The Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Working Paper 224 December 1981

Program Understanding through Cliché Recognition

Daniel Brotsky

Abstract

We propose research into automatic program understanding via recognition of common data structures and algorithms (*clichés*). Our goals are two-fold: first, to develop a theory of program structure which makes such recognition tractable; and second, to produce a program (named Inspector) which, given a Lisp program and a library of clichés, will construct a hierarchical decomposition of the program in terms of the clichés it uses.

Our approach involves assuming constraints on the possible decompositions of programs according to the teleological relations between their parts. Programs are analyzed by translating them into a language-independent form and then parsing this representation in accordance with a context-free web grammar induced by the library of clichés. Decompositions produced by this analysis will in general be partial, since most programs will not be made up entirely of clichés.

This work is motivated by the belief that identification of the clichés used in a program, together with knowledge of their properties, provides a sufficient basis for understanding large parts of that program's behavior. Inspector will become one component of a system of programs known as a *programmer's apprentice*, in which Inspector's output will be used to assist a programmer with program synthesis, debugging, and maintenance.

1. Introduction

We propose research into automatic program understanding via recognition of common data structures and algorithms (*clichés*). Our goals are two-fold: first, to develop a theory of program structure which makes such recognition tractable; and second, to produce a program (named Inspector) which, given a Lisp program and a library of clichés, will construct a hierarchical decomposition of the program in terms of the clichés it uses.

This work is motivated by the belief that identification of the clichés used in a program, together with knowledge of their properties, provides a sufficient basis for understanding large parts of that program's behavior. Rationale for this belief is given in Section 2.

Of the two results of this work, the more immediately useful will be the program Inspector. From a methodological perspective, however, a theory of program structure is a necessary precursor of Inspector's existence, and is of interest in its own right. Section 3 explains this point of view.

Inspector is intended to become one component of a system of programs known as a *programmer's apprentice*, described fully in [21]. Inspector's representation for programs and initial library of clichés are the results of earlier research by Charles Rich, Howard Shrobe, and Richard Waters. Section 4 summarizes this earlier work.

Our approach involves assuming constraints on the possible decompositions of programs according to the teleological relations between their parts. Programs are analyzed by translating them into a language-independent form and then parsing this representation in accordance with a context-free web grammar induced by the library of clichés. Section 5—the bulk of this paper—presents our theory of recognition and a scenario of Inspector analyzing a program. It also sets forth the criteria by which the success of the proposed work will be judged.

Finally, section 6 reviews the literature.

2. Why Cliché Analysis?

One goal of this research is a program which recognizes standard forms in programs. We claim that the ability to do such recognition is useful for an automatic program understanding system. In this section, we support this claim by viewing programming from an engineering perspective.

The Nature of Understanding

The measure of an engineer's understanding of a device is what he can do with it. Can he explain how it works? Modify it to do something else? Fix it when it breaks? As a special case of this, a programmer's understanding of a program is demonstrated in his ability to document, modify, and maintain it. If we are to

believe that an automatic program understanding system understands a program, it must demonstrate its understanding in the same ways.

Since understanding is always diplayed indirectly, the nature of an engineer's understanding of a device is not obvious. It may be that his understanding consists solely of his ability to reason about the device in the ways necessary to work with it. We believe, however, that his understanding has content apart from his reasoning ability, in the form of knowledge about the device which is shared among many problem-solving and reasoning processes.[1] This research is concerned with how appropriate knowledge of this type is brought to bear on a given problem.

Analysis by Inspection

Programmers understand programs not as monoliths but as hierarchical structures of components. When a programmer encounters a new program, he tries to understand it a piece at a time. When he encounters something familiar—a hashing function, or a linked list used as a stack—he can apply his pre-existing understanding of that piece to facilitate work on the program. He also integrates that knowledge with understanding of other familiar pieces to understand the program as a whole.

We call this process, wherein understanding of the whole is assembled from prior understanding of familiar parts, *analysis by inspection.* Its use runs throughout all engineering disciplines, not just programming. It provides a partial explanation for the expertise which an engineer accrues with experience: Not only does an engineer's depth of understanding of common components grow as he uses them repeatedly, but his familiarity with an increasing number of components allows him to easily analyze a broader range of devices.

Analysis by inspection is to be the primary program understanding technique of the *programmer's apprentice,* a group of programs intended to aid programmers with program synthesis, maintenance, and debugging. The first step in analysis by inspection—recognition of common program components—is the goal of this research.

3. Methodology

The methodology of this research is inspired by David Marr's distinction between theory, algorithm, and implementation [14]. In this approach, one separates the tasks of specifying the problem domain (theory), manipulating domain objects (algorithm), and representing domain objects (implementation), treating them as far as possible in that order. This separation is motivated by the observation that algorithms are dependent on those domain constraints which make their task tractable. Thus, design of a successful algorithm is possible only after such constraints have been identified by a domain theory; also, one adequate theory may provide enough "leverage points" on the problem that several methods of

[1] Arguments supporting this view are manifold. See, for example, [20 6 23].

solution become apparent. In addition, the theory which constrains the solutions to one problem in a given domain may be sufficient to constrain those of others; a fact which may not be apparent if the theory is never made explicit but remains embedded in an algorithm which uses it.

Having adopted this approach, our first goal is to specify those constraints on programs which make cliché recognition possible. Our theory will by no means address all aspects of the nature of programs; we will be concerned only with those features relevant to recognition. In particular, we will focus on how programs are built up from smaller pieces, since *a priori* knowledge of the possible structures for a given program allows quick location and identification of its standard forms.

Having identified the constraints which make recognition possible, our next step will be to construct a recognition algorithm. This algorithm, in addition to its utility in program analysis, will be of interest as a test of the program structure theory it embeds.

As to implementation, Inspector's implementation is being integrated with that of other parts of the PA system, such as an interactive program synthesis module. Thus the exact implementation structures used will depend in part on the needs of those parts. In general, aside from the use of a truth maintenance system [16], we do not anticipate Inspector's implementation to use mechanisms that are new or of especial interest in themselves.

4. Plans and the Library

This section reviews the theory and structures which Inspector inherits as part of the Programmer's Apprentice (PA) project. These include a theory of program semantics and inter-program relationships, as well as a library of program modules commonly found in symbolic manipulation programs. Before proceeding with the actual review, however, we spend some time investigating the relationship between the work proposed here and previous PA research.[2]

4.1. PA Research Directions

As mentioned in Section 2, the PA's primary tool for understanding programs is analysis by inspection. As the PA project has matured, so have its inspection techniques. The work proposed here is best understood in light of this historical development.

The PA system was first described by Charles Rich and Howard Shrobe in [19]. Also described in this document was a language-independent representation for programs called the *plan calculus*. As is common in ground-breaking work in AI, the original form of the plan calculus was inspired less by a desire for rigor than for intuitional accuracy and utility in problem-solving. A formal basis has since

[2]This section is not intended as a self-contained review of the PA. Readers who desire one should consult [19 20 22].

been supplied, but the original conception was so useful that it has been preserved almost in its entirety as the presentational form of the current plan calculus (see below).

The first use of the plan calculus in program analysis was made by Richard Waters [27]. In this work, Waters used the control and data flow relations between program operations (which are made explicit in the plan calculus) to segment programs into a hierarchy of loosely-coupled subsystems. This segmentation was desirable for two reasons: first, its groupings were natural from a programmer's point of view; and second, the function of each grouping could be deduced from those of its constituents (as shown in [9]).

Although Waters's approach captured the hierarchical feeling of analysis by inspection, it made no provision for the use of special knowledge about clichéd segments. In addition, it ignored entirely what has been called [25] the *teleological* role of the segments it identified; for example, that an operation which concatenates two lists may be viewed as taking the union of two sets represented by those lists.

These issues were the focus of work by Rich [23] in which he constructed a library of clichés used in symbolic manipulation programs. With this library, we need rely less on deduction to discover the functions of segments; we can look them up in the library and read their function as an annotation. Similarly, teleological annotations relating library entries show how they may be used to implement each other, and other types of annotations provide information that may be used in program synthesis or modification.

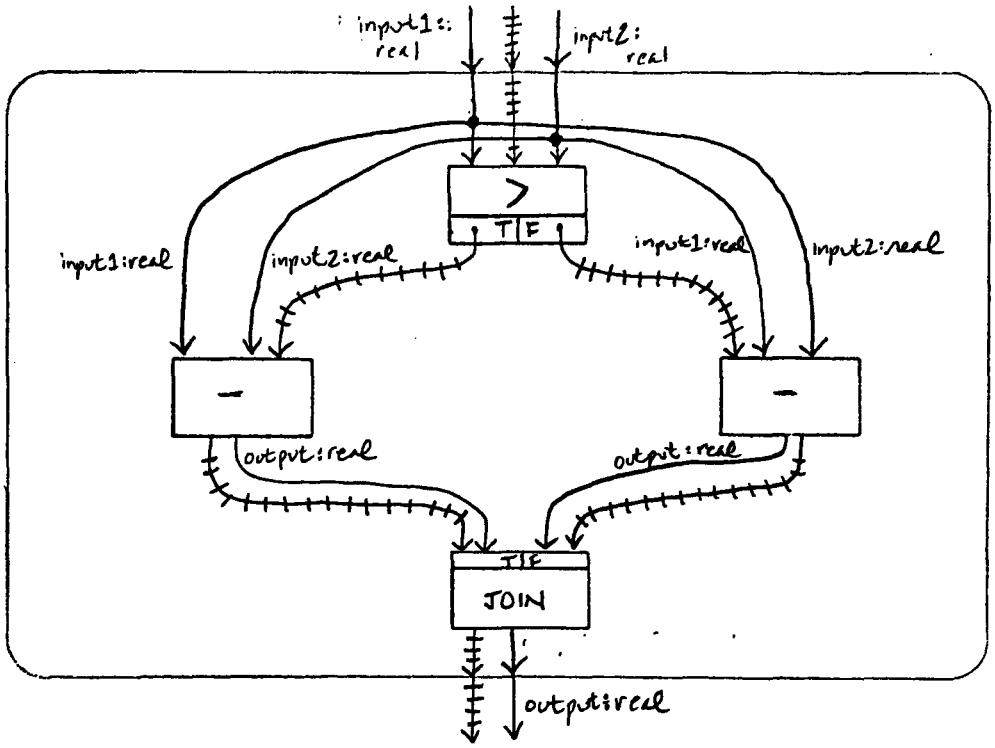
Unfortunately, we may not do program analysis simply by running Waters's segmentation algorithm and then looking up the resulting segments in Rich's cliche library. Since the choice of library entries made by a given designer may be somewhat idiosyncratic, we can not expect the segmentation imposed by a library-independent algorithm to agree with that imposed by a particular library. The aim of this research is an algorithm which takes both a program and a library as input, and decomposes the program according to the library. This is the first step in analysis by inspection.

4.2. Plans

The *plan calculus* is Inspector's representation for programs at all levels of abstraction. Inspector's input is a plan prepared by Waters's plan segmenter [27], and its output is a hierarchy of library plans representing possible derivations for parts of the input plan.

In this section we summarize those features of plans which are crucial to an understanding of the scenario in section 5. We do not present a complete description of the plan calculus, nor are we concerned with justifying the plan calculus as Inspector's chosen representation. Readers interested in these topics should consult [24] and [23].

Figure 1. The plan for ABS-DIFF



4.2.1. Plans as Flowcharts

To a first approximation, the plan for a program may be thought of as an augmented flowchart for that program. Its skeletal structure is that of a flowchart—operation and test boxes connected by control flow arcs—but to this structure are added logical annotations such as data flow information [8] and operand types or values.

For example, figure 1 shows the plan for the following program, which computes the magnitude of the difference of its operands:

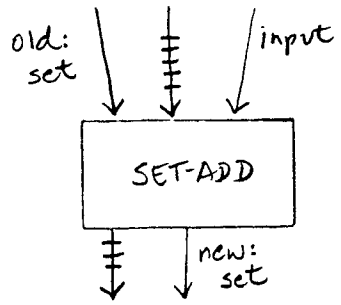
```
(DEFUN ABS-DIFF (X Y)      ;; computes |x-y|
  (COND ((> X Y) (- X Y))
        (T (- Y X))))
```

In this figure, rectangular boxes represent program operations. Arrows with hatched shafts represent control flow; those with no hatches represent data flow.

Although the operations in this plan are LISP primitives,[3] it differs from a traditional flowchart in significant ways. For one thing, the variables X and Y have been replaced by explicit data flows, which become the inputs and outputs of operation boxes. For another, the plan is annotated with the fact that the inputs to the initial test are real numbers. The relationship between such logical annotations on plans and their graphical skeletons, as well as the relationship between plans and programs, will be clarified below.

[3]Plans meeting this criterion are called *surface* plans.

Figure 2. The plan SET-ADD



The plan for ABS-DIFF shown above is known as a *temporal plan*, since it consists of multiple operations that are temporally ordered by control flow. Temporal plans are built out of three simpler plan types, each of which has an analogue in traditional flowcharts: single-operation plans (called *input/output specifications*) which correspond to flowchart boxes, control-flow splitting plans (called *tests*) which correspond to flowchart diamonds, and control-flow merging plans (called *joins*) which correspond to merges in flowchart control flow. For example, the plan for ABS-DIFF contains one test (the > test), two input/output specifications (the subtractions), and one join (the last box). Note that Plan joins have more structure than the control-flow joins in traditional flowcharts. Their purpose is to specify explicitly the dependency relation between branches in control- and data-flow; that is, that the data-flow resulting from a branch depends on which side of the branch was taken. In traditional flow-charts, this is left implicit in the variable assignments used on either side of the branch.

Figure 2 presents an input/output specification from Inspector 's plan library. This is the SET-ADD operation, in which an element is added to a set. Notice that this plan has exactly the same form as the input/output specifications in the ABS-DIFF plan shown above: there are input and output data flows with logical conditions both on and between them. There is no syntactic distinction between this abstract plan and the surface plan for SUBTRACTION;[4] this will allow Inspector to replace groups of operations in surface plans by the more abstract operations that they implement without violating their plan structure.

4.2.2. Plans as Predicates

The flowchart-like *presentation form* for plans presented above greatly facilitates insight into program structure. The semantics of such plans are rigorously defined by means of the *plan calculus* (PC), a variant of standard predicate calculus developed by Rich [23].

In the PC, a computer is modelled as an infinite set of *objects*, which are functions mapping *situations* to abstract mathematical structures called *behaviors*. This is a starting point common in modern programming language semantics (see,

[4]The plan calculus is thus a *wide-spectrum* language in the sense of [3].

e.g., [13] and [4]), and may be motivated by taking the objects to be computer memory cells, situations to be states of memory at given times, and behaviors to be the integers represented by the bit strings inside memory cells.

A plan calculus *computation* is an n-tuple of objects and situations. Intuitively, the objects represent memory cells involved in the computation, and the situations represent different states of memory that are reached during the computation. For example, a computation which adds two real numbers might be represented in the PC as a 5-tuple consisting of two situations—the memory states just before and just after the addition—and three objects—the two cells holding the inputs and the cell holding the output.

A plan calculus *plan* is a first-order predicate calculus formula whose variables range over objects and situations. Intuitively, this formula is viewed as a predicate on computations, and we say that a computation *satisfies* a plan if the plan is made true by substituting the computation's components for the plan's free variables.[5] For example, consider the plan:

$$\begin{aligned}&\texttt{real-addition}(\kappa = \langle s_1, s_2, o_1, o_2, O_3\rangle) \equiv\\ &\quad real(o_1(s_1)) \wedge real(o_2(s_1))\\ &\quad \wedge(O_3(s_2) = o_1(s_1) + o_2(s_1))\end{aligned}$$

which is part of the library plan for addition of reals. This plan asserts that, when the computation starts, both input cells contain reals, and that, when it ends, the output cell contains the sum of the two input values.

In practice, referring to components of computations by subscript or arbitrary variable, as was done above, makes plans impossible to read. The PC avoids this by introducing *role functions* to select the appropriate component; for example, we might write the above plan as:

$$\begin{aligned}&\texttt{real-addition}(\kappa) \equiv\\ &\quad real(input1(\kappa)(in(\kappa))) \wedge real(input2(\kappa)(in(\kappa)))\\ &\quad \wedge(output(\kappa)(out(\kappa)) = input1(\kappa)(in(\kappa)) + input2(\kappa)(in(\kappa)))\end{aligned}$$

4.2.3. Formal vs. Presentational Forms

The formal and presentational forms of a plan represent the same information in different ways, and translation between them is straightforward. For example, the formal plan for an input/output specification is practically a transliteration of the presentation plan; we can look at the diagram for SET-ADD shown above and read off the following:

$$\begin{aligned}&\texttt{set-add}(\kappa) \equiv\\ &\quad set(old(\kappa)(in(\kappa))) \wedge set(new(\kappa)(out(\kappa)))\\ &\quad \wedge input(old(\kappa)) \in new(out(\kappa))\end{aligned}$$

[5]Plan predicates are considered type predicates. A computation which satisfies a plan is said to be an instance of that type.

There are just a few subtleties worth mentioning. First, note that the "memory cells" of the plan calculus "virtual machine" are capable of holding *any* mathematical structure, from integers to sets. In the next section (about the library), we will introduce the plan-to-plan abstraction functions which allow us to work our way up from the bit strings of real machines to the sets and other structures of abstract plans.

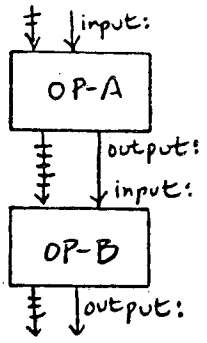
Second, note that the actual library plan for SET-ADD also contains the following two clauses, which insure that the *only* change made to the input set is the element insertion:

$$\wedge \forall x[x \in old(\kappa)(in(\kappa)) \rightarrow x \in new(\kappa)(out(\kappa))]$$
$$\wedge \forall x[(x \in new(\kappa)(out(\kappa)) \wedge x \neq input(\kappa)(in(\kappa))) \rightarrow x \in old(\kappa)(in(\kappa))]$$

In general, such clauses are left implicit in the presentation form. Their idiosyncratic nature is one reason the PC is used to provide an unambiguous semantics for the presentational form.

The relationship between the formal and presentational forms of tests and joins is very similar to that for input/output specifications. The differences center around the fact that tests have more than one output situation, only one of which will be reached in any given computation. This requires some formalism which we need not explore here; interested readers should consult [24].

Temporal plans are defined inductively in terms of other plans. The control- and data-flow connective tissue of these plans become clauses in the fomal plans which relate the output objects and situations of one operation with the inputs of another. For example, given this temporal plan:



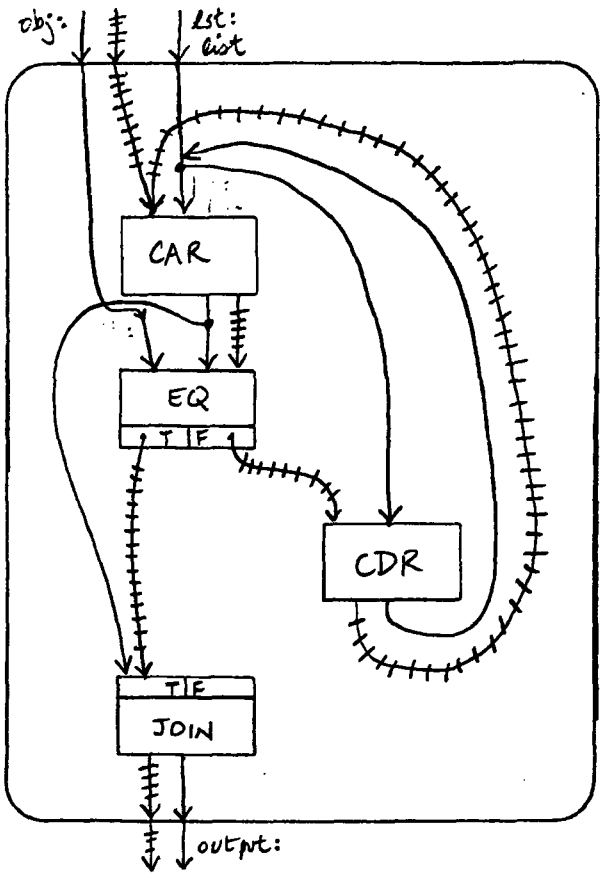
we would obtain its formal form essentially by concatenating formal plans for OP-A and OP-B and then adding clauses similar to the following:

$$\wedge cflow(out(\texttt{OP-A}), in(\texttt{OP-B}))$$
$$\wedge output(\texttt{OP-A})(out(\texttt{OP-A})) = input(\texttt{OP-B})(out(\texttt{OP-B}))$$

4.2.4. Recursion and Temporal Abstraction

Consider the following program, which returns the first member of a list equal to an input:

Figure 3. Circular plan for loop of LIST-FIRST




```
(DEFUN LIST-FIRST (OBJ LST)
  (PROG ()
   LP   (COND ((EQ OBJ (CAR LST)) (RETURN (CAR LST)))
              (T (SETQ LST (CDR LST))
                 (GO LP)))))
```

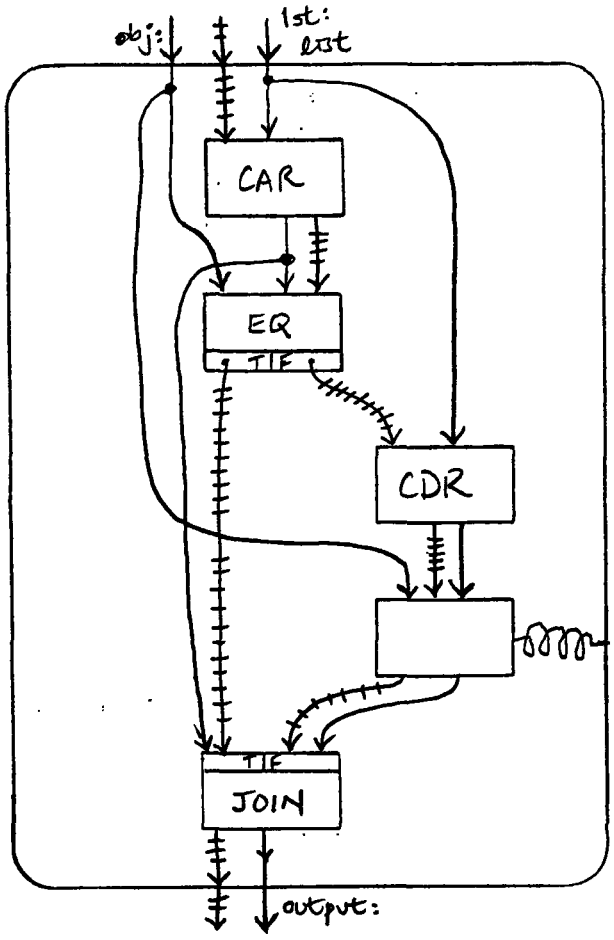
This program is iterative, and we might expect its plan to contain circular control-flow relations like those shown in figure 3. In the plan calculus, however, the program is represented in its equivalent tail-recursive form:

```
(DEFUN LIST-FIRST (OBJ LST)
  (COND ((EQ OBJ (CAR LST)) (CAR LST))
        (T (LIST-FIRST OBJ (CDR LST)))))
```

This gives rise to the tail-recursive plan shown in figure 4, in which the spring-like connecting line indicates that the inner plan is in fact identical to the outer plan.

The primary feature of note in tail-recursive plans is their similarity to plans for *non-recursive* procedural composition. For example, consider the plan for the following procedure, shown in figure 5:

Figure 4. Recursive plan for loop of LIST-FIRST




```
(DEFUN MARK-COPY (ARG)
  (COND ((MARKED? ARG) ARG)
        (T (MARK (COPY ARG)))))
```

This structure of this plan differs from that for LIST-FIRST primarily in that the inner procedure call is not recursive.

The resemblance between plans for recursive and non-recursive compositions suggests that analysis techniques intended for one type might be applicable to the other. Investigation of this possibility led Waters, Rich, and Shrobe to develop a representation technique known as *temporal abstraction*, in which a tail-recursive plan is represented as the non-recursive composition of other, special-purpose plans.

The essence of temporal abstraction is the observation that, since all the "iterations" of a tail-recursive plan have the same control- and data-flow structure, any given point in the presentation form of these plans is reached once in each iteration. If we think of copying the plan once for each iteration, each data flow in the plan for the initial iteration is replicated in each of the others. This gives rise to a *temporally distributed sequence* of data objects, all of which are outputs of the same operation. For example, in the above plan LIST-FIRST, we might consider

Figure 5. Plan for MARK-COPY

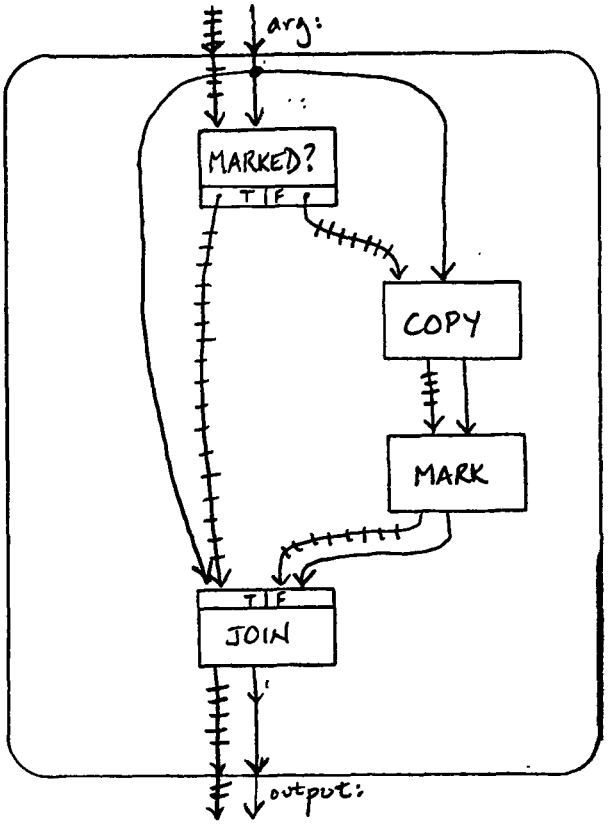
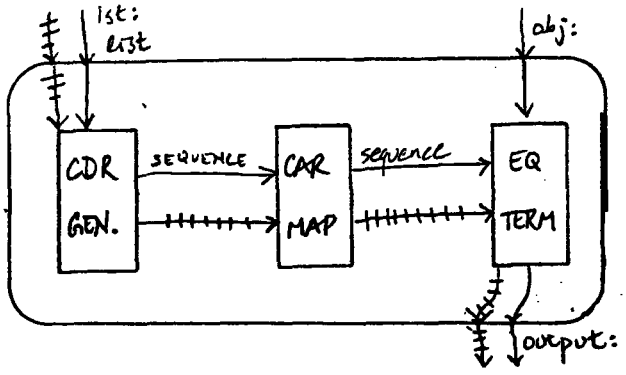


Figure 6. Temporally Abstracted loop from LIST-FIRST

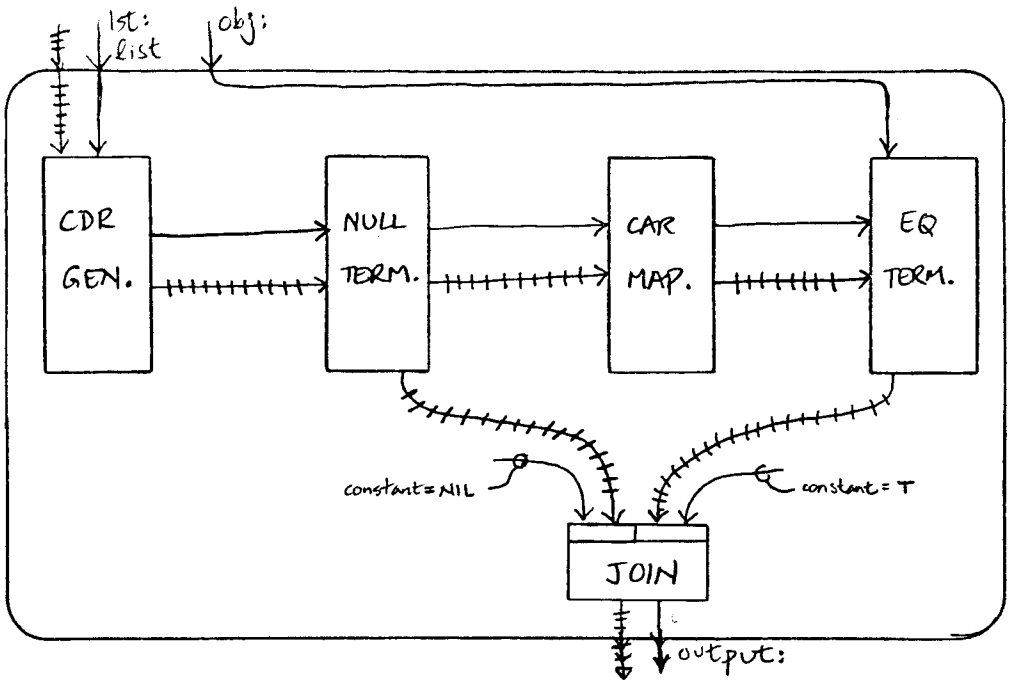


the sequence of outputs of the CAR or CDR operations.

We can *temporally abstract* such a sequence by considering a vector composed of its elements, and viewing operations in the plan as vector operations which operate on each element in the temporal sequence simultaneously. For example, the CDR operation in LIST-FIRST becomes a CDR GENERATOR which generates a vector of lists, while the CAR operation becomes a CAR MAPPING which accepts a vector of lists and outputs the vector consisting of their CARs.

Figure 6 shows the temporally abstract plan of the loop from the LIST-FIRST

Figure 7. Temporally Abstracted plan for LIST-MEMBER



procedure. The CDR operation has become a *generator* which takes a standard data object as input and produces a temporal sequence as output. The CAR operation has become a *mapping* which inputs and outputs temporal sequences. Finally, the exit test has become a *terminator*, which accepts a temporal sequence as input and outputs a standard data object. Note that the control- and data-flows connecting these operations are *temporal flows* which result from the temporal abstraction, not the standard flows of a temporal plan.

In general, the *temporal decomposition* of an iterative (tail-recursive) plan may lead to a number of generators, mappings, and terminators.[6] For example, consider the two-exit tail recursion used in the following code, which implements a list-membership predicate:

```
(DEFUN LIST-MEMBER (OBJ LST)
  (COND ((NULL LST) NIL)
        ((EQ OBJ (CAR LST)) T)
        (T (LIST-MEMBER OBJ (CDR LST)))))
```

This gives rise to the temporal composition shown in figure 7.

4.2.5. Programs vs. Plans

A program may be viewed as defining a set of computations, *viz*, the set of all computations which are gotten by executing the program on all valid inputs. Intuitively, the plan for a program should define the same set the program does;

[6] [27] discusses this decomposition in great detail.

that is, it should be satisfied by a computation if and only if that computation is the result of executing the program on some valid input.

In practice, a surface plan for a program may be obtained by taking a single-element plan for each of the various programming-language primitives used in the program, and then symbolically evaluating the program in order to determine the control- and data-flow connecting these plans. Program-to-plan translators which use roughly this technique have been written for LISP [19], FORTRAN [28], and COBOL [9].

As mentioned above, the plans Inspector will analyze are prepared from LISP code by a translator implemented by Waters [28]. The control-flow relations in these plans differ slightly from those in the original code, in that only the control-flow necessitated by data-flow and conditionals is preserved. For example, the control-flow ordering of these two assignment statements would not affect the execution of a program containing them:

```
(setq x (+ 3 4))
(setq y (+ 4 7))
```

In the plan prepared from these two statements, Waters's translator would not put a control-flow link between the two additions; they would remain unordered by control flow.[7]

4.3. The Plan Library

Inspector's initial library of plans for clichéd operations was developed by Rich [23]. Entries in the library are annotated with information about their relationships to other entries; this section summarizes the two most important types of annotations.

4.3.1. Specialization

A plan P_1 *specializes* another plan P_2 if $P_1 \rightarrow P_2$; that is, if the defining constraints of P_1 are stronger than those of P_2. In practice, specializations of plans are often derived by adding constraints on their inputs or outputs. For example, the library plan SORTED-LIST-MEMBER, whose input is constrained to be a sorted list, is a specialization of the LIST-MEMBER plan presented above.

4.3.2. Implementation

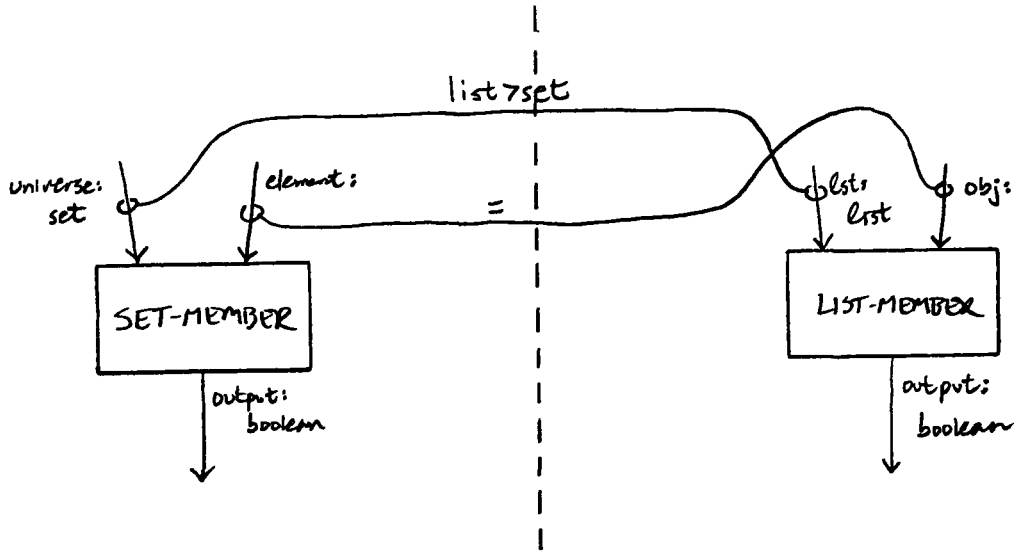
Some library entries are commonly used to implement others. For example, lists are often used to implement sets, and set membership operations are often implemented as list membership operations. Implementation relationships in the library are represented via special entries known as *overlays*.[8]

Formally, overlays are functions from instances of one plan type to instances of another. The image of an instance under an overlay is typically an abstract view

[7]Note that determining which control-flows are required by data-flows may be very hard in the presence of general side effects. Waters's approach to this problem is outlined in [27].

[8]Most of this section is taken directly from [24].

Figure 8. Overlay LIST-MEMBER>SET-MEMBER



of that instance; for example, one library overlay maps instances of linked lists to sets whose elements are the list elements. In practice, instances of a given plan type may often be implemented by their pre-images unders some overlay.

An example of an overlay between computations is given in figure 8. Intutitively, this overlay expresses how a SET-MEMBER operation may be implemented by the LIST-MEMBER operation introduced earlier, given that the set being tested is implemented as a list. Note that the diagram fro an overlay is made up of a plan on the left hand side (which is the domain type of the mapping), a plan on the right hand side (which is the range of the mapping), and a set of hooked lines showing a set of *correspondences* between roles of the two plans (which define the mapping). In this case, the role correspondences may be summarized as follows:

```
(defoverlay list-member>set-member (list-member)
  (= (element set-member) (obj list-member))
  (= (universe set-member) (list>set (lst list-member))))
```

In general, correspondences between roles are either simple inequalities, as in

```
(= (element set-member) (obj list-member))
```

which says that the OBJ object in the LIST-MEMBER computation (which is tested for membership) corresponds to the ELEMENT object in the SET-MEMBER operation; or they are defined in terms of other overlays, as in

```
(= (universe set-member) (list>set (lst list-member)))
```

which says that the set being searched in the SET-MEMBER operation is the set composed of the elements of the list being searched in the LIST-MEMBER operation.

One final point to note about overlays is that they are not only part of the taxonomic structure of the library, they are also used to construct the refinement trees of programs. When Inspector "recognizes" a SET-MEMBER operation in a

program, it will in fact be recognizing a surface plan which, through a sequence of library overlays, can be used to *implement* a SET-MEMBER operation.

5. Scenario

In this section we present our recognition theory and a scenario of Inspector at work. We begin by describing an overly simple approach to the recognition problem, but one that nevertheless formed the basis of our approach. This intuitive description is followed by a more formal treatment of our current thinking, which leads naturally into a scenario of Inspector's operation. We conclude by stating criteria for judging the success of the proposed work.

Throughout what follows, we will use the terms "plan analysis" and "program analysis" interchangeably. While the goal of this work is program analysis, existing systems which translate programs into plans [27] make it sufficient to solve the problem of plan analysis. Since Inspector will make little reference to the actual code of its input programs, this section deals entirely with analysing the output of the translation system—a plan.

5.1. Overview of the Problem

A well-designed program embeds a hierarchical decomposition of the problem it solves. The task of cliché recognition is: given a library of problem decompositions and a program, identify those pieces of the program hierarchy which occur in the library. This is the first step in bringing library knowledge about such pieces to bear on engineering work involving the program.

An intuitive picture of a possible recognition algorithm is provided by imagining library plans drawn on transparent slides, such as those used in overhead projectors. Given the plan of a program to be analyzed, we take the library slides one at a time and slide them around on top of the program plan. Wherever the components of a library plan match up with pieces of the program plan, we have recognized some cliché, and we add to the program plan an extra box representing the entire cliché.

We then repeat this process, but this time we move the library slides around on top of our augmented program plan. The plans on these slides may match up with parts of the original program plan or with previously recognized library plans; the latter case giving rise to a hierarchy of recognized plans. As before, we augment the program plan with recognized clichés and repeat the matching process again, stopping when we can no longer find matches for any of the library plans.

This process strongly resembles bottom-up string parsing, suggesting an analogy which proves apt. Just as strings may be generated using grammar rules to expand single *symbols*, programs may be generated using library entries to expand single *operations*, such as a hash-table insert or lookup. Parsing techniques, which in the string case recover a generating sequence of rule applications, may be used in our case to recover a generating sequence of library entries, that is, cliché usages.

We proceed, then, by embedding the process of cliché-based plan (program) synthesis in a context-free grammar framework. This embedding allows cliché recognition by parsing for those program plans (or parts thereof) that can be generated from the cliché library. Special problems arise in the parsing of such plans; we consider these and show a sample parse. Finally, we note that standard optimizations applied to real programs lead to derivation structures outside of the context-free framework. We designate as *well-structured* those plans whose derivations are "optimized context-free," and adapt our parsing techniques to handle them. These well-structured plans will be those in which we can do cliché recognition; only experimentation can show how broad a class this is.

5.2. Plan Synthesis in a Context-Free Framework

Throughout our discussion of grammar-based plan synthesis and analysis, it will be useful to keep in mind an analogy with string grammars. For example, consider the grammar for Pascal [12]: In synthesis, it tells us that we can implement a statement of the language as an assignment statement, an `IF-THEN-ELSE` statement, or many others. In analysis, it tells us that an expression of the form `VAR[INDEX]` is to be interpreted as an array reference. Similarly, in synthesis our *cliché grammar* should tell us that we can implement a set as a list; in analysis it should tell us that the statement

```
(COND ((< X 0) (- X))
      (T X))
```

may be interpreted as taking the absolute value of `X`.

5.2.1. Graph Grammars

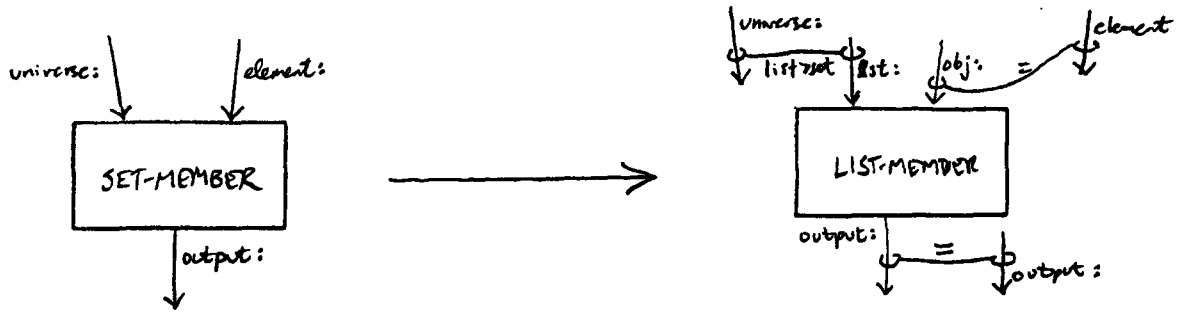
Before we can express plan derivations in a context-free framework, we must extend our concept of grammar to derive more general graphs than just strings. We will adopt the formalism of *web grammars*, developed in [18] and extended in [17].

The intuition here is that strings may be thought of as directed acyclic graphs (DAGs), all of whose nodes have restricted in- and out-degree. Just as a context-free string grammar generates strings by successively replacing nodes with strings; a context-free web grammar generates more general DAGs by successively replacing nodes with DAGs. Since the presentation forms of plans are just DAGs, web grammars are ideal for expressing derivations in which a single operation of an existing plan is replaced (implemented) by those of some other plan.

5.2.2. Library Entries as Grammar Rules

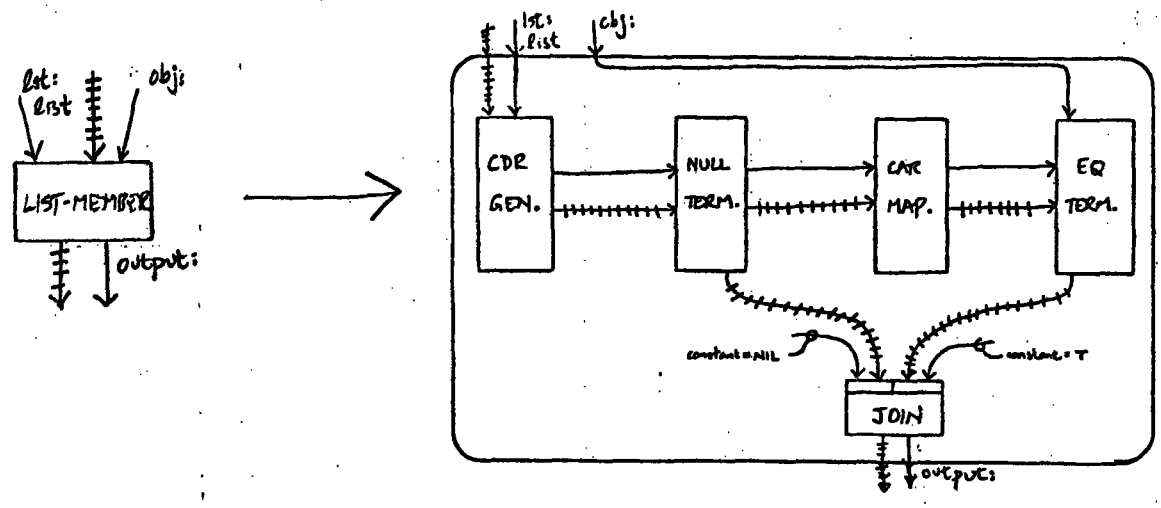
Now that we can derive plans using a web grammar, we must decide on the particular rules this grammar should contain. Intuitively, the left hand side of such a rule (a single box from a plan diagram) will specify something to be implemented, and its right hand side (an entire plan diagram) will specify a possible implementation. Thus, every entry in our cliché library will induce a rule which expresses its embedded implementation relationship, leading to the term *implementation grammar* as a synonym for cliché grammar.

An overlay with domain P and range Q specifies how an instance of plan type P implements one of type Q. Thus for an overlay P>Q we construct a rule whose left hand side is Q and whose right hand side is P. For example, section 4.3 introduced the overlay LIST-MEMBER>SET-MEMBER, which induces the following rule:



Note that overlay-induced rules take single nodes to single nodes; they express changes in the programmer's view of an operation or data item. We call them *abstractions*, and there may be many abstractions with the same range (*e.g.*, many implementations for sets).

Temporal and data plans show how single operations or data items may be constructed out of others. Thus a plan P with roles $R_1, \ldots, R_n$ becomes a rule whose left side is the node P and whose right side is P's presentation form (a DAG with nodes for $R_1, \ldots, R_n$). For example, section 4.1 introduces the plan LIST-MEMBER which implements a list membership operation as a list enumeration followed by a sequence-search. This plan induces the rule:



As described in [18], the rule specifies how the connectivity of the left side determines the embedding of the right side in any graph to which this rule is applied. Rules of this type (induced by temporal or data plans) describe clichéd groupings of operations; we call them *aggregations*.

Finally, the plans for primitive programming language operations are the terminals of our cliché grammar. Their presentation form consists of a single node, either an input/output specification or a test. The plans Inspector will parse are

Figure 9. Set implemented as a Headed, Irredundant List

```
(DEFUN SET-CREATE ()
  (LIST NIL))

(DEFUN SET-MEMBER (E S)
  (MEMBER E (CDR S)))

(DEFUN SET-ADD-NEW (E S)
  ;;precondition: (not (set-member e s))
  (RPLACD S (CONS E (CDR S)))
  S)

(DEFUN SET-ADD (E S)
  (COND ((SET-MEMBER E S) S)
        (T (SET-ADD-NEW E S))))

(DEFUN SET-DELETE (E S)
  (PROG (THIS NEXT)
        (SETQ THIS S)
   LP   (SETQ NEXT (CDR THIS))
        (COND ((EQUAL E (CAR NEXT))
               (RPLACD THIS (CDR NEXT))
               (RETURN S)))
        (SETQ THIS (CDR THIS))
        (GO LP)))
```

constructed from these terminals by Waters's program translation system [27], about which we say more in section 5.5.

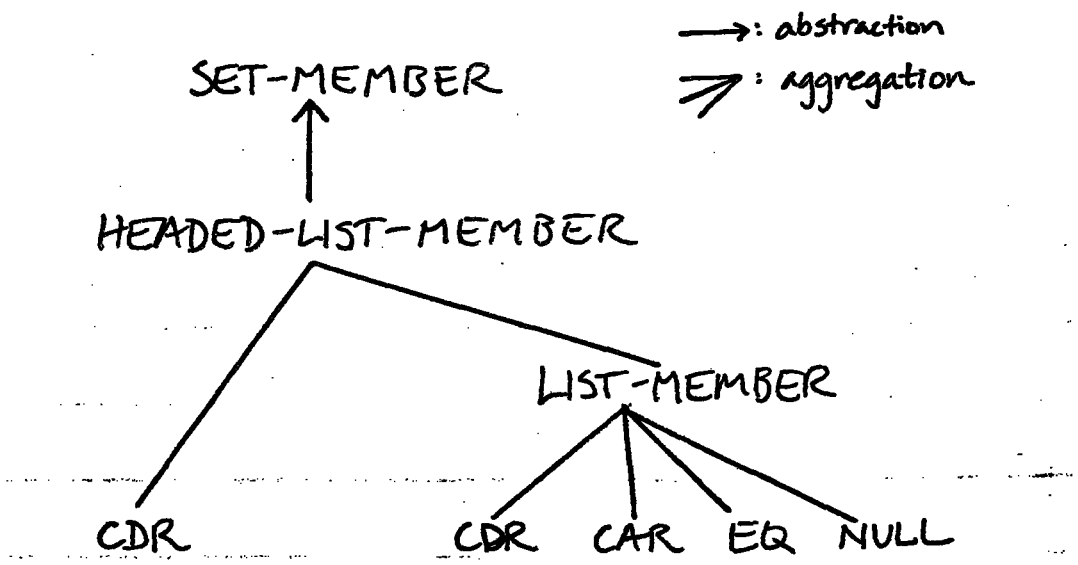
5.2.3. A Sample Derivation

Consider, as shown in figure 9, implementing a set as a list with a header cell and no duplicate entries.[9] A simplified version of the derivation tree for the SET-MEMBER operation is shown in figure 10. (Control and data flow relations have been suppressed, and the connections to the final code simplified, so as to produce a tree with a more conventional appearance.) This derivation makes use of the LIST-MEMBER plan mentioned above.

One subtle feature of this derivation system must be mentioned; to wit, the interplay of "syntactic" and "semantic" concerns. In traditional programming language grammars, the semantic consistency of derived strings (*e.g.*, agreement of operand types in expressions) is not considered; only the form of allowable

[9]The header cell allows operations by side-effect—a useful technique if a set is to be pointed at from multiple locations. The lack of duplicate elements is part of the interface between the element insertion and deletion operations.

Figure 10. Derivation Tree for SET-MEMBER



expressions is specified. In our case, since the constraints of the domain semantics (*i.e.*, computations) are part of the plans themselves, no distinction may be made between the syntactic and semantic applicability of a rule. From a synthetic view, this means that *all* derivable plans are meaningful.[10] Its ramifications for analysis are discussed below.

5.3. Parsing Tree-derived Plans

We are now ready to consider parsing plans which are derivable using the context-free web grammar system described above. We begin by considering the requirements our parsing methods must meet, and some of the decisions we have made in satisfying them.

5.3.1. Partial, Bottom-up Recognition

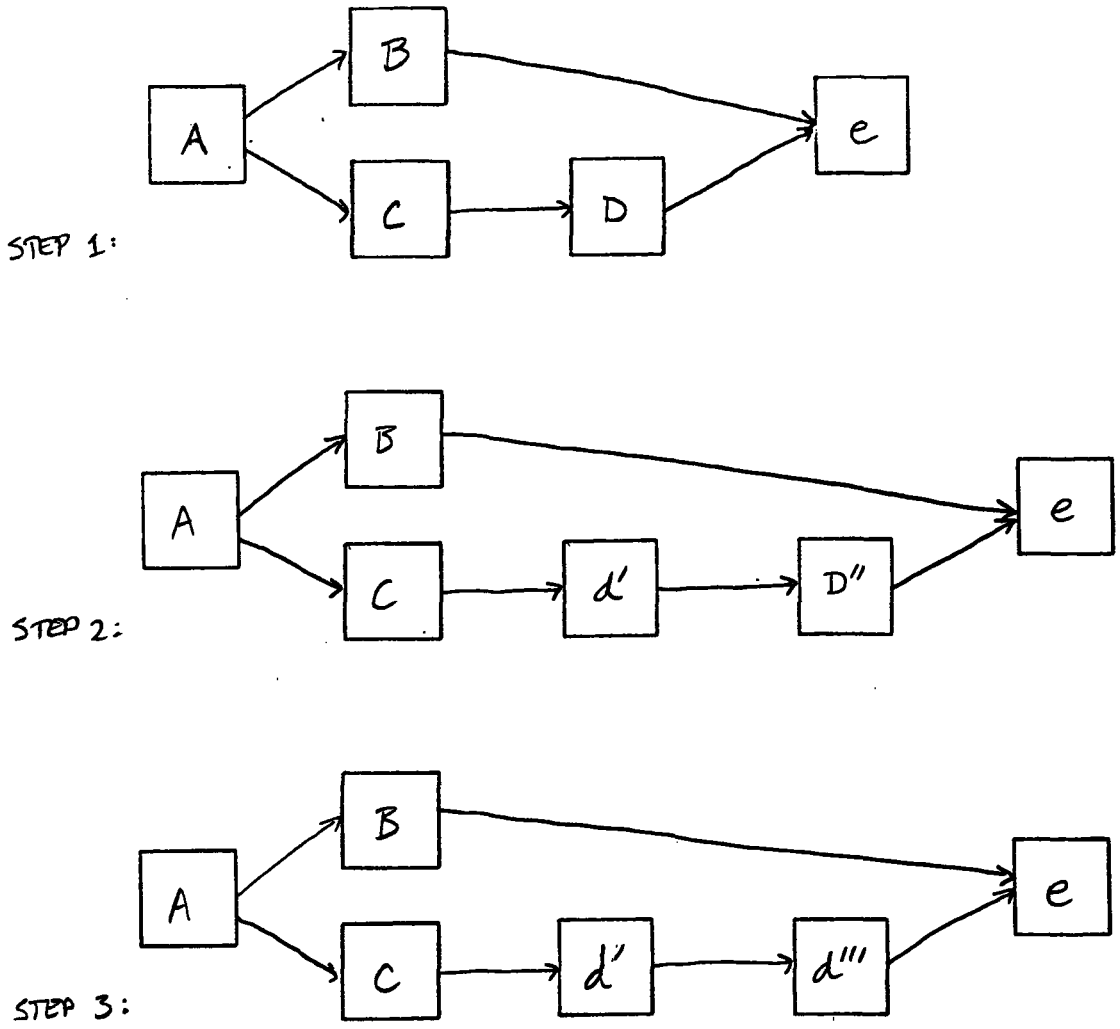
Real programs are not composed entirely of clichés. Thus, our parsing technique must pick out of a program's plan those sub-plans which may be derived from our library. The "sliding transparency" method of section 5.1 comes to mind, and its suggestion that we work bottom-up.[11]

In traditional left-right bottom-up string parsing, one tries to recover a rightmost derivation for the input by reducing rules in their inverse order of application. The lack of a total order on plan nodes prevents adopting this strategy directly; there is no clear referent for "rightmost derivation" or "right sentential form." However, since plan nodes are partially ordered by their control flow links, we can generalize the string case by trying to recover a *right-minimal derivation*, that is,

[10] This is certainly not the case for all syntactically correct program strings!

[11] Reader familiarity with some traditional shift-reduce parsing methods is assumed in the following material.

Figure 11. Sample Right-minimal Derivation (partial)



one in which rules are only applied to nodes who have no non-terminal successors in the partial order.

To clarify this concept, consider the sample right-minimal derivation of figure 11, in which terminal elements are labelled with lower-case letters and non-terminals with capitals. At step 1, only nodes B and D are eligible for rewriting; node C is succeeded by D, which is a non-terminal. Even after node D is rewritten once (to give step 2), node C may not be rewritten: it still has a non-terminal successor. Only in step 3, once its successors are entirely terminals, may C be expanded. Of course, node B is still eligible for rewriting in both steps 2 and 3.

Our strategy, then, will be to find a right-minimal derivation of the plan being parsed. We will do this with a generalized shift-reduce technique which finds a left-minimal handle of the plan being parsed, and reduces it as usual. We give more

details below, but first we deal with some of the considerations imposed by the merging of syntax and semantics mentioned above.

5.3.2. Logical vs. Structural Constraints

Plans are constructed out of two sorts of constraints. There are the *structural* constraints shown most clearly in the presentation form: the number, arity, ordering (control-flow), and connectivity (data-flow) of its operations. There are also the *logical* constraints on or between operations and data elements, such as their type (*e.g.*, integer or set-of-integers), intrinsic and relative properties (*e.g.*, that a list is irredundant, or that one list is longer than another), and other special features. Some plans (*e.g.*, `ABSOLUTE-VALUE`) have very few logical constraints; others (*e.g.*, `SET-MEMBER`) have almost no structural constraints.

While structural and logical constraints have the same formal status (predicates on plan components), it would not be wise to treat them *computationally* uniformly. Structural constraints are made explicit when plans are coded in any standard programming language, and they are rarely the subject of complicated deductions. Logical constraints, on the other hand, are rarely made explicit in a program's code, and are often obtained only as the result of much deduction. We must expect that, in general, the necessary computations involving logical constraints will cost much more than those involving structural ones. Wherever possible, we should delay work on logical constraints until all possible structural work has been completed.

To effect such a division of labor, we group grammar rules into classes with structurally identical right-hand sides, and break the reduction process into two parts: First, we locate a handle which belongs to one of the structural classes; and second, we go through the members of that class until we find one whose logical constraints are satisfied by the handle. This breakdown is analogous to the traditional syntax/semantics distinction used in programming language parsing: first, the form of a rule's right hand side is located, and then its semantic applicability is checked.

Note that our structural classes will be closely related to the specialization links in the cliché library. Plans which are related via specialization are structurally identical, differing only in their logical constraints. Once the rule induced by some plan is found to be a candidate for reduction, all rules induced by generalizations of that plan must also be candidates.

5.3.3. The Structural Task

We are now ready to explain the structural side of our parsing strategy. The next section explains how this interfaces with the verification of logical constraints.

As mentioned above, we will be using a generalized shift-reduce strategy. We will stick to a simple algorithm, taking as our handle the first left-minimal subplan which structurally matches some rule's right-hand side, and backtracking if this leads to an incorrect choice. This strategy is motivated by the desire to locate *all* instances of cliché usage, even those which were not intended by the programmer and whose recognition is thus inconsistent with recovery of a complete derivation.

Although there are faster parsing algorithms, such as the LR(k) techniques, they use context to avoid such accidental reductions, and so are not suited for our purposes.

Our central structural task, then, is to find a *structural handle*: a left-minimal subplan which structurally matches the right-hand side of some rule. To do this, we use standard shift-reduce techniques on each of the *threads* (linear subgraphs) of a plan, and then assemble the results of these parses into a parse of the plan as a whole. For example, given the library plan for absolute value shown in figure 12(a), we start by factoring it into its threads, `A-V-LEFT` and `A-V-RIGHT` (see figure 12(b)). We then parse the threads of our program plan using the `A-V-LEFT` and `A-V-RIGHT` rules: if they appear in two threads that share the appropriate nodes (as in figure 12(c)), we can combine them and reduce the `ABSOLUTE-VALUE` plan in its entirety.

Note that this algorithm, while it may be costly, has the necessary property of finding derivations even for subplans of the program plan. We will delay a detailed algorithmic description and complexity analysis until this work is completed.

5.3.4. Checking Logical Constraints

Once a structural handle has been found, we must find a rule in its structural class whose right-hand side logical constraints are satisfied by those of the handle. Unfortunately, as mentioned above, the handle's logical constraints may not be explicit in the input plan. For example, the input to `ABSOLUTE-VALUE` must be a number; in the code:

```
(LET ((X (- (LENGTH LIST1) (LENGTH LIST2))))
  (COND ((> X 0) X)
        (T (- X))))
```

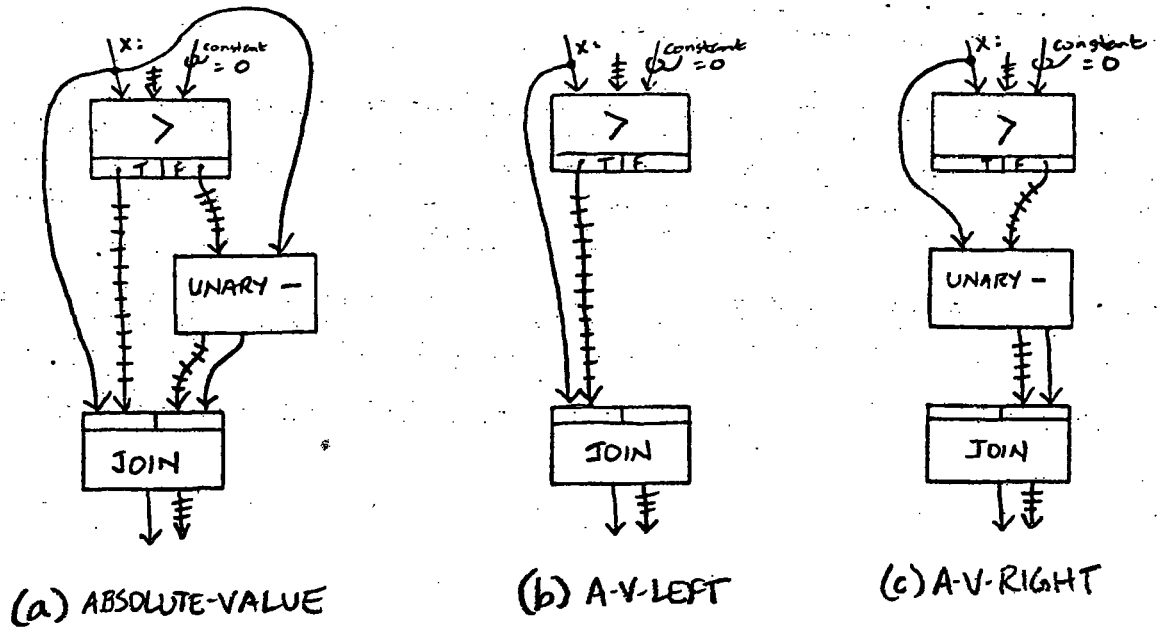
it is only library knowledge of the `LENGTH` function that allows us to recognize this condition on `X`.

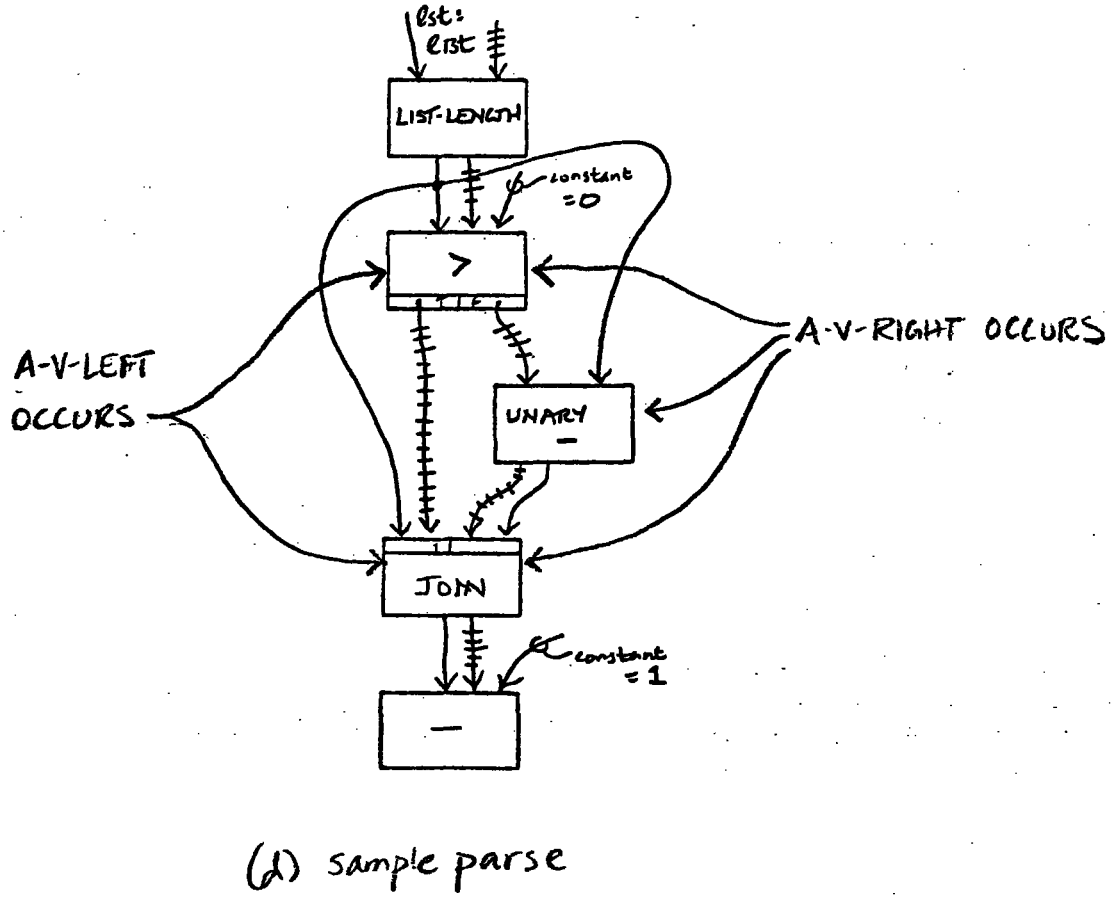
As this example shows, in order to complete the process of logical constraint checking, we will have to propagate information about parts of the plan already parsed to those being parsed. Unfortunately, there may be cases in which this technique may not be sufficient to recover all the implicit logical constraints, and we may be forced to make assumptions about the applicability of rules. Both these processes—propagation and hypothesis formation—lie at the heart of the parsing procedure, and both require deduction using logical constraints. We first discuss the necessary deductive facilities, and then how these processes may be handled.

Deduction using Logical Constraints

Since we will be making deductions based on rule applications (such as that described above), and since our backtracking parser may undo such applications (and so invalidate deductions based on them), our system must facilitate the retraction of deductions when their hypotheses are invalidated. In addition, it must support deductions based upon equalities of terms; for example, that a predicate true about the output of some operation is also true of an input which is equal

Figure 12. Plans for ABSOLUTE-VALUE, its chains, and a sample parse





to that output. (This is the nature of the deduction used in the absolute value example above.)

The construction of such a system is well beyond the scope of this research, so it is well that some already exist. We will be using a Reasoning Utility Package (RUP) developed by McAllester [15] which satisfies these requirements.

Propagation of Information

Inspector's information-propagation process is based on the distinction made in the library between an entry's defining constraints and its *properties*—other predicates which follow from its use. For example, the plan ABSOLUTE-VALUE constrains its input to be a real, so that the ABSOLUTE-VALUE rule may not be reduced unless this constraint holds on the current handle. However, it is a *property* of the plan ABSOLUTE-VALUE that its output is positive, so that having reduced the ABSOLUTE-VALUE rule we can deduce this about its output in the resulting plan.

Propagation of information is thus performed conveniently at reduction time. Once a structural handle is found, its logical constraints are looked up in the RUP and used to find an applicable rule (if any). Once this rule is reduced, any properties associated with its left-hand side are then asserted in the RUP, with the rule reduction as justification. These properties are then available as constraints on successive handles, successfully propagating information derived from analyzed plan portions to others not yet parsed. (If the rule reduction is later retracted during backtracking, the RUP will automatically retract any deductions based upon it.)

Hypothesis Formation

In cases where the known logical properties of a handle do not satisfy the logical constraints on the right-hand side of any rule, it may be appropriate to assume the constraints necessary for some reduction. For example, to recognize in the code:

```
(DEFUN ABS (X)
  (COND ((> X 0) X)
        (T (- X))))
```

the plan for ABSOLUTE-VALUE, it may be necessary to make the assumption—fairly obvious from operation types—that the input is a real. Or, more subtly, in order to recognize the code

```
(DEFUN SORTED-MEMBER (INT LST)
  (COND ((OR (NULL LST)
             (> (CAR LST) INT))
         NIL)
        ((= (CAR LST) INT) T)
        (T (SORTED-MEMBER INT (CDR LIST)))))
```

as a list membership operation, it may be necessary to assume both that the input is an integer and that the input list is of integers sorted in increasing order.

It is our belief that efficient, intelligent hypothesis formation may be the key to recognizing large classes of programs.[12] One of our motivations in distinguishing structural and logical constraints is the hope that the structure of the handle may guide guessing about logical constraints. (For instance, the structure of the conditional in the sorted list membership operation above is characteristic of all priority queue operations.) Here, once again, the specialization relations in the library may be closely involved: useful assumptions may be those necessary to specialize an applicable rule to one of its desired specializations. In general, the question of how to make appropriate assumptions may only be answered via experimentation.

5.3.5. A Sample Parse

We are now ready to parse the SET-MEMBER operation derived above. We will show only the main path followed in the parse, ignoring most of the false starts. In addition, we will suppress many details of the parsing algorithm, leaving these unspecified until the work is completed.

Figure 13 shows the output of Waters's segmenting translator when run on the following code:

```
(DEFUN SET-MEMBER (E S)
  (COND ((NULL S) NIL)
        ((EQ E (CAR S)) T)
        (T (SET-MEMBER E (CDR S)))))
```

Notice that, although the program itself generates its return value of T or NIL at whatever recursion level it terminates, the translator has noticed that these data values are constants and moved them out of the recursive section. The resulting structure is that of a conditional whose test is the two-exit recursively implemented loop. It is this structure that Inspector takes as input.

The first structural handle Inspector isolates is the top-level conditional whose branches contain no actions. This is a structural match for the library plan ENFLAG+NIL (see figure 14), which embodies the LISP cliché in which a value is tested and the result flagged as NIL if the test fails, some other value otherwise.[13]

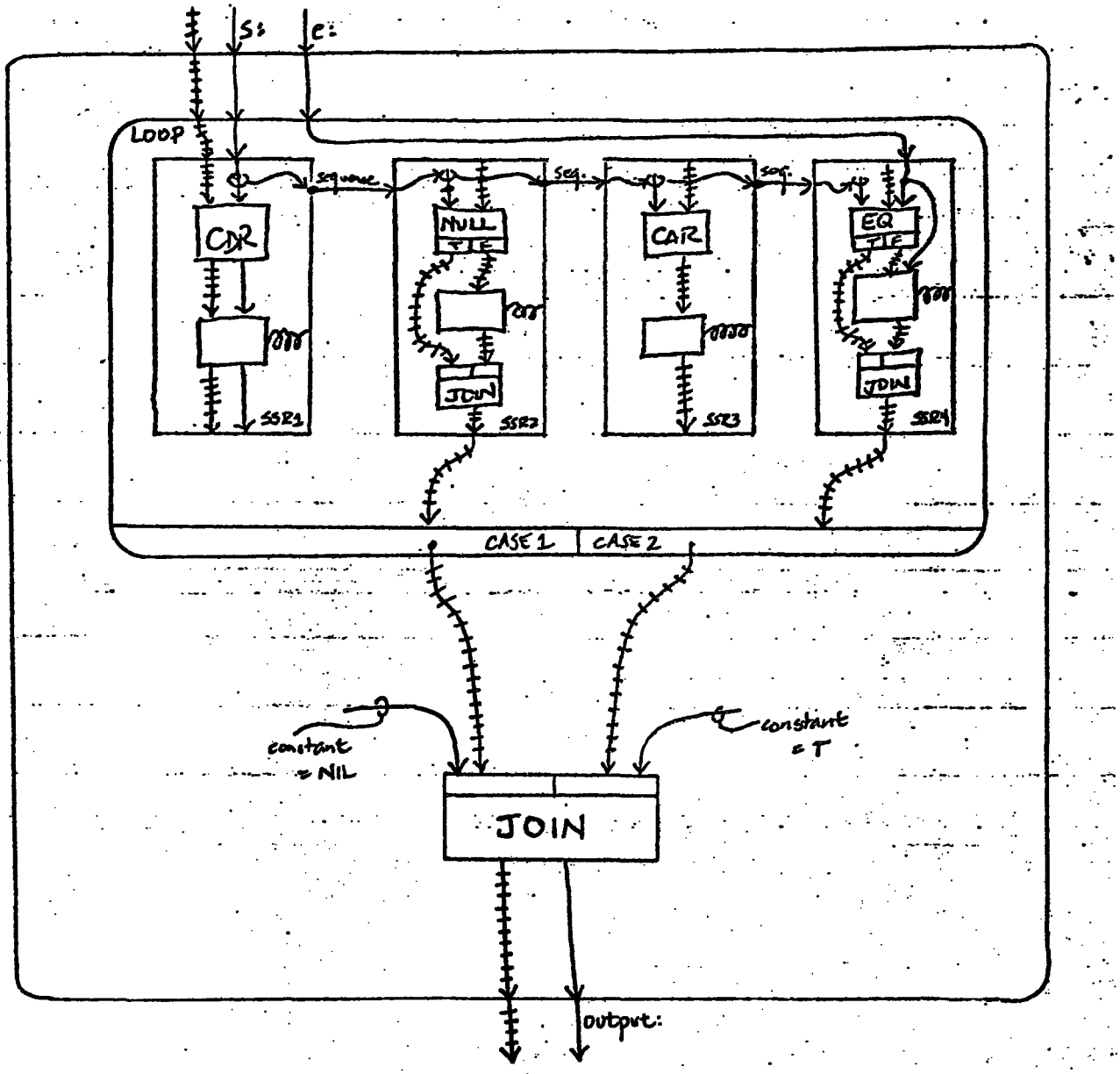
The logical constraints on ENFLAG+NIL are then checked, starting with the data flow constraints. In this case, the logical constraints on the input plan satisfy those of a specialization of ENFLAG+NIL, LISP-PREDICATE, whose success flag is T. (A specialization not matched is TESTED-VALUE+NIL, which returns the tested value itself after a successful test.) Since the constraints of LISP-PREDICATE are satisfied by the input plan, this reduction is made, yielding an initial analysis of the routine as a LISP predicate whose criterion is computed by the loop.

Inspector now continues with the analysis of the loop itself, shown in figure 15. Since loop plans contain both temporal and standard control flows, their control

[12] We believe, in fact, that this is one area which gives expert programmers an edge over novices in analysis.

[13] The flag is then usually used as the predicate value in a COND clause.

Figure 13. Waters's Segmented Output for SET-MEMBER



flow ordering is not well defined. Inspector's approach is to first eliminate the standard control flows by parsing each recursive segment individually. This results in a sequence of temporally abstract plans connected by temporal control flow. While we will not go into all the details here, the following gives a general idea of how this procedure works.

The first operation grouping surrounded by temporal control flows is the recursive application of CDR, labelled SSR1.[14] This is a structural match for the library plan LISP-LIST-GENERATOR, but fails to satsify its logical constraints as the input is not known to be a list. Rather than assuming this hypothesis, however,

[14] "SSR" stands for "single self-recursion".

Figure 14. Library Plan for ENFLAG+NIL

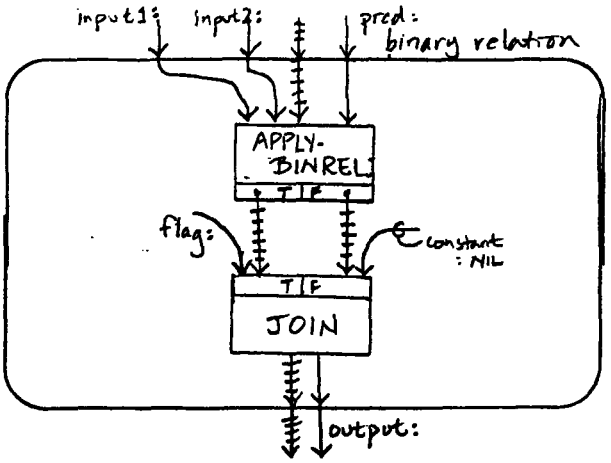
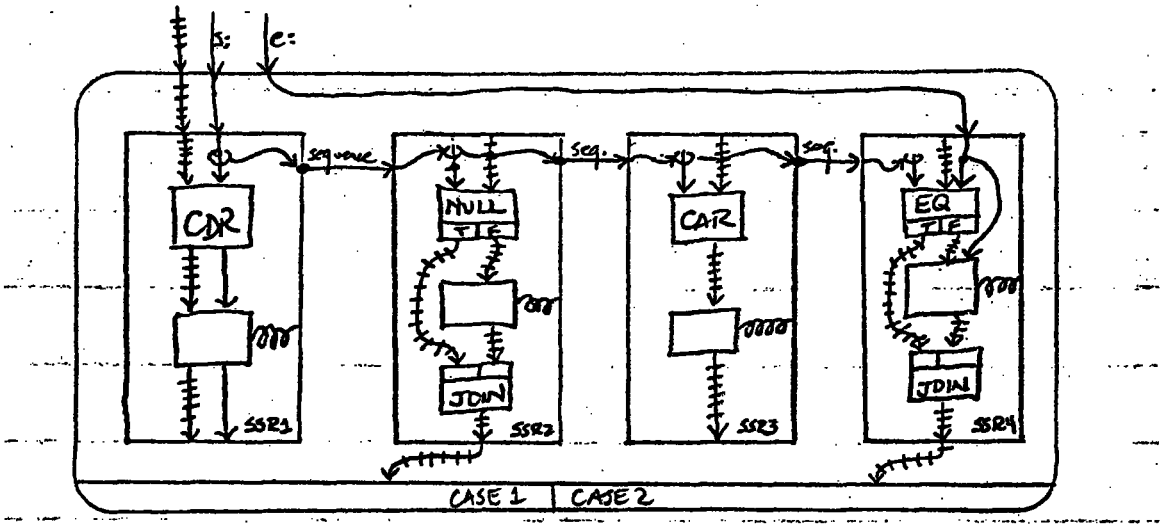


Figure 15. The loop from SET-MEMBER



Inspector is able to deduce it as a result of its built-in assumption that primitive language operations are correctly used in the input program.[15] Thus it assumes that the input to CDR is a cons-cell, and is able to apply the overlay CONS-CELL>LIST to satisfy the constraints on LISP-LIST-GENERATOR.

The analysis of the next two SSR segments proceeds similarly, identifying them as a terminator whose criterion is NULL and a map whose action is CAR. Finally, Inspector analyzes the final SSR in two stages, one in which the test is transformed into a unary predicate on the temporal input,[16] and one in which the resulting SSR is recognized as a terminator with the transformed criterion.

At this point, Inspector is ready to parse the resulting sequence of temporally abstract operations, shown in figure 16. Note that this sequence is not a true

[15]In future work on bug detection via analysis by inspection, we may wish to relax this assumption.

[16]This uses the fact that one of the arguments to the EQ test is the same in all iterations

Figure 16. Temporally Abstracted Loop from SET-MEMBER

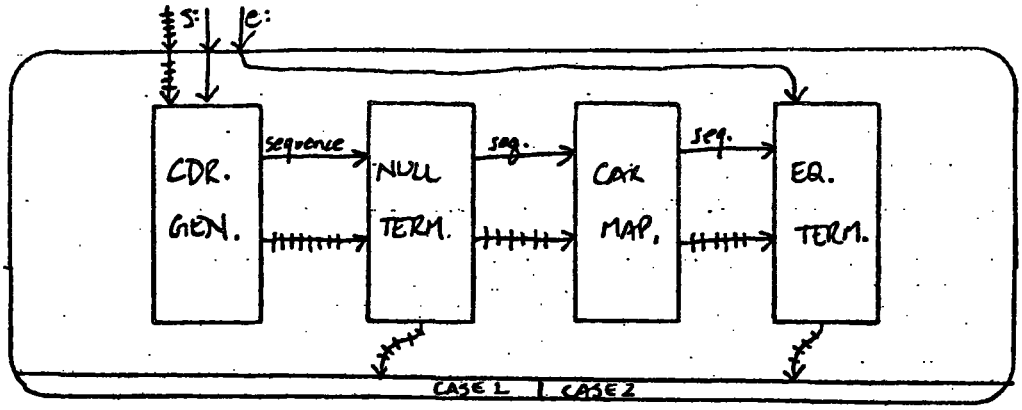
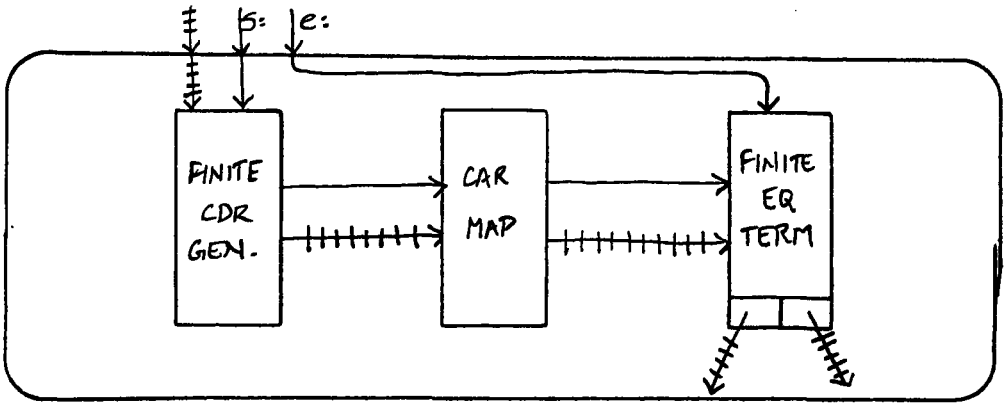


Figure 17. Transformed Loop from SET-MEMBER



composition, as the last two actions may not be executed every time the first two are. However, we consider this non-compositionality to be an optimization commonly used in loops to simplify the generation of a sequence of values.[17]. In this case, Inspector treats the generator/terminator pair as a *finite generator*. The knowledge that the sequence generated is probably finite is used to transform the final terminator into a *finite terminator* whose additional exit is used in case the sequence ends before the terminating condition is satisfied. (The result of these transformations is shown in figure 17.)

Inspector is now ready to resume parsing as usual on the transformed loop. This leads to two reductions, in which the first two operations are recognized as a LISP-LIST-ENUMERATOR which takes a LISP-style list as input and outputs its top-level elements as a temporal sequence, while the terminator is recognized as an EARLIEST operation which takes a temporal sequence and a predicate as input and outputs the first element of the sequence which satisfies the predicate (or none if there is none).

Inspector continues by reducing this two-operation plan to the plan for LIST-

[17]This is a view shared by others. See, for example, [1].

Figure 18. Set implemented as a hash table (partial)

```
(DEFUN HASH-TABLE-MEMBER (E S)
  (SET-MEMBER E (HASH E S)))          ;note that HASH returns a bucket

(DEFUN HASH-TABLE-ADD-NEW (E S)
  ;;precondition: (not (hash-table-member e s))
  (SET-ADD-NEW E (HASH E S))
  S)

(DEFUN HASH-TABLE-ADD (E S)
  (LET ((BKT (HASH E S)))
    (COND ((SET-MEMBER E BKT) S)
          (T (SET-ADD-NEW E BKT)
             S))))

(DEFUN HASH-TABLE-DELETE (E S)
  (SET-DELETE E (HASH E S)))
```

MEMBER,[18] analyzing the SET-MEMBER function as a lisp predicate on an object and a list which returns T or NIL as the object is or is not present in the top level of the list.

At this point, the loop being analyzed consists of a single LIST-MEMBER predicate, so the only applicable rules are abstaction rules, such as SET-MEMBER and SEQENCE-MEMBER. The obvious heuristic applied to the function name[19] allows Inspector to finish by reducing the LIST-MEMBER>SET-MEMBER rule, so that its complete analysis of SET-MEMBER is as a lisp predicate on an object and a set which returns T if and only if the object is in the set. Inspector's derivation tree for SET-MEMBER is exactly that developed in the previous section (see figure 2).

5.4. Optimizations

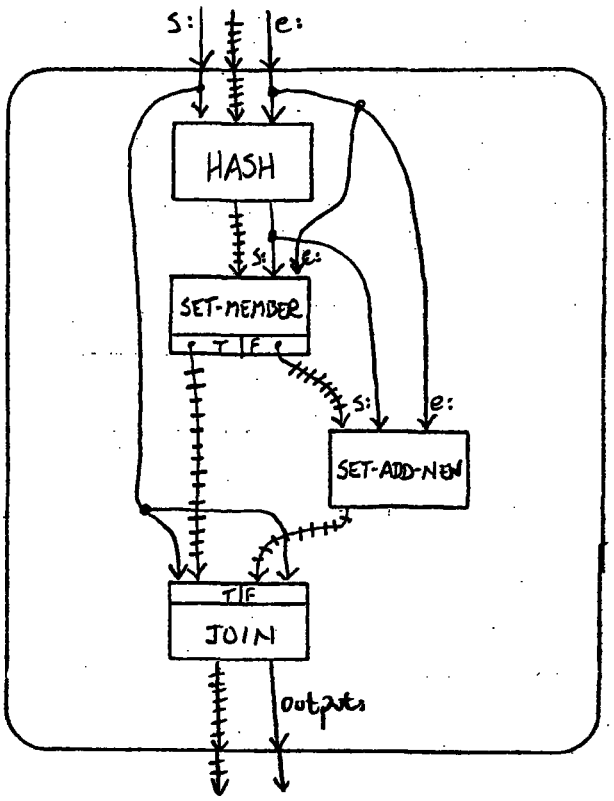
Any programmer could tell us immediately why what we have proposed won't often work: programs don't really grow in trees. Even if programs are designed in a tree-like fashion, various sub-trees will have identical nodes. Since separate coding of these nodes may lead to great inefficiency, standard engineering practice holds that they should be shared among the sub-trees in which they appear.

For example, suppose we wish to implement a set using a hash table. We can use our previous set implementation for the buckets of this table, adding the table superstructure as shown in figure 18. All goes as expected except for HASH-TABLE-ADD, where we notice that the expected code:

[18]This reduction depends on the predicate input to EARLIEST being of the proper form.

[19]Inspector will analyze function and variable names using a parser developed for this purpose by Chapman [5] in his work on test case maintenance.

Figure 19. Plan for HASH-TABLE-ADD




```
(DEFUN HASH-TABLE-ADD (E S)
  (COND ((HASH-TABLE-MEMBER E S) S)
        (T (HASH-TABLE-ADD-NEW E S))))
```

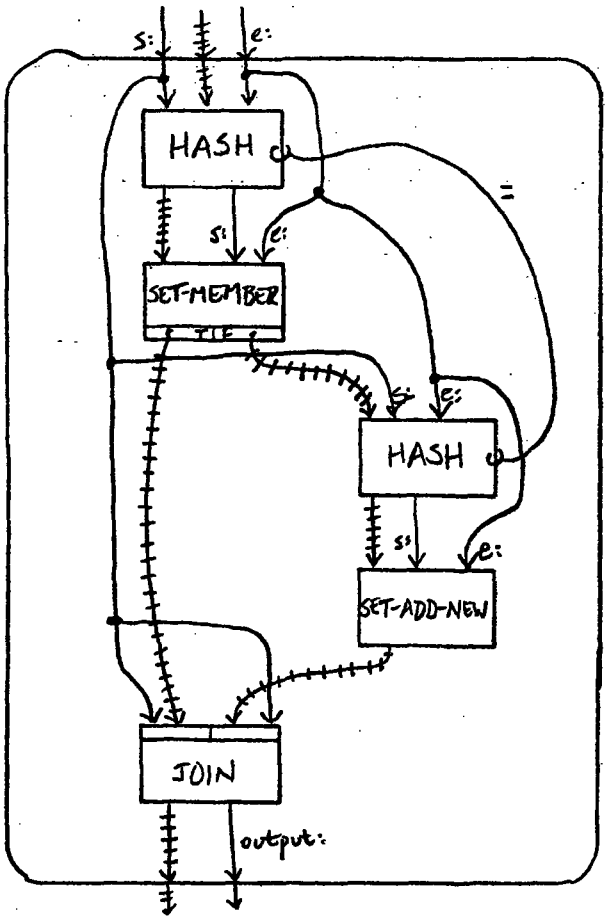
may hash the added element *twice* (once each in HASH-TABLE-MEMBER and HASH-TABLE-ADD-NEW). This leads to the optimized version used in figure 18, where the hash is remembered and the SET- routines called directly.[20]

This type of *sharing for optimization* is both ubiquitious and desirable, so that not allowing it in our parsable programs would restrict them unreasonably. To see how our strategy may accomodate it, we must consider its effects on the derived plans.

A plan for HASH-TABLE-ADD is shown in figure 19. Note that both the plan for HASH-TABLE-ADD-NEW and that for HASH-TABLE-MEMBER are subplans of this plan, reflecting its derivation as the merge of the two. Since our plan parsing algorithm is sensitive to partial parses, both of these plans will be recognized, and we can use their overlap as a key that sharing has occurred. We can then force the intuitively correct derivation by copying the shared plan elements (in this case the HASH) and parsing as usual. In this case (see figure 20), the resulting derivation for HASH-TABLE-CHECKING becomes just that for the unoptimized version shown above; the fact that an optimization was used may be retained by annotating the derivation tree.

[20]Note that this would not be considered a good design unless the hash is expensive. In the absence of other considerations, programmers really do prefer pure trees!

Figure 20. Plan for HASH-TABLE-ADD with copied hash function



In summary, we define a *well-structured program* as one whose implementation structure is context-free except for sharing induced by optimization. Well-structured programs are those for which our recognition algorithm will be effective. Their context-free implementation structure allows recognition by parsing, in which shared program segments (indicated by ambiguities in the parse) are unshared through duplication.

5.5. Success Criteria

Our proposed success criterion is purely operational. Appendix A contains a number of programs, all of which may be derived using Rich's cliché library. Inspector should be considered a success when it is able to parse all of these programs.

Note that we would eventually like to have Inspector recover derivations for clichéd fragments of programs even if the entire program is not derivable from the cliché library. While we believe we have taken an approach which should make this possible, we have not made this a criterion for Inspector's success because we

believe that doing it efficiently in the general case may present problems beyond the scope of a master's thesis.

6. Review of the Literature

This work has roots in a number of research areas without belonging entirely to any one of them. It draws on previous work in program analysis and synthesis, pattern recognition, and engineering problem solving; but extends and adapts this work with an essentially new perspective. We here examine briefly, area by area, these historical connections.

Engineering Problem Solving

Engineering problem solving (EPS) is the study of the techniques used by human engineers in their daily work. The goals of EPS research are twofold: first, the construction of design tools (such as the PA) whose utility is enhanced due to their understanding and use of these techniques; and second, the design of educational programs which teach such techniques explicitly, rather than through example.

The work proposed here is part of the EPS project at MIT. Early work in this project [25] identified a problem solving approach now known as AID,[21] which may be summarized as follows:[22]

(i) Simplify the problem description by concentrating on just those features which are most important.

(ii) By inspection, find a standard implementation (cliché) which solves the problem.

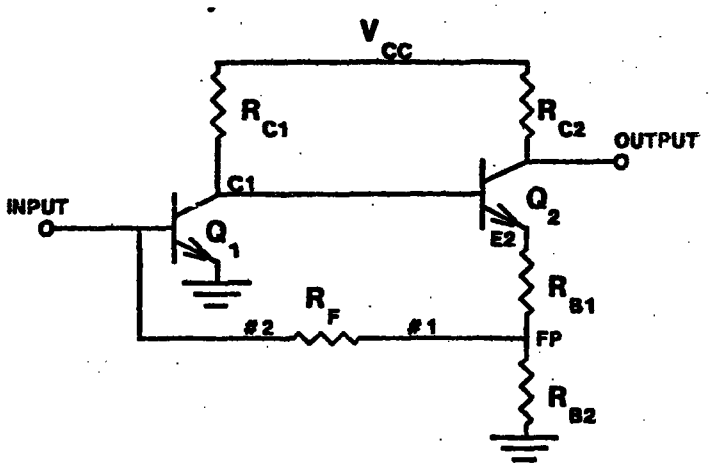
(iii) Debug the solution by recursively applying this technique to those parts of the problem specification not yet satisfied.

The AID paradigm, which emphasizes the use of inspection methods, has been applied to electronic circuit analysis [EL 6] and synthesis [7], mechanical analysis [10], and programming [19]. The electronic circuit work is continuing in the DPL project [26], and the programming work in the PA project. The work proposed here is thus properly viewed as research into the use of AID in programming.

The prior EPS research most relevant to this work is that of DeKleer [6], in which he tried to automatically recover the rough functional specifications of an analog circuit by examining its circuit diagram. For example, the circuit:

[21] AID is an acronym for "Abstraction, Inspection, and Debugging." The method was originally entitled "problem solving by debugging almost-right plans."

[22] This description is a summary of that in [22].



was correctly identified as a two-stage amplifier.

DeKleer used a two-stage analysis strategy which bears a strong resemblance to ours. He first used a form of symbolic evaluation which he called "causal analysis" to move from the circuit diagram representation to one in which changes in electrical quantities and their cause/effect interrelationships were made explicit. He then matched these "causal graphs" against those characteristic of common circuit structures, such as amplifiers and power supplies. These clichéd graphs were annotated so that the functions of individual components (such as a shunt resistor) in the original circuit could be recovered.

In our work, plans take the place of deKleer's causal graphs. Waters's analyser performs the symbolic evaluation needed to recover a plan from the "circuit diagram" provided by a program's code. Inspector will do the matching of program and library plans done in deKleer's second stage.

An instructive difference between our work and deKleer's lies in the relative complexities of the two stages. In deKleer's work, most of the complexity was in the causal reasoner, which had to correctly model the behavior of circuit components in various situations. Once the causal graph was obtained, the pattern matching stage presented few difficulties, primarily because (i) the library of clichéd graphs was small, as was each entry in the library, and (ii) the logical relations between causal components (e.g., voltage or current variation) were derivable directly from the circuit diagram.

In our work, while the initial symbolic evaluation is still difficult (especially in the correct recognition of side effects and temporal compositions), the pattern matching stage has grown enormously in complexity. Each library entry is still fairly small, but there are hundreds of them (as opposed to deKleer's few dozen) at many levels of abstraction. We must also be able to recognize patterns occuring as fragments of plans, whereas deKleer always considered the circuit as a whole. Finally, almost none of the logical constraints needed for recognition are made explicit in the output of the symbolic evaluation, so guessing and verification strategies play a very important role.

The commonality of our work and deKleer's, however, is also instructive. Both deKleer and the Programmer's Apprentice group started by formalizing some of the intuitions engineers (in these cases circuit designers and programmers) have about their domain. These formalizations led to domain representations amenable to the types of reasoning used by engineers, and translation methods between these intuitive representations and standard ones used in the fields. The two stage process characteristic of both systems arises from the desire to separate the reasoning used to recover useful information left implicit in the standard representation used in the field (*e.g.*, blueprints, circuit diagrams) from that needed to manipulate this information so as to perform engineering tasks.

Pattern Recognition

The initial recognition strategy outlined in the previous section was motivated by work done by Winston [29]. As our ideas developed, however, it became clear that most of the traditional pattern-matching work would not be as relevant as we first believed. At this point, the only point of contact between this research and pattern-matching research are the web grammars.

Web grammars were initially intended as pattern generators used in feature detection work and their use in our application is, as far as we know, entirely new. Prior research using web grammars has concentrated primarily on generative questions, such as grammar equivalence and characteristics of the generated language. Thus, little work has been done on parsing grammar-generated webs (labelled graphs), and we have not been able to find relevant material.

Program Synthesis and Verification

The synthesis ideas presented in chapter 5 resemble much of the work done on program synthesis via transformations. With regard to Inspector's analysis methods, however, only a few of the transformational ideas are relevant, primarily some of the work on loop transformations (see, *e.g.*, [2]). This is because most current transformation systems are pattern directed and driven only in the synthetic direction, so that the final program representations do not preserve the modularity of the patterns which triggered the transformations.

More useful in analysis is some of the verification literature, especially that concerned with hypothesis generation (*e.g.*, [11]). We expect that, as Inspector needs more sophisticated hypothesis generation schemes (in the constraint checking phase of the parse), it will use some of the presented there.

Appendix

```
(DEFUN HASH-TABLE-MEMBER (E S)
  (SET-MEMBER E (HASH E S)))          ;note that HASH returns a bucket

(DEFUN HASH-TABLE-ADD-NEW (E S)
  ;;precondition: (not (hash-table-member e s))
  (SET-ADD-NEW E (HASH E S))
  S)

(DEFUN HASH-TABLE-ADD (E S)
  (LET ((BKT (HASH E S)))
    (COND ((SET-MEMBER E BKT) S)
          (T (SET-ADD-NEW E BKT)
             S))))

(DEFUN HASH-TABLE-DELETE (E S)
  (SET-DELETE E (HASH E S)))

(DEFUN SET-CREATE ()
  (LIST NIL))

(DEFUN SET-MEMBER (E S)
  (MEMBER E (CDR S)))

(DEFUN SET-ADD-NEW (E S)
  ;;precondition: (not (set-member e s))
  (RPLACD S (CONS E (CDR S)))
  S)

(DEFUN SET-ADD (E S)
  (COND ((SET-MEMBER E S) S)
        (T (SET-ADD-NEW E S))))

(DEFUN SET-DELETE (E S)
  (PROG (THIS NEXT)
        (SETQ THIS S)
   LP   (SETQ NEXT (CDR THIS))
        (COND ((EQUAL E (CAR NEXT))
               (RPLACD THIS (CDR NEXT))
               (RETURN S)))
        (SETQ THIS (CDR THIS))
        (GO LP)))
```

Bibliography

[1] R. Balzer, "Transformational Implementation: An Example", *IEEE Trans. on Software Eng.*, Vol. 7, No. 1, January, 1981.

[2] S.K. Basu and J. Misra, "Proving Loop Programs", *IEEE Trans. on Software Eng.*, Vol. 1, No. 1, pp. 76-86, March, 1975.

[3] F.L. Bauer *et al*, "Towards a wide spectrum language to support program specification and program development," *ACM SIGPLAN Notices*, Vol. 13, No. 12, pp. 15-24, 1978.

[4] R. Cartwright and J. McCarthy, "First Order Programming Logic", *6th Annual ACM Symposium of Principles of Programming Languages*, 1979, pp. 68-80, 1979.

[5] D. Chapman, "A Program Testing Assistant", MIT/AIM-651, November, 1981.

[6] J. de Kleer, "Causal and Teleological Reasoning in Circuit Recognition", (Ph.D. Thesis), MIT/AI/TR-529, September, 1979.

[7] J. deKleer and G.J. Sussman, "Propagation of Constraints Applied to Circuit Synthesis", *International Journal of Circuit Theory and Applications*, Vol. 8, No. 2, April 1980.

[8] J.B. Dennis, "First Version of a Data Flow Procedure Language", *Proc. of Symposium on Programming*, Institut de Programmation, U. of Paris, April 1974, pp. 241-271.

[9] G. Faust, "Semiautomatic Translation of COBOL into HIBOL", (M.S. Thesis), MIT/LCS/TR-256, March, 1981.

[10] M.J. Freiling, "The Use of a Hierarchical Representation in the Understanding of Mechanical Systems", Ph.D. Thesis, Mathematics Dept., M.I.T., 1977.

[11] S.L. Gerhart, "Knowledge About Programs: A Model and Case Study", in *Proc. of Int. Conf. on Reliable Software*, June 1975, pp. 88-95.

[12] K. Jensen and N. Wirth, "PASCAL User Manual and Report", Springer-Verlag, New York, 1976.

[13] Z. Manna and R. Waldinger, "Problematic Features of Programming Languages: A Situational-Calculus Approach; Part I: Assignment Statements", Stanford Univ., Weizmann Institute and the Artificial Intelligence Center, August, 1980.

[14] D. Marr, "Artificial Intelligence: A Personal View", *Artificial Intelligence*, Volume 9, 1977, pp. 37-48.

[15] D.A. McAllester, "The Use of Equality in Deduction and Knowledge Representation", MIT/AI/TR-550, January, 1980.

[16] D.A. McAllester, "An Outlook on Truth Maintenance", MIT/AIM-551, August, 1980.

[17] U.G. Montanari, "Separable Graphs, Planar Graphs, and Web Grammars", *Information and Control* 16:3, March, 1970, pp. 243-267.

[18] J.L. Pfaltz and A. Rosenfeld, "Web Grammars", *Proc. Int. Joint Conf. on Artificial Intelligence*, Washington, D.C., 1969, pp. 609-619.

[19] C. Rich and H.E. Shrobe, "Initial Report On A LISP Programmer's Apprentice", (M.S. Thesis), MIT/AI/TR-354, December 1976.

[20] C. Rich, H.E. Shrobe, R.C. Waters, G.J. Sussman, and C.E. Hewitt, "Programming Viewed as an Engineering Activity", (NSF Proposal), MIT/AIM-459, January, 1978.

[21] C. Rich and H. Shrobe, "Initial Report on A Lisp Programmer's Apprentice", *IEEE Trans. on Software Eng.*, Vol. 4, No. 5, November, 1978.

[22] C. Rich, H. Shrobe and R. Waters, "Computer Aided Evolutionary Design for Software Engineering", (NSF Proposal), MIT/AIM-506, January, 1979.

[23] C. Rich, "Inspection Methods in Programming", MIT/AI/TR-604, (Ph.D. thesis), December, 1980.

[24] C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice", *Proc. of 7th Int. Joint Conf. on Artificial Intelligence*, Vancouver, Canada, August, 1981.

[25] G.J. Sussman, *A Computer Model of Skill Acquisition*, (Ph.D. Thesis), American Elsevier, New York, 1975.

[26] G.J. Sussman, J. Holloway, and T. Knight, "Computer Aided Evolutionary Design for Digital Integrated Systems", MIT/AIM-526, May, 1979.

[27] R.C. Waters, "Automatic Analysis of the Logical Structure of Programs", MIT/AI/TR-492, (Ph.D. Thesis), December, 1978.

[28] R.C. Waters, "A Method for Analyzing Loop Programs", *IEEE Trans. on Software Eng.*, Vol. SE-5, No. 3, May 1979, pp. 237-247.

[29] P. Winston, "Learning and Reasoning by Analogy", *Comm. of the ACM*, Vol. 23, No. 12, December, 1980.