Ball-3 Size Small Pos) (Ball-2 Color Red Pos) (Block-1 Color Green Pos)

(Block-1 Color Green Pos) (Floor-2 Color Red Neg)

**Floor-1 Color Pos**

Manual (Ball-1 on Table-1 Pos) (Ball-1 Near Block-1 Pos) (Ball-2 on Block-1 Pos)

(Ball-3 in Box-2 Pos) (Block-1 on Table-1 Pos) (Box-1 on Table-1 Pos) (Box-2 on Floor-1 Pos)

**Floor-1 on Table-1 Neg**

18. INPUT TEST IS "THERE IS A LARGE GREEN BALL IN THE BOX ON THE FLOOR BESIDE THE BALL IN THE BOX ON THE FLOOR."

**Adding** Size Large (Pos) to Ball-4

**Adding** Color Green (Pos) to Ball-4

**Adding** Ball Ball-4

**Adding** Obj-2 Airing Obj-1 Box-1 Box-2 ...

**Adding** Obj-3 Refers Floor-1

**Adding** Obj-2 Box-1 on Floor-1 Pos

**Adding** Obj-2 Refers Box-2

**Adding** Ball-4 in Box-2 (Pos)

**Adding** Obj-4 Airing Obj-1 Ball-1 Ball-2 ...

**Adding** Obj-5 Airing Obj-1 Box-1 Box-2 ...

**Adding** Obj-6 Refers Floor-1

**Adding** Obj-5 Box-1 on Floor-1 Pos

**Adding** Obj-5 Refers Box-2

**Adding** Ball-5 in Box-2 (Pos)
PROBLEM STATEMENT:

The problem involves finding a region around a random point ( \( x_1 \), \( y_1 \)) given a space of points \( X \times Y \). The task is to find a region of size \( T \) around the given point. This is achieved by identifying points within the specified region and then finding a path that connects these points.

THEORETICAL BACKGROUND:

The region is defined by a grid representation of the space. Each point on the grid is either inside or outside the region. The pathfinding algorithm involves traversing the grid to find a path from one point to another within the region.

ALGORITHM:

1. **Initialization**:
   - Define the grid size \( X \times Y \) and the region size \( T \).
   - Set the initial point as \( (x_1, y_1) \).

2. **Region Definition**:
   - Identify points within the region using a region detection algorithm.

3. **Pathfinding**:
   - Use a pathfinding algorithm to find a path from \( (x_1, y_1) \) to another point within the region.

4. **Output**:
   - Return the path found.

IMPLEMENTATION:

- Use a grid-based representation to store the points inside and outside the region.
- Implement a pathfinding algorithm such as Dijkstra's or A* to find the path.
- Handle edge cases and boundary conditions to ensure the path is valid.

Example:

Given a space of points and a random point \( (x_1, y_1) \), find a path of size \( T \) around this point. Ensure the path is valid within the defined region.
V4: TESTED(POS) & LOCATION(X,Y,Z) & SATISFY(ERUPTED HANDPL(1)

V4: TESTED(LOC) & LOCATION(C) & SATISFY(ERUPTED HANDPL(1)

V4: TESTED(LOC) & LOCATION(C) & SATISFY(ERUPTED HANDPL(1)

V4: TESTED(LOC) & LOCATION(C) & SATISFY(ERUPTED HANDPL(1)

V4: TESTED(LOC) & LOCATION(C) & SATISFY(ERUPTED HANDPL(1)

END.

NEW PROGRAM LISTING

END.
FIRST BLOCK

1 INPUT TEST IS "PUT THE SMALL RED BLOCK ON THE BLUE BLOCK"

OBJ-3 MAPIC S3-1 BLOCK-1 PYRAMID-3 ...
OBJ-1 MAPIC R4-1 BLOCK-3 PYRAMID-1 ...
OBJ-1 REFERS BLOCK-1
OBJ-2 MAPIC R1-1 BLOCK-3 PYRAMID-2 ...
OBJ-2 REFERS BLOCK-5
RELINCN OBJ-1 R5-1 ON BLOCK-5 POS
STARTING GD PUTON BLOCK-1 INTO BLOCK-5

GOAL-1 CLEARKING BLOCKS
GOAL-2 GETPIPER PYRAMID-1
REJECTING (760 721 8)
LOOKING AT (780 660 8)
REGION AT (600 660 8) TOO SMALL
LOOKING AT (1785 871 8)
FOUND REGION (600 8 A1 TO (1200 660 8)
GOAL-3 PUT PYRAMID-1 (900 451 8)
GOAL-4 CLEAR PYRAMID-1

12 G-5 CLEARKING BLOCKS

1 SUCCEEDS
1 MOVING HAND FROM (100 400) TO (150 150 200)
1 GRASPING PYRAMID-1
1 SUCCEEDS
1 LIFTING PYRAMID-1 FROM (100 100 100) TO (980 451 8)
1 TAKING PYRAMID-1 FROM STACK-3 DISMANTLED
1 LETTING GO OF PYRAMID-1
1 ADDING PYRAMID-1 ON TABLE-1 (POS)

2 SUCCEEDS

SUCCES

GT SUCCEEDS 2

REPLY (1 (OKAY))

CLEARKING (BLOCK-1) (BLOCK-3) (PYRAMID-1) (PYRAMID-2) (PYRAMID-3)

MAPIC-1 COLOR RED POS (BLOCK-1) SIZE SMALL POS (BLOCK-2) COLOR GREEN POS

(BLOCK-2 SIZE LARGE POS) (BLOCK-3 COLOR RED POS) (BLOCK-3 SIZE LARGE POS)

(BLOCK-2 COLOR GREEN POS) (BLOCK-4 SIZE LARGE POS) (BLOCK-5 COLOR BLUE POS)

(BLOCK-2 SIZE LARGE POS) (BLOCK-3 COLOR GREEN POS)

(PYRAMID-1) SIZE SMALL POS (PYRAMID-2 COLOR BLUE POS)

(PYRAMID-2) SIZE SMALL POS (PYRAMID-3 COLOR RED POS)

(PYRAMID-3) SIZE SMALL POS

MAREK (BLOCK-1) ON BLOCK-5 POS (BLOCK-2 ON TABLE-1 POS)

(BLOCK-3 ON TABLE-1 POS) (BLOCK-3 ON TABLE-1 POS) (BLOCK-5 ON TABLE-1 POS)

(BLOCK-2 ON TABLE-1 POS) (BLOCK-1 ON TABLE-1 POS) (BLOCK-3 ON TABLE-1 POS)

(PYRAMID-3-2 ON BLOCK-1 POS)

MAPIC-1 SIZE SMALL POS (BLOCK-1) (BLOCK-2) (BLOCK-3) (BLOCK-4) (BLOCK-5)

(BLOCK-2 SIZE LARGE POS) (BLOCK-3 COLOR RED POS) (BLOCK-3 SIZE LARGE POS)

(BLOCK-4 COLOR GREEN POS) (BLOCK-5 SIZE LARGE POS) (BLOCK-5 COLOR BLUE POS)

(MAPIC-1) SIZE SMALL POS

REPLIES (1) (OKAY)

2 INPUT TEST IS "WHAT IS BELOW THE SMALL RED BLOCK"

OBJ-2 MAPIC R5-1 BLOCK-1 PYRAMID-1 ...

OBJ-2 MAPIC R4-1 BLOCK-1 PYRAMID-3 ...

OBJ-2 REFERS BLOCK-1

REPLIES (1) (OKAY)

OBJ-1 MAPIC R1-1 BLOCK-2 BLOCK-3 ...

2
(6) MOVING HAND FROM (700 500 300) TO (560 630 500)
(7) GRABBING PYRAMID-1
G-6 SUCCESSES
(8) LIFTING PYRAMID-1 FROM (300 840 400) TO (590 890 1)
(9) LETTING GO OF PYRAMID-1
ADDING PYRAMID-1 ON BOX-1 (POS-1)
ADDITION PYRAMID-1 IN BOX-1 (POS-1)
G-6 SUCCESSES
GT SUCCESSES
2

REPLY 11 (OKAY)

LAB3

10. INPUT TEXT IS "WHAT IS THE TO LEFT OF THE BOX"
OBJ-2 PERTS BOX-1
RELATE OBJ-1 WI-1 TO LEFT OF BOX-1 POS
OBJ-1 MOVES WI-1 BLOCK-1 BLOCK-5

REPLY 11 (THE SMALL RED BLOCK) 12 (THE LARGE BLUE BLOCK)
13 (THE SMALL RED PYRAMID) 14 (THE TABLE)

LAB3

11. INPUT TEXT IS "WHAT IS IN FRONT OF THE BOX"
OBJ-2 PERTS BOX-1
RELATE OBJ-1 WI-1 IN FRONT OF BOX-1 POS

OBJ-1 REFERS TABLE-1
2

REPLY 11 (THE TABLE)

CLEANUP BLOCK-1 (BLOCK-2) (BLOCK-3) (PYRAMID-1) (PYRAMID-2) (PYRAMID-3)
HOLD (BLOCK-1) SIZE SMALL POS (BLOCK-2) COLOR RED POS (BLOCK-2) SIZE LARGE POS
(BLOCK-2) SIZE LARGE POS (BLOCK-3) COLOR RED POS (BLOCK-3) SIZE LARGE POS
(BLOCK-3) SIZE LARGE POS (BLOCK-4) COLOR GREEN POS (BLOCK-4) SIZE LARGE POS
(BLOCK-4) SIZE LARGE POS (PYRAMID-1) COLOR GREEN POS
(PYRAMID-1) SIZE SMALL POS (PYRAMID-2) COLOR BLUE POS
(PYRAMID-2) SIZE LARGE POS (PYRAMID-3) COLOR RED POS
(PYRAMID-3) SIZE SMALL POS

MODEL (BLOCK-1 ON BLOCK-5 POS) (BLOCK-2 IN BOX-1 POS) (BLOCK-3 IN BOX-1 POS)
(BLOCK-4 ON BLOCK-2 POS) (BLOCK-5 ON TABLE-1 POS) (BOX-1 ON TABLE-1 POS)

(PYRAMID-1 IN BOX-1 POS) (PYRAMID-2 IN BOX-1 POS) (PYRAMID-3 ON BLOCK-5 POS)

POSITION BLOCK-1 (POS-1) (POS-2) (POS-3) (POS-4)

MOVE BLOCK-1 BLOCK-5 BLOCK-3 BLOCK-4

LOCATE BLOCK-1 (POS-1) (POS-2) (POS-3) (POS-4)

HOLD-1 (POS-1) (POS-2) (POS-3) (POS-4)

10.10 SECONDS ELAPSED: 31 HARMONIC NEW OBJECTS
PAK - SAP SIZE LENGTH
156
CORE TYPE FULL: 1352; 2311 USED (1383; 1441)
FIXED 42 OUT OF 48 MODS

FOURTH SEGMENT

12. INPUT TEXT IS "PUT A SMALL PYRAMID AND A SMALL PYRAMID AND A GREEN BLOCK
AND THE SMALL RED BLOCK ON THE LARGE RED BLOCK"
OBJ-1 MOVES 51-1 BLOCK-1 PYRAMID-1
OBJ-1 MOVES 51-1 PYRAMID-1 PYRAMID-2
OBJ-2 MOVES 51-1 BLOCK-1 PYRAMID-1
OBJ-2 MOVES 51-1 PYRAMID-1 PYRAMID-2

13. INPUT TEXT IS "PUT A SMALL PYRAMID AND A SMALL PYRAMID AND A GREEN BLOCK
AND THE SMALL RED BLOCK ON THE LARGE RED BLOCK"
OBJ-1 MOVES 51-1 BLOCK-1 PYRAMID-1
OBJ-1 MOVES 51-1 PYRAMID-1 PYRAMID-2
OBJ-2 MOVES 51-1 BLOCK-1 PYRAMID-1
OBJ-2 MOVES 51-1 PYRAMID-1 PYRAMID-2

RUN TIME 1 MIN 11.4 SEC
EXEY 'TRI' FIRE WHATEV CUT EXT TAT
PRO 547 547 1700 5.95 0.78 1.18
0.120 0.54 0.670 0.210 SEC AVG

1011 SUSPECTS DELETES 31 HARMONIC 14 NEW OBJECTS
PAK - SAP SIZE LENGTH
156
CORE TYPE FULL: 1532; 2311 USED (1383; 1441)
FIXED 42 OUT OF 48 MODS
10 INPUT TEST IS "PUT A LARGE GREEN BLOCK IN THE BOX"

OBJ-1 AMBIG LT-1 BLOCK-2 BLOCK-3

OBJ-1 AMBIG LT-1 BLOCK-2 BLOCK-4

OBJ-2 PETERS BOX-1

RELACE OBJ-1 DS-2 IN BOX-1 POS

OBJ-1 PETERS BLOCK-2

BACKUP OBJ-1 PETERS BLOCK-4

PUTIN STARTS WITH PUTIN

STARTING GT PUTIN BLOCK-4 ONTO BOX-1

GOAL G-1 CLEAROFF BLOCK-4

G-1 SUCCEEDS

LOOKING AT (274 913 1) REGION AT (890 940 1) TOO SMALL

LOOKING AT (274 944 1) REGION AT (890 940 1) TOO SMALL

LOOKING AT (137 989 1) FOUND REGION (791 560 1) TO (1286 580 1)

GOAL G-2 PUT BLOCK-4 (944 648 1)

G-2 SUCCEEDS

(1) MOVING HAND FROM (790 945 561) TO (454 399 260)

(2) GRASPING BLOCK-4

G-3 SUCCEEDS

(2) LIFTING BLOCK-4 FROM (354 299 0) TO (964 688 1)

ADDING BLOCK-4 ON BOX-1 (POS1)

(3) LETTING GO OF BLOCK-4

ADDING BLOCK-4 IN BOX-1 (POS1)

G-2 SUCCEEDS

GT SUCCEEDS

REPLY 1 (OKAY)
RUN TIME 5 MIN. 32.3 SEC  
EXEC TYP IPAC UACT E/A E/T  T/R  
3393 7966 0.928 0.988 0.184 0.264  
  0-05 INSERTS 729 DELETES 319 WARNINGS 19 NEW OBJECTS  
MAX (SPKR LENGTH 172  
CPCE (FREE, FULL) (1431) 2981600 (1481 154)  
FIRIRED 44 OUT OF 440 PRODS  

SIXTH SEGMENT  
18.0 INPUT TEST IS "PUT IT IN THE BOX"  
OBJ-1 PYRAMID BLOCK-9  
OBJ-1 PYPAR BOX-1  
RELOCATE OBJ-1-123-1 IN BOX-1 POS  
PUTIN STARTS WITH PUTIN  
STARTING GT PUTIN BLOCK-9 ON BOX-1  
GOAL G-1 CLEARMOFF BLOCK-9  
G-1 SUCCEEDS  
LOOKING AT (609 263 1)  
REGION AT (600 640 1) TOO SMALL  
REJECTING 1001 036 1)  
LOOKING AT (600 480 1)  
REGION AT (600 310 1) TOO SMALL  
REJECTING 1035 747 1)  
LOOKING AT (600 241 1)  
REGION AT (600 810 1) TOO SMALL  
REJECTING 1052 1027 1)  
LOOKING AT (600 640 1) TOO SMALL  
REJECTING 1004 1036 1)  
LOOKING AT (604 1150 1)  
REGION AT (600 1140 1) TOO SMALL  
REJECTING 1004 1036 1)  
LOOKING AT (604 1150 1)  
REGION AT (600 840 1) TOO SMALL  
REJECTING 1004 1036 1)  
REGION AT (600 1140 1) TOO SMALL  
FINDSPACE LIMIT EXCEEDED  
NO SPACE TO PUT BLOCK-9 BOX-1  
GT FAILS  
GOAL G-2 CLEARMOFF BOX-1  
GOAL G-3 CLEARMOFF BLOCK-9  
REJECTING 1455 135 0)  
LOOKING AT (490 139 0)  
REGION AT (100 0 0) TOO SMALL  
REJECTING 1493 520 0)  
LOOKING AT (1000 500 0)  
FOUND REGION (200 200 0) TO (1200 800 0)  
GOAL G-4 PUT BLOCK-5 (984 267 0)  
GOAL G-5 GRAPPI BLOCK-5  
REJECTING 1740 124 0)  
LOOKING AT (1740 200 0)  
FOUND REGION (200 200 0) TO (1200 800 0)  
GOAL G-7 PUT BLOCK-9 (984 307 0)  
GOAL G-8 GRAPPI BLOCK-9  
(6) LIFTING BLOCK-9 FROM (600 940 1000) TO (985 307 0)  
(11) LETTNG GD OF BLOCK-9  

ADDITIONS ON TABLE-1 (POS)  
G-1 SUCCESS  
G-6 SUCCESS  
GOAL G-9 CLEARMOFF BLOCK-5  
G-9 SUCCEEDS  
(2) MOVING HANDLE FROM (600 940 200) TO (985 307 0)  
(3) GRAPPI BLOCK-5  
G-5 SUCCEEDS  
MOVE TO (1010 317 400) OVERLAPS BLOCK-5 WITH BLOCK-9  
(4) LETTNG GD OF BLOCK-9  
G-6 SUCCESS  
G-9 SUCCESS  
GOAL G-10 DELRIF DD BLOCK-5  
REJECTING 1455 135 0)  
LOOKING AT (600 1070 0)  
REGION AT (600 547 1) TOO SMALL  
REJECTING 1049 480 0)  
LOOKING AT (1039 480 0)  
REGION AT (600 547 1) TOO SMALL  
LOOKING AT (1258 332 0)  
FOUND REGION (100 200 0) TO (600 307 0)  
GOAL G-11 PUT BLOCK-5 (253 293 0)  
GOAL G-12 GRAPPI BLOCK-5  
GOAL G-13 CLEARMOFF BLOCK-5  
G-13 SUCCEEDS  
(5) GRAPPI BLOCK-5  
G-12 SUCCEEDS  
(6) LIFTING BLOCK-5 FROM (600 1040 1) TO (253 293 0)  
(7) LETTNG GD OF BLOCK-9  
ADDED BLOCK-9 ON TABLE-1 (POS)  
G-11 SUCCESS  
G-10 SUCCESS  
GOAL G-14 DELRIF DD BLOCK-4  
REJECTING 1455 135 0)  
LOOKING AT (600 35 0)  
FOUND REGION (600 0 1) TO (1200 307 0)  
GOAL G-15 PUT BLOCK-4 (1583 65 0)  
GOAL G-16 GRAPPI BLOCK-4  
GOAL G-17 CLEARMOFF BLOCK-4  
G-17 SUCCEEDS  
(8) MOVING HANDLE FROM (483 253 400) TO (1049 700 0)  
(9) GRAPPI BLOCK-4  
G-16 SUCCEEDS  
(10) LIFTING BLOCK-4 FROM (600 700 1) TO (253 293 0)  
(11) LETTNG GD OF BLOCK-4  
ADDED BLOCK-4 ON TABLE-1 (POS)  
G-15 SUCCESS  
G-14 SUCCESS  
GOAL G-18 DELRIF DD BLOCK-2  
LOOKING AT (141 425 0)  
FOUND REGION (600 800 1) TO (600 900 0)  
GOAL G-19 PUT BLOCK-2 (197 351 0)  
GOAL G-20 GRAPPI BLOCK-2  
GOAL G-21 CLEARMOFF BLOCK-2  
G-21 SUCCEEDS  
(12) MOVING HANDLE FROM (100 105 200) TO (1049 700 0)  
(13) GRAPPI BLOCK-2  
G-20 SUCCEEDS  
(14) LIFTING BLOCK-2 FROM (940 400 1) TO (1087 381 0)  
(15) LETTNG GD OF BLOCK-2  
ADDED BLOCK-2 ON TABLE-1 (POS)  
G-19 SUCCESS  
G-18 SUCCESS  
GOAL G-22 DELRIF DD BLOCK-2  
LOOKING AT (793 475 0)  
REGION AT (10 393 0) TOO SMALL  
LOOKING AT (1370 730 0)  
FOUND REGION (200 551 0) TO (600 1288 0)  
GOAL G-23 PUT BLOCK-2 (100 999 0)  
GOAL G-24 GRAPPI BLOCK-2  
GOAL G-25 DELRIF DD BLOCK-2  
G-25 SUCCEEDS  
(16) MOVING HANDLE FROM (297 463 200) TO (1080 740 0)  
(17) GRAPPI BLOCK-2  
G-24 SUCCEEDS  
(18) LIFTING BLOCK-2 FROM (1049 400 1) TO (253 293 0)  
(19) LETTNG GD OF BLOCK-2  
ADDED BLOCK-2 ON TABLE-1 (POS)  
G-23 SUCCESS  
G-22 SUCCESS  
GOAL G-25 DELRIF DD BLOCK-3  
LOOKING AT (1823 527 0)  
REGION AT (600 587 0) TOO SMALL  

VI-110
<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Moving</td>
<td>Block 1</td>
</tr>
<tr>
<td>2</td>
<td>Replacing</td>
<td>Block 2</td>
</tr>
<tr>
<td>3</td>
<td>Cleaning</td>
<td>Block 3</td>
</tr>
<tr>
<td>4</td>
<td>Adjusting</td>
<td>Block 4</td>
</tr>
<tr>
<td>5</td>
<td>Testing</td>
<td>Block 5</td>
</tr>
<tr>
<td>6</td>
<td>Adding</td>
<td>Block 6</td>
</tr>
<tr>
<td>7</td>
<td>Reacting</td>
<td>Block 7</td>
</tr>
<tr>
<td>8</td>
<td>Reacting</td>
<td>Block 8</td>
</tr>
<tr>
<td>9</td>
<td>Reacting</td>
<td>Block 9</td>
</tr>
<tr>
<td>10</td>
<td>Reacting</td>
<td>Block 10</td>
</tr>
</tbody>
</table>

**Notes:**
- Moving the blocks is the first step.
- Replacing and cleaning are necessary for proper functioning.
- Adjusting ensures that everything is in place.
- Testing is crucial to ensure all components work as intended.
- Adding new elements enhances performance.
OVERL.

- I OMIG

10.8 INPUT TEST IS "PUT IT IN THE BOX"

OBJ-1 RETYPES BLOCK-8
OBJ-2 RETYPES BLOCK-8
RELIEF OBJ-1 IN BOX-1 POS
PUTTING STARTS 4TH PUT:
STARTING CT PUT BLOCK-8 ON BOX-1

GT SUCCEES

REPLY (1 DAY)

...
SEVENTH SEGMENT

ADDING SIZE LARGE (POS) TO BLOCK-A
ADDING BLOCK BLOCK-A

19 INPUT TEXT IS * STACK UP A LARGE PID BLOCK AND A SMALL BLOCK AND IT AND A SMALL PYRAMID AND A BLACK BLOCK AND A LARGE OPEN BLOCK AND A SMALL PYRAMID *  
OBJ-1 AMBIG L-1 BLOCK-2 BLOCK-3  
OBJ-2 AMBIG S-1 BLOCK-1 PYRAMID-3  
OBJ-2 AMBIG L-1 BLOCK-3 BLOCK-5  
OBJ-3 AMBIG S-1 BLOCK-1 PYRAMID-3  
OBJ-4 AMBIG P-1 BLOCK-1 PYRAMID-3  
CHOOSE PYRAMID-3 FOR OBJ-9  
OBJ-9 AMBIG B-2 BLOCK-6 BLOCK-7  
CHOOSING BLOCK-6 FOR OBJ-5  
OBJ-5 AMBIG L-3 BLOCK-5 BLOCK-6  
CHOOSING BLOCK-6 FOR OBJ-5  
OBJ-6 AMBIG L-3 BLOCK-2 BLOCK-3  
OBJ-6 AMBIG L-1 BLOCK-6 BLOCK-2  
OBJ-6 AMBIG L-3 BLOCK-1 BLOCK-2  
OBJ-7 AMBIG L-3 BLOCK-1 BLOCK-2  
OBJ-7 AMBIG P-1 BLOCK-1 PYRAMID-3  
CHOOSING PYRAMID-1 FOR OBJ-7  
STARTING GT SETUP

GOAL G-1 PUT BLOCK 3 ON TABLE-1  
GOAL G-2 CLEAR PYRAMID-3  
GOAL G-3 CLEERTOP PYRAMID-3  
REJECTING #28 B 01  
LOOKING AT #17 01  
FOUND REGION 1023 901 TO 1023 900  
GOAL G-1 PUT BLOCK 3 PYRAMID-3  
Parts: C-2 01  
GOAL G-3 CLEERTOP PYRAMID-3  
LOOKING AT #55 347 01  
FOUND REGION 1023 901 TO 1023 900  
GOAL G-1 PUT BLOCK 3 PYRAMID-3  
Parts: C-2 01  
GOAL G-3 CLEERTOP PYRAMID-3  
LOOKING AT #55 347 01  
FOUND REGION 1023 901 TO 1023 900  
GOAL G-1 PUT BLOCK 3 PYRAMID-3  
Parts: C-2 01  
GOAL G-3 CLEERTOP PYRAMID-3

G-8 SUCCESS
(1) MOVING HAND FROM (1023 900 100) TO (1023 900 100)
(2) GRABBING PYRAMID-3  
G-9 SUCCESS
(1) MOVING HAND FROM (1023 900 100) TO (1023 900 100)
(2) GRABBING PYRAMID-3  
G-10 SUCCESS
(1) LIFTING PYRAMID-3 FROM (1023 900 100) TO (1023 900 100)
THERE PYRAMID-3 FROM STACK-0  
STACK-0 DISMANTLED
(1) LETTING GO OF PYRAMID-3
(2) ADDING PYRAMID-3 ON TABLE-1 (POS)
G-11 SUCCESS
G-12 SUCCESS
G-13 SUCCESS
REJECTING #141 01  
LOOKING AT #141 01  
REGION AT #141 01 TO LARGE  
LOOKING AT #141 01 TO LARGE  
FOUND REGION (1023 900 100) TO (1023 900 100)
GOAL G-14 PUT BLOCK-1 (1023 900 100)
GOAL G-15 CLEERTOP BLOCK-1  
GOAL G-16 CLEAR PYRAMID-3  
GOAL G-17 CLEERTOP BLOCK-1  
GOAL G-18 SUCCESS
(1) MOVING HAND FROM (1023 900 100) TO (1023 900 100)
(2) GRABBING BLOCK-1  
(3) LIFTING BLOCK-1 FROM (1023 900 100) TO (1023 900 100)
(4) LETTING GO OF BLOCK-1
ADD BLOCK-1 ON BLOCK-1 (POS)
MOST STACK-0 ON BLOCK-1 (POS)
G-19 SUCCESS
G-20 SUCCESS
GOAL G-21 PUT BLOCK-1 ON TABLE-1
GOAL G-22 SUCCESS
(1) MOVING HAND FROM (1023 900 100) TO (1023 900 100)
(2) GRABBING BLOCK-1  
G-23 SUCCESS
(1) LIFTING BLOCK-1 FROM (1023 900 100) TO (1023 900 100)
THHERE BLOCK-1 FROM STACK-0  
STACK-0 DISMANTLED
(1) LETTING GO OF BLOCK-1
(2) ADDING BLOCK-1 ON TABLE-1 (POS)
G-24 SUCCESS
G-25 SUCCESS
G-26 SUCCESS
G-27 SUCCESS
G-28 SUCCESS
G-29 SUCCESS
LOOKING AT (065 880 0)
REGION AT (0 0 0) TOO SMALL
REJECTING (065 746 0)
LOOKING AT (065 698 0)
REGION AT (015 351 0) TOO SMALL
REJECTING (510 150 0)
LOOKING AT (151 290 0)
REGION AT (546 290 0) TOO SMALL
REJECTING (729 330 0)
LOOKING AT (169 330 0) TOO SMALL
REJECTING (99 433 0)
LOOKING AT (2 413 0)
REGION AT (0 0 0) TOO SMALL
REJECTING (412 771 0)
FINDSPACE LIMIT EXCEEDED
GOAL G-3 GRASP BLOCK-4
G-3 SUCCEEDS
LOOKING AT (427 376 100)
REGION AT (12 254 100) TOO SMALL
REJECTING (188 333 100)
LOOKING AT (088 304 100)
REGION AT (12 254 100) TOO SMALL
LOOKING AT (182 291 100)
REGION AT (12 254 100) TOO SMALL
REJECTING (187 314 100)
LOOKING AT (187 304 100)
REGION AT (12 254 100) TOO SMALL
LOOKING AT (163 263 100)
REGION AT (12 254 100) TOO SMALL
REJECTING (156 304 100)
LOOKING AT (106 304 100)
REGION AT (12 254 100) TOO SMALL
FINDSPACE LIMIT EXCEEDED
G-3 EXHAUSTED
(12) LETTING GO OF BLOCK-A
(13) MOVING HAND FROM (106 829 100) TO (500 789 101)
(15) GRASPING BLOCK-4
(16) LIFTING BLOCK-4 FROM (800 699 293) TO (306 720 498)
TAKING BLOCK-4 FROM STACK-14
STACK-14 DISMANTLED
ADDED BLOCK-4 ON BLOCK-C (POS)
ADDED BLOCK-4 TO STACK-12
GOAL G-35 PETRA G Erford BLOCK-4
GOAL G-34 GRASP BLOCK-4
G-34 SUCCEEDS
REJECTING (92 267 300)
LOOKING AT (132 655 300)
REGION AT (136 679 300) TOO SMALL
REJECTING (149 796 300)
LOOKING AT (131 655 300)
REGION AT (136 679 300) TOO SMALL
REJECTING (135 780 300)
LOOKING AT (139 655 300)
REGION AT (136 679 300) TOO SMALL
REJECTING (139 780 300)
LOOKING AT (128 655 300)
REGION AT (136 679 300) TOO SMALL
REJECTING (135 780 300)
LOOKING AT (131 655 300)
REGION AT (136 679 300) TOO SMALL
REJECTING (131 781 300)
FINDSPACE LIMIT EXCEEDED
GOAL G-35 GRASP BLOCK-4
G-35 SUCCEEDS
FOUND REGION CLEAN TOP BLOCK-6
GOAL G-36 PUT BLOCK-6 (150 280 0)
GOAL G-37 GRASP BLOCK-4
G-37 SUCCEEDS
(14) LIFTING BLOCK-4 FROM (306 720 498) TO (100 0 200)
TAKING BLOCK-4 FROM STACK-12
(15) LETTING GO OF BLOCK-4
ADDED BLOCK-4 ON BLOCK-C (POS)
STACKING BLOCK-15 ON BLOCK-6
G-36 SUCCEEDS
G-5 SUCCEEDS
G-5 SUCCEEDS
(16) MOVING HAND FROM (250 100 460) TO (106 829 100)
(17) GRASPING BLOCK-4
G-4 SUCCEEDS
REJECTING (106 829 0)
GOAL G-38 PUT BLOCK-8 (150 280 0)
GOAL G-39 GRASP BLOCK-4
G-39 SUCCEEDS
(18) LIFTING BLOCK-4 FROM (306 720 498) TO (250 0 200)
TAKING BLOCK-4 FROM STACK-12
STACK-12 DISMANTLED
(19) LETTING GO OF BLOCK-4
ADDED BLOCK-4 ON TABLE-1 (POS)
G-30 SUCCEEDS
G-5 SUCCEEDS
G-5 SUCCEEDS
(20) MOVING HAND FROM (125 207 100) TO (400 870 300)
(21) GRASPING BLOCK-9
G-1 SUCCEEDS
(22) LIFTING BLOCK-3 FROM (306 670 293) TO (306 670 300)
G-7 SUCCEEDS
REPLY (1 (QNA))
0-4 Succeeds

WARNING (7) ALREADY UNDER TRACING =
INSERTING (SUCCESS) GT (TRACING T) GT (SUCCESS O-4) V3/V4/S/NOT/

1-190. WS-1 "BUCK TOP"
USING (SUCCESS GT) (PARALLEL GT O-3) WARNING (7) ALREADY UNDER TRACING =
INSERTING (SUCCESS) GT (TRACING T) GT (SUCCESS O-4) V3/V4/S/NOT/

1-191. WS-1 "BUCK TO" USING (SUCCESS) TRACING GT SUCCESS

1-192. WS-1 "BUCK TOP" USING (SUCCESS GT) (PARALLEL GT O-3)

1-193. WS-1 "BUCK TOP" USING (SUCCESS GT) (PARALLEL GT O-3) WARNING (7) ALREADY UNDER TRACING =
INSERTING (SUCCESS) GT (TRACING T) GT (SUCCESS O-4) V3/V4/S/NOT/

1-194. WS-1 "BUCK TOP" SUCCESS

WARNING (7) ALREADY UNDER TRACING =
INSERTING (SUCCESS) GT (TRACING T) GT (SUCCESS O-4) V3/V4/S/NOT/

1-195. WS-1 "BUCK TO" USING (SUCCESS GT) (PARALLEL GT O-3)

1-196. WS-2 "CLEAR"
USING (SUCCESS GT) (PARALLEL GT O-3) WARNING (7) ALREADY UNDER TRACING =
INSERTING (SUCCESS) GT (TRACING T) GT (SUCCESS O-4) V3/V4/S/NOT/

1-197. WS-1 "BUCK TO" USING (SUCCESS GT) (PARALLEL GT O-3) WARNING (7) ALREADY UNDER TRACING =
INSERTING (SUCCESS) GT (TRACING T) GT (SUCCESS O-4) V3/V4/S/NOT/

1-198. WS-1 "BUCK TO" USING (SUCCESS GT) (PARALLEL GT O-3) WARNING (7) ALREADY UNDER TRACING =
INSERTING (SUCCESS) GT (TRACING T) GT (SUCCESS O-4) V3/V4/S/NOT/

1-199. WS-2 "CLEAR"
USING (SUCCESS GT) (PARALLEL GT O-3) WARNING (7) ALREADY UNDER TRACING =
INSERTING (SUCCESS) GT (TRACING T) GT (SUCCESS O-4) V3/V4/S/NOT/

1-200. WS-1 "BUCK TO" USING (SUCCESS GT) (PARALLEL GT O-3) WARNING (7) ALREADY UNDER TRACING =
INSERTING (SUCCESS) GT (TRACING T) GT (SUCCESS O-4) V3/V4/S/NOT/