MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A. I. Memo No. 701 February, 1983

Computational Introspection

by John Batali

Abstract: Introspection is the process of thinking about one's own thoughts and feelings. In this paper, I discuss recent attempts to make computational systems that exhibit introspective behavior: [Smith, 1982], [Weyhrauch,1978], and [Doyle, 1980]. Each presents a system capable of manipulating representations of its own program and current context. I argue that introspective ability is crucial for intelligent systems -- without it an agent cannot represent certain problems that it must be able to solve. A theory of intelligent action would describe how and why certain actions intelligently achieve an agent's goals. The agent would both embody and represent this theory: it would be implemented as the program for the agent; and the importance of introspection suggests that the agent represent its theory of action to itself.

This report describes research done at the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505.

CONTENTS

0. Introduction

It's amazing just how hard these cartoon trees are!
-- Bugs Bunny

We can think about lots of things. We can also think about ourselves. We can *introspect*, and think about our own thoughts and feelings. This is certainly a striking ability, presumably it is important.

It is possible that the same sorts of reasoning we do about things in the world can be used to think about mental operations. If this is true, thinking may be viewed as a certain kind of action, and introspection (thinking about thinking) is just thinking about action. And so on. If this is indeed possible, one could hope that the power a reasoning system has with respect to its object domain could be applied to its own operations.

Naturally, such talk dredges up a host of philosophical issues. Some such issues are explored and certain positions defended in this paper. But an overarching (undefended) assumption in this work is that the mind may profitably be viewed as a computational entity. So an understanding of the mind proceeds by understanding the architecture and programming of that computational entity.

To the degree that "thoughts and feelings" are computational entities, computational introspection would require the ability of a process to access and manipulate its own program and its current context.

In this paper I discuss the problem of constructing computer programs that can be said to introspect. In particular, I examine three recent approaches to the problem:

Reflection and Semantics in a Procedural Language, by Brian Smith [Smith, 1982];

Prolegomena to a Theory of Formal Reasoning, by Richard Weyhrauch [Weyhrauch, 1978];

A Model for Deliberation, Action and Introspection, by Jon Doyle [Doyle, 1980].

From the discussion of these papers and related issues, four concerns are emphasized:

1. The importance of the ideas of *theory relativity* and *causal connection* in discussions of introspection in particular and mental operations in general.

A term is theory relative to the degree that it is defined by, and defines, other terms in some theory. The importance of the idea of theory relativity is that it allows us to use

mentalistic terminology -- so long as we are careful to make explicit the theory in which the term resides. It is also important from the point of view of an introspective program, because such a program must have some idea what to expect to see when it looks at itself. And it must be able to manipulate itself for introspection to have any effect. Both of these operations require that the program have some theory of itself so that it can view and manipulate the structures the theory posits.

Actions or events are causally connected if one can be said to cause the other. The importance of this idea for introspection lies in the fact that if introspection is to be useful, it must cause things to happen. A mind's theory of itself not only describes itself, it also allows the mind to control itself.

2. The architecture of introspection.

How can a program that "thinks about itself" ever actually work? Computer languages are usually implemented in other languages such that the new language works by being processed by the old one. An introspective processor, it would seem, implements itself to the degree that it acts in virtue of causally connected reasoning about how it should act. Happily, this problem has been solved -- at least for a procedural language. The solution (Smith's) is presented and I discuss ways in which the solution might be used to implement problem solving systems.

3. The representational requirements of introspection.

To think about oneself, one must represent oneself to oneself. So the design of introspective programs requires some representational mechanisms that are capable of representing a theory of the program to the program. The representational requirements of introspection thus include the ability to represent programs or agents or some such, as well as the relations among them. It may be that these requirements are more easily met then general knowledge representation. It is also possible that a representational system capable of supporting an introspective architecture can be extended by the introspective program itself into a more powerful formalism.

4. The importance of a theory of action

For a program to think about what it should do, it must understand what it is to *do* something. And it must have some theory of why some things are better to do than others. An introspective system needs some theory of action which it can use to evaluate and represent situations, potential actions, and their consequences. To the degree that reasoning is a particular kind of action, an introspective program might be best viewed as an implementation of a theory of action.

0.1 Outline

Chapter 0 is an introduction. It introduces the work, summarizes the main concerns, and presents an outline.

In chapter 1, I present working definitions of several important terms, including introspection itself. I also discuss how some things can represent others -- this is important in trying to understand how a program can reason *about* itself. Finally, I defend my use of mentalistic terms.

Chapter 2 is devoted to the utility and importance of introspection. Several important capabilities (including learning, and planning) are argued to be introspective in nature. It is also argued that introspective ability is crucial for intelligence.

Chapter 3 summarizes Smith's approach to introspection. Chapter 4 is a summary of Weyhrauch's, chapter 5, of Doyle's.

The three systems, and a few related approaches are compared chapter 6.

In chapter 7 I argue for the importance of theory of action and present the first sketches of such a theory.

Chapter 8 is a discussion of some of the paradoxes associated with introspective activity and self reference.

Chapter 9 discusses what the next introspective program ought to look like and examines some potential domains.

Chapter 10 is a conclusion and recapitulation of the main points.

0.2 Acknowledgements

This paper was originally written for my area exam. I thank the committee: Marvin Minsky, Carl Hewitt, and Tomas Lozano-Perez for letting me read the papers I wanted to, and for an entertaining defense of my paper.

Hal Abelson, Gerry Sussman, Ken Forbus, Ken Haase, Phil Agre, Dave Chapman, Dan Weld, Neil Mayle, and John Lamping provided encouragement and interesting discussions.

John Lamping and Penny Berman carefully read drafts of this paper and suggested many improvements.

I was supported by an IBM VLSI fellowship when this work was done. I thank them for that.

Finally, I thank Smith, Weyhrauch and Doyle for helping to open a very interesting territory indeed. I hope to follow.

1. Definitions

This paper is about "programs that reason about themselves." Before the discussion can get started, we must ask: What is a program? Is a program something that can reason? How can we tell what anything is reasoning about? And so on. In this chapter I define some terms that are used throughout the paper in a way that, at least, allows the above sorts of questions to be asked.

Virtually all terms defined and used herein are theory relative. So the definitions are to be taken as parts of a developing theory of computation and mental activity, and must be assessed in the terms of the theory. The ideas must have some intuitive appeal -- otherwise the worth of the theory can be questioned -- but they must not be expected to fully account for all aspects of our intuitions about the subject. We are in the process of developing theories, not explicating fully completed ones.

1.1 Programs and Processes

Let us begin by examining the notion of a *program*. I suggest that a program can be profitably viewed as a description of a certain object's behavior. The program will describe states the object can be in, and will describe what the object will do when in those states. We can then speak of an *abstract machine* or *processor* which is just the object described by a program. To describe a real object as *implementing* the abstract machine, is to say that the behavior of the object is described by the program. I speak of an *agent* as an implementation of an abstract machine that is viewed as interacting with some *environment*, defined roughly as everything else in the world but the agent.

Programs contain representations of certain states of affairs and the relationships between them. But programs do more then just describe, they also prescribe -- the expressions in a program have imperative force. This is to say that the program is *causally connected* to the machine whose behavior it describes. This is unlike, say, the description of the patterns of weather. In such a case the description is not causally connected to the actual behavior, however correct it may be. For a program we can say that the program *causes* the behavior it also describes. Not only that, it also causes the behavior in a theory relative way -- the process does *what the program says it will do*.

So one way that a theorist can attempt to explain a sequence of events is to postulate that some agent is performing them, and then produce a program for that agent. There are two ways such an explanation could go: either the program is taken to be an explanatory device, used for its practicality (for example as the basis for simulation); or the program could be taken literally, with the assumption that the agent really is the implementation of the machine described by the program.

The second approach is that of a *computational reduction*. (The term is from Smith.)

In a computational reduction, one must take into account the fact that a program presumes the existence of an implementing processor. The activity of the agent is viewed as the result of the processor acting as directed by some program.

The class of computational reduction studied in depth by Smith since it is (he claims) true for most existing programming languages, is that of *interpretive reduction*. In interpretive reduction, the process is viewed as the behavior of a single interior process interacting with a *structural field*. The interior process can itself be describable as a program which may then be reduced. (Of course, eventually one of the reductions is not computational -- this is the one from machine language to physics performed by integrated circuits or whatever.)

1.2 Aboutness

The way we use computers is to encode in a program some representation of our understanding of the world. We judge the correctness of the program to the degree that its behavior successfully corresponds to that understanding. So when I type `(+ 2 3)` to an interpreter, I am pleased when it responds with the result `5`, because I know that the sum of two and three is five. The *program* doesn't know, of course, it is just manipulating formal symbols according to the specification of whoever wrote the interpreter. But I take it to be manipulating representations, which are *about* things in the world (or abstract objects like numbers). Though the *operation* of computers may be viewed formally, the *use* of computers is as manipulators of meaningful entities -- of symbols. Computation is like linguistics: we are dealing with the use of meaningful tokens.

For the purpose of this paper, I use phrases like "reason about" or "represent" with regard to computation processes. As a first attempt at an account of aboutness, I simply suggest that one can determine what a process is reasoning about by examining what its variables are supposed to represent. If they are supposed to represent, say, frogs, the program may be viewed as reasoning about frogs. This becomes important for introspection, because we want to say that a program is reasoning about "itself." My suggestion would be that a program is reasoning about itself just in case the variables therein are bound to representations of the program, as well as its context. Smith is quite explicit about this, in fact a major portion of his project is to develop a theory of computation based on aboutness.

Certainly an important part of AI is the theory of representation. The way that some physical object can be taken to represent other objects or states of affairs has puzzled people for a long time. A consensus has developed that there are at least two important aspects to the content of a representation (the content is "what the representation means"). The first is that content involves a relation to the world. The sentence "Ronald Reagon fears olives." is true just in case the actual person fears the actual objects. The other aspect of the content of a representation is the *inferential role* it plays in the mental activity of the agent holding it. Such relations

include: what evidence is there for the sentence?; What follows from it?; What sorts of things could someone intend by telling it to me? and so on. Both of these aspects of the content must be elucidated in a complete theory of representation.

1.3 Mentalistic Terms

Throughout this paper I use mentalistic terms in the discussions of various programs and ideas. I suppose that a more conservative approach would be to hedge with double quotes or adjectives like "putative" or "pseudo". But I won't do that. AI is the study of how minds work. We must be allowed the benefit of the doubt: we can hardly *explain* minds if we aren't allowed to talk about them. Also, in many cases, the use of mentalistic terms is justified by the fact that there is virtually no other way to describe such things as "choosing," "belief" and so on. Thus such usage could be taken as metaphorical, but I don't mean it that way. I mean real choosing, real beliefs, real *minds*.

Another objection to mentalistic terminology is that such terms may be meaningless. It may be that concepts like "belief" and "desire" will have as much to do with future psychology as "phlogiston" and similar terms have to present science. I don't deny this possibility outright, but I will argue against it.

My argument is as follows: We first must establish the utility of introspection in a behaving agent. I later discuss details of what introspection must be, but for now let us recall that a vital property it must have is that of *causal connection*. This means that if a certain representation is accessed during introspection, and is modified, then the modification will affect the operation of the system. Furthermore, this causal connection is *theory relative*, that is, the representations so accessed will be described and defined by a theory of the operation of that system. So introspection requires causally connected theory relative access to the internal representations of an agent by the agent.

Suppose that strong arguments can be made for introspecting in certain situations. If this is the case, the argument would require that the agent have a theory of its operation and has the ability to make causally connected modifications to itself. An important question: must the theory be correct?

I think that the correctness of the theory is determined entirely by the causally connectedness requirement: a theory is correct just in case it allows causally connected access. Or to put it another way: the correct theories are just those which provide such access. But causal connection implies that the theories are really the *program* of the agent.

Furthermore, if we view the mind as modifying itself in the process of learning, and recall that such modification must be theory relative, it would seem that an agent's theory of itself would be mostly self-constructed. Many or most of the aspects of the

agent's program will have been constructed by the agent itself. So it will know what it put there.

All of this argues for the general correctness of an agent's introspective view of itself. In fact it argues that whatever the agent thinks it sees when it looks inside is essentially correct, so long as it is causally connected. Perhaps this argument holds for our concepts of belief, desires, and so on. I think that it does. "If we think they're there, and they seem to do what we think they do, *they're there*."

1.4 What is Introspection?

I have been dealing with a rather rough idea of what introspection is up to now. In this section I present a more explicit account of what sorts of things will count as introspective behavior. The general idea is that a computational system (an agent preferably) embodies a theory of reasoning (or acting, or whatever). This is what traditional AI systems are -- each system embodies a theory of reasoning in virtue of being the implementation of a program written to encode the theory.

Now consider the construction of an explicit representation of the theory of inference the program embodies, in a way that the program can handle. The program now represents and embodies a theory of problem solving. The final step is to give the program *access* in a causally connected way to the representation of its program. This access must have the following properties (suggested by Smith):

1. The program must have access, not only to its program, but to fully articulated descriptions of its state available for inspection and modification.

2. The program must be able to resume its operation with the modified state information, and the continued computation must be appropriately affected by the changes.

3 Introspection must be able to recurse. In general, the problem of modifying and inspecting the state of the program may need the power of reflection also.

4. Introspective must be *theory relative*. This is to say that the representations of the program and the context must be represented in terms of the theory the program is implementing.

The point is that introspection involves doing whatever the program does to the object domain to itself. It thus must be able to modify representations of itself in a way describable in the theory it implements. Presumably it can apply whatever ability it has in the object domain to successfully deal with the domain of its own state. Doyle argues that introspective reasoning is "similar in kind" to reasoning about the world.

I should mention why I use the term "introspection" in the title to this paper. Having

been reading words like "introspection," "deliberation," "reflection," "circular reasoning" "meta-reasoning" and so forth, I gave up and looked in the dictionary. I found that introspection was defined as "examination of one's own thought and feeling." [Webster's New Collegiate Dictionary]. I expand this definition slightly to include the notions of access and modifications to ones own thinking process.

And finally, I should mention that this paper has nothing to do whatsoever with "consciousness," in the sense of "private, subjective, feelings." It seems that a program could be introspective without being conscious -- consciousness could be a gift from God that comes with our immortal souls. In any case I agree with [Nagel, 1974] that subjective experiences might resist scientific explanation by their very nature. On the other hand I can't help suggesting, in the spirit of Nagel's paper, that consciousness might just be "what it is like" to be an intelligent, introspective agent.

2. Why Introspect?

Viewed as a potential feature under consideration for addition to a reasoning system, introspection must be supported by whatever power it can bring to the system. In this chapter I discuss several kinds of activities that minds should be capable of and argue that the capacities are:

1. Introspective in their nature: which is to say that the activity requires access to some representation of the program and/or its state.

2. Facilitated by an introspective architecture.

In the last section I give arguments for why introspection is crucial in an intelligent system.

2.1 The Importance of Representation

Smith presents the "knowledge representation hypothesis":

> "Any mechanically embodied intelligent process will be comprised of structural ingredients that (a) we as external observers take to represent a propositional account of the knowledge that the overall process exhibits, and (b) independent of such external semantical attribution, play a formal but causal and essential role in engendering the behavior that manifests that knowledge." (page 2)

The importance of this hypothesis is the assumption that a part of an intelligent program will be some identifiable "propositional account" of the knowledge embodied by the program. To some degree any program is a sort of propositional account of what it does and thus of the knowledge it embodies. However the hypothesis is stronger than this: it asserts that certain portions of the structure are taken to be representations, and that it is in those specific portions of the structure that the knowledge resides.

Smith doesn't really defend this hypothesis, though it certainly seems to be shared by many researchers, including virtually all workers interested in questions relating to introspective and "meta" reasoning. The reason is that an explicit use of certain structures and kinds of structures as representations allows reasoning processes to be defined explicitly over representations. *What* the representations represent is not part of the description of the reasoning algorithms. This is in the spirit of the "knowledge representation" research in AI (for example: KRL, FRL, KLONE). The payoff comes in the ease with with the knowledge representations structures are used to represent arbitrary domains, and potentially the reasoning process itself.

Not all research in AI is conducted within the knowledge representation framework,

however. Many systems exploring analogy, or learning. or various kinds of planning encode the knowledge implicitly in the program rather than explicitly in some representation language. The general reason for this is that the researchers are interested not in the representation of their theories, but whether or not they work. One of the problems adherents of the knowledge representation hypothesis must deal with is the question of representing such procedurally encoded knowledge in a way that both loses no power or speed. A theory of how such procedural knowledge works would help.

Such a theory would also help to elucidate some of the constraints knowledge representation systems must satisfy. A theory of procedural knowledge would be one of the things that a representational system in an introspective architecture would have to represent.

2.2 Some Introspective Activities

In this section I present a number of activities which are "by their nature" introspective. By this I mean that the activity involves some sort of access to, or modification of, representations of either the processor itself, or representations of its actions, skills or abilities. These are to be contrasted with non-introspective activities which need only represent the object domain.

The activities I present, while introspective under the above interpretation, nevertheless might not require the sort of full-blown introspection discussed later in this paper. At the end of this chapter, I will discuss the amount of introspective ability it might be best to have.

2.2.1 Learning

A most general definition of learning would describe the agent as modifying itself as a response to certain activities, in such a way that the changes would affect later situations related to the one in which the changes were suggested. We might want to include some requirement that the changes *improve* the behavior of the agent along some dimension, else self-inflicted damage might count as learning. The important thing is that learning involves change to oneself, specifically change to one's mental operations.

2.2.2 Planning

A "plan" is a representation of a sequence of activities the agent will perform. Elements in the plan represent the taking of certain actions by the agent. Such representations are introspective not only in their representation of the agent and its taking of actions, but also in the fact that the descriptions must be theory relative. The

descriptions must be in terms that the program can understand or else it can't perform the actions. Constructing a plan also probably involves much introspective inspection of the agent's abilities, skills, knowledge, goals, and so on.

2.2.3 Advice Taking

An intelligent agent must be able to improve its actions relative to its goals if someone simply *tells* it what it ought to do. For this to work, the agent must be able to understand the advice: it must be able to represent its own reasoning processes. The advice "eat more roughage" can only be understood in the context of understanding that one might be sometime deciding about what to eat. These issues were raised, of course, in [McCarthy, 1968].

2.2.4 Assumptions

An assumption is a choice taken by a program to behave as if a certain state of affairs obtains, in cases where it is impossible (or unduly expensive) to actually tell. An *assumption*, as opposed to a mistake, or a too readily jumped-to conclusion, is something that the agent must be able to deal with effectively if it turns out to be incorrect, or when asked to explain its actions. Note that assumptions are introspective about the representational facilities of the agent, rather than its acting abilities.

2.2.5 Serendipity

Agents must be able to realize that their goals are met even if it happens by accident. For that to work, the agent must be able to represent what those goals *are*. Representing the goal of a problem probably requires (due to the theory-relative language the goal will be expressed in) representation of the relations of goals, the concept of a solution, and so on.

2.2.6 Defaults

Like assumptions, defaults represent a certain commitment a problem-solver makes to a state of affairs with inadequate evidence. An interesting feature of defaults, noticed immediately by everyone who attempts to formalize them, is that they are not a first-order concept. Even expressing what a default is requires the ability to represent the inference scheme. I mention defaults separately from assumptions, because defaults are asserted unless specifically overruled, and are easily overruled, whereas assumptions are more carefully taken, and their removal requires considerable readjustment. I don't claim that the distinction is necessarily very important.

2.2.7 Debugging

Programs fail sometimes. The concept of failure is an introspective one, requiring an understanding of goals, and all that such understanding entails. Beyond that, the actual fixing of failures is quite definitely an introspective skill. One can't fix anything without knowing how and why it works. The elaborate truth maintenance systems and dependency directed backtracking mechanisms recently under serious study are all attempts to construct systems with these skills. In all cases, a crucial aspect of each system is a representation of the inferential activity of the problem solver.

2.2.8 Efficiency

One problem in AI is that as programs get smarter, they tend to get slower. Access to large quantities of knowledge tends to choke existing computers. And any computer will perform better at some tasks than others. A major portion of the skill of programming large systems involves constructing algorithms that take maximum advantage of the capabilities of the machines on which the programs will run. With respect to AI programs, much of the behavior of the program is under the control of the knowledge, rather than directly under the control of the programmer. We would like programs to be able to solve their problems efficiently, this would require that the program have access to representations of the computational capabilities of the machine it is running on.

2.3 Why Introspection is Crucial

Each of the above activities is introspective in the sense that I described at the start of the chapter. I concede that the activities do not need be implemented in a fully introspective interpreter. Certainly much work has been done in each domain without introspection. What the discussion ought to have shown is that since the activities are really introspective, a true understanding of them requires a deeper understanding of the nature of introspection.

What I wish to argue in this section is that introspection is *crucial* for intelligence, not only to understand or prove the correctness of implementations, but actually that intelligence requires introspective abilities. There are two parallel arguments here, one concerned with the *power* introspection provides -- it maintains that anything but an introspective system just wouldn't be able to solve the sorts of problems it must to be intelligent. The other argument is concerned with *understanding*. It maintains that one cannot be said to understand or use representations, unless one understands "how one understands."

2.3.1 The Control Problem

Every program, at every instant of its operation, must solve the problem "what do I do next?" (the control problem). For most programs, the problem is "solved" by the language, and ultimately the machine -- the program is wired to, say, increment the program counter and take the next instruction from that location. In fact, the study of "computer science" is the study of systems such that we can direct them precisely to go into certain states in certain conditions.

I claim that there is no general solution to the control problem. Any specification of what to do in a certain situation, it would seem, could be overruled by more specific knowledge. Therefore introspection is crucial for the ability it provides: if all steps available to the processor are represented, and the processor is able to specify any potential action as the one it will actually take, the program will be able to perform better than any system whose response to a certain situation is "wired in" but less than optimal for that situation. Introspection is not a magic solution to the control problem, but it provides a system with the ability to represent and entertain any particular solution to some particular instance of the control problem.

Consider an alternative position: This would have to claim that a general solution to the control problem *does* exist. So there would be some set of rules that constitute a context-free problem solving mechanism. And the program need not have access to them because they, simply, are always the correct ones to apply to problems. But the assumption of a general problem solving mechanism is, I think, rather rash. There is little except optimism to support it. One might, in fact be able to maintain that the arguments of philosophers like Kant, and the mathematical results of Gödel, lead to a feeling that such a general method simply does not exist. Furthermore, even in the presence of such a general mechanism, it would still be important for a system to be able to reason about the comparative efficiency of using the general mechanism versus a special-case approach. And finally: if full introspective ability is possible, and relatively easy, *why not* provide a system with it? Why limit the domain of its reasoning? The only conceivable reason for this would be that introspective ability actually hurt performance. I discuss the relevance of some self-reference paradoxes to introspection in a later chapter.

So: to be able to solve the control problem, the agent must have access to its internals in such a way as to be able to see what the relevant options are, and the potential solutions. Since even the decision to take this step is an action, the successful solution of it would seem to require introspective ability.

2.3.2 Understanding Representations

A complete theory of what it means to "understand something" is certainly beyond the scope of this paper, but several comments can make it clear why introspective ability is necessary for the understanding and use of representations. The general

idea is this: to understand a representation, one must understand *that* it is a representation. The meaningfulness of, say, a road sign depends on it being a sign and thus a representation of something else. The use of representations requires the ability to treat some things as being representative of other things.

But to understand representation, the agent must have an explicit representation of a theory of representation. This follows from the knowledge representation hypothesis which argues that to intelligently reason about a domain, one must make use of a representation of that domain. But the "theory of representation" that the agent must represent is the theory that the agent itself implements. And one way to represent the theory is to represent the program that implements it. Another way is to represent the theory declaratively and have the program run by implementing the theory. In any case, this implies introspective abilities.

My claim, essentially, it that to say that an agent understands the content of some representation, one must be able to say that the agent understands the content of some set of other representations. The concept of "broccoli" cannot be understood without the concepts of "green" and "vegetable". But even more than this: there is a set of concepts that must be understood to understand *anything* at all -- the concepts of a theory of representation.

3. Smith

Smith's project is to explore issues relating to introspection in the context of procedural programming languages. I discussed criteria for introspection above, as well as some of the issues relating to the question of aboutness in a computational framework. Smith chooses to examine LISP and to develop a reflective calculus as a dialect of LISP.

Smith speaks of designing a "reflective" calculus. For the most part, his use of the term is identical with my use of the term "introspective". I will maintain Smith's usage in this chapter for two reasons: one is that it is simpler; the other is that since the calculus he develops is purely procedural it can't really be said to think about anything: much less itself. The distinguishing feature of Smith's system is that the processor has access to descriptors of its own state and program. So I will treat "reflection" as a technical term, referring to the sort of limited introspective abilities that Smith's interpreter does have.

Before reflection can be discussed, argues Smith, LISP must be semantically characterized in such a way that we know what we are doing. He argues that standard LISP is "confused and confusing" in its notions of evaluation and the way it handles meta-structural expressions. So he characterizes the semantics and behavior of three dialects of LISP with the goal of providing a reflective formalism. 1-LISP is supposed to be a "distillation of current practice," it is essentially LISP 1.5. 2-LISP is a "semantically rationalized" statically scoped dialect. 3-LISP is virtually identical to 2-LISP, but includes the capacity for reflection. It turns out that the power gained by reflective abilities actually makes 3-LISP slightly simpler than 2-LISP: several procedures which must be primitive in 2-LISP (e.g. set, catch, lambda) can be defined as reflective procedures in 3-LISP.

The general strategy he takes is one that has been urged by other writers (especially: [Hayes, 1977] and [Woods, 1975]) which is, roughly put, to get the semantics right before the code is written. So we must understand what forms in the language "mean" and "do", before writing the interpreter. Then when an implementation is produced, it is possible to say that it is correct.

Smith admits early on that the languages he is describing are not representation languages: the expressions therein are not *representations* of anything. One can't actually *say* anything with them. A real problem-solver must be able to represent states of affairs. But the crucial problem of actually creating an introspective dialect is difficult enough for a procedural language, for a representational language, it may be even more difficult.

3.1 Designational Semantics

Consider the case of EVAL in LISP. In explaining to a new student of the language, we might say that (+ 2 3) "will evaluate to (will return)" 5. Also, 'A returns A. And if A is bound to 3, A will evaluate to 3. And so on. Rather than explicitly mentioning the evaluation process, Smith would prefer that we capture our pre-computational intuitions about expressions, and design a language that conforms to those intuitions. He would prefer that we develop a theory of what expressions mean before the language is implemented. Otherwise the student's question "why does EVAL work that way?" can't be answered.

We begin by distinguishing two ways in which expressions may be described. The *declarative import* of an expression is what it stands for, what it signifies, what it *names*. The *procedural consequence* of an expression refers to the results and effects the expression produces when it is processed.

Here is where Smith starts: Consider the numeral 5. It is understood from the nature of numerals that a numeral is not a number itself, a numeral stands for, or *designates* a number. That is to say: a numeral is a linguistic object, a number is whatever numbers are, and there is some understood relationship between the designator and designatee. Smith's approach to the declarative import of S-expressions is to try to extend the notion of designation to virtually all kinds of expressions. He contends that an interpreter should be understood by treating all forms in the language as designators. To understand the processing of forms one must keep the designation relationship in mind.

Now consider the LISP expression (+ 2 3). We are pleased when an interpreter produces the numeral 5 as a result of this expression. Why does this seem like the right thing? Here is the story Smith would like to tell: The expression + designates the addition function, the numeral 2 designates the number two and the numeral 3 designates the number three. So the expression designates the result of adding two and three which is five. We designate five with the numeral 5 and that is what is returned.

The understanding of functions is crucial in Smith's semantics. He distinguishes between the concept of *reduction*, which is a meta-structural notion, and that of function *application* which is a semantic notion. Functions are taken to be abstract things, they can be applied to abstract things (like numbers) and the result is another abstract thing. Reduction, on the other hand, is the process of taking some syntactic form and producing another syntactic form, such that the two forms designate the same thing, given that the first form is taken to designate the application of a function to its arguments. So in the above example, the form (+ 2 3) is reduced to the form 5. We understand what is going on in terms of the addition function being applied to the arguments two and three.

Smith's approach to the semantics of LISP is to determine, for all types of

expressions, how their designations depend on their subexpressions. An expression is *extensional* if the designation of the compound expression depends only on the designations of the constituent expressions, it is *intensional* otherwise. For example, the expression (+ A B) is extensional because it depends on what +, A and B designate. (QUOTE A) on the other hand, is intensional, because it depends on not what A designates but the symbol A itself.

If we suppose that we have an account of what expressions designate, we can then turn to the question of the the language processor does with them. I will discuss Smith's treatment of side-effects later, for now, let us consider what the processor returns after processing an expression. Smith refers to the function from expressions to their designations as Φ and the function from an expression to what it returned by the processor as Ψ. His project is to show how Ψ depends on Φ, this relationship is called Ω. Ω is a *semantic* relationship between two expressions that depends on their designations.

Consider mathematical logic. Two relations between expressions are considered. One is the derivability relation: $\vdash$. This relation is defined in terms of the inference rules of the language. The other important relationship is that of entailment $\models$. Entailment is defined in terms of satisfaction by a model: A $\models$ B just in cause B is satisfied in all models in which A is satisfied. Note that $\vdash$ and $\models$ are not the same relation -- proofs of soundness are important because they show that $\vdash$ and $\models$ are related in interesting ways, for example: $\vdash \leftrightarrow \models$. In Smith's terms, $\vdash$ is logic's Ψ. For logic, Φ is the interpretation mapping that takes terms into elements of the model. Logic's Ω is $\models$, this is the semantic relation between an expression and another expression.

In the lambda-calculus, Ψ is given by the rules of conversion, while Φ is a mapping between lambda expressions and, for example, the elements of a lattice in the Scott-Strachy approach. The relation between the two mappings is that the rules of conversion *preserve designation*. Whatever the expression designated before the conversion, it designates the same thing afterwards.

3.2 1-LISP

With these considerations in mind, let us examine standard LISP. Before attempting to characterize the Φ and Ψ for this language, we note one important difference LISP has between logic and lambda-calculus and, indeed, many programming languages: LISP has *meta-structural* facilities. This means that certain LISP forms can evaluate to other forms which may themselves be evaluated. To make this ability actually useful, LISPs have the primitive processor functions, EVAL and APPLY available to the user. It is thus possible, though standard practice argues great care in these situations, to perform computations by having LISP programs construct expressions, which are then given as arguments to EVAL.

Let us take the designational semantics view and assume that LISP forms designate things. We have argued that it is natural to take the numeral 5 as designating the number five and the symbol + as designating the addition function. We also take a form whose CAR is a symbol to designate the result of applying the function designated by the car to the sequence of arguments determined from the CDR of the form.

Further, let us specify that (QUOTE <x>), where <x> is some LISP form, *designates that form*. That is, QUOTE is meta-structural not only in what it returns (in standard LISPs), but also in (we stipulate) what it designates.

What about variables? Suppose that the state of the computation is such that evaluating the symbol A returns 5. There are two ways we could describe what is going on, in terms of what A is bound to and what it designates. On one account, we can say that A is bound to the *number* five. We could also say that A designates that number. But when A is evaluated, it returns the numeral 5 (because it can't return a number). So this account would have us say that A designates what it is bound to, but returns not the thing that it is bound to, but instead something that *also* designates what the variable is bound to. A problem with this approach will come up when we attempt to reify environments so that reflective code can manipulate them as LISP objects. Since this account has A bound to a number, the environment can't represent that binding literally because LISP objects can't contain numbers. The best that can be done is to contain something that *represents* numbers.

This analysis suggests that binding be thought of as *co-designational*, which is to say that a variable be bound to an expression which designates the same thing as the variable. Thus A would be bound to the numeral 5, but A would designate the number five because it shares the designation of its binding.

Rather than explore the various consequences of these decisions, I will now present Smith's major objection to the behavior of EVAL. The analytical apparatus I have constructed is sufficient to show that there is at least a problem. This demonstration will be used to motivate the semantic rationalization in the design of 2- and 3-LISP.

Here are two examples of the behavior of the 1-LISP evaluator:

```
(+ 2 3) ==> 5
(QUOTE A) ==> A
```

In the first case, as discussed above, the evaluator returns an expression which designates the same thing as the input expression. (+ 2 3) designates the result of applying the addition function to the numbers two and three, which is the number five, which the numeral 5 designates.

In the second case, we have (from above) that the input expression designates the symbol A. But the result designates whatever the variable A designates, however the designation of variables is imagined to go.

We seem to have two different behaviors with respect to the designations of forms. In one case, EVAL produces a result that designates the same thing as its input, in the other, EVAL produces what the input designates. In Smith's terms, sometimes it *de-references* the input expression. In fact, it de-references in just those cases when it can -- when the designation of the form is another LISP expression. In all other cases, when the designation is some abstract object like a number or a truth value, EVAL returns a co-designating form.

Note that the inconsistency in LISP comes from its meta-structural abilities. If LISP had no functions like QUOTE, and if EVAL weren't available to the user, all expressions would be taken into co-designating forms and EVAL would, like the lambda-calculus and logic, preserve designation.

3.3 Semantic Apparatus

In this section, I will present the tools Smith uses to develop the semantics of S-expressions in preparation for the development of a semantically rationalized dialect of LISP. The project, as I mentioned above, is to develop those semantics independently of the behavior of the processor, and then to demonstrate that a processor behaves correctly in terms of those semantics. He does not argue that this cannot be done for 1-LISP, in fact he develops a semantics of 1-LISP and describes how one could prove the "evaluation theorem" which describes the relationship between EVAL and de-referencing.

The general idea of the semantic rationalization is that the language should be "semantically flat." That is: the processor should not, by default, de-reference expressions. Rather than continue the use of the term EVAL, Smith prefers the concept of *normalisation* for the Ψ of his language. In his dialects, the result of normalising an expression designates the same thing as the expression, in all cases.

Normalisation has another property: its results are in *normal form*. Smith defines a normal-form expression to be that which has the following properties:

1. Environment independent.
2. Side-Effect free.
3. Idempotent

Thus, by 1, a normal form expression designates what it designates independent of the environment it is normalised in. By 2, the result of normalising an expression in normal form has no side effects, and by 3, normal form expressions normalise to themselves.

Another property of Smith's dialects is that they are *category aligned* which means that the syntactic type of a normal form designator is determined from the semantic type of the referent. Thus, by looking at the thing designated, one can tell what kind

of S-expression will designate it. Thus numerals designate numbers and so on. This is important from the point of view of the APPLY/REDUCE distinction developed above: a rationalized processor must be able to return something that designates the result of the application of a function.

1-LISP is not category aligned. For example a pair (cons) can designate either a pair of values, or the application of a function to arguments. To distinguish these cases Smith uses pairs only to designate function applications, and introduces a new structure, a *rail* to designate sequences of things. Rails are notated by their elements between square brackets: [EEP FEEP ASKEEP].

To handle side-effects, the relevant context in which the processing is done must be described. Smith introduces the context as having three parts: the *environment*, the *continuation* and the *structural field*. The environment is a function from symbols to co-designating bindings. The continuation encodes the state of the processor, the structural field is the set of S-expressions accessible from the computation.

Now we can be more precise about Φ and Ψ. Φ is a function from contexts (fields, environments and continuations) and an expression onto the semantic domain (which consists of the structural field plus numbers and truth values and objects in the user's world.). Ψ is a function which also takes a context and an expression, but produces another expression, that is: another element of the structural field.

To handle side effects, we speak of the *full procedural consequence* function Γ which takes an expression and a context into a result expression, and a new, possibly modified context. An expression is side effect free just in case its full procedural consequence does not affect the environment or structural field.

All these notions are collected into the *full computational significance* function Σ, which takes an expression and a context to its result, its designation, and a new context.

3.4 Rationalizing Lisp

Armed with this theoretical apparatus, Smith develops the language 2-LISP, a "semantically rationalized version of LISP." The important features:

2-LISP is a higher-order language, like SCHEME, thus functions are first-class objects. Closures are defined as normal-form function designators. Closures must contain the environment in force when they are created. Smith argues that this static-scoping best captures the intensions of procedures.

The main processor function NORMALISE takes expressions onto normal-form co-designators of their referents. Thus meta-structural expressions aren't normally de-referenced. A special syntactic category *handle* is defined. A handle is an

expression whose designation is another expression, They are notated with a quote. Some examples:

```
(+ 2 3) ==> 5              ;; both designate numbers
(+ '2 3) ==> <error>       ;; '2 designates a numeral
(CAR '(FACT 5)) ==> 'FACT ;; the handle designates a pair
[2 (+ 1 2)] ==> [2 3]      ;; a rail is in normal form if its elements are
```

Note the behavior of the form `(CAR '(FACT 5))`. Whereas normal LISP would return `FACT`, 2-LISP returns `'FACT`. The simplest way to understand that this is the appropriate behavior is to recall the mandate that normalisation be idempotent. If the selector function returned `FACT`, another normalisation would return whatever `FACT` was bound to (in this case, a designator of the `FACT` procedure). Also notice that the handle `'(FACT 5)` represents a pair of two S-expressions whose first element is the symbol `FACT`. So the `CAR` of that pair must be a symbol. A symbol is designated by a handle, thus `'FACT` is the result. In fact, normalisation *never* will return a symbol -- symbols are not normal form designators in 2-LISP.

Two functions are defined for explicit shifts of reference levels. The function `NAME` returns a designator of its argument, the function `REFERENT` returns what its argument designates. Some examples:

```
(NAME 5) ==> '5
(NAME (+ 2 3)) ==> '5
(NAME '(+ 2 3)) ==> ''(+ 2 3))
(REFERENT 5) ==> <error>
(REFERENT '5) ==> 5
```

2-LISP supports three kinds of functions. EXPRs are ordinary extensional functions, the arguments are normalised in the current environment before the function is reduced with them. IMPRs don't normalise their arguments, the function is given access to the un-normalised argument expression to do with what it will. It turns out that IMPRs can't do very much interesting with the expression, because they have no access to the environment. (2-LISP is statically scoped, and environments are not LISP objects.) MACROs are essentially IMPRs whose resultant expression is then normalised instead of the expression given to `NORMALISE` as input. MACROs thus solve some of the problems with IMPRs because the second normalisation takes place in the current environment.

3.5 Procedural Reflection

Though 2-LISP seems to be a useful language, the problems with IMPRs and difficulties defining such primitives as `IF` and `SET` suggest the need for more access to the context than 2-LISP provides. Specifically, the proper handling of closures requires that the environment be a LISP object. And control-structure primitives like `IF` require some ability to access and modify the continuation. The argument is not that such access is required to make the language useful. Rather, there is a set of fundamental ideas required to *understand* what the implementation of a language is doing, these ideas concern the operation of certain constructs. Smith argues that

these constructs (the environment and continuation) be reified and made available to the user.

Consider a debugging program. Such a program would presumably be entered when some specified condition is met, and would allow the user to examine and modify the environment, as well as the continuation. For example the user might want to look at or change the value of a variable (environment modification); or perhaps abort the current computation, or substitute some other forms for the one being processed (modifying the continuation structure). Such programs exist in the support environment for virtually every programming language, certainly users must understand the relevant notions in order to use them. But few programming languages give the user access *in the language* to those constructs.

Of course, the whole project all along was to consider a reflective dialect. The fact that reflection will be useful for programming, as well as for problem-solving, is to some degree just a happy result. Of course had no uses for reflection been found in a procedural formalism, much of what Smith argues for in his work might seem less relevant, since after all, he only examines such procedural formalisms.

Reflective programs have access to the environment and continuation in force when they are called. The elements of the context can be examined and modified, since they are valid 3-LISP objects. Also, they can be used to de-reflect and continue the computation in progress before the reflection. Smith considers a number of ways that such access can be possible, the most obvious being primitive functions that return designators of the current continuation and environment.

The solution ultimately used by Smith is to create a certain kind of procedure which is called on arguments including designators of the current continuation and environment. These *reflective* procedures are just 3-LISP programs, but since they manipulate designators of a lower level of processing, they can be viewed as running at the same level as the processor.

3.5.1 The 3-LISP Processor

The meta-circular 3-LISP processor is shown in figure 1. For the most part, it is a straightforward meta-circular continuation passing processor. The use of continuation passing makes it easy to reify the state of the processor -- it is just the current continuation.

The procedure `NORMALISE` takes an expression, an environment and a continuation. If the expression is in normal form or is atomic, it simply passes a result to the continuation. RAILs and PAIRs are normalised specially.

The behavior of `REDUCE` has been discussed above. It calls `NORMALISE` to normalise the procedure argument, with a continuation (called `C0`) that dispatches according to the

```
(DEFINE NORMALISE
  (LAMBDA SIMPLE [EXP ENV CONT]
     (COND [(NORMAL EXP) (CONT EXP)]
           [(ATOM EXP) (CONT (BINDING EXP ENV))]
           [(RAIL EXP) (NORMALISE-RAIL EXP ENV CONT)]
           [(PAIR EXP) (REDUCE (CAR EXP) (CDR EXP) ENV CONT)])))

(DEFINE REDUCE
  (LAMBDA SIMPLE [PROC ARGS ENV CONT]
     (NORMALISE PROC ENV
        (LAMBDA SIMPLE [PROC*]                          ;; C0
           (SELECTQ (PROCEDURE-TYPE PROC*)
               [REFLECT ((EXTRACT-SIMPLE PROC*) ARGS ENV CONT)]
               [SIMPLE (NORMALISE ARGS ENV (MAKE-C1 PROC* CONT))]))))))

(DEFINE MAKE-C1
  (LAMBDA SIMPLE [PROC* CONT]
     (LAMBDA SIMPLE [ARGS*]                             ;; C1
        (COND [(= PROC* ↑REFERENT)
               (NORMALISE !(1ST ARGS) !(2ND ARGS) CONT)]
              [(PRIMITIVE PROC*) (CONT ↑(!PROC* . !ARGS*))]
              [$T (NORMALISE (BODY PROC*)
                             (BIND (PATTERN PROC*) ARGS* (ENV PROC*))
                             CONT)])))))

(DEFINE NORMALISE-RAIL
  (LAMBDA SIMPLE [RAIL ENV CONT]
     (IF (EMPTY RAIL)
         (CONT '[])
         (NORMALISE (1ST RAIL) ENV
            (LAMBDA SIMPLE [ELEMENT*]                   ;; C2
               (NORMALISE-RAIL (REST RAIL) ENV
                  (LAMBDA SIMPLE [REST*]                ;; C3
                     (CONT (PREP ELEMENT* REST*)))))))))
```

Figure 1: The 3-LISP meta-circular processor. ! is a reader macro such that !EEP expands into (REFERENT EEP). ↑ is a reader macro that expands ↑EEP expands into (NAME EEP). The named continuations C0 through C3 are discussed in the text.

type of the procedure.

Reflective procedures will be discussed below. Simple ones evaluate their arguments, this is done by calling NORMALISE on the rail of arguments with the continuation C1.

The C1 continuation accepts a normalised rail and dispatches according to whether the procedure is primitive or not. The procedure REFERENT is handled specially. Compound procedures have their bodies normalised in the environment created by binding the arguments to the formal arguments of the procedure.

NORMALISE-RAIL is just a continuation-passing way to evaluate a sequence of arguments and collect the results together. The continuation C2 takes the evaluated first argument and calls NORMALISE-RAIL tail-recursively with continuation C3 which hooks the rail together.

3.5.2 Levels of Processing and Designation

For the moment, pretend that the above processor works: pretend that the code, written in 3-LISP, actually implements 3-LISP as well. Consider the evaluation of a form such as

```
(NORMALISE EEP)
```

Using the semantic apparatus developed previously, let us analyze the designational story to be told here. First, the argument EEP is normalised (since NORMALISE is a simple procedure). The binding of EEP designates something, let us say D, it also normalises to something, N, which also designates D. The form (NORMALISE EEP) on the other hand, designates the result of normalising the binding of EEP, that is to say, it designates N, and thus must return a form which designates N, that is: 'N. The point of this is the following: Think of levels of designation, such that if A designates B, then A is one level of designation above B. By this analysis, we can say that (NORMALISE EEP) is one level of designation above the binding of EEP. Also we can see that normalising (NORMALISE EEP) will be the same as normalising the binding of EEP at the level below.

In 3-LISP, one can also speak of levels of *processing* such that a simple procedure runs one level below that at which NORMALISE is running. One of the results of making the dialect semantically flat is that one can roughly align levels of designation with levels of processing.

Note the line in the procedure REDUCE of the form

```
[REFLECT ((EXTRACT-SIMPLE PROC*) ARGS ENV CONT)]
```

This is a potential dispatch depending on the type of the procedure argument to continuation C0. We see that if a procedure is of type REFLECT, a procedure is extracted that *runs at the same level* as the processor and has access to the variables ENV and CONT, which were variables of the processor. Whatever level REDUCE was

running at, the extracted procedure runs at the same level -- which is one level above that of simple procedures. Thus is reflection engendered.

3.5.3 Using Reflection

We haven't shown yet that this can actually work. It has the strange, paradoxical flavor of circularity in it. Let us examine a few simple reflective programs and see that, at least, reflective abilities seem useful.

```
(DEFINE SILLY (LAMBDA REFLECT [[ARG] ENV CONT]
                      (NORMALISE ARG ENV CONT)))
```

This program just duplicates what would happen if the argument were normalised with no reflection at all. In fact, we could take it as a requirement of the implementation that (SILLY <exp>) have the same effects and return the same result as <exp> itself.

```
(DEFINE SET (LAMBDA REFLECT [[VAR BINDING] ENV CONT]
              (NORMALISE BINDING ENV
                (LAMBDA SIMPLE [RESULT]
                   (CONT (REBIND VAR RESULT ENV))))))
```

This is 3-LISP's version of SETQ. It reflects, and then normalises the binding in the current environment with a new continuation that calls REBIND (which side-effects the environment) and gives the result to the original continuation. Note that here reflection was used to get an un-normalised argument, as well as to gain access to the environment. Since the arguments are not normalised in reflective procedures, one can do without IMPRs and MACROs as primitive functions in 3-LISP, defining them as reflective procedure if it seems useful.

In the above examples, very little of interest was done at the reflective level. To get more power, we must look more closely at what is defined in the 3-LISP processor. In particular, since reflective code is also running in 3-LISP, it would seem well-defined to be able to reflect out of reflective code also. This is in fact the case. Smith urges us to view the 3-LISP system as an infinite tower of meta-circular processors, each processing forms in the level below it. We assume that at some point in time, God (or some functional equivalent) starts the top of the tower running the driver loop, READ-NORMALISE-PRINT:

```
(DEFINE READ-NORMALISE-PRINT
   (LAMBDA SIMPLE [ENV]
      (BLOCK (PROMPT (LEVEL))
             (LET [[NORMAL-FORM (NORMALISE (READ) ENV ID)]]
                (BLOCK (PROMPT (LEVEL))
                       (PRINT NORMAL-FORM))
                       (READ-NORMALISE-PRINT ENV)))))
```

Note the use of the identity continuation in the call to NORMALISE. This causes the call to NORMALISE to actually return a value, which will then be printed. So this call to NORMALISE is *not* tail-recursive, and thus requires that the level above be running and maintaining its own continuation. (This illustrates why it is crucial that God be involved. Only the "uncalled caller" can run READ-NORMALISE-PRINT without another

level above.)

We then imagine that each level of the tower down to ours is started out running that loop until we get, at our level, the prompt. From the point of view of the bottom level, the interpreter at the next level up is running READ-NORMALISE-PRINT. From the point of view of that one, the next one up is running it and so forth. Let's not worry about implementing this at this point, we will simply argue that it is at least defined.

With this understanding in place, we can describe some other interesting code that can be written reflectively. For example consider:

```
(DEFINE QUIT (LAMBDA REFLECT ARGS '$T))
```

This procedure throws away the continuation in force (as well as the rest of its arguments) and simply returns a designator of a boolean. From the structure of the READ-NORMALISE-PRINT code it can be seen what will happen as a result of this: the call to NORMALISE at the next reflective level up will return. If that call happened to be the one in READ-NORMALISE-PRINT (as it will usually be), $T will be printed out, and READ-NORMALISE-PRINT will accept another form to be processed at the level below.

We can now define an important control structure:

```
(DEFINE UNWIND-PROTECT
      (LAMBDA REFLECT [[PROTECTED-FORM CLEANUP-FORM] ENV CONT]
         (LET [[NORMAL (NORMALISE PROTECTED-FORM ENV ID)]]
           (BLOCK (NORMALISE CLEANUP-FORM ENV ID)
                  (CONT NORMAL)))))
```

The idea is to normalise the protected form with the identity closure in the same way as in READ-NORMALISE-PRINT. So if that procedure ever returns, either normally, or as a result of someone throwing away the closure, as in QUIT, the CLEANUP-FORM will be processed.

Notice how most of the reflective procedures examined eventually call either NORMALISE or their continuation argument. From the analysis earlier in this section, normalising a form whose procedure is NORMALISE is equivalent to normalising the argument at the level below. I will show below that the same is true for most of the continuations passed to reflective code. Thus the functions presented here all reflect down with their results at the end of their processing -- or at least can be thought of as doing so from our point of view at the bottom of the tower of interpreters.

3.6 Implementing Reflection

The above description of the 3-LISP processor as an infinite tower of READ-NORMALISE-PRINT loops might be very disconcerting to anyone interested in actually implementing 3-LISP. Certainly only a finite machine can be built. Recall, however, the requirements that an implementation must meet: a system can be said to implement a program just in case the system's behavior can be described by the

program. So long as no behavior is infinite (whatever that means), the program ought to be implementable on a finite machine. More practically: since the programs all start from the bottom, and all *programs* are finite, reflection will only go some finite amount up the tower. So we really only need to implement enough of the tower to run the program. (Of course, there could be errors which would cause a program to reflect indefinitely upwards -- an infinite reflection loop -- but no *useful* program would ever have to reflect infinitely before producing a result.)

As a matter of fact, we really only need to actually worry about two levels of the tower: the one that the code is currently running in, and the level above that, the level of the processor running the current code. The rest of the tower below the level of the code is all designated by variables in the code: it is not running directly. Of the portion of the tower above the current level there are two situations: a finite number of levels directly above the current one have been visited -- programs have reflected to those levels and then dropped back down. Above the levels that have been visited, each level in the rest of the infinite tower is doing the same thing -- running the standard `READ-NORMALISE-PRINT` loop. We need to actually simulate only the level of the tower currently running. The state of levels above the current one that have been visited must be saved in case they are visited again. Levels not visited yet need not be represented, since they are all doing the same thing. They can be constructed when and if some code reflects up to one of them.

Smith presents an implementation of 3-LISP that works as described above. I will explain it in some detail here because I feel that the specific techniques are important for the actual implementation of any introspective program.

The main idea of the implementation is to write, in MacLISP, procedures that perform the same operations as the 3-LISP meta-circular procedures. For non-reflective code, this is a straightforward implementation of a statically-scoped LISP. Thus, one writes a `FAKE-NORMALISE`, which dispatches according to the type of the argument, calling `FAKE-REDUCE` if the argument is the MacLISP representation of a 3-LISP pair, for example. Rather than explicitly construct and run the continuations `C0` through `C3`, one needs only to represent their environments and do a jump to their bodies, MacLISP programs such as `FAKE-C0`, etc.

But if a reflective function is encountered, one must be able to provide it with representations of the environment and continuation in force. They must be constructed from whatever representations have been being used.

Smith's solution is as follows: Each of the processor procedures explicitly creates the continuations. But continuation `C0` (which dispatches according to the type of procedures) looks at the procedure it is about to apply. If the procedure is one of the processor continuations, it ignores the created continuation, and simply jumps to the MacLISP code that directly does whatever processing the continuation would have done. If `FAKE-C0` doesn't recognize the closure it is about to apply, it must reflect up one level and process the procedure explicitly.

When making an upward reflection, the implementation must construct new representations of new level. Consider the first jump to a specific level. The environment of the new level is easy: since nothing has ever visited it before, no variables could have been bound at this level, thus the environment at the level is the global environment.

What is the continuation in force at a new level? Note that all reflection occurs inside the body of a REDUCE and inherits unchanged the continuation argument for the reduce. REDUCE, in turn, is only called by NORMALISE, and inherits the continuation of NORMALISE, also unchanged. And each level is assumed to be running READ-NORMALISE-PRINT. The continuation in force for the call to NORMALISE in READ-NORMALISE-PRINT is the identity continuation. So the continuation in force at each new reflective level is the identity continuation. Thus reflecting to a new level means that the implementation must simply use the global environment and a new copy of the identity continuation. If the level has been visited before, the implementation will have stored enough information to reconstruct the state in effect when the processor dropped levels. This is simply the environment and continuation that were in force when the drop occurred.

Dropping down a level is slightly more difficult, because the processor must recognize that it is about to apply certain procedures that it can run directly at the level below. Consider the analysis of forms such as (NORMALISE EEP). We concluded that this form designates whatever the binding of EEP normalises to. So normalising (NORMALISE EEP) at one level of the processor is the same as normalising the binding of EEP at the level below. Similarly for REDUCE, and the continuations C0 through C3. In each case, one can get the same result by dropping a level.

So the C1 continuation (which actually applies normalised procedures to normalised rails) checks to see if the closure it is about to apply is one of the processor closures. If so, it drops a level, saving the current environment and continuation, and runs the fake version of the procedure.

3.7 Criticisms

Smith has succeeded in his self-defined goal of providing a reflective dialect of LISP. 3-LISP satisfies the requirements laid out at the start. Furthermore the implementation of the reflective 3-LISP processor is a major achievement: it demonstrates that such a thing can be done, whether or not the language implemented is very much like 3-LISP or not.

A deficiency with 3-LISP, which Smith readily admits, is this: 3-LISP is not a representational language. Its forms have no *assertional force*. One can't say anything to or with 3-LISP, one can just cause forms to be normalised. That is: the expression (= 3 (+ 1 2)) normalises to $T, but it in no way says anything about the relationship between three and the sum of two and one. In fact something of the

opposite sort occurs: the processor returns $T, and we are satisfied because we *know* that the relationship holds. If the processor behaved otherwise, it would be grounds to call it broken.

Since 3-LISP is not a representational language, it can certainly not represent a theory of itself. Thus one can say that it has computational access to something that designates itself but there is no way the program can be said to know that. It seems that a vital criteria for reflection be that the program be able to represent that the structures it is manipulating represent itself somehow. The systems of Doyle and Weyhrauch, discussed later, have this property.

In particular, if 3-LISP had a representation of itself, a representation of what it is doing, it ought to be able to understand the way that it was implemented: the way in which the implementation satisfies its description (program). This may be a generally important type of reasoning for programs: converting declarative descriptions of what they ought to do into sequences of operations that can somehow be primitively performed.

Smith also discusses a shortcoming of not only his system but the current state of philosophy: the lack of a theory of intensionality. Logic deals with extensional expressions, intensional ones have been treated by various modal logics, but no adequate theory has been developed. In particular we need a theory of *procedural intension* which would describe "what programs do" in the terms the programmer intends. What, for example, is in common among all programs that compute square roots by Newton's method?

Understanding the intension of a procedure is crucial for a system that is going to understand programs. It is important specifically in 3-LISP because the reflective interpreter must determine when it can reflect down. Smith's implementation checks if a procedure about to be reduced is structurally identical to one of the primitive processor functions. But what we really require, surely, is that the procedure to be applied *intensionally* equivalent to one of the primitive processor functions -- that it "do the same thing" as one of the primitive processor functions.

A theory of procedural intension is a theory of how to act: of how procedures behave. It would describe a procedure as engendering certain behavior in the processor, and would relate intensions in terms of those related behaviors. Such a theory would begin to look more like a theory of action than a theory of computation. The idea of a theory of action will be developed in a later chapter.

4. Weyhrauch

Weyhrauch describes FOL: a "theorem prover's assistant" program that carries on a conversation with the user about first order mathematical theories. A user can make declarations about the symbols, functions, predicates and constants in a first-order theory. The user can introduce axioms and have the system check proofs of theorems, or prove simple theorems itself. FOL is described most completely in [Weyhrauch & Thomas, 1974].

The crucial idea for those of interested in introspection is the realization that the metatheory of many theories and programs (including, for example, Smith's system) is often a first order theory. Such is the case for FOL itself. Since FOL is a program to manipulate representations of first order theories, it can therefore manipulate representations of *its own* metatheory. This turns out to be a very natural, simple, and powerful step to take.

4.1 Simulation Structures

FOL has a "syntactic simplifier" which attempts to use the axioms and theorems of a language to deduce new theorems or to check ones entered by the user. This portion of the system is similar to many theorem proving programs.

An important part of the FOL system is the introduction of "simulation structures" which are intended by Weyhrauch to represent a model of the theory under investigation. The idea of a simulation structure, as its name implies, it to simulate operations in the model of the theory. For example we could represent the axioms of Peano arithmetic in FOL. But instead of doing all sums by deriving the results as theorems, we could declare that the simulation structure for Peano arithmetic is the LISP function + over the LISP representations of the integers. In some cases, when the simulation is defined, much can be learned about the domain by studying its model. Naturally, for reasons described also by Smith, we can't really have the model around to examine (unless the system is representing something else in the machine.) Thus we *simulate* it.

Simulation structures are declared by making "attachments" to the constant, predicate, and function symbols of the first order language under consideration. A "semantic simplifier" can then be used, along with the syntactic simplifier, to prove or check theorems by running procedures in the simulation structure. So to determine the value of a particular sum (in a theory of arithmetic), the semantic simplifier looks for attachments to the arguments. If, as we assume, it finds them, it calls the LISP function + on them. Simulation structure functions and predicates can either return a result or a "don't-know" response. If the answer is returned, a representation of it is constructed in the theory and the result returned.

Note how the flavor of this process mirrors the notion of REDUCE in 3-LISP. In both

cases we go from designators of some abstract object, determine the result of applying an (abstract) function to (abstract) arguments, and then construct a designator for that result. Of course, as Smith points out, first order logic is, like 3-LISP, based on the notion of a designational semantics.

In many cases, the simulation structures will be theories also, with their own simulation structures. A FOL environment with many theories in place, using each other as simulation structures, looks to Weyhrauch very much like the interconnected structures found in human's models of the world.

4.2 Metatheories

A first order theory in FOL may be declared to have a "metatheory" which is another FOL theory. The relation of theory to metatheory is that, for example, constants in the metatheory represent theorems, proofs, and simulation structures in the theory. Functions in the metatheory might generate proofs in the theory. The metatheory has predicates which describe such things as "proves" and "follows from" in the theory.

An interesting result of representing the metatheory relationship between theories is that the user of FOL can often make use of the metatheory of a theory to simplify the task of generating and proving theorems. In the case of a metatheory which includes proof generation procedures for its theory we may, as Weyhrauch puts it: "change theorem proving in the theory into evaluation in the metatheory."

Suppose a user is attempting to prove a theorem `T` in a theory. If desired, the metatheory can be invoked by issuing the instruction:

```
REFLECT T;
```

Now, FOL attempts to find or prove a theorem of the form `(THEOREM T)` in the metatheory. If it is found, `T` may be added to the set of theorems in the theory.

Note the similarity between the notions of reflection in Weyhrauch and Smith. Especially similar is the utilization of the relation between levels of designation and levels of the processor. Just as `(THEOREM T)` is true in the metatheory if and only if `T` is a theorem in the theory, so the designation of normalising `(NORMALIZE EEP)` is the same as the designation of `EEP`, viewed from the bottom level of the 3-LISP tower.

4.3 Representing FOL in FOL

FOL can be described as a first-order theory, whose constants are axioms and theorems and whose relations are those of theorem proving: proves, depends on, simulates, and so on. So an axiomatization of FOL can be described to FOL, and FOL can prove theorems about how it would behave in certain situations. Weyhrauch calls FOL's representation of itself META.

META contains constants representing the various first order theories in FOL as well as the relationships and functions used by FOL to prove and verify theorems. META is, essentially, a description in first order logic of how FOL works. It is the sort of thing that would be offered as an account of the semantics of the program FOL. I was impressed by how similar the statements in META about FOL looked like statements in Smith's metatheory about 2- and 3-LISP.

An obvious question to come up when it is suggested that FOL be represented in FOL is this: what do we attach to META as its simulation structure? And the answer is almost as obvious: the LISP code for FOL. This move allows many interesting sorts of reasoning to be carried out, it also grounds META so that it can actually run. Notice the similarity with Smith's solution to the problem of implementing reflection. In both cases there are really two representations of the processor, one is represented in the language itself and it is this one that, for example, FOL could prove theorems about. The other is implemented in some other language, with the stipulation that it have the same behavior as the language explicitly represented. In both cases the implementing language acts as if it really were represented. Although Weyhrauch suggests the possibility, he doesn't demonstrate in anywhere near the detail of Smith how introspective activity could be made to work.

4.4 FOL's Promise

Weyhrauch's paper is short, but it contains some very powerful ideas. First of all, FOL really is a representational system. The axioms of the system have the assertional force that the LISP expressions considered by Smith lack. Furthermore the attachment of FOL to its own code allows the system to represent virtually everything about itself. Weyhrauch doesn't explore the various issues involved in such reasoning very much -- like Smith, he is most concerned for the moment with demonstrating the feasibility of introspective reasoning than actually trying it on examples.

A more general criticism of Weyhrauch's approach is the question of the importance of theorem proving to intelligence. Logic is static, yet an agent must dynamically adapt to a changing environment. Weyhrauch doesn't mention the ability of FOL to handle non-monotonic logics, though Doyle's work with a very similar representation language suggests that it would have the ability. The real problem is the representation of and the understanding of change, action and time. It is not explained how FOL represents a program, whether it is a sequence of states, or a set of theorems which depend on previous steps being proved, or what. This lack of an action theory limits the choice of applications of FOL to AI domains -- even the domain of using FOL. It is not obvious how the user could be represented in FOL.

But at the very least FOL has promise. For one thing, it is a successful attempt to express two non-first order relationships. One is the relation between a theory and its model. The model is represented by the simulation structure of the theory. The other

non-first order relationship is the relationship between a theory and its metatheory. Though both facilities are easily enough *provided* to the user, representing FOL in FOL allows even these relationships to be represented. To an imaginative sort, as Weyhrauch certainly is, the power of FOL seems staggering.

5. Doyle

Doyle's work is concerned specifically with the problem of deciding what to do, the task Aristotle referred to as *deliberation*. For the most part, artificial intelligence work on this subject has gone by the name of "problem solving." A process is viewed as having access to certain information about the problem domain, some methods for solving certain kinds of problems, and some goals. The process must then act in such a manner that eventually the problem is solved.

Before taking any action in the world, such programs typically spend a while planning or otherwise thinking about the problem. Usually the planning process is guided only by information about the problem domain. Doyle argues that "reasoning is a species of action" and thus merits recognition as a domain for reasoning itself. Doyle's project then, is to develop a theory of deliberate action, where action is either in the world or in the mind of the reasoner. An important assumption he makes is that the same general mechanisms can be used for reasoning in both domains.

Doyle's theory of deliberate action attempts to provide a foundation for the use of such mentalistic terminology as *reason*, *belief*, *intention*, *desire* and so on. This approach must be judged, not by our pre-theoretic intuitions about these terms, but rather in the context of the theory developed. The arguments in [McDermott, 1981] could be taken as against any use whatsoever of such terms, but I take the force of the arguments to be against wanton use of the terms without the sort of theoretical justification that Doyle explicitly provides.

The approach in Doyle's work is to attack the entire problem of constructing an intelligent agent. First, the problem of the requirements of knowledge representation are explored. Second, the system is given a theory of knowledge with which to represent various states of belief and their justification. With this in place, a theory of action is developed, based essentially on the paradigm of deliberating on possible actions, selecting one, and performing it. A general plan for deliberation is presented. Finally, plans and policies for deliberate changes to one's own processes and structures are discussed.

5.1 Representing Theories

Doyle's mechanisms for representation are for the most part precisely as in Weyhrauch, with a few additions. Like FOL, Doyle's representation system (Structured Description Language, SDL) is based on the idea of a theory, consisting of a language, axioms and theorems, paired with a simulation structure. The major addition of Doyle's approach is the ability for theories to have other theories as parts.

The simplest way that a theory can include another is via the attachment mechanism for simulation structures. Since a SS can be viewed as a way of getting answers about questions in a theory without having to go through the proofs, we can imagine

that an SS could be *another theory*. In this case the questions may take much work to answer, but the work will be done by another theory.

In SDL, furthermore, all attachments of theories are not actual links but *virtual copies*, which provide efficient representation of shared structure. The importance of virtual copies from the point of view of their use in SDL is that while each instance of a copy is initially identical to the original, changes can be made to each copy without affecting the original or other copies. Virtual copies allow one to attach theories to other theories in the language, and then modify a copy locally. For example, the following declaration would describe the use of a theory of addition to construct a theory about the sum of two particular numbers:

```
IN T-1
        INDIVIDUAL-CONSTANT T-1;
        ATTACH T1 T1;
        INDIVIDUAL-CONSTANT ADDER;
        ATTACH ADDER ADDER;
        AXIOM VC (T1, ADDER);
        ATTACH A1 3;
        ATTACH A2 5;
```

This declaration describes a new theory `T-1`, which consists of a virtual copy of the adder theory modified to have particular values for two of its three variables. An interesting notion, suggested by [Ken Haase, personal communication] is of making a theory of virtual copies, such that each theory has a copy of that theory and can modify it as appropriate. Such an approach might be useful for implementing arbitrary inheritance schemes.

A more complete way to compose theories is with the `TYPED-PART` declaration, which attaches a theory, names it, sets up pathnaming mechanisms and performs other bookkeeping functions. The idea of all this is to provide a mechanism wherein the relations between concepts (which are theories) are made explicit. For the most part, SDL theories are just collections of facts stated in a local vocabulary.

As in Weyhrauch, Doyle embeds a theory of his program in his description language. This will be discussed in more detail below, where it is shown how that description is actually transformed into actions, a step Weyhrauch never demonstrates.

5.2 Reasons and Beliefs

One of the most common failings of problem-solving programs is the lack of information about *why* certain decisions were made. Many programs simply take certain of several possible paths for whatever reason the programmer felt like. Whether they solve, or fail to solve their appointed problem, there is no way to recount the series of steps and justifications that went into the process. The language PLANNER introduced the technique of chronological backtracking as a strategy to recover from errors, but still was unable to account for why certain choices led to the failure in the first place.

One of the first attempts to record the dependencies a program's progress through a task domain was in [Sussman & Stallman, 1975]. In this system, whenever an assumption was made, the supporting facts were recorded in a dependency record. Thus if an error was later found, the program could trace back the dependencies to determine the set of facts potentially responsible for the error.

There are at least two reasons to record the dependencies of the program's results. One is the ability thus engendered to locate the assumptions or actions responsible for errors, giving the ability to recover from errors by changing only those facts or actions that had something directly to do with it. Another important reason is for learning. When an action is successful, it is useful for an agent to record why it worked, so a similar strategy can be used in similar situations. The dependency records provide the sort of summary of reasoning that can be used to compare situations for this purpose.

The Reason Maintenance System (RMS) is based around the task of representing the systems state of belief with respect to a content statement. The content statement is a statement in some SDL theory. A statement can be IN or OUT, the two states corresponding roughly to the system's belief or non-belief, respectively, in the content statement. A statement may be supported by a set of *justifications* which are essentially arguments for the belief of the statement, in terms of other statements. A statement and its justification is called a *node*. The idea of the IN/OUT mechanism is to implement a system of beliefs for a reasoner. It can be made to handle assumptions, default reasoning, reasoning from incomplete sources of information, and other kinds of reasoning modes.

The most common kind of justification in RMS is the "support list" justification. In this kind of justification, a statement is IN just in case all of a set of other nodes in the statement's "in-list" are IN and all of the nodes in the statement's "out-list" are OUT. A statement is IN just in case at least one of its justifications is satisfied.

An important property of this system is that the justifications are *non-monotonic* in the sense that a node can be taken OUT (made non-believed) if other nodes are brought IN. Consider the following case of a node justified by the lack of belief in its opposite:

```
(NODE P () ((NOT P)))
```

Here the second element of the list is the content statement, the third is the IN-list and the fourth is the OUT-list. From this statement, `P` can be brought IN just in case `(NOT P)` is not IN. If `(NOT P)` is ever brought IN, `P` will be taken OUT. This is an example of believing something "unless there is reason to believe otherwise."

Another important aspect of the RMS is that it records the dependencies of virtually everything the system does. Since the program of the reasoner will be represented in SDL, the system will eventually bring IN statements that will assert the doing of some primitive action by the problem solver. When the action is then performed, the dependencies of its results will also be recorded. So the program will have a record

not only of its reasoning but of its actions as well.

5.3 Dialectical Reasoning

A problem any reasoner must confront is that there are many decisions that must be made among incomparable objects. Despite the well-known problems, a shopper with limited funds must sometimes choose between apples and oranges. In fact most beliefs of a system are the result of choosing among statements which cannot be compared along a single dimension. The power of the RMS mechanism is that it allows many sorts of ways to justify the belief in a statement, and records the justification. RMS requires one to support not only the statement in question but also the *reasoning process* that led to it.

The importance of this is that it allows virtually any aspect of a program's belief system to be questioned. Conversely, it allows all elements of the belief system to be supported. We see the value of this approach for introspection: the program represents not only what it believes, but also why. And for any reason for why something is worth believing, there is a reason for *that*. And so on.

As situations change new facts emerge, new beliefs must be brought IN, others need to be taken OUT. We should thus think of the belief system as an active set of agents, each attempting to maintain the consistency and correctness of the belief system. Doyle introduces the method of *dialectical reasoning* as the technique for maintaining the belief system. In dialectical reasoning, each node can be thought of as an agent attempting to argue why it should be brought IN. So each node must produce a sequence of supporting nodes and their justifications. The sides then examine each other's arguments. One side may succeed in causing of the supports of another side's conclusion to become OUT, defeating that side's conclusion. The process continues, with each supporting node attempting to argue for its validity.

At any moment, there is a current set of believed statements which can be used by the reasoning agents, or by other programs. Presumably there are certain beliefs that rarely change, others are continually changing as new information is made available.

5.4 Intentions, Desires, and Action

Chapter 4 of Doyle's thesis presents what amounts to a theory of deliberate action. All of the information that an agent uses to decide what to do is represented in SDL as theories. Support for the theories is justified in the RMS. The agent is conceived of as using the IN rules, together with its beliefs about the current state of affairs, to decide what to do. These decisions are made in terms of the theory of action.

An agent is considered to have a set of *desires* which describe states of affairs. The idea is that it is somehow in the agent's interest for the state of affairs (called the *aim*

of the desire) to obtain.

An *intention* describes a state of affairs (also called its aim) as well as a commitment to actually perform some action to cause the aim to become true. The difference between the two states are discussed by Doyle as he explains why the more common idea of a "goal" was not used. The argument is that the two states differ in logical form: a desire is satisfied if its aim obtains; while an intention could be said to be satisfied just by attempting to carry it out.

5.5 Plans

A *plan* describes a sequence of actions the agent can take to actually carry out some intention. The plans available to the agent are stored in a library, each is annotated with descriptions of its utility and potential problems. The plan library and representation of plans are influenced by [Sussman, 1975] and [Rich and Shrobe, 1976]. A plan consists of a set of desires, intentions, and subplans that the system will use to try to realize an intention. The presently active set of desires, intentions and plans is called the "current state of mind." Invoking a particular plan, then, consists of entering a particular partial state of mind.

Plans are related in various ways, large plans are constructed out of simpler ones, some groups of plans are inconsistent, some share common prerequisites. So as the agent constructs plans in the process of deciding how to satisfy an intention, these relationships must be understood and dealt with. Doyle uses an approach very similar to the NASL interpreter of [McDermott, 1979] to describe the relationships among the states of partial fulfillment of an intention. The relationships among the various stages of the realization of an intention by activating and executing plans is shown in figure 2.

The relationships among intentions and plans can be described using the justification mechanisms of RMS. So a plan step can be kept OUT until the completion of its prerequisites allows it to be brought IN for active consideration. In fact this kind of mechanism is the way that sequential procedures are represented in SDL/RMS: the completion of a previous step is required for the following step to be considered. This control structure is roughly that of data-flow analysis and like that formalism, could be implemented in parallel. The justification relationships maintained by the RMS system effectively break up the reasoning into independent groups of nodes that can be processed concurrently.

5.6 TORPID

Doyle's theory of action provides a kind of semantics for the interpreter he then presents. He has described desires, intentions, beliefs, and so on independently of any implementational issues (just as Smith urges). His interpreter TORPID is a fairly

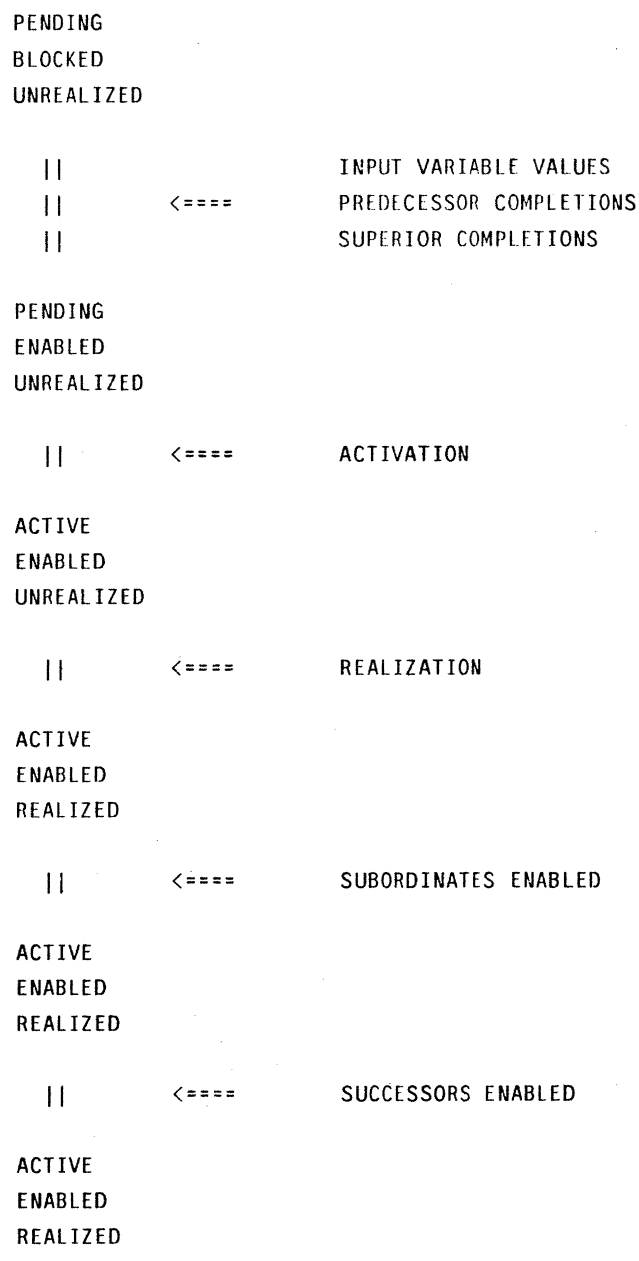

Figure 2: Intention progress state transitions. A pending intention is blocked until it has input variables, and its predecessors and superiors are complete. The intention then becomes ACTIVE, and when realized by a plan, is followed by its subordinates and successors.

straightforward translation of what has been described so far into SDL, with the relationships between the steps of the plan maintained by RMS. The idea is that the system always has an intention to "continue." Associated with this intention in the plan library is the plan TORPID whose steps are as follows:

1. Find a desire to turn into an intention.
2. Find an intention to pursue.
3. Find a plan to solve the intention.
4. Execute the plan.
5. Repeat.

Recall that the RMS is sitting in the background, maintaining the relationships among intentions. Crucially, the RMS will cause intentions to be taken OUT once they are realized.

Each step of TORPID is subject to deliberation: the system can decide what to do at each step, as well as whether to take the step at all. So TORPID is an outline of the strategy to use when deciding how to continue acting. But it is more than that: the plan TORPID is the system's representation of the theory of action it embodies. The theory of action defines desires, for example, as things that can be turned into intentions. TORPID represents this fact explicitly as the first step of its plan.

Each of the steps in TORPID represents the intention to carry out the described action. Attached to each of these intentions is a default plan. For example, in step 3 the default method to find a plan to solve an intention would be to find the default plan for the intention.

Doyle then presents the code for a TORPID interpreter. Like Smith's, the interpreter has two levels. One is written in SDL. The other is written in LISP and basically directs the full TORPID through its top-levels. The LISP version, MICRO-TORPID, has behavior identical to TORPID, in the simple cases it is applied -- mostly selecting problems for the full TORPID to work on. Each of the steps in MICRO-TORPID explicitly calls TORPID's default method for handling the step.

To get the system going, one starts up MICRO-TORPID with a single intention: to "continue." As mentioned above, the default way to continue is to use the plan TORPID. So MICRO-TORPID activates TORPID, activating the desires, intentions, and plans which constitute TORPID. The next step of operation, from the point of view of MICRO-TORPID, is to apply the default method to find a TORPID desire to turn into an intention. At step 4 of TORPID, when a plan is actually executed, the desires and intentions that constitute the selected plan are activated, and MICRO-TORPID goes to work on these.

Unlike 3-LISP, no explicit level shifting takes place in TORPID. Though each step of the TORPID plan can involve comparable amounts of reasoning as the whole plan does, the program just works by worrying about a set of beliefs desires and intentions.

Of course, the operation of the interpreter changes that set, and it is this process that actually directs the operation of the interpreter. Thus, instead of a reified environment and continuation, TORPID may inspect and modify its state by looking at the enabled (IN) statements about, for example, the current intention and its progress towards satisfaction.

5.7 Deliberation

I mentioned above that each step of TORPID was subject to consideration. The first three steps all instruct the system to make a certain choice: of a desire to turn into an intention, of an intention to find a plan for, of a plan to realize an intention. The important assumption in Doyle's work now comes forward: reasoning is a species of action. Therefore TORPID, which implements a theory of action, ought to be used to make these decisions.

In fact, these particular actions are crucial to the operation of TORPID because they must be taken for TORPID to work at all. So to complete a description of how TORPID works -- indeed to show that it *can* work -- Doyle must describe plans for the sorts of "finding" actions described in the first three steps of the TORPID plan. Doyle calls this kind of action "deliberation" and presents a general deliberation plan.

All introspective problem solving systems must be shown to be capable of solving the deliberation problem. They must be more than just systems for representing or organizing problem solving activity. Since they are introspective, and since their introspection is causally connected to their actions, it is required that they have methods for solving the deliberation problem, and that the methods be shown to work. Doyle's solution to this problem is to present a very general deliberation strategy, defend it, and then point out that it will rarely be used: in most cases more domain specific strategies will be successful.

The desire/intention cycle is reasonable as an account of the behavior of the agent *vis a vis* its environment, but Doyle is especially concerned with interior reasoning, the selecting of desires to pursue, the construction of plans, and so on. An important concept in this kind of reasoning is that of a a *policy* which Doyle defines as an "intention with a hypothetical aim." An example of a policy would be "If I'm hungry I should find something to eat." The most general interpretation of a policy is that it is roughly like a situation-action pair, where the action is not taken directly, but is considered. In the terms of RMS, a policy can justify the consideration of a certain intention. So the above policy has the effect of justifying an intention to find something to eat if the agent is hungry. Policies don't cause the agent to actually do anything about about the aim, but they cause certain intentions to satisfy the aim to be supported. Policies embody the values of the reasoner, because they represent the sorts of considerations it will apply to issues. One of the ways a reasoner can modify itself is to install various policies.

When TORPID realizes that it must make a decision, it forms a "decision intention" and begins to look for ways to make the decision. Relevant policies are applied to construct and support a set of options. The general deliberation method, if invoked, proceeds by first deciding among several alternatives:

1. Return a solution.
2. Delay.
3. Give up.
4. Pursue a subordinate decision.

This choice requires "second-order" deliberation -- it is a choice *about* deliberating

Assuming, as Doyle argues, that there are few second order deliberation strategies, and thus such deliberation will eventually halt with an outcome, we continue the general deliberation strategy. If the result of the second order deliberation was option 4, of pursuing a subordinate decision, that decision is then made. Most often this will result in a new state of reasons, beliefs and desires for the system. So the strategy loops back to consider what to do next.

5.8 Hints to a Reflective Program

The last part of Doyle's thesis describes suggestions and heuristics that a reflective problem solver might find useful. Though presented in English, the hints make use of the vocabulary given a theoretical basis by the rest of the paper. Many of the suggestions seem obvious, but, again, the fact that they can be phrased in a way that (at least potentially) can be understood by a machine is somewhat of an achievement. Many of the hints concern deliberate changes of the set of beliefs, desires and intentions of the system.

Here are some second-order deliberation strategies:

```
POLICY-3: If there us exactly one option, and it is IN, take it as the first-order
outcome.

POLICY-5: If there are no IN options, do not make a decision.
```

Here are some strategies about maintaining one's belief system:

```
B1: If someone informs me of a fact, try to explain it from my previous beliefs or try to
detect its inconsistency with them.

B2: If the observed effects of an action conflict with the effects I predicted, then
try to explain the failure of the predictions.
```

A program must be able to maintain, and occasionally reorganize the way it

understands the world. For example:

```
C1: If one concept (e.g. animal) has too many specializations (dog, perch, horned
toad) in the hierarchy for efficient searching, create new intermediate concepts
(mammal, fish, reptile) to decrease the branching factor and capture commonalities.
```

5.9 What has been Achieved

Before presenting a criticism of Doyle's work, let me reiterate what I feel are the rather major achievements. First of all, it attempts to be a theory of intelligent action. It takes the situation of an agent in an environment and presents an account of how such an agent should be equipped in order to figure out what to do. In some sense, this is an important direction to take in AI, perhaps *the* important direction.

In the process of developing the theory of action, Doyle technically defines and uses concepts from folk psychology such as beliefs, desires, intentions, policies and so on. This is also an achievement of note, both because such terms are only to be used if they are used carefully; and also because folk psychology has intuitive appeal (obviously). Thus Doyle's theory can be said to be at least *plausible* in that it does not wildly contradict our intuitions. (This is not to say that folk psychology is non-contradictory.) Of course the plausibility could derive directly from our flawed intuitions about the workings of minds. The *possibility* of this problem does not argue against Doyle directly, in fact since he has presented a (mostly) coherent and defined theory based on intuitive concepts, arguments against the intuitive concepts must take the existence of Doyle's theory into account.

Doyle's theory is mostly in need of distillation. This is not to say that a theory of the mind need be simple. Rather, it is not clear what the necessary elements in his theory are, and how they relate to one another. Doyle himself wonders whether the RMS mechanism ought to be represented in SDL (he concludes that it should). To a large degree Doyle seems not to attempt to make the various parts of the theory cohere in any way. There is no sense that a new construct is being introduced as a primitive or as a necessary addition to the theory. The theory of action he develops seems to be insufficient for the analysis of such questions.

The problem may be in the lack of an adequate notion of *action* itself. While he has primitives that can be run and policies that can suggest reasoning strategies, there is no well-developed notion of the program taking an action. This problem is best illustrated in the relationships of temporal ordering that must be represented by RMS in the TORPID code. While this representation is, as I said, useful for deducing ordering constraints and allowing the analysis of computational relationships, it never just represents the idea of *doing something*.

Though minimalism is certainly not a goal to be pursued at all costs, it seems as if a set of core ideas could be distilled from Doyle's work and implemented, with the rest

of the ideas defined in terms of the core. The interpreter could then be simple enough for the program to reason about, but as potentially powerful as Doyle's is now.

6. Comparisons

I have compared the approaches somewhat in the course of presenting the summaries. In this chapter I concentrate specifically on issues raised by more than one of them and also discuss some related approaches.

In general: Smith presents a narrow but deep analysis of what he sees as the core issue: the construction of an introspective processor. Along the way, he describes a designational semantics, but it is clear that this semantics is meant primarily to be a tool for analysis of the architecture he describes. Weyhrauch is concerned on the one hand with theorem-proving and logic, but his heart really lies with introspective processors also. He shows how one might represent a system in itself, but isn't very clear on the details. Certainly his representational system is more powerful than Smith's -- it can really be said to represent itself. Doyle's approach is wider, shallower, and aimed more at the issue of problem solving and intelligent behavior. His representation language is basically that of Weyhrauch. Like Weyhrauch, he is not specific enough on the actual architecture of introspection, though he presents a detailed theory of how it could be used.

6.1 The Architecture of Introspection

I am most struck by a theme that is discussed in all three approaches though specifically mentioned in none. It seems that full reflective power is available at the price of actually implementing only *two* levels of the basic processor. Doyle and Weyhrauch's systems are admittedly two-level, one for the declarative description of the processor and the other which implements it in some other language. Smith speaks of 3-LISP as an infinite tower of processors, and that is probably the best way to think about the language, but the implementation is also done in a two-level approach: one level of 3-LISP written in 3-LISP, the other level implementing 3-LISP, written in MacLISP.

This position seems intuitive: I can think, I can think about thinking. Do I ever (except when writing papers such as this) think about thinking about thinking? Perhaps, but what I imagine happens is that there are really two levels going on here. One is the object level -- the things I am thinking about. The other is the level I am thinking in. When I think about thinking, I transfer my thinking processes to the object level. This, of course, is precisely what Smith's processor actually does upon reflection.

What can Smith's particular implementation strategy tell us about implementing other introspective systems? It seems that the key idea is the realization that the implementation language can fake out the language it is implementing, so long as it can construct representations of the state the program would have been in had it really been running as defined. In order to be able to reflect down, the implementation must be about to tell that it is about to process a form that consists of a call to the processor to process another form. In that case, it can process the

second form directly.

For example, we might have a problem solver whose main function determines (WHAT-TO-DO-NEXT <situation>). It might be useful to introspect about what to do when trying to solve this situation. So it must be able to reify its own state, producing a state such as this:

```
(WHAT-TO-DO-NEXT (WHILE-CONSIDERING (WHAT-TO-DO-NEXT <situation>)))
```

While there may be special procedures to run when contemplating a WHILE-CONSIDERING situation, the problem solver must be able to consider that it can de-reflect this problem back into the original one if it wants. For example, it might modify the representation of the situation and then call WHAT-TO-DO-NEXT on the modified situation. As in 3-LISP, the system must drop a level, and run the fake implementation of the main processor on the new situation.

Doyle's system changes the objects of reflection, not by an explicit level shift, but by modifying the set of desires, intentions and plans, adding some description of the deliberation process. Though this approach may not make the levels of designation quite as explicit as in Smith's system, it has the advantage of representing what it is doing in SDL. So the program can, as it were, describe the fact that it is "thinking about thinking" to whatever degree. FOL would seem to have the same ability. This representation must be done carefully, of course, intuition suggests that one can get very confused thinking about thinking. One of the advantages of Smith's rationalized semantics and concept of a "tower of processors" is that it makes very simple to keep track of the degree to which the program is introspecting.

6.2 TORPID in 3-LISP

An obvious question that comes up is that of using 3-LISP as an implementation language for a introspective program like Doyle's. Smith considers this possibility also and admits that 3-LISP is a nice enough programming language to use. But a crucial fact is that reflective ability does not cross implementational levels. A language will not magically be reflective simply in virtue of being implemented in a reflective language. The ability to reflect must be a primitive part of whatever language is to be used. Of course, in TORPID, it is.

One possibility is to use 3-LISP as the procedural half of a dual-calculus approach, attaching to it a description language, such as SDL. Then one could describe the sorts of semantical equations Smith presents in his analysis of the dialects, as well as representing the code of the processor and a theory of how the code is to be interpreted. Such a problem solver could use the reflective abilities of 3-LISP to gain access to the state of the process, then reason about the best way to proceed. When it had finished this reasoning, it would process 3-LISP forms it had constructed from the environment and continuation arguments.

Such a program might not be able to do very much in terms about reasoning about "what to do next," but it could be the correct target language for a reflective compiler. Such a program would compile declarative descriptions of what it ought it do in certain situations into more or less runnable code for particular situations. The target code would use the reflective ability as a way to call these compiled routines, once an analysis of the computational context had been done.

6.3 Meta-ness

A crucial difference between the representational systems of Doyle and Weyhrauch on the one hand, and Smith on the other, is the role of the model of the expressions in the system. On Smith's view, expressions designate objects in the model, and processing is defined to preserve designation. In the SDL/FOL systems, the model is simulated. While Smith's viewpoint is cleaner conceptually, the notion of simulation seems to have power for cases when one wants to reason about a theory by examining the model.

It seems that there are at least two axes of "meta-ness". One goes from a process to the process it is running in. The other goes from a theory to its model. Reasoning in the model is reasoning about the theory, just as reasoning at the level of the interpreter is reasoning about the theory below. Weyhrauch's system, at least, provides the ability to do both sorts of reasoning, certainly they are both useful.

In fact Smith's account of designational aboutness may be too limited. While such limitations make it possible to say very clear things about the operations of his interpreters, it may not be enough to succinctly express the many relations representations can have with one another. Since Smith has only one kind of aboutness (namely designation), he can represent that relation implicitly in the nesting of environments and continuations in the environment of the reflective level. I get the impression from SDL/FOL that the levels of relations among representations would be more explicitly stated. This would seem to be an advantage.

6.4 Multiple Agent Theories

In examining each theory, one must be careful not to be misled by intuitions about present day computers. The intuitions can lead one both into false confidence of the success of a theory, or to a skepticism that such a thing could ever be implemented. In this section, I will discuss the architecture of the proposed systems, especially in terms of the ways that the processing could be distributed among independent agents.

At a coarse level of detail, Smith's system is a straightforward single-processor LISP interpreter. Yet it is understood as an infinite tower of processors each working on the next lower level. Even so, there is only one locus of real activity, which moves

between levels of the tower. I suppose that 3-LISP could be given some parallel primitives that could speed up its processing. Each copy of 3-LISP would then be an independent tower, communicating, perhaps, via the shared environment structure.

The multiple theories in FOL could be thought of as independent experts in their own domains. Requests from a theory to its simulation structure (also a theory) could be viewed as requests from one agent to another. Doyle's presentation of his system relies quite heavily on multiple-agent images. The method of dialectical argumentation can be viewed as a process wherein multiple agents each try to prove their own conclusions, and defeat each other's support.

In general, I think that a reasoning system could be viewed as a system of experts for various kinds of reasoning as well as domains. Higher-order experts would be invoked to adjudicate differences between other experts. A system of this flavor is presented in [Kornfeld and Hewitt, 1981].

There is no reason to suppose any special difficulty with introspection in a multiple agent theory of mental activity. If a multiple agent system can solve a real world problem, it can also solve one that involves introspection. If a multiple agent theory is more powerful at solving a real world problem than a single agent theory, then it might also be more powerful at solving an introspective problem.

However many agents there are, there is still one *system*. And the system is described by its program. Agents can, singly or in multiples, examine, reason about, and modify this program. There might be some changes in terminology: perhaps in such a system the agents should speak of "we" rather than "I".

6.5 Reasoning Experts

Several systems have been described which can be described as "reasoning about reasoning" though not introspective because they don't thing about *their own* reasoning. Eventually, one would hope, these efforts could be applied to introspective reasoning also.

Perhaps the first program of this variety was described in [Sussman, 1975]. This system reasoned about *programs* and represented the concepts of bugs, and debugging, all introspective ideas. Certainly an introspective program needs a well-developed theory of programming concepts. A more complete approach to the problem of programs that understand programs is the Programmer's Apprentice project [Rich and Shrobe, 1976]. Here, a very detailed representation system is set up for programs. The concept of a *plan* and its representation figure prominently in the effort.

[Lenat, 1982] is studying the creation and utility of heuristics. These may be viewed as hints to a behaving agent. As such they require a well-articulated description of

what the agent is doing. In any case, a developed theory of such hints would be valuable to an introspective program.

7. Needed: A Theory of Action

All of the approaches discussed in this paper have had to justify and support assumptions and points of view by relying on a more or less explicit theory of what it is that constitutes intelligence. In this chapter, I will argue that intelligence can be profitably studied in terms of the generation of behavior. A theory of *intelligent action* would elucidate the sorts of behaviors that would count as intelligent.

The emphasis on the generation of behavior is to specify what the intelligence is for, rather than any attempt to limit the scope of allowable items in the theory. This is not logical or psychological behaviorism, both disciplines holding that organism-level behavior only constitute the basis for explanation. I feel that interior actions are also behaviors and thus to be included in a theory of intelligent action.

A theory of intelligent action will be useful to determine what form knowledge about action should take. In fact it may prove valuable to approach the problem of knowledge representation from this direction, rather than as a problem in itself demanding a general solution. An action-theorist might adopt as a maxim: *Knowledge is in the service of action.* A representational system must be able to represent the relevant aspects of the theory of action. What *more* it may need to represent will be a later project.

It may very well be that minds are nothing more than ways to generate intelligent behavior. The recent development of "functionalist" philosophy, which holds that mental states and events are individuated by their abstract functional or computational roles and relations with one another, has led to the suggestion that minds are not natural kinds. That is: there is no set of essential properties, no necessary and sufficient conditions, no *definition* of what it is to be a mind. In any case, the task of a theorist in the functionalist framework is to try to determine, not what minds *are*, but what they are *for*. Claim: What minds are for is the generation of behavior.

I have argued previously and will simply hold as an assumption that there are no general rules for the answer to "what do I do next?". If that is the case, what sorts of questions can a theory of intelligent action answer? Most importantly, such a theory would provide a *vocabulary* for the discussion of relevant issues. The theory would have to elucidate the ideas of "goals," "program," "failure" and so on. For the most part, a theory of action would be like the theory of a programming language: providing the terms and forms of expressions for describing certain kinds of actions, but really solving very few cases. The theory would specify how the special-purpose solutions necessary for real world problem solving could be generated, maintained, and controlled.

7.1 Simplifying Doyle

Of the three papers considered here, only that of Doyle comes close to a full theory of intelligent action. Doyle presents a theory of deliberate action based on beliefs, desires, intentions, and so on. He describes ways in which these entities could interact to produce behavior. He also presents a general solution to the crucial problem of deliberation.

The problem is that Doyle's approach is not focused enough. Doyle's project was not simply to present a theory of action but also to present many things to do with it. Much of the thesis consists of specifications on how to use a theory of action, once it is in place. Also, Doyle gives no hints about the portions of his theory which are necessary, or are derivable in terms of others, or are simply frills.

On the other hand, Doyle has presented a relatively clear indication of how such a theory would go. Doyle's theory attempts to give theoretical justification to certain folk-psychology ideas, and seems to succeed. The framework he erected could probably be used to construct a more rigorous theory of action.

7.2 A Triangular Definition

In this section, I will briefly present a sequence of steps on the path to what I feel is a fairly simple theory of intelligent action. The technique is to start from a fairly general description of intuitions about intelligent action, and proceed by adding as little as possible until a reasonably general framework is established.

We first realize that it is a theory of intelligent *action* we are after. Thus an agent must be able to act, and its actions must modify the environment (internal or external) in predictable ways. So a theory of intelligent action must include terms about actions, it must be able to describe simple actions and their effects, and how actions can be combined and represented as compound actions.

There must be some reason *why* the agent should act. We must be able to say that certain actions are better or more intelligent than others. For this, let us introduce the notion of "goal". A goal is a state of affairs which we observers, in our understanding of the agent, realize is worthwhile for the agent to strive for. We certainly tend to say that actions that result in the satisfaction of goals are intelligent.

But do we? It seems that it is too easy to describe behaviors as intelligent if we simply postulate goals. What about situations where the achievement of a goal is by luck or accident? I think that a solution is to postulate that the agent has certain kinds of beliefs, such that it is aware that the action will satisfy one of its goals. I will call these "action beliefs" which represent the following kinds of things: "In situation S, action A will result in consequence C." Note that we have characterized a very specific kind of belief, and furthermore, the characterization is *semantic* -- the characterization is in

terms of what the beliefs are about, rather than any form they must take.

I believe we now have the minimal set of attributions necessary to describe behavior as intelligent. Intelligent behavior is precisely the behavior that makes use of actions based on action beliefs to achieve the goals of the agent. Note that "goal" must be defined in terms of intelligent behavior (my use of the term "strive" above) and beliefs must be defined in terms of actions and results.

So the terms are defined circularly. Actually, since there are three, we may say that the terms are defined "triangularly." This seems appropriate, there is no reason to suppose that, say, belief can be understood without the idea of an agent who is going to use those beliefs. And so on for the other three sides.

There may be more to a complete definition then just the three terms, I merely present the outline of a theory here. For example, we must specify how the agent comes to be aware that the situation part of an action belief is occurring, how the agent actually generates an action, and so on. I am fairly certain that it can't make do with *less*, though, and I feel it may be possible to describe other kinds of phenomenon in the terms of the triangle.

The advantage of such a simple theory is that it is fairly easy to represent, and thus could form the basis for a problem-solving interpreter. The basic action of the interpreter is to locate an action belief such that its situation is satisfied and its result would be the satisfaction of some goal of the agent -- and then perform that action. Naturally, each of those steps is an action and thus recursively follows the same procedure.

Note as I said at the outset, the theory mostly just defines terminology that can then be used to describe more complicated mental events. But, crucially, it also has imperative force: the theory actually specifies *what to do*, albeit in the most general terms possible. But this is correct, all we want the agent to do in general is to "be intelligent." There are no specific actions such that if an agent does them it is intelligent. Specific actions must be coded in domain-specific ways. And an intelligent agent is one who performs the correct domain-specific actions if it knows about them and wants what they cause.

A theory of action must be able to explicate notions of "program," "implementation," "process" and so on. It seems reasonable to suppose that an agent would make use of such concepts in planning and generating behaviors. In general, the understanding of the notion of a "plan" and things like "maps," "recipes," and so on are all terms from the same sort of language: the representation of action.

The representation of plans and programs is essentially an introspection affair, as argued above, thus a theory of action will probably elucidate the role that introspection will play in intelligent action. I will assume for now that the role is as follows: an agent takes representations of action beliefs and its current situation, and

determines (each step being an intelligent action itself) what is the right thing to do. This involves processing and analyzing the representations of goals and action beliefs. But, given the simplicity of the basis of the whole structure (something like the triangular definition) the program for the agent is just that set of representations. There is no principled separation between the description of the agent (its program) and the agent's description of its beliefs and goals. Introspection is thus not an additional skill -- it is just the way the agent works.

7.3 Compiling

A processor takes a description of action and turns it in to actual action. In a deliberative system, such descriptions may be very involved, with many potential actions to be considered. We can imagine that there would be several levels of descriptions, each more directly turned into action. The levels may correspond to levels of generality or applicability. The important aspect of the distinction is that with the more quickly applied plans, the agent need do much less deliberation for each step once it has decided that the plan as a whole is justified. This seems to be related to the concept of "compiling" in computer science. Compiled code runs faster because the processor has to do less in figuring out how to run it. Compiling also suggests (though does not require) optimization: which can be accomplished by determining, before the plan is run, what situations and steps can be ignored or simplified.

An important aspect of the process of compilation is that the compiled program *preserves the intension* of the source. Compiling a program is to produce a new representation of the same process (the same procedural intension) but in a form that is more immediately executed. This notion of compiling is also in line with the first usage of the term, when a compiler was something that took a high-level language like FORTRAN into machine language. The idea is that the process takes specifications of what to do in a declarative language, and turns it into a more procedural language. Compiling is also partial interpretation, with some steps performed at compilation time, and only the situation-specific tests and actions retained in the object code.

The compilation metaphor is a solution to how a program can both represent and embody some declarations of how it should work, such as a theory of action. Smith's method to make 3-LISP actually run was to analyze the operation of the interpreter and determine, for example, that the upper-level interpreter can be represented by a finite state machine requiring a very small amount of information to simulate. An introspective interpreter could determine similar things about itself. Such a program would run compiled: at any moment it would be stepping quickly through pre-compiled plans, but much of its activity would be concerned with compiling plans for later situations.

8. Paradoxes

The idea of self-representation or self-reference has been a major source of interesting paradoxes in the history of philosophy and mathematics. How might such paradoxes affect the feasibility of an introspective architecture? A simple answer is to say that if they do the problems can't be that great, given Smith's existence proof of a reflective interpreter. On the other hand, as noted, Smith's formalism doesn't really *represent* anything, let alone itself, so perhaps the problems still lurk. In this chapter I discuss three potential problems that seem to come with the representation of one's own inferential processes: the problem of instantaneous and unbounded reflection; problems raised by the halting problem and Gödel's theorem; and problems relating to self-prediction and free will.

8.1 Instantaneous Reflection

A problem with the notion that reasoning is a species of action, and that operations like "choosing" and "deciding" are actions, is that this seems to imply that one must "choose to choose" and so on, ad infinitum. This problem, and many like it were pointed out by Gilbert Ryle in *The Concept of Mind*, [Ryle, 1949]:

> "According to the legend, whenever an agent does anything intelligently, his act is preceded and steered by another internal act of considering a regulative proposition appropriate to his practical problem. But what makes him consider the one maxim which is appropriate rather than any of the thousands which are not? Why does the hero not find himself calling to mind a cooking-recipe, or a rule of Formal Logic? Intelligently reflecting how to act is, among other things, considering what is pertinent and disregarding what is inappropriate. Must we then say that for the hero's reflections how to act to be intelligent he must first reflect how best to reflect how to act?" (page 31)

It would seem that such an agent could never get anything done. If we are committed to the notion of a system explicitly representing potential actions, including all potential actions available, it would seen that this problem could arise.

Doyle's system, especially, would seem to be susceptible to the problems described by Ryle. The general deliberation procedure explicitly considers whether or not to reflect about the next question. And *that* question is one that is open for deliberation. Indeed Doyle makes no attempt to show how or when a program in TORPID could be relied upon to ever stop deliberating and come up with anything to do.

Though I do not claim that Doyle's system is immune to this problem, there is no reason all introspective programs succumb to it. The reason that the paradox doesn't work is the difficulty between the *ability* to do something, and actually doing it. We

require that an introspective interpreter be able to reason about any aspect of its inferential processes -- we don't require that it always do so. It is certainly within the framework herein developed that there could be some actions which in some situations are taken without introspection.

So the way a system could avoid the paradoxes of instantaneous reflection would be to not do it. Instead the program would only introspect about other situations. In some cases the result of those introspections would be compiled procedures to run when appropriate situations arise. This suggestion is developed more fully in the chapter on a theory of action.

8.2 Mathematical Paradoxes

The result of the proof that the halting problem is not recursively computable is that it is impossible to recursively compute virtually anything interesting about the behavior of a program. It seems to me that this problem is of virtually no consequence to introspective programs. For even given the problem, one can still make some true statements about some programs. Reasonable compilers can tell if certain kinds of programs will enter certain kinds of infinite loops. And "program-understanding" programs are able to represent much information about many kinds of programs.

Of course, there is a limit to how much can be said. But there is a more practical limit as well: in general an intelligent program should not compute everything it can about anything. We certainly would not count as intelligent someone who, when shown Euclid's axioms, immediately began generating theorems, ignoring eating, sleeping, or communicating results to others. The point is that an intelligent program must understand how to limit itself when considering a problem, whether the problem is solvable in principle or not. So the fact that it *can't* solve certain kinds of problems is masked by the fact that it *shouldn't* solve certain kinds of problems.

This same sort of consideration applies to the relevance of Gödel's theorem to the possibility of introspection. If the theorem is taken to show that certain true results are not provable in a system, then an introspective program will not be able to prove them. But, again, it probably shouldn't even try. If one takes the result of the theorem as demonstrating the limits of knowledge of a formal system, then the result holds for people also, inasmuch as they are computational systems.

8.3 Self-Prediction

Consider an agent with a perfect self-model. Suppose it's model is so good that it can predict, the day before the actual event, what it would decide in a situation it knows is coming up. On the one hand, the prediction can't really be correct because the agent could decide the next day, perhaps for the express purpose of thwarting the prediction, to do something else. On the other hand, if the agent actually does as

predicted, when did the act of choosing take place? It would seem that the action taken on the day of the choice wasn't really a choice at all, given the prediction of the previous day. On the other hand, the event of prediction was a prediction about a choice taken with multiple alternatives.

These considerations suggest that either of the notions of choosing or self-modeling are faulty. I suppose that a stance favorable to those who view mental operations as computational would be to deny that agents ever choose: that all actions follow computationally from the situation and state of the computation. Another position would be to deny the concept of self-knowledge either at all, or to the degree necessary for the paradox to go away. Or else one could deny that people are programs. I will argue that none of these positions need be taken: we are free to actually choose, and complete self knowledge is possible.

The resolution comes from the fact that, first of all, most predictions need not be made on the basis of a detailed model of the self. In many cases, a simple model will do. To the degree that such a model is causally connected it will be useful. It solves the problem by postulating that the self-model be incomplete or inaccurate. Such an approach is taken by [Minsky, 1965].

However, my line is even stronger than this. Consider the case of an agent with a complete representation of its program. It still can't accurately predict what it will do the next day because on the next day it will be a *different* program, assuming that it can make changes to itself. It will have a day's more perceptions, a day's more problems solved and so on. One of the differences will be that it will recall having made the prediction the day before. So the predictions made by a self-knowledgable agent are really just guesses, the actual choice will be made when it must be.

So at least the notion of choice is not inconsistent with that of complete self-predictive ability. But now a problem seems to remain with respect to free will. If the program has self knowledge enough to predict how it will act in certain situations, how can it be said to be free?

I certainly don't intend to solve this problem here, but I will remark on some ways in which the theories being developed for introspective action might deal with them. For example: an important application of free will is to justify the assignment of responsibility to agents. Morality seems to require that an agent be able to choose its actions freely, else what can it be held responsible for?

But notice that the notion of responsibility is actually discussed in Doyle's system independently of any notions of freedom. His RMS explicitly records the nodes "responsible" for a certain result. And a theory of action would account for the behavior of an agent in terms of (for example) its beliefs and desires. A notion of responsibility could be developed to take these determinates of action into account. Note that this notion would be vital to a multi-agent system interested in improving its own performance: the credit for good behavior of the whole must be divided among

those agents responsible for it.

Freedom, in fact, may play an important role in a theory of intelligent action. The concept would embody the idea that an agent must in general take action when all the facts are not in: it must choose freely among alternatives. The causally connected notion of freedom might simply represent the state of uncertainty between "what do I do next?" and actually doing it. A similar stance is taken by Kant in *Groundwork for the Metaphysic of Morals* [Kant, 1785]:

> "Every being who cannot act except under the idea of freedom is by this alone -- from the practical point of view -- really free." (page 100)

9. The Next One

In this chapter I discuss issues relating to the next attempt at an introspective interpreter. Some of the conclusions have been discussed above in the context of the discussions of the papers and more speculative ideas. I examine what such a system would take from each of the three papers, what sorts of domains it should be tested in, and what to do with the results.

The general features of the "next" program would be as follows: The main outlines of the system would be based on a theory of action similar to that of Doyle -- possibly the triangular definition outlined in section 7.2. The theory would be as simple as possible. This would make implementation easier and also help to determine precisely what is essential. The representation language will be basically as in Doyle with that system's major debt to the work of Weyhrauch. The contribution of Smith will be twofold: In formulating a theory of action, one will be taking Smith's advice about the relationship between determining *what* is to be represented before implementing anything. Also, an approach very similar to Smith's implementation of 3-LISP will be used to make the problem solver introspective.

9.1 A Domain

The most important problem with the systems of Doyle, Smith, and Weyhrauch is the fact that they have never been run on any real problems. They have never been given the chance to demonstrate the supposed power of introspective reasoning. I don't mean this as a criticism of the work, in fact I think that the first steps in such endeavors as this must always be speculative. However the existence of three formalisms, all roughly successful at what they set out to do, suggests that the time has come to try some real applications.

Unfortunately, implementing adequate applications may be extremely difficult. There are two major reasons for this:

1. Introspective programs will be very large and complex. Maintaining the dependencies of TORPID in an RMS, for example, involves hundreds of pointers just to go through the main interpreter loop. The system will probably have a large rule base which must be searched, or else clever hashing or discrimination-net schemes must be developed. Without a doubt, such programs will run extremely slowly on present machines. Parallel architectures will alleviate some of these sorts of problems, but it is not obvious how much.

2. To be a reasonable domain for introspective reasoning, the domain must be difficult enough so that introspection is in some sense necessary. In introspective tic-tac-toe playing program would be silly. So implementing an introspective program really amounts to exploring a fairly difficult problem with relatively untested techniques. [Davis, 1981] describes other criteria for an introspective domain, here

too, the general feeling is that introspection is more valuable as the problems get harder.

With respect to problem 1, a hope is that the program could reason about its own efficient use of the resources of the machine. And with respect to problem 2, the choice of a domain must be based on its being "just hard enough" to need to do introspection, but easy enough to imagine that a solution is possible. In fact it seems that a domain ought to be chosen in which other solutions exist already so that known problems can be avoided and methods of known promise can be pursued within an introspective framework.

9.1.1 An Intospective RMS

An earlier work by Doyle on a RMS-like system [Doyle, 1979], attempted to deal with inconsistencies in the data-base by using graph-theoretic algorithms on the dependency structure. This turns out to be very difficult, both in terms of finding appropriate algorithms and in terms of the complexity of such algorithms. Furthermore there are no general retraction mechanisms guaranteed to satisfy requirements on the state of the data base if it can represent programs and formal systems -- this must be true on halting-problem, and Gödel-theorem grounds. Doyle thus concludes that, in general, the problem of dealing with inconsistencies in the RMS data base is difficult enough to require the full power of the problem solver.

Truth maintenance systems have been in use in many domains for several years now, so quite a bit of experience has grown up about their power and limitations. Most existing systems either just record dependencies and let domain-specific programs resolve contradictions, or use general retraction mechanisms that are good enough for most cases. An introspective RMS would give the user the ability to specify ways to keep the network consistent and up to date, using domain-specific knowledge when the general purpose mechanisms fail.

Another reason to consider implementing an introspective RMS as the next domain for the study of introspection is that such a system would almost certainly be necessary for any other application program.

9.1.2 Planning

As mentioned in the chapter on a theory of action, planning actions or compiling declarative descriptions of actions may be a crucial part of the activity of an intelligent system -- it may indeed by the only way the system can ever do anything at all. Thus an important domain will be that of "self-compilation": writing introspective programs that can figure out what the agent should do in certain situations, given its knowledge and goals.

One potential application would be the problem of generating implementations of circular specifications of the sort presented in the 3-LISP meta-circular interpreter. It might be possible that such a program, given the specification of the 3-LISP processor, could produce an implementation of it in some non-reflective implementation language, along the lines described in section 3.6. The problem of compilation is hard, yet well enough understood to provide a domain for introspection.

Actual planning of the actions of an agent requires explicit access to representations of the skills the agent possesses as well as its requirements. An introspective implementation of Sussman's HACKER would be a good place to start, especially because what HACKER is supposed to be doing is writing programs itself.

9.1.3 Perception

Perception is the problem of constructing a description of the present state of affairs, from general knowledge about the world, and sensory input. A remarkable fact about human perception is that it is unbelievable fast, yet has virtually unlimited combinations of potential results. Somehow the descriptions are assembled as quickly as they are requested.

A theory of perception must account for how declarative knowledge about the world is compiled into perceptual procedures which describe not only what to look for, but how to interpret the results. Perception is influenced by expectations, knowledge, experience, and so on. All of this suggests that perception is very intimately associated with processes that pre-compile sequences of actions to perform when situations arise.

The process of perception is directed by the results of sensory input, yet the direction it can actually take are constrained by the perceiver's knowledge. So the compilation of a perception program must record the connections between states of the process, and the suggested percepts at each state.

9.2 Architectural Analysis

During the construction of an introspective interpreter for a particular domain, the researcher must be careful to understand the architectural requirements of the proposed process. How does it gobble memory? Where and how can a parallel implementation help? How does the acquisition of new knowledge affect the performance of the program? Other questions that must be asked deal with the interpreter itself: can it be shown to be finite? Are there situations in which it will spend forever recursively introspecting?

The answers to these kinds of questions are important for two reasons: First of all, if

the program is to effectively improve its efficiency, it must have some representations of these issues. Thus the programmer's understanding of the architecture of the system must be represented by the program itself.

Another important reason that such analysis must be performed is that it is a generally good thing to do in AI. We are at the point of coming to the limits of serial machines, the point where the constructing of advanced architectures is under active consideration. If, as I feel, introspective architectures are crucial to intelligence, their architectural requirements are of great importance and the machines of the future must meet them. The precise characterization of those requirements is therefore a vital aspect of this research.

10. Conclusion

The three papers considered here have made very valuable steps into a crucial domain. Any inadequacy of the approaches is more than made up for by the fact that the work has begun the solidification of what was previously a very ethereal topic.

Let me restate what I feel are the major points explored in this paper:

First, introspection can be described for the time being as causally connected, theory relative access of a program to itself. The program must know how it works: it thus needs a theory of itself. And the program must be able to change itself in predictable ways: it needs causally connected access to itself.

Second, Smith has actually presented us with the architecture of a system that is at least arguably introspective. At the very least, the implementation of his interpreter contains a "programming trick" that may be of some use and will present itself for consideration in the design of any introspective system.

Third, I have argued that the requirements for the representation system needed by an intelligent program be viewed from the standpoint of introspection: What must the program know to be able to know about itself? It must know *at least* this much just to be introspective, thus the representation system has a lower bound on its competence criteria. How much more must be represented remains to be seen.

Finally, the actual shape of the sort of theory an introspective system must have of itself has been suggested by Doyle. A fully introspective system embodies and represents a theory of intelligent action. The theory must allow one to express the relationship between actions and their desired consequences.

With respect to the relationship of introspection and artificial intelligence: Where other fields of research construct theories *about* their domains, the importance of introspection suggests that artificial intelligence theories will not only be about minds, the theories must be *in* minds as well.

References

Davis, R. (1980) *Meta-Rules: Reasoning About Control,* MIT AI Lab Memo 576.

Doyle, J. (1979) "A Truth-Maintenance System," *Artificial Intelligence* 12:231-272.

Doyle, J. (1980) *A Model for Deliberation, Action, and Introspection*, MIT Artificial Intelligence Laboratory Technical Report 581.

Hayes, P. (1977) "In Defense of Logic," in: *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* pp 559-565.

Kant, I. (1785) *Grounding for the Metaphysics of Morals,* trans. by J. Ellington, Hackett Publishing, 1981.

Kornfeld, W. and Hewitt, C. (1981) "The Scientific Community Metaphor," *IEEE Transactions on Systems, Man and Cybernetics,*, Vol. SMC-11, No. 1.

Lenat, D. (1982) "The Nature of Heuristics," *Artificial Intelligence,* 19.

McCarthy, J. (1968) "Programs With Common Sense," in *Semantic Information Processing,*, M. Minsky, ed. MIT Press.

McCarthy, J. and Hayes, P. (1969) "Some philosophical problems from the standpoint of artificial intelligence," *Machine Intelligence 4*, (B. Meltzer and D. Michie, eds.), American Elsevier.

McDermott, D. (1979) "Planning and acting," *Cognitive Science*, 2.

McDermott, D. (1981) "Artificial intelligence meets natural stupidity," in *Mind Design*, J. Haugland, ed. MIT Press.

Minsky, M. (1965) "Matter Mind and Models," *Proc. of the IFIP Congress.*

Nagel, T. (1974) "What is it like to be a bat?" *Philosophical Review*, 83.

Rich, C. and Shrobe, H. (1976) "Initial report on a LISP programmer's apprentice," MIT AI lab TR 354.

Ryle, G. (1949) *The Concept of Mind*, Peregrine Books.

Smith, B. (1982) *Reflection and Semantics in a Procedural Language*, MIT Laboratory of Computer Science Technical Report 272.

Sussman, G. (1975) *A Computer Model of Skill Acquisition,* Elsevier Press.

Sussman, G. and Stallman, R. (1975) "Heuristic Techniques in Computer-Aided Circuit Analysis," *IEEE Transactions on Circuits and Systems*, CAS-22, 857-865.

Woods, W. (1975) "What's in a link?" in *Representation and Understanding*, D. Bobrow, and A. Collins, eds. Academic Press.

Weyhauch, R. (1978) "Prolegomena to a Theory of Formal Reasoning," *Stanford AI Memo AIM-315.* Also in: *Artificial Intelligence,* Vol. 13, No. 1,2 April 1980.

Weyhrouch, R. and Thomas, A. (1974) "FOL: A Proof Checker for First-order Logic," *Stanford AI Memo AIM-235.*