SESSION VII A TUTORIAL ON COGNITIVE SIMULATION

Alan Lesgold, *Presider*

The why, when, and how of cognitive simulation: A tutorial

DAVID KIERAS

University of Michigan, Ann Arbor, Michigan

Practical advice is given on cognitive simulation modeling; information is provided on why a simulation effort is worthwhile, when it should be undertaken, and how to do it. A cognitive simulation model is defined as a computer program that represents a hypothetical mental process that operates on symbolic structures.

For purposes of this presentation, a *cognitive simulation model* is a computer program that represents a hypothetical mental process that operates on symbolic structures. Note that there are other forms of computer simulation models, such as those used in certain areas of perception and memory research, in which the simulated processes involve quantitative entities such as activation levels. These models are clearly important in their own right, but this paper addresses the class of models involved with symbolic structures, such as semantic network models of memory processes and process models for comprehension.

Although simulation models have become the accepted form of advanced theory in cognitive psychology, very few researchers actually construct and test such models. Thus, there is a substantial gap between the central role of such models in theory and the extent to which this approach is actually used. A major obstacle to the increased use of the simulation modeling approach is simply that it is a difficult methodology to get started in, and for many researchers, the potential returns often seem small compared with the initial costs.

This paper is intended to provide some practical advice on why a simulation modeling effort might be worthwhile, when it is likely to pay off, and, especially, how to go about actually getting started and carrying out this approach.

WHY DO A COGNITIVE SIMULATION?

There are two major reasons for constructing a cognitive simulation model. The first is the traditional one, that constructing a rigorously specified computer program is an excellent way to convert a set of vague ideas into a more specific and precise theory. It is important to keep in mind, however, that the simulation program is *not* the theory, but only a specific realization of the theory. That is, we normally have a set of theoretical ideas that we are *really* interested in, but which are usually stated in a very general and imprecise way. In constructing a simulation model (and also, a traditional mathematical model), we are specifying a precise system that is based on these ideas. Attempting to specify the system in a way that runs on the computer reveals the consistency, completeness, or clarity of our theoretical ideas. However, if this effort is unsuccessful, there may be alternative realizations of the same general theoretical ideas that are successful.

Although this is the traditional reason for constructing a simulation model, the resulting benefits can be remarkable; I believe that cognitive theory would advance much faster if, for this reason alone, more researchers constructed simulation models.

The second reason for constructing a simulation model is to account for empirical data. Clearly, this is the standard goal for the traditional mathematical modeling approach; but simulation models have traditionally been excused from this standard, or questions have been raised about whether they could ever be held to it. However, recently efforts have been made to use simulation models in a serious and thorough way to account for data. These efforts are especially interesting because both the model and the cognitive processes are quite complex, and the

Work on this paper was supported by the Office of Naval Research Personnel and Training Research Programs under Contract N00014-81-C-0699, Contract Authority Identification No. NR 667-453. Requests for reprints should be sent to the author at the Technical Communication Program, College of Engineering, 1223 E. Engineering Building, University of Michigan, Ann Arbor, MI 48109.

accounting is done at a very fine level of detail in the data. On some occasions, the model functions essentially as a data-analytic tool; if the model can account for certain aspects of the data, then these aspects can be explained in terms of the model processes. If there are other aspects of the data that the model cannot account for, then there must be other processes involved in addition to those in the model. See Kieras (1981, 1984) for some examples. This allows, in essence, a theory-based way to examine the data.

The fact that such accounting can be done at a very fine level of detail is important because the level of detail possible with modern on-line data-collection methods is simply not exploited by conventional verbal or mathematical models, whereas simulation models are intrinsically committed to a very high level of detail.

Thus, a major reason for constructing a simulation model is to account for detailed empirical data on very complex phenomena and processes.

WHEN IS CONSTRUCTING A SIMULATION A GOOD IDEA?

There are several situations that indicate that a simulation model may be worth developing. However, given that simulation modeling is primarily a theoretical tool, it follows that it works best in the context of some theoretical ideas. That is, I do not recommend developing a simulation model unless you have some definite ideas about what theoretical principles must be involved. Otherwise, there are simply no constraints on the model, and it is likely to end up being simply a computer program, and not a realization of a cognitive theory. The situations are as follows:

(1) You have a set of verbally stated theoretical ideas or intuitions that you wish to make more specific and understand better, in the sense discussed above. An ideal situation involves a fairly well-worked-out verbal theory or a partially formalized theory that you want to turn into a detailed and well-defined process model. For example, Kieras (1982) started with some basic concepts of the van Dijk and Kintsch (van Dijk, 1977; Kintsch, 1977) partially formalized theory of macrostructures and developed a simulation model that represented some of these mechanisms.

(2) You have a well-characterized empirical phenomenon that you want to explain in detail, on the basis of theoretical ideas. Note that you cannot construct a computer program without a specification of what the program should do. You should specify precisely what behavior the model should have, so that the questions all involve *how* the model produces the behavior, not what the target behavior is. Thus, starting with a clear phenomenon to be explained is highly desirable.

(3) If you have started empirical work using a new paradigm or dealing with a new phenomenon, working out a simulation model for the task can help clarify what subjects might be doing in the task, and thereby lead to more

precise studies. Be on the lookout for this possibility.

(4) If your theoretical work is far enough along, developing a simulation model can help in the design of complex theory-based experiments. That is, you can try out your experimental manipulations on the simulation, and choose a set that is theoretically supposed to produce the desired effect. In complex tasks, unaided intuition about the effects of a manipulation can be wrong; the simulation provides a formal way to examine the manipulations before doing the experiment. In addition, if the predicted effects are obtained, the results provide a very strong confirmation of the theory.

An example of this approach appears in Kieras and Bovair (1984).

HOW TO DO A SIMULATION MODEL

Choice of Programming Language

Apparently, many would-be simulation modelers are deterred by the prospect of having to learn the LISP programming language, currently the most popular for cognitive simulations. This reluctance is unjustifiable. During the peak of popularity of mathematical models, we were expected to learn calculus and probability theory. Why is it too burdensome to learn a major programming language?

The recommendation here is simple: LEARN LISP! The reasons are: (1) LISP is the language used in almost all artificial intelligence (AI) work, which is the primary source of ideas and techniques for cognitive simulation. (2) The structures of the LISP language are very closely related to the concepts in current cognitive theory; in fact, much cognitive theory looks as though it were based on LISP. (3) Most LISP implementations provide a very powerful programming environment, such as amazing debugging facilities, and the language itself is more powerful than any other. Simulation programming is difficult enough; the researcher needs all of the help possible.

Over the years, various "packages" were supposed to allow nonprogrammers to construct simulation models. To date, most of these packages have not been popular, although some new efforts, such as Pat Langley's PRISM package, appear more promising. The package concept works in statistics because the situations in data analysis are quite well characterized; people essentially agree on the "theory" of an ANOVA. But, cognitive theory is not yet at this point. Thus, trying to use a simulation package involves adopting someone else's basic theoretical positions, which an individual may not agree with. Furthermore, current packages are all embedded in a LISP environment, meaning that they will be difficult to use without knowing at least the rudiments of LISP. Finally, if the package is not quite what you want, modifying it will require LISP expertise, and can be as much work as developing your own programs.

Thus, learning LISP is unavoidable. There are excellent textbooks, such as that by Winston and Horn (1981), that make the task much easier.

LISP basics. A very brief overview of LISP is provided to illustrate the basic simplicity and power of the language. Although this brief presentation is far from complete, it incorporates the essence and elegance of LISP. I will often compare LISP with the familiar FORTRAN, but keep in mind that LISP is made to manipulate symbols, whereas FORTRAN is specialized for number crunching.

LISP expressions are either *atoms* or *lists*. An atom is simply a symbol, such as NODE, G0343, 12.42, or MY-SYMBOL. Atoms can have *properties* attached to them, each consisting of a *property name* and a *property value*. An important symbol property is the VALUE assigned to the symbol itself; this enables an atom to behave as a variable name does in FORTRAN. Another property that can be attached to a symbol is the definition of a function, meaning that the symbol corresponds to a FORTRAN subroutine name. Finally, arbitrary property name-value pairs of the user's choice can be attached to atoms. Among other things, this makes it very easy to build complex network data structures. Each node in the net is just a symbol. The links leaving that node are represented by a property name for the link type. The node that the link points to is just the property value. Similar structures in FORTRAN are very clumsy to implement.

A list is in the form: (ITEM ITEM ITEM . . .). The ITEMS can be any LISP expression, such as another list, permitting a list to be of arbitrary length and complexity. This capability is excruciatingly difficult to obtain in FORTRAN-like languages. Thus, complicated nested expressions, which appear in work on chunking, for example, can be directly expressed in LISP.

The basic component of a LISP program is the *function*, which is similar to a FORTRAN FUNCTION subroutine. A LISP function takes some number of input parameters and returns a single value, and may also have some side effects, such as modifying the values of other variables or data structures. Writing a LISP program consists of defining a collection of functions that call each other to perform the desired computation. Breaking up an algorithm into simple functions is a key part of LISP programming style; structured programming is idiomatic in LISP.

Normally, LISP is run in the context of an interactive interpreter that does nothing more than wait for the user to type in a LISP expression. It then evaluates the expression and types out the returned value. The basic rule for the manner in which expressions are evaluated is as follows: If the expression is an atom, then its value is returned. If the expression is a list, then the first element of the list is considered to be the name of a function. The rest of the list comprises arguments to be passed to the function. The returned value is specified by the function definition.

There are two other features of LISP that are remarkable compared with FORTRAN. The first is that it is easy to operate on variable names as well as values, because variables are simply symbols that can be manipulated like

any other symbols. Their "variableness" comes only from an associated property. Thus, it is easy to actually construct a variable name, whose value is another variable name, whose value might be part of another variable name, and so forth. The second feature is that, unlike in FORTRAN, there is no built-in distinction between program and data; LISP code is simply LISP expressions that are subjected to evaluation. This makes it easy to implement advanced debugging facilities, and also to write code that generates and runs other code. More interestingly, it is easy to implement specialized higher level languages, such as production systems, by working in LISP.

Table 1 gives an example of a simple LISP session. The lowercase words are my comments explaining what is going on.

"Pure" vs. "vulgar" LISP. The LISP that appears in textbooks is usually what I call "pure" LISP. One of the distinctive features of LISP is the ease of using recursive functions, that is, functions that call themselves. Pure LISP involves doing everything in terms of recursive functions, and without the normal sequential flow of control that is basic to FORTRAN-like languages. For example, Table 2 provides a pure LISP definition of the factorial function. This body of the definition consists of only a single function, the COND conditional function. If $N = 1$, the expression (EQ N 1) will be true, the value of N will be returned, and the rest of the COND will be ignored. If not, the returned value will be N times the factorial of $N - 1$. When this function is called, it computes the result by calling itself $N - 1$ times, and then returns back up through the nest of calls, doing the multiplications on each returned value. This is elegant, but distinctly counterintuitive for the ex-FORTRAN programmer.

Table 2 also shows the "vulgar" version of the same function, which resembles closely the corresponding FORTRAN code. The body of the definition is a PROG expression, which allows definition of the function using sequential flow of control, statement labels, and GOTOS, as in FORTRAN. The function first sets the local variable PRODUCT to N and then enters a loop in which N is decremented; then PRODUCT is set to N times the PRODUCT. When N reaches 1, the value of PRODUCT is returned.

Compared with vulgar LISP, pure LISP is difficult to understand (unless your mind is naturally recursive), is often inefficient, depending on the implementation, and can be quite difficult to write. However, certain applications, such as searching networks, can be far simpler to write as recursive functions, since LISP can automatically provide much of the bookkeeping involved in keeping track of where you are in the search. On many occasions, this power alone is adequate justification for LISP.

I suspect that one reason why many people are hesitant to try to learn LISP is that most textbooks present the pure approach, making it look like an individual has to be a recursive genius to use LISP. My advice is to use vulgar LISP, especially in the beginning, and save recursion for

Table 1
A Sample LISP Session (Comments are in Lowercase)

1>(SETQ L '(A B C D))	this assigns the list of symbols, A,B,C, and D as the value of L; the quote mark means to take the following expression literally, rather than evaluating it
(A B C D)	the value returned by SETQ
2>L	evaluating the atom L gives the value
(A B C D)	
3>(CAR L)	CAR returns the first element of the list, which is the symbol A
A	
4>(CDR L)	CDR returns the rest of the list
(B C D)	
5>(SETQ L (APPEND L L))	set the value of L to L appended to itself
(A B C D A B C D)	
6>(DEFINEQ (MY-FUNC (LAMBDA (X) (PRINT X))))	define a function, MY-FUNC, which prints and returns the value of its argument
(MY-FUNC)	
7>(MY-FUNC (CAR L))	(CAR L) is evaluated first, then MY-FUNC
A	this is the effect of PRINT, which then returns its argument, which MY-FUNC then returns, giving a second "A"
A	
7>(SETQ ZOT '(MY-FUNC (SETQ X 'HELLO)))	this results in ZOT being set to the list expression that is then returned
(MY-FUNC (SETQ X 'HELLO))	
8>(EVAL ZOT)	we can ask that a datum, the value of ZOT, be evaluated or "run" as though it were a program; the inner SETQ is evaluated first, followed by MY-FUNC
HELLO	the effect of PRINT
HELLO	the returned value
9>X	evaluate X, which as a side effect of the previous step, now has the value returned
HELLO	

the situations in which it is actually the simplest approach. With experience, the investigator will probably want to try his or her hand at pure LISP.

Two brief examples of LISP coding will be shown to demonstrate that this language makes sense for representing cognitive processes.

Semantic networks. Semantic networks represent a standard theoretical concept that can be implemented in LISP trivially, using the LISP primitive functions PUT-PROP and GETPROP, which access the properties of atoms.

Suppose that we want to represent the net shown in Figure 1. Each node is represented simply by a symbol, which for convenience is the name of the concept (it can equally well be an arbitrary generated symbol, such as G0343). In the example, these nodes will be the symbols FROGS, GREEN, BUGS, and LEGS.

The network can be constructed simply by having LISP evaluate a series of expressions, one for each link, in the form of (PUTPROP FROM-NODE LINK-TYPE TO-NODE). For example, (PUTPROP 'FROGS 'COLOR 'GREEN) puts onto the property list of the atom FROGS the name-value pair COLOR-GREEN. The inverse links are constructed by evaluating expressions such as (PUT-

PROP 'GREEN 'COLOR-INVERSE 'FROGS). After the network is constructed, we can access any of the nodes with (GETPROP NODE LINK-TYPE), such as (GETPROP 'FROGS 'COLOR), which returns the symbol GREEN.

Thus far, we can have only one node pointed to by each type of link. This can be handled by a somewhat more complicated approach of having the property value for

Table 2
"Pure" Versus "Vulgar" LISP

Pure LISP
(DEFINEQ (FACTORIAL (LAMBDA (N)
(COND ((EQ N 1) N)
(T (TIMES N (FACTORIAL (DIFFERENCE N 1))))))
Vulgar LISP
(DEFINEQ (FACTORIAL (LAMBDA (N)
(PROG (PRODUCT)
(SETQ PRODUCT N)
LOOP
(COND ((EQ N 1)(RETURN PRODUCT)))
(SETQ N (DIFFERENCE N 1))
(SETQ PRODUCT (TIMES PRODUCT N))
(GO LOOP)
)))

Figure 1. A simple semantic network example.

a link consist of a *list* of the nodes pointed to by that type of link. Thus, if there were more than one reference to the GREEN concept, (GETPROP 'GREEN 'COLOR-INVERSE) would return (FROGS LEAVES EGGS HAM). We would add a new node for a link type by first getting the old node list, adding the new node, and replacing the list as the property value, thus:

```
(PUTPROP FROM-NODE LINK-TYPE
(CONS NEW-NODE (GETPROP FROM-NODE
LINK-TYPE)))
```

Of course, we can define functions that construct or access networks by using variables to specify the nodes and links. For example, part of a comprehension model will consist of a function that takes a list of the nodes and links specified by an input sentence, and constructs the corresponding semantic network. The different models of semantic memory differ basically only in the set of property names used for link types.

Production Systems. Production systems are a popular approach to building cognitive models, for reasons discussed below. There are many technical issues involved in implementing production systems efficiently, but here we can demystify the programming and illustrate some important LISP features.

Table 3 has the code for a simple production system interpreter. It is written in a plain, open style, rather than using any advanced LISP features. This style offends LISP purists, but I have used it here because it is easy to understand and transport to other LISP dialects. The production rules are assumed to be in a list, supplied as an argument, in which a rule is in the form:

```
(IF (condition expression) THEN (action expression))
```

The first line of the code defines INTERPRETER as a LISP function with one argument, PRODUCTION-LIST. The LAMBDA has to do with the mathematical ancestry of LISP, and can be ignored. In the second line, the remainder of the function definition is given as a PROG, which allows us to write the code as though it were a FORTRAN-like language with sequential flow of control, statement labels, GOTOs, and so forth. See the Table 1 example for descriptions of some of the common functions, such as SETQ and CAR.

Basically, the algorithm consists of getting each production in the list and then evaluating its condition; if the condition returns true, then the action expression is evaluated ("run"). This process is repeated until we scan the list without firing a production, whereupon we stop, returning FINISHED.

Thus, the basics of a production system interpreter are quite simple in LISP. The complexities of a practical interpreter involve making the pattern matching specified by a condition adequately efficient, and avoiding the brute-force serial exhaustive scan of the production conditions. Although solving these technical problems can be difficult, it is relatively easy to accomplish in LISP.

Approach and Evaluation

Arbitrary vs. architectural code. An important issue is determining the basic approach to writing the simulation program. A natural tendency is to write arbitrary code and data structures without regard to their relation to the psychological theory or any overall consistency. The only requirement of the code is that it generate the correct behavior using processes that map onto the psychological theory at a gross level. For example, there might be a function that scans a short-term memory representation, and another function that implements various strategies for long-term memory search, all written in whatever way seems easy and convenient at the time.

Another approach is an architectural one, in which the basic architecture of the cognitive system is defined, and all of the coding is done in those terms. The use of production systems, instead of arbitrary programs, is a good example of this approach; the use of semantic networks instead of arbitrary data structures is another. Another example is the Anderson and Bower (1973) decree that all memory retrieval is done with a single, uniform process, the MATCH process.

The advantage to the architectural approach is that the link to the psychological theory is much easier to establish, and it is thus much easier to explain the simulation,

Table 3
A Simple Production System Interpreter

```
(DEFINEQ (INTERPRETER (LAMBDA (PRODUCTION-LIST)
(PROG (PRODUCTION CONDITION ACTION FIRED-FLAG PL)
(SETQ FIRED-FLAG T)
START-CYCLE
(COND ((NOT FIRED-FLAG)(RETURN 'FINISHED)))
(SETQ FIRED-FLAG NIL)
(SETQ PL PRODUCTION-LIST)
NEXT-PRODUCTION
(COND ((NOT PL)(GO START-CYCLE)))
(SETQ PRODUCTION (CAR PL))
(SETQ PL (CDR PL))
SETQ CONDITION (CADR PRODUCTION))
(SETQ ACTION (CADDR PRODUCTION))

(COND ((EVAL CONDITION)
(EVAL ACTION)(SETQ FIRED-FLAG T)))

(GO NEXT-PRODUCTION)
))))
```

if there is a uniform cognitive architecture. Since the goal is to develop cognitive theory, as opposed to simply getting a program to run, having the theoretical basis explicit in the architecture is vital.

Evaluation of a simulation. Once a simulation model is constructed, how do you tell whether it is any good? This is a classic problem, discussed elsewhere in some detail (Kieras, 1983, 1984). I offer the following criteria:

(1) Does the simulation model display apparently realistic behavior? That is, does it do what it is supposed to do, in a way that is conceptually reasonable? Normally it will, because this is the usual goal of the programming process. But note that a language comprehension model that did not handle the relationship of active and passive paraphrases would not seem very realistic, no matter its other virtues. Thus, we demand that, at a minimum, the model be in the "right ballpark" in its behavioral domain.

(2) If you based the model on a theory, can you explain how the model works in terms of the theory in a straightforward way? If so, the model is a good realization of the theory. If you cannot explain the model without becoming involved in the details of the code, you have a situation in which the relation of the model to the theory has been lost. For example, if you must say things such as, "At this point we compute the the APPEND of the CADR of the MEMORY-ABSENT-LIST with the SEARCH-OBJECT-LIST of the GETPROP of the active node," you've got a problem. There should be a level of analysis of the model at which there is a one-to-one mapping to psychological concepts.

(3.) If you used an architectural approach, were you able to stay within the architecture, or did you have to "kludge" the code to get the system to work? For example, if many of your production rules have arbitrary LISP code in their actions, there must be a gap in your architectural assumptions. Go back to the drawing board!

(4) If you are dealing with a well-characterized phenomenon, your model should address the data reasonably precisely, and perhaps even quantitatively. Depending on the nature of the data and the model, you can use some standard statistical and model-testing techniques, as in Kintsch and van Dijk (1978) and Kieras (1981, 1984).

On this subject, simulation models are often criticized as having so many "degrees of freedom" that they can "fit" any set of data. In a sense, this is true, because the entire program can be considered as a giant free parameter, which can be modified to fit the data. However, in an architectural approach, arbitrary modifications of the program that violate the architecture can be ruled out. In fact, it has been my experience that these "degrees of freedom" are an illusion; it is usually quite difficult to get apparently realistic behavior out of a fixed architecture, much less arbitrarily good fits to data.

Practical Advice

Requirements. In order to do simulation work, you need any decent LISP system, that is, one that can handle full-sized programs, has an editor, debugging facilities, and disk file capability, runs interactively, and has

a compiler. At this time, whether any micro-based LISPs are adequate is not clear, but there are many mainframe and supermini LISPs that are. Despite an apparent current popular belief, you do not need a "LISP machine," which is a hardware-software package specialized for running LISP, such as the Xerox 1108 Dandelion. These machines have their advantages, but are not essential. They are quite expensive, and are not useful for anything else. A suitable general-purpose machine may be a better acquisition. You will need, however, a lot of computer time—a dedicated or "free" system is ideal; working late at night on most time-sharing systems is a good idea.

Perhaps the most difficult requirement to meet is that of time—you will need a lot of time. Allow at least a month to become fluent in LISP if you are already an experienced programmer. Try to get long blocks of uninterrupted time; this work requires concentration. Finally, simulation modeling does not result in a high rate of publication, so junior faculty should be especially careful about committing the large amounts of time involved.

Style and techniques of programming. A basic feature of simulation programming is that, unlike most other programming, you don't really know what the code is supposed to do before you write it; you are constructing a model of the cognitive processes, whose operation you don't already understand! Consequently, compared with most programming, simulation programming involves a lot of tentative programming. Your code is modified, re-modified, scrapped, and started over. This is where you are doing the psychology; you are defining in detail the actual possible cognitive processes. There are some steps that can be taken to make this process easier:

(1) Code in plain LISP. Being cute and tricky just makes it harder to understand later what the code is doing.

(2) Commenting the code is especially important. It is amazingly easy to forget how the code works.

(3) Take advantage of LISP's encouragement of structured programming. Dividing the code into clear modules makes revisions easier.

(4) Good LISP implementations have a lot of convenience features, such as automatic updating of source files. These should be learned, because they can save a lot of time.

(5) Study AI sources to get ideas about specific techniques. Charniak, Riesbeck, and McDermott (1980) is currently the only AI textbook that attempts to present advanced techniques in detail, but it strikes many as difficult to understand. Keep in mind that many of the approaches of special interest to cognitive psychology are not necessarily of major interest to AI researchers.

How to get started. I recommend the following steps to get started in simulation modeling:

(1) Learn LISP, using Winston and Horn (1981) and any LISP system that you can spend unlimited time on, even if it is a limited micro-based implementation.

(2) Get access to a full-scale LISP implementation, such as Franz LISP or COMMON LISP on a VAX.

(3) Focus your theoretical ideas, and, if possible, choose a specific phenomenon to account for, so that you

have a clear goal. Do not try to build a large-scale or comprehensive model until you have worked out a smaller system to your satisfaction.

(4) If possible, obtain a copy of an existing simulation model that does things of the same general type that you are interested in. Study this model. This provides a framework for starting and allows you to learn LISP usage in a meaningful context. Change this model so that it does what you want, rewrite it, or use your experience to start your own model from scratch.

(5) If the coding seems to become too difficult, or the model too slow and unwieldy, stop and consider whether your architectural assumptions or basic theoretical ideas are still good, and react accordingly.

(6) After the model is working, compare it with some data in detail, paying close attention to where the model fits and where it doesn't. Does the model explain puzzling aspects of the data? Where it fails to fit, is this due to defects in the basic assumptions, or to the presence of unrepresented processes?

REFERENCES

- ANDERSON, J. R., & BOWER, G. H. (1973). *Human associative memory*. Washington, DC: Winston.
- CHARNIAK, E., RIESBECK, C. K., & McDERMOTT, D. V. (1980). *Artificial intelligence programming*. Hillsdale, NJ: Erlbaum.
- KIERAS, D. E. (1981). Component processes in the comprehension of simple prose. *Journal of Verbal Learning and Verbal Behavior*, **20**, 1-23.
- KIERAS, D. E. (1982). A model of reader strategy for abstracting main ideas from simple technical prose. *Text*, **2**, 47-82.
- KIERAS, D. E. (1983). A simulation model for the comprehension of technical prose. In G. H. Bower (Ed.), *The psychology of learning and motivation* (Vol. 17). New York, NY: Academic Press.
- KIERAS, D. E. (1984). A method for comparing a simulation model to reading time data. In D. Kieras & M. Just (Eds.), *New methods in reading comprehension research*. Hillsdale, NJ: Erlbaum.
- KIERAS, D. E., & BOVAIR, S. (1984, June). *The acquisition of procedures from text*. Paper presented at the Sixth Annual Conference of the Cognitive Science Society, Boulder, CO.
- KINTSCH, W. (1977). On recalling stories. In M. Just & P. Carpenter (Eds.), *Cognitive processes in comprehension*. Hillsdale, NJ: Erlbaum.
- KINTSCH, W., & VAN DIJK, T. A. (1978). Toward a model of discourse comprehension and production. *Psychological Review*, **85**, 363-394.
- VAN DIJK, T. A. (1977). Semantic macro-structures and knowledge frames in discourse comprehension. In M. Just & P. Carpenter (Eds.), *Cognitive processes in comprehension*. Hillsdale, NJ: Erlbaum.
- WINSTON, P. H., & HORN, B. K. P. (1981). *LISP*. Reading, MA: Addison-Wesley.