Towards a Hermeneutic Understanding of Programming Languages
Clay Spinuzzi

1. Composition teachers--and theorists--have recently begun to turn to *post-process theory* to discuss how people communicate (Kent, *Post-Process*). In a nutshell, post-process theory postulates that language is not inherently systematic and cannot be taught or analyzed as a system. That's quite an about-face for composition pedagogy, in which one theory after another has laid out a systemic understanding of communication. For instance, composition teachers in the early half of the twentieth century taught a rather rigid system of grammar and syntax that James Berlin later labeled "current-traditional rhetoric." In the 1980s, sociocognitivists began to discuss language in terms of a system of processes and schema. And later in that decade, social constructionists theorized a system of social language conventions that included not only syntax and grammar, but also expressions, tropes, and other characteristics of languages used by particular discourse communities. These theories of communication, although they differed radically from one another, are all rooted in the notion that language is inherently systemic. Post-process theory questions this notion, and thus questions how rhetoric has been taught since Aristotle.

2. But in this article, I'm less concerned with how rhetoric has been taught than I am in how a post-process theory of rhetoric can be applied to other domains. Here, I discuss one post-process theory, *paralogic rhetoric*, as it is portrayed in Thomas Kent's 1993 book of the same name. Kent, drawing heavily from analytic philosopher Donald Davidson, argues systemic understandings of language can explain part but not all of communicative interaction. Communication, he says, is not logical but *paralogic*, that is, "beyond system" (to borrow the title of one of Kent's articles). Kent and Davidson argue this point through numerous thought experiments involving malapropisms, neologisms, slang, ambiguities, and similar language phenomena. (Some of these thought experiments will be discussed later in the article.) And in these thought experiments, one recurring trope is that of an *interpreting machine*, a hypothetical automaton or program that is "set to grind out the meaning of an arbitrary utterance" (Davidson 445). Kent and Davidson demonstrate in their thought experiments that such a machine or system ultimately cannot account for the sorts of hermeneutic interactions we take for granted. Certainly machines can *process syntax*, if by that we mean that they can follow a tightly bounded set of instructions; but people *communicate*, and in a way that cannot be explained by such a codified process. The quotes on the left side of Figure 1 offer a couple of samples of this trope.

3. On the right side of Figure 1 is an entirely different type of quote--the sort of text that Kent and Davidson explicitly disallow from their discussion. This is a chunk of programming code written in the computer language called C. In fact, it's from the kernel of the Linux operating system that you may have heard about. And it's utterly systematic. That is, once a programmer writes code, she uses a program called a compiler to convert it into machine code that the computer can run. The compiler processes the code by converting the tightly controlled, systematically defined syntax and vocabulary into a set of bits. The compiler
does not interpret the code in any meaningful sense any more than a combination lock interprets the combination that one keys into it. The combination simply moves some tumblers in a particular sequence; the code simply sends a set of simple instructions to the computer's microprocessor in a particular sequence. On the face of it, there seems to be no real connection between this code on the right side of the screen and the paralogic rhetoric described on the left.

**Paralogic Rhetoric**

"[W]e have discovered no learnable common core of consistent behavior, no shared grammar or rules, no portable interpreting machine set to grind out the meaning of an arbitrary utterance."

(Davidson 445)

"Clearly, we can talk about regulative sign systems in the sense that we share rulelike conventions of language in order to communicate, but the ability to generate these marks and noises should not be confused with the ability to communicate with other language users. Machines, for example, can be programmed to generate words and sentences, but machines cannot triangulate; they cannot, in other words, communicate. . . . From an externalist point of view, to say that machines cannot communicate is simply to say that communication is more than just wiring and that no model of cognitive processes or brain function can duplicate the complex relation that exists between language and human social life." (Kent, *Paralogic* 158)

**Programming Code**

```c
asmlinkage long sys_getpriority(int which, int who) {
    struct task_struct *p;
    long retval = -ESRCH;
    if (which > 2 || which < 0)
        return -EINVAL;
    read_lock(&tasklist_lock);
    for_each_task (p) {
        long niceval;
        if (!proc_sel(p, which, who))
            continue;
        niceval = 20 - p->nice;
        if (niceval > retval)
            retval = niceval;
    }
    read_unlock(&tasklist_lock);
    return retval;
}
```

4. Fig. 1.
Paralogic rhetoric opposes system; programming code seems to embody it.

5. But there is a connection. What is not understood by most of us outside of computer science--and forgotten even by many inside that discipline--is that this code is not actually for the machine's use. It's a collaborative tool meant to help programmers share and review their work with others. Although the computer does not interpret this code, programmers do. Indeed, there has been considerable study of how programmers comprehend code, and that literature has startling parallels with the tenets of paralogic rhetoric laid out.
Furthermore, this literature does something that Kent, Davidson, and others in the humanities have not done: it has gone beyond thought experiments to support its points with empirical studies.

6. In this paper, I explore some of these parallels, matching some of Kent's tenets with similar tenets outlined in a seminal paper that Ruven Brooks wrote in 1983, ten years before Kent published *Paralogic Rhetoric*. I briefly review some of the empirical studies that have supported Brooks' points. Finally, I discuss some of the differences between Kent's and Brooks' ideas and talk about the implications for paralogic rhetoric.

**Hermeneutic Guessing vs. Recursive Refining of Hypotheses**

7. Perhaps the most distinct tenet of paralogic rhetoric is that of hermeneutic guessing. Kent argues that linguistic interpretation, rather than being a systematic and codifiable process, is ultimately guesswork; we can't rely on cognitive processes or defined syntax or discourse communities to provide us with a surefire method of interpretation, and in fact it's quite unlikely that the contents of the reader's thoughts will ever be identical to those of the writer. Rather, we come to a communicative interaction with a prior theory of what a text might mean, and we develop on-the-spot passing theories to constantly adjust or improve our hermeneutic guesses about what the text means. Interpretation, then, is interactionist in that it relies on continuous interactions or adjustments in the reader's understanding of the text. (See the left side of Figure 2.)

8. Brooks describes a very similar phenomenon. He breaks with the assumptions of his predecessors, who saw program comprehension as an essentially linear process that mimics the work of the compiler. Although program comprehension does sometimes involve this sort of stepwise examination, Brooks argues, it usually involves a very different approach: "the successive, top-down refinement of hypotheses about other knowledge domains and their relationship to the executing program" (Brooks 545). These hypotheses begin, Brooks asserts, as soon as the programmer hears the program's name, a short description of it, or an explanation of what it is intended to do. Obviously, none of these elements is structural or inherent in the code itself. The programmer's hypotheses are successively, interactively refined as the programmer reads the code, just as hermeneutic guesses are progressively refined as the reader examines the text. (See the right side of Figure 2.)

9. The program comprehension literature since 1983 is full of evidence for Brooks' assertion. For instance, Robertson et al. found that programmers frequently revisited code that they had already read--or written--in order to sharpen their hypotheses about the program. And Von Mayrhauser and Vans found that the programmer they observed would draw on an arsenal of hypotheses--the equivalent of prior theories--for understanding code. Unlike a compiler, the programmer would opportunistically abandon hypotheses that became too time-consuming, instead formulating different ones that fit better with what he was observing in the code.
<table>
<thead>
<tr>
<th>Hermeneutic guessing; prior and passing theories</th>
<th>Recursive refining of hypotheses</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;[T]o get things done in the world, we must bringinto agreement or match not only our codifiable forestructures ofwords, sentences, and linguistic conventions; we must match as well our forestructures of judgment and belief, and these forestructures never match precisely.&quot; (Kent, Paralogic 16)</td>
<td>&quot;One view of the process of understanding a program is that it is fundamentally bottom-up and that the programmer begins by looking at the individual lines of code or at groups of lines and assigning them interpretations. The interpretations are in terms of domains close to that of the program text, such as that of simple algorithms. . . . While most programmers will recall instances when they have attempted to behave in this way, this theory asserts that this sort of inductive behavior is only a degenerate special case of a more powerful process. This process is based on the successive, top-down refinement of hypotheses about other knowledge domains and their relationship to the executing program.&quot; (Brooks 545)</td>
</tr>
<tr>
<td>&quot;When we guess, we shift ground linguistically in order to convey information or decipher information, and the guesswork that allows us to shift ground relies on paralogical elements of language use, elements like skill, intuition, taste, and sympathy.&quot; (Kent, Paralogic 40)</td>
<td>&quot;This theory asserts that the hypothesis generation and verification process begins with a primary hypothesis that is generated as soon as the programmer obtains any information about the task that the program performs. . . . It is claimed that the primary hypothesis is often created by the time the programmer has heard only the name of the program or a brief descriptive phrase[.]&quot; (Brooks 546)</td>
</tr>
</tbody>
</table>

10. Fig. 2. Parallels between hermeneutic guessing and the recursive refining of hypotheses.

**Triangulation vs. Constructing Domain Information**

11. Kent says that hermeneutic guessing works because of triangulation, in which two language users converge on a shared meaning by referring to a shared world. It's not enough for language users to swap utterances because without a shared world to which those
utterances refer, the language users cannot improve (that is, confirm or deny) their guesses. David R. Russell gives an example of triangulation in an oft-quoted passage:

For example, a seven-month-old child who has not yet learned her first word reaches in the direction of a spherical object and babbles. Her parent, seeing this, puts the object in her hands and says, "Ball! You want to play with the ball?" Sooner or later—usually sooner—the child learns that adults may play with her using spherical objects and that certain sounds ("ball") and certain activities accompany human interactions with such objects. She learns through observing others' actions and her own that making the sound "ball" in certain situations often produces certain effects in others. Triangulation has been achieved—and learning. (181)

12. The parent and child can converge on a meaning here because they have a shared artifact to which they can refer, one that they perceive similarly. Without a shared world to which we can interactively refer, Kent argues, we cannot improve our hermeneutic guesswork; we would simply be swapping syllables and marks. (See the left side of Figure 3.)

13. Kent, Davidson, Russell, and others who deal with paralogic notions of language tend to rely on examples of radical interpretation—examples in which two language users have almost no language in common. (Davidson is fond of bringing space aliens into his thought experiments.) Although Brooks does not assume such a purely radical situation, he does introduce a parallel to triangulation, one that is a vital part of his theory and that has become very important to subsequent studies of program comprehension. He argues that to gain more than a strictly functional understanding of the code—to understand the original programmer's intentions—the programmer who reads the code must understand the domain in which the program is intended to work, the problem it is meant to solve, and even the hardware on which it should run. Without that understanding, Brooks points out, the programmer will not be able to make useful sense of the program (i.e., productively refine hypotheses) or be able to modify the program in a satisfactory way. (See the right side of Figure 3.)

14. Empirical evidence of this point has been relatively thin in program comprehension research, primarily because much of that research focuses on typified low-level programming moves or artificial problems rather than naturalistic studies of programming. But Nanja and Cook's 1987 study suggests that Brooks' account is sensible. In this latter study, experienced programmers who attempted to debug a program tried to understand it as a whole—its intent and its context—and were better prepared to identify and debug logical errors than non-expert programmers.

<table>
<thead>
<tr>
<th>Triangulation</th>
<th>Constructing domain and hardware information</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;For Davidson, the intimate and seemingly subjective knowledge that each of us possesses about our own mind arises only through triangulation with the other, other language users and other objects that...&quot;</td>
<td>&quot;[T]he task of understanding a program for a programmer becomes one of constructing, or reconstructing, enough information about the modeling domains that the...&quot;</td>
</tr>
</tbody>
</table>
constitute our shared world. . . . In order to surmise if our marks and noises create any effect in the world, we require at least another language user and objects in the world that we know we share. In order to communicate, we need to triangulate." (Kent, *Paralogic* 90)

original programmer used to bridge between the program and executing program. For example, if the programmer is given the task of modifying the dynamic processing algorithm in [a] cargo rate program[, the objects and operations in the domain of the original algorithm must first be learned about. Then, the programmer must find out about how these objects and operations appear in the particular programming language in which the program is written. If the algorithm is sensitive to issues of arithmetic precision, the programmer will also have to learn about the characteristics of the hardware on which the program will be run." (Brooks 545)

15.

Fig. 3.
Triangulation is a well developed concept in paralogic rhetoric; it is paralleled by Brooks' less developed concept of domain and hardware information.

**Genre vs. Beacons**

16. Let's return to paralogic rhetoric. To flesh out an account of hermeneutic guessing, Kent turns to genre. Genre is an important issue for Kent, so important that he devotes an entire chapter of *Paralogic Rhetoric* to it. Genre is an interpretive unit, not a syntactic one, and it provides a shared prior theory for language users. For instance, someone who writes a memo will likely adhere to the genre's conventions by starting with the memo header; someone who reads that memo can recognize this convention and guess that the text will in fact be a memo. Genre is not a surefire way to synchronize prior theories, but it does present a starting point for making, confirming or disconfirming, and refining hermeneutic guesses. (See the left side of Figure 4.)

17. Similarly, Brooks introduces the concept of beacons. Like a genre, a beacon is an interpretive unit, not a syntactic one, and it provides programmers with a starting point for developing, confirming or disconfirming, and refining hypotheses about segments of the code. For instance, take a look at the program code that I introduced in Figure 1. The individual lines in this code fragment are governed by strict syntactic rules. But there is latitude in how these instructions are combined and sequenced. For instance, notice that we have a LOCK statement, then a FOR statement, then an UNLOCK statement. These don't have to be executed in this particular order, and in fact the programmer could theoretically provide an infinite series of LOCK statements; the compiler doesn't care. But the particular combination here is familiar to any programmer who is moderately experienced with file
access. It gives programmers an idea of what the original programmer's intention was and helps them to refine their guesses about how this chunk of code helps to meet the program's goals. (See the right side of Figure 4.)

18. Beacons have been extensively studied in the program comprehension literature, as have related concepts such as plans, schema, kernels, and cliches. These studies suggest that programmers segment code in similar (but not identical) interpretive units; that the more experienced the programmers are, the more subunits they identify; and that when researchers deliberately obscure beacons, programmers experience poorer comprehension and recall. In other words, programmers seem to use beacons to improve their guesses and converge on meaning, but beacons do not provide structural guidance that leads to identical interpretations.

<table>
<thead>
<tr>
<th><strong>Genre</strong></th>
<th><strong>Beacons</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bakhtin's notion of genre &quot;corresponds to an open-ended and uncodifiable strategy for hermeneutic guessing that comes into being through triangulation, or what Bakhtin formulates as open-ended dialogue.&quot; (Kent, <em>Paralogic</em> 128)</td>
<td>&quot;Sets of features that typically indicate the occurrence of certain structures or operations within the code will be referred to as 'beacons' for that structure or operation.&quot; (Brooks 548)</td>
</tr>
<tr>
<td>&quot;Because we must guess about the genre to which an utterance belongs in order to recognize the boundaries of an utterance and to recognize its tenuous finality, all communicative interaction, therefore, derives from our ability to recognize and generate genres, for we only communicate through the utterance as genre.&quot; (Kent, <em>Paralogic</em> 136)</td>
<td>&quot;[T]he discovery of a beacon results in a further refinement and specification of the hypothesis itself.&quot; (Brooks, 549)</td>
</tr>
<tr>
<td>&quot;As we read a text, we continually try to match our hermeneutic strategy with another's strategy, even if the other is us. When we write something, we attempt to match our hermeneutic strategy with the strategy we think someone else might employ to interpret what we write, and obviously, we may be wrong. So, in both the reception and production of texts, we make hermeneutic guesses about how texts should or will be interpreted. Within this guessing game, genre constitutes the essential interpretive element, for only through genre . . . may we begin to</td>
<td>&quot;[T]he programmer first forms a primary hypothesis about the overall functions performed in the program. From this primary hypothesis, a cascade of subsidiary hypotheses are formed until a level is reached at which the programmer can verify the code against beacons in the program text.&quot; (Brooks 549)</td>
</tr>
</tbody>
</table>
formulate a hermeneutic strategy that, in turn, will allow us to make sense of an utterance." (Kent, *Paralogic* 145)

19.  
Fig. 4.  
Genres have strong parallels with beacons.

**Similarities and Differences**

20. Well - so what?

21. On the plus side, the program comprehension literature provides empirical support for concepts that so far have been supported primarily through thought experiments. It also allows us to get at the difference that Kent has drawn between hermeneutic and systematic aspects of language, particularly because the systematic aspects of computer languages are far more regulated than those of conversational languages. Program comprehension, in other words, presents an important testbed for paralogic rhetoric's tenets. Finally, this comparison leads us to see programming as another form of writing, another form of human communication, and it gives us new avenues for cross-disciplinary work. Such avenues are particularly interesting to rhetoricians as we begin to do our own programming--of applications, Web sites, databases, and other software. (See the left side of Figure 5.)

22. But we should also heed the differences between Kent and Brooks, and thus between the two bodies of literature they represent, because these two bodies of literature do portray communication in quite different ways. Although I've documented strong parallels, ultimately, Brooks and others in the program comprehension literature still see program comprehension as a logical rather than a paralogic phenomenon. They do reject the older understanding of program comprehension as a linear process, but they continue to see it as a process that can ultimately be comprehensively modeled. (There are counterexamples in the literature, however [see Guindon et al.; Koenemann and Robertson].) Finally, interpretation does not go "all the way down": theoretically, a programmer can reach a complete understanding of a given program. So we must be cautious as we attempt to apply the program comprehension literature to the study of writing. (The right side of Figure 5 summarizes these differences.)

<table>
<thead>
<tr>
<th><strong>Kent and Brooks: Similarities</strong></th>
<th><strong>Kent and Brooks: Differences</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Brooks and others studying program comprehension have developed and empirically examined an understanding of program comprehension and production that is--at some points--strikingly similar to Kent's view as described in <em>Paralogic</em></td>
<td>However, Kent and Brooks differ on many points. Brooks' theory of program comprehension--and most studies in this literature--assumes the following decidedly un-paralogic things:</td>
</tr>
</tbody>
</table>
Rhetoric. According to Brooks' understanding,

- Program comprehension is interactive, recursive, and involves more than simply "processing" lines of code.
- Program comprehension necessarily involves triangulating the intentions of the producer and the comprehender against a shared world (including the project domain and the machinery on which the program will run).
- Experienced programmers draw on "beacons," interpretive units that signal common programming moves. These interpretive units are not inherent in the structure of the program and are not necessarily algorithmic or procedural.

Program comprehension, although it is more than a linear, mechanical process, is still understood as a process that can be codified, studied, and taught.

Program comprehension is portrayed as a relatively orderly, systematic process.

Hypothesis generation does reach an end; a programmer can theoretically understand a program completely. Interpretation does not go all the way down.

Fig. 5.
A summary of parallels and differences in Kent and Brooks.

Works Cited

Please cite this article as Currents in Electronic Literacy Spring 2002 (6), <http://www.cwrl.utexas.edu/currents/spring02/spinuzzi.html>.