Rational choice and framing devices: Argumentation and computer programming

Seana Coulson
Department of Cognitive Science
University of California, San Diego
La Jolla, CA 92093-0515
coulson@cogsci.ucsd.edu

Nick V. Flor
Department of Cognitive Science
University of California, San Diego
La Jolla, CA 92093-0515
flor@cogsci.ucsd.edu

Abstract

The argumentative discourse of computer programmers engaged in a collaborative programming task were analyzed as instances of ecologically valid reasoning behavior. Teams of expert programmers were brought into a laboratory setting to work cooperatively on a software maintenance task. Arguments which occurred spontaneously in the course of the task were examined with respect to: (a) their effect on task performance; and (b) to reveal the sorts of inferential machinery programmers use when they reason with one another. Arguments were found to be important in the formulation of plans as well as the negotiation of strategic priorities with respect to the task. Pragmatic features of the programmers' discourse revealed extensive use of framing devices whose efficacy depended upon interpretation in the context of linked pragmatic scales.

Introduction

Traditional approaches to reasoning in cognitive science involve the use of prescriptive formalisms to guide research. This approach involves comparing people's actual behavior in the performance of deductive and inductive inference with that predicted by formal theories of logical and statistical inference. Classic examples include studies by Tversky and Kahneman (1981), who demonstrate the inadequacy of their human subjects to make decisions in accordance with Bayesian probability theory, and Wason and Johnson-Laird (1966), who demonstrate a similar disparity between the character of their subjects' capacity for deductive inference and that dictated by first order predicate calculus.

Although the traditional approach has succeeded in generating a productive research program, it is problematic in two respects. First, the use of prescriptive formalisms is at odds with the goal of a descriptive account of rational behavior. Rather than starting with a prescriptive formal system, a more empirical approach to the investigation of rational behavior would start with observations of people's behavior in situations which paradigmatically require rational thought. Second, because the traditional approach has relied upon carefully controlled experimentation, considerations of ecological validity have often been lacking.

Recent research indicates that intelligent behavior is largely the result of the exploitation of historically established knowledge bases (e.g. physics), culturally established symbol systems (e.g. calculus), and socially manufactured tools (e.g. computers) (Hutchins, 1994). Further, the pared down setting of experimental studies often lacks the sorts of contextual clues which enable cognitive activity in natural settings. A full account of rational inference requires supplementation of traditional studies with observational analyses of ecologically valid reasoning behavior which occurs in the course of other, goal-oriented behavior.

One resource which has been underutilized by cognitive scientists interested in human reasoning ability is pragmatics. The gap between the literal content of speakers' utterances, and, the representation which the hearer eventually constructs on the basis of that information, gives testimony to the human capacity for inference. Moreover, pragmatic phenomena which operate in rational argumentation are particularly germane to the study of reasoning.

Because the defining characteristic of rational argument is that it appeals to one's opponents' reason, argumentation can be seen as a paradigmatic exemplar of rational behavior. Consequently, we propose rhetorical theory as an alternative guide to reasoning research. Rhetoric is a tradition with over 2000 years of development which catalogues the use of persuasive language. In contrast to formalisms whose appeal must be enforced by an external authority, rhetorical theory points to techniques which have proven to be intrinsically compelling.

Arguments that occur between programmers in the performance of a collaborative programming task thus present an ideal opportunity for the study of ecologically valid reasoning behavior. First, computer use is a good, real-world application domain in which to study complex behavior (Norman, 1986, 1987). Software development clearly involves cognitive activity in the pursuit of well-defined goals. Moreover, software development can also be a highly social activity involving frequent interactions between programmers and with their development tools. The arguments discussed in this paper are those which occurred spontaneously in the course of the performance of a complex cognitive task.

The goal of this study was first, to uncover the ways in which argumentative discourse affects programmers' performance of the task; and, second, to examine the pragmatic features of the programmers' argumentative discourse to reveal the sorts of inferential machinery which they exploit when they reason with each other. We consider how socially shared representations develop over the course of the task and how they function to organize and direct actions. We begin by reporting some general findings and proceed to examine one example in detail.

The Functional Role of Argument

To investigate the interactions of programmers at work, we brought 6 teams of 2-3 expert programmers into the laboratory to work together to add new commands to an existing computer game program. All subjects had had prior experience working together in industry or on programming projects for computer science classes. In order to complete the task, programming teams had to figure out how the existing program worked and then to figure out how to modify it to meet the task demands. From a maintenance perspective, the program was fairly complex, involving two modules of approximately 1500 lines each, spread out over 14 source files in two subdirectories. During the session, both subjects had their own computer terminal and were seated next to one another. Two video cameras were set up in sight of the subjects. One recorded the programmers' displays, while the other was positioned to record changes in body positions.

Transcription and coding. After each session, a complete transcription of the session was prepared and incidents of argumentative discourse were coded independently by two coders. Argumentative discourse is defined as discourse which contains an expressed disagreement between parties. In an argument, then, one member of the programming team espouses a belief or plan of action with which the other programmer expresses doubt or disapproval. Intercoder agreement was 90%. After coding, instances of intercoder discrepancy were then evaluated jointly by the coders to determine whether particular instances of discourse fulfilled the criteria of the definition. This discussion resulted in the labeling of several sections of discourse as pseudoarguments, discourse initially coded as an argument by one of the coders, and subsequently discarded.

When do arguments occur?

It was initially hypothesized that arguments, when they occurred, would occur at transition points in the programming task. Although there are, theoretically, an infinite number of algorithms one can deploy to solve a programming problem, realistically speaking the choice of a particular algorithm is quite constrained. One might guess that arguments function to constrain the choice of algorithms. Because the task involved modification of an existing program, transition points were defined as periods in which the team was actively editing the code. Additions, modifications, and deletions of the code as revealed by script (a UNIX routine which exhaustively records all characters displayed on the computer screen) were marked in the verbal transcript and their position was compared to the position of argumentative incidents in the transcript. While arguments occasionally occur at points of transition, this is not necessarily the case.

Thus the hypothesis that the functional role of argument lies chiefly in the planning phase task was disproved in its strongest form. However, argumentative discourse clearly plays some role in the team's plan formation process. Arguments frequently occur because of contradictory plans about how to proceed and thus involve discourse over two or more competing plans. One such example is discussed in detail in the section labelled the Right Way versus the Not So Right Way.

Why do arguments occur?

Coders divided the arguments into six categories: implementation details, responsibility, environment, completion, division of labor, and strategy. The relative proportion of each type of argument in the four two-person programming teams is displayed in Chart 1. By far the most frequent sort of argument concerned *implementation details*, which comprised 45% of the disputes among two-person teams. Implementation details concern contradictory proposals by the programmers about how to write specific elements of the code. This includes things such as how to implement a specific idea, where to add a specific subroutine, whether or not to implement a particular way, and arguments about the interpretation of existing code.

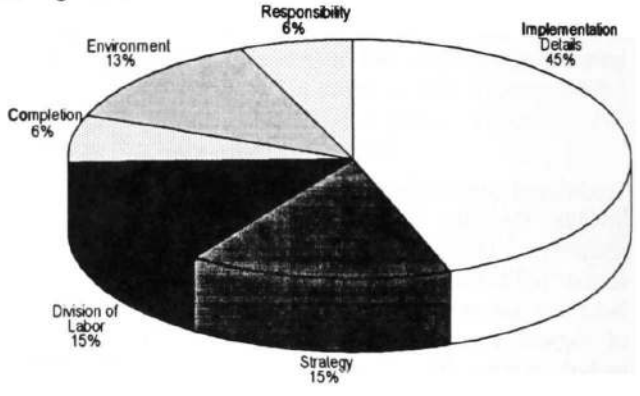

Chart 1. Distribution of argument topics.

Environment disputes included arguments concerning the computational environment, such as the location in the code, or the way client-server works. They comprised 13% of the arguments. Arguments about both the environment and about implementation details help the programming teams construct shared mental models of the program's structure which ultimately serve to determine their course of action. We discuss the details of how arguments aid the establishment of shared mental models in forthcoming work (Coulson & Flor, in progress).

Responsibility disputes involve disagreement over which programmer was to receive credit or blame for a given

piece of the code, and issues such as who was doing more work. Only 6% of the arguments were specifically about responsibility. However, 15% of the arguments concerned the division of labor. *Division of labor* involves disputes concerning which programmer should perform which task. Because of the constraints of the computing environment, a good way to divide the labor is to have one participant dictate changes while the other implements them. Analysis reveals that both responsibility disputes and explicit disputes over the division of labor affect who occupies the dictator's role. However, it is not the case that the person who wins the most arguments gets to dictate the changes. Rather, it is the person whose arguments turn out to be the most well-founded. Responsibility disputes, because they can help establish the relative validity of the programmers' competing strategies, seem to play an important role in the social organization of the team.

Argumentative discourse is also involved in the establishment of shared goal structures with respect to the task. Normative aspects of programming were discussed chiefly in the context of *completion* arguments, which comprised 6% of the arguments, and *strategy* disputes, which comprised 15% of the arguments. Completion arguments concern whether a particular piece of code, or the endeavour as a whole, should count as finished. *Strategy* disputes involve the programmers' general approach to the task. While arguments coded as strategy arguments frequently began as disputes regarding implementation details, they are not necessarily implementation related. Strategy arguments frequently concern issues of time-management and involve discussion about the relative merits of top-down versus bottom-up approaches, understanding the task versus acting to complete the task, and the elegance versus the ease of a particular implementation. The upshot of these arguments frequently has direct ramifications for the completion of the programming task.

Moreover, analysis of the arguments examined in this study uncovered several persistent trends in the nature of the appeals the programmers made in the course of their arguments. This agreement in the face of disagreement suggests that the programmers who participated in this study share certain fundamental assumptions and values. These include the virtue of simplicity, minimal standards for code, and the necessity of a speed/elegance trade-off:

- *Simplicity is a virtue*: other things being equal, code which is easiest for the programmers to implement is the preferred solution.
- *Minimal standards for code*: when modifying code written by other programmers, it is unacceptable to write code which is of a lower standard than the existing code.
- *Speed/Elegance trade-off*: it is permissible to sacrifice the elegance of your code if the speed of implementation is more important.

The Right Way vs. the Not-so-right Way

The example examined in this section concerns a strategic argument, an argument which turns on a decision about what sort of strategic emphasis the team should employ. In the following example, the issue concerns whether to employ a 'quick and dirty' versus a more elegant solution to a particular programming problem. The programmers are implementing a command that allows players of the game to eavesdrop on one another. To do this, they have to find a way to store the names of the players who are eavesdroppers. Their ensuing argument can be found in Table 1.

Table 1. Transcript of argument between R, L, and A.

1/R	We don't wait we don't need to make a list or anything do we?
2/	All we need is a character array to store the name of the person who's eavesdropping on this person.
3/L	But what, we need a list of we need a list of the
4/R	Oh, in case a lot of people are eavesdropping
5/	[whispers] no no right now this only supports two people,
6/	let's only make it work for two [laughs]
7/A	[laughs] He didn't go through the trouble to put down more users
8/R	Yeah, so let's not let's worry about that
9/	that's a future enhancement
10/A	That's for the next rev
11/R	Yeah, so let's just do it the simple way
12/	Let's go back to
13/ L	Noo that's
14/ R	my global variable way
15/A	Wait, wait, wait, wait, wait, well, well, see
16/L	No, let's not do it that way, let's do it the right way.
17/A	Are you guys hungry or not? [laughs]
18/ L	C'mon
19/A	[looks at Laura] twenty till eight
20/L	It's not that hard.
21/R	There are three of us here because we're supposed to vote on things.
22/	The quick way
23/L	[pushes A's head down, A laughs] She's out of the picture
24/A	[laughs]
25/R	or the Laura way [pause] the quick way [raises hand]
26/A	Whatever.
27/L	You mean the right way or the wrong way.
28/A	[laughs]
29/R	Define those.
30/A	There's the right way and the semi-right way, yeah?
31/R	The fast way and the Laura way.
32/L	You're not starving.
33/A	Kinda sorta.
34/	I'm unemployed remember [laughs].
35/R	Okay, okay, okay, we'll make a list if you wanna, lists are easy.
36/A	Okay so let's let's do it [pause] check out dot h.

The team considers two different ways in which the eavesdrop command could be implemented. R's suggestion is to use a character array, a data structure capable of representing the name of a single eavesdropper. L, however, insists that what is needed is actually a linked list of character arrays. The difference between the two suggestions is that a character array can store the name of one eavesdropper, while a linked list can store an indefinite number of eavesdroppers' names.

R's comments in lines 4 and 5, his whispering, and his laughter all suggest that he is aware of this difference and that he considers his own suggestion to be less than optimal. However, R points out that the program is currently set up so that a maximum of 2 players can play the game. Because there will never be more than one eavesdropper in a 2-player game, all that is really needed to make the function work is a data structure to store a single player's name. A character array and a linked list will thus function equally well. While the character array is easier to implement, the linked list is preferable from a maintenance perspective because it can remain unchanged if and when the game is modified to support more users.

R's proposal meets with immediate support from A. Further, A's justification in line 7 appeals to the sentiment that the standard of the new code need not exceed the standard of the existing code. This justification occurred frequently in programmers arguments to justify suboptimal programming procedures, such as the failure to comment code or to provide warnings to users to alert them when they doing something illegal.

Even the most cursory inspection of the transcript points to the extent to which the programmers' arguments rely on the particular way in which they describe the two suggested solutions. These descriptions serve to frame the suggestions in ways which have definite implications for the choice of which solution to adopt. For instance, contrast the terms used by L with those used by R to refer to the two candidate solutions. Whereas L refers to the linked list as *the right way* (line 16), R frames the linked list suggestion as *a future enhancement* (line 9). Both phrases (*the right way* and *a future enhancement*) cast the linked list suggestion in a favorable light. However, while R and A imply that the linked list suggestion should be implemented by a future team, L implies that it is she, R, and A who should implement the linked list.

R's argument in favor of the character array solution is based on an appeal to simplicity: the character array is easier for the programming team to implement than the linked list. Appeal to simplicity, like the appeal to the standard of the existing code, is another justification which occurs repeatedly in programmers' arguments. The programmers who participated in this study clearly consider simplicity to be a virtue, especially when it refers to the complexity of the tasks for which they are responsible. So, by framing his own solution (in line 11) as *the simple way*, R implies that the character array is the preferable course of action.

In lines 12 and 16, L objects to R and A's joint proposal and offers her own way of framing the two solutions. L refers, in line 16, to the character array solution as *that way*, and to the linked list solution as *the right way*. L's use of the distal deictic in *that way* (line 16), (as opposed to the proximal deictic in *this way*) suggests that she is distancing herself from the character array solution. Moreover, her use of the definite rather than the indefinite determiner in *the right way* (as compared to *a right way*) emphasizes L's commitment to the linked list suggestion as the unique solution to the problem at hand.

Change over time

Tables 2a and 2b contain the words which each of the three programmers uses to refer to the linked list solution (2a) and the character array solution (2b). The way in which each programmer's characterization of the two solutions changes over the course of the argument is instructive as to their changing commitments to the respective solutions. As noted above, L initially refers to the character array as *that way* (line 16), and later as *the wrong way* (line 27). R, on the other hand, initially characterizes this suggestion with the relatively neutral phrase, *a character array* (line 2), later shifting to a characterization consonant with his appeal to simplicity. R refers to the character array solution as *the simple way* in line 11, *the quick way* in lines 22 and 25, and *the fast way* in line 31.

Similarly, L shifts from a relatively neutral characterization of the linked list as simply *a list* (line 3), to one which strongly suggests her commitment to it. By line 16 the linked list solution has become *the right way*, is described as being not *that hard* (line 20), and is referred to again in line 27 as *the right way*. Furthermore, R's characterization of the linked list solution undergoes a parallel shift in the opposite direction. He initially refers to the linked list solution with the neutral phrase *a list* (line 1). However, by line 9, it has become *a future enhancement*. And, in lines 25 and 31, R calls the linked list solution *the Laura way*. Referring to the solution in this way uniquely identifies L as responsible for the possible costs and benefits of the proposed solution and underscores the fact that she (at least at line 25) is its sole proponent.

Table 2a. Linked list solution.

L	R	A
	1/ a list	
3/ a list		
	9/ a future enhancement	
		10/ for the next rev
16/ the right way		
20/ not that hard		
	25/ the Laura way	
27/ the right way		
		30/ the right way
	31/ the Laura way	
	35/ easy	

Table 2a. Each column contains phrases used by programmers L, R, and A to refer to the linked list solution. Each phrase is tagged with its line number from the transcript in Table 1.

In this discourse, A emerges as the least firmly committed. Her only explicit reference to the character-array solution occurs quite late in the excerpt, as *the semi-right way* in line 30. This description contrasts both with R's phrase *the quick way* and with L's phrase *the wrong way* in a way which marks A as occupying a position intermediate to those of the other 2 programmers. Moreover, A's utterances with respect to the linked list solution indicate her initial alliance with R (*for the next rev* in line 10), and later capitulation to L's position (*the right way* in line 30).

Table 2b. Character Array Solution

L	R	A
	2/ a character array	
	11/ the simple way	
16/ that way		
	22/ the quick way	
	25/ the quick way	
27/ the wrong way		
		30/ the semi-right way
	31/ the fast way	

Table 2b. Each column contains the phrases used by programmers L, R, and A to refer to the character array solution.

In spite of his appeal to the democratic process, R eventually accedes to L's manner of framing the linked list solution. The contrast between R's early implication that the linked list solution is more difficult than the situation warrants and his characterization of lists as easy in line 35 clearly signal R's capitulation to L's framing of the solution.

Linked Pragmatic Scales

Several of the programmers' utterances point to their use of *pragmatic scales* as the context against which their statements should be interpreted. A pragmatic scale consists of objects or scenarios ordered along relevant semantic dimensions (see Levinson, 1983 for review). Once ordered, statements which concern one member of the scale entail propositions about other members of the scale. The programmers' discourse contains reference to (i) varying degrees of the ease of each implementation (*simple* versus *not that hard*), (ii) the expected speed of completing each implementation (*quick way*, *fast way*), (iii) the normative correctness of each implementation (*the right way*, *the semi-right way*, and *the wrong way*), and (iv) the degree of their own hunger (*hungry* versus *not starving*).

It is the interpretation of A's utterances in the context of linked pragmatic scales which lends them import extending beyond their literal content. A's rhetorical question, *Are you guys hungry or not?* (line 17), as well as her mention of the time (in line 19) suggest that she favors the character array solution because it will take less time to implement. All three of the programmers know that the faster they finish the program, the sooner they can go out to dinner. The admission of hunger by any one of the programmers, then, can be interpreted as the expression of the desire to finish the programming task quickly. Moreover, because it is usually faster to implement a simple solution than a more complicated solution, the goal of finishing the task quickly is best accomplished by implementing the simplest solution.

The logic behind A's mention of hunger (in line 17) and her statement of the time relies on the existence of linked pragmatic scales. For example, Figure A depicts scales linked by the assumption that the ease of implementing a particular piece of code is related to how quickly it can be implemented. Thus each point on the Ease of Implementation scale has a corresponding point on the Speed of Implementation scale. When scales are linked in this manner, locating an object on one scale carries the implication that it occupies the corresponding point on the linked scale. This makes it possible to implicate how quickly a solution can be implemented by making reference to the perceived ease of implementation. Similarly, stating that it will take a long time to implement a particular function implicates the difficulty of the task.

Figure A. Quick and easy. Figure A depicts two linked scales: the scale of the ease of implementation and the scale of the speed of implementation. Once linked, statements which locate an object on one scale implicate a corresponding location on the adjacent scale. For example, stating that a function will be implemented quickly, implicates that it will be relatively simple to implement.

Figures B and C depict scales linked by the knowledge of the specific situation of the programming team comprised of R, L, and A that the sooner they finish the programming task, the sooner they can eat dinner. Thus A's utterances in lines 17 and 19 implicate speed as an important factor in the decision of which solution to adopt. The combination of the link between speed and ease of implementation, R's framing of the character array solution as the simple way, and the links between lateness, hunger and the importance of speed, all conspire to promote the character array as possessing a quality which has been contextually established as the relevant quality on which the group's decision is to be based.

Viewed against the backdrop of the scales linked in Figures A, B, and C, L's statement about the linked list solution, (*It's not that hard* in line 20) can be construed as an attempt to relocate the linked list solution on the ease scale. Because the ease of implementation is linked to the speed scale, L implicates that her solution will not take an inordinate amount of time to implement. Note that, at least in line 20, L has not taken issue with any of the general principles. She does not argue against the principle that speed is related to difficulty (in fact, she relies on it). She does not argue against the notion that speed is a virtue, that simplicity is a virtue. And neither does she argue that

eating is an important goal. Instead she stays within the framework of the linked scales established by A.

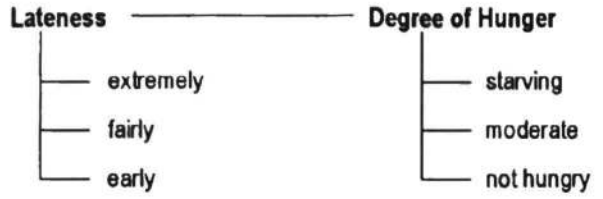

Figure B. Lateness and Hunger. Figure B depicts the linked scales of Lateness and the Degree of Hunger.

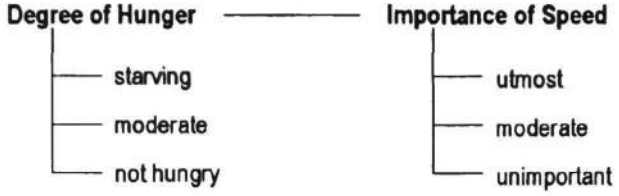

Figure C. Hunger = Importance of Speed. Figure C depicts the linked scales of Hunger and the Importance of Speed.

However, in the face of the alliance between A and R, L changes the character of the debate. Rather than trying to relocate the linked list solution on the scales of simplicity and speed, L reformulates the contrast between the two candidate solutions. Whereas R has framed the contrast as the *quick way* versus *the Laura way* (line 22 and 25), L frames the contrast as that between *the right way* and *the wrong way* (line 27). Right versus wrong is a bivalent distinction which clearly implicates the linked list solution as the only reasonable choice. A's response to this move by L is particularly interesting because it introduces a scalar reading of a categorical distinction. Calling the character array solution the wrong way implicates that it is not a viable candidate for implementation. However, a semi-right solution may be permissible if its implementation is sufficiently fast, easy, and warranted by the speed/quality trade-off.

Although the specific issue of debate concerns which of two possible ways the team should implement a particular facet of the eavesdrop command, the larger issue involves the negotiation of a speed/quality tradeoff. Another prevailing trend among the programmers who participated in this study, was first, the belief that writing high quality code (in general) takes longer than writing low quality code does; and second, a willingness to implement code which was -- in their own estimation -- less elegant, or suboptimal when time constraints were stringent.

The existence of a general model linking the importance of speed to the importance of quality coupled with the situation-particular model in which the degree of hunger is linked to the importance of speed, yields a link between the degree of hunger and the importance of quality. Because the general model dictates that the importance of speed is inversely proportional to the importance of quality, the programmers are able to link scales such that the degree of their hunger is inversely proportional to the importance of quality.

Conclusion

The above example shows how arguments about strategy work by successive framings and reframings of a particular task until the group settles onto one. Note how negotiation of general priorities is integrally connected with arguing about specific facets of the task itself. Thus both general priorities and subtask priorities fall out of these sorts of arguments. The data examined here suggest that strategy arguments do not generally concern the value of competing solutions. That is to say, all of the programmers agree on the attributes of each of the suggested solutions. Moreover, there is agreement as to which of those attributions are to count as favorable. The issue in the programmers' strategic argument is how their agreed upon values are to be brought to bear on the particular situation at hand.

While Tversky and Kahneman (1981) point to their discovery that people's decision was affected by differential framing as irrational behavior, we demonstrate how the ability to differentially frame situations is a component of rational behavior, where rationality is empirically construed. From the data examined herein, it would seem that Kahneman and Tversky's findings would come as no surprise to the participants in our study. The fact that they utilize framing techniques in their arguments concerning the choice between two programming solutions suggests, first, that they possess a tacit awareness of the influence of differential framing on decision making; and, second, that they are able to exploit this knowledge in their attempts to influence one another's behavior with respect to the task at hand.

References

Coulson, S. and Flor, N. (1994, in progress). The establishment of mutual mental models among collaborating programmers.

Flor, N. and E. Hutchins. (1991). Analyzing distributed cognition in software teams: A case study of team programming during perfective software maintenance. In *Empirical Studies of Programmers 4*. Norwood, NJ: Ablex Publishing Corporation.

Grice, H.P. (1967). *Logic and Conversation.* Unpublished MS. of the William James Society.

Hutchins. (1994, in press). *Cognition in the Wild.* Cambridge, MA: MIT Press.

Levinson, S. (1983). *Pragmatics*. Cambridge, England: Cambridge University Press.

Norman, D. (1986). Cognitive engineering. In D.A. Norman & S.W. Draper (Eds.), *User Centered System Design*. Hillsdale, NJ: Erlbaum.

Tversky, A. & D. Kahneman. (1981). The framing of decisions and the psychology of choice. *Science*, 1981 Jan, v211 (n4481):453-458.

Wason and Johnson-Laird. (1975). In R.J. Falmagne (Ed.) Reasoning: Representation and process in children and adults. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.