Technical Report 354

Initial Report on A LISP Programmer's Apprentice

Charles Rich and Howard E. Shrobe

MIT Artificial Intelligence Laboratory

This blank page was inserted to preserve pagination.

INITIAL REPORT ON

A LISP PROGRAMMER'S APPRENTICE

by

Charles Rich and Howard E. Shrobe

Massachusetts Institute of Technology

December 1976

Revised version of a dissertation submitted to the Department of Electrical Engineering on August 15, 1975 in partial fulfillment of the requirements for the degree of Master of Science.

This report describes research done at the Artificial Intelligence Laboratory of 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 N00014-75-C-0643. This research was also supported in part under Naval Research Contract N00014-75-C-0522.

Abstract

This is an initial report on the design and partial implementation of a LISP programmer's apprentice, an interactive programming system to be used by an expert programmer in the design, coding, and maintenance of large, complex programs.

The conceptual basis of the system lies in three forms of program description: (i) definition of structured data objects, their parts, properties, and relations between them, (ii) input-output specification of the behavior of program segments (specs), and (iii) a hierachical representation of the internal structure of programs (plans). The major theoretical work reported here is a representation for program plans which includes data flow, control flow, and also goal-subgoal, prerequisite, and other dependency relationships between the segments of a program. Plans are utilized in the apprentice both for describing particular programs, and also in the compilation of a knowledge base of more abstract knowledge about programming, such as the concept of a loop and its various specializations, such as search loops and enumeration loops.

As a practical matter, we have designed a system with three major performance modules: a deductive system, a surface flow analyzer for LISP code, and a program recognition module. The deductive system is based on the technique of symbolic evaluation using a situational data base, and can be used to verify the correctness of plans. The surface flow analyzer is the only programming language-specific component of the system; given a particular LISP program written by the programmer, it abstracts a superficial plan which represents the control flow and data flow present in the LISP code. In a program verification scenario, this superficial plan is acted upon by the recognition component which, using the knowledge base and aided by the programmer's written comments, builds a deeper and more complete plan description of the program, which can then be verified by the deductive system. Initial implementations of the deductive system and the LISP flow analysis module are described; the recognition component has not yet been implemented.

As compared to automatic programming research, the programmer's apprentice emphasizes a cooperative relationship between the computer and the human programmer, wherein the computer provides primarily support facilities. This is seen as a more realistic, interim solution to current software problems; as techniques for automated program synthesis are perfected, these can be incorporated into the apprentice environment in an evolutionary manner, with the computer eventually taking over more and more of the programming responsibility. The present work also entails an extension of symbolic evaluation and verification techniques to programs which have side effects on complex data structures.

Acknowledgements

We would like to express our sincere gratitude to our thesis advisors, Professors Gerry Sussman and Carl Hewitt, and to all the graduate students of the M.I.T. Artificial Intelligence Lab, from whom we have benefited greatly in the exchange of ideas and by their intellectual and moral support. In particular we wish to acknowledge here our many discussions on the topics of this report with Dick Waters, Johan DeKleer, Richard Steiger, Robert Moore, Mike Genesreth, Keith Nishihara, and Mark Miller.

We would also like to thank Mitch Marcus and Will Clinger for their careful help in proofreading this document, and Karen Prendergast for preparing the figures.

TABLE OF CONTENTS

3. The Deductive System

4. Surface Analysis Of LISP Programs

5. Proposed Apprentice Environment

6. Relationship To Other Work

TABLE OF FIGURES

CHAPTER ONE

INTRODUCTION

1.1 The Complexity Barrier

During the past decade the power of computational facilities has increased by several orders of magnitude. The transition from tab equipment to modern day computer utilites has taken little more than two decades. Moreover, we are faced with the prospect that current hardware will soon be superseded by technologies capable of housing in a desk drawer computers more powerful than those of the present generation.

Software development has also proceeded at an amazing rate, taking about two decades for the transition from the first FORTRAN compiler to structured languages, optimitizing compilers, sophisticated data base systems, complex operating systems, and the like. In the artificial intelligence research community, similar progress has taken place from the batch LISP 1.5 to high powered interactive dialects such as MACLISP and INTERLISP. In addition, new specialized A.I. languages have been developed, such as PLANNER, CONNIVER, and QA4.

Each advance in computing hardware or in the power of programming languages has spawned a new generation of larger and more complex programs. Software engineering is now caught on the horns of a dilemma. The sheer size of modern software systems seemingly dictates that they be produced by a group of people, in which responsibilities for modules are parcelled out. However, the interactions between these modules are frequently so diverse that they prevent the effective coordination of labor between individuals who cannot each see the entire picture. The result has been that software is both notoriously late and unreliable. The alternative to division of labor, i.e. that design and even coding be accomplished by a single individual, is no better since one person's capacity for keeping track of a large number of interactions is also quite limited. Thus the problem is not simply due to the size of programs, but also the fact that as size increases, the number of relationships between modules grows much more quickly.

The economics of large scale software production bring about additional difficulties. The elite expert programmer who crafted a system and stayed with it for years has been replaced by armies of young college graduates who change jobs frequently. Thus any particular module in a large system may be modified and remodified by many successive generations of programmers, each time increasing the likelihood that a bug will be introduced because of the current programmer's distance from the original design.

Terry Winograd has labelled this cluster of phenomena the "complexity barrier" <Winograd, 1973>. He experienced it working in the research environment of the M.I.T. Artificial Intelligence Laboratory. Programs such as his natural language understanding system, SHRDLU, grew larger and more complex than could be handled by either an individual or a large group. Furthermore, it was clear that the current generation of A.I. programs fell far short of the complexity and size necessary to achieve the levels of performance to which the field aspires. Thus Winograd and others have concluded that overcoming this complexity barrier is crucial to continued research in A.I.

1.2 A Solution: The Programmer's Apprentice

The purpose of this report is to describe our concept of a programmer's apprentice. The programmer's apprentice approach to the software problems described above is an instance of a general orientation towards symbiotic man-machine interactions, in which responsibilities are divided between the two according to their respective strengths and weaknesses. Thus our immediate goal is not to replace programmers, but rather to design an "intelligent" computer system which can help an already competent programmer. Chapter Six is a review of other similar and alternative approaches to software problems.

Our concept of a programmer's apprentice is based largely on the metaphor of a relationship between the senior expert programmer and a very junior colleague in a joint project. The senior partner has the accumulated experience, the sense of aesthetics, and the high level planning that are the basis of quality programming. The apprentice helps him mostly by keeping track of the mundane but crucial low level details that guarantee a program will work. A computer is well suited to keeping track of a large number of precisely specified details, which people cannot do very well. People on the other hand, have a unique ability to solve problems by the application of vague but powerful global reasoning strategies.

In order for the expert programmer and the apprentice to work together, they must be able to communicate effectively. We believe that communication in this situation must rest not only on the use of a mutually understood syntax, but also on a significant base of shared knowledge. The apprentice must know what a loop is, what a list is, that loops have tests, that lists have CAR's , and so on. Some of this knowledge is basic to all programming, and should therefore be "built-in" to the apprentice. Other knowledge pertains only to a particular program, and is therefore built up during interaction with the programmer. Chapter Two is concerned with how to represent this shared knowledge.

A Unified Environment

In a simple view, programming activity can be divided into three phases: design, coding, and maintenance. As we will illustrate in the following scenarios, our apprentice is aimed at helping the programmer in all three phases.

Design is the most abstract activity. Given specifications which tell what the program is supposed to do, a programmer must decide how to do it. Sometimes, the answer to "how" is to use a well-known standard plan, as for example using the Newton-Raphson method of successive approximation for finding the roots of an equation. Other times, a programmer is more creative, putting together a number of basic operations in a new way in order to achieve the desired result. In both cases, the apprentice can help the programmer by recording the design plan, and if possible verifying that the plan indeed satisfies the specifications.

Once the programmer has settled on a design, or part of a design, he usually starts the coding of the program. Here again, the apprentice can be of assistance. If the coding is done in an orderly fashion, and adequately commented, an apprentice should be able to verify that the code is a correct implementation of the programmer's design intentions. If there are incompatibilites between the code and the design, these can be pointed out to the programmer by specific reference to the design.

Finally, even after a program has been correctly designed and coded with respect to a particular set of specifications, there is continuing maintenance. Sometimes the specifications change, sometimes the programmer changes his mind on how he wants to achieve them. In any case, the apprentice can help the programmer (particularly a replacement person who was not involved in the original programming) by recalling and explaining the design decisions and particular knowledge that was originally used. Also, the apprentice can apply its reasoning abilities to help point out potential inconsistencies caused by changes to either the specifications, the design, or the code.

1.3 Scenarios

To complete our explanation of the kind of programmer's apprentice we are trying to implement, we now give two excerpts from an imagined interaction between a LISP programmer's apprentice (PA) and a human programmer (Pgmr). In these sessions, the programmer is working on a storage and retrieval system for keyed records, implemented as a hash table. This particular programming task is one which we will use repeatedly for illustrative examples throughout this report. The programming errors and modifications that arise in the scenarios are realistic in the sense that they actually occurred while the authors were programming a similar task.

In this presentation, all dialogue is in English. This is primarily for ease of reading. Although we feel that the system we are designing will ultimately have adequate knowledge to support communication in natural language, this has not been an immediate concern thus far. Nothing in this report will deal with the question of natural language communication.

First Scenario: Design and Coding

> [The scenario joins the programmer and apprentice in the middle of the design of a hash table system. We assume the programmer has already explained the structure of the main data objects: the hash table is an array, whose buckets are lists; and each entry is a CONS cell whose CAR is the key. Furthermore, the lookup and hashing routines have already been coded.]

Pgmr: Here is the design for the insert routine. Inputs are a key and a data object. Build an entry out of them. The key is hashed to get the correct bucket. The entry is then added to the bucket to create the new bucket. Finally, the new bucket is stored in the slot of the array indexed by hashing the key.

PA: OK. By the way, you only have to hash the key once.

> [The apprentice uses the programmer's outline of the design to build for itself a representation of the program's intended data flow. This representation, called the

deep plan, forms the basis of the apprentices further understanding of the program. Section 5.1 will discuss how plans may be built up interactively. Notice that the plan is more modular than the programmer's description, since in building the plan a redundant computation was exposed.]

Pgmr: Here's the code:

```
(DEFINE INSERT (KEY DATA1 DATA2)
        (STORE (HASH KEY)
               (CONS (CONS DATA1 DATA2) KEY))))
```

[Mnemonic identifiers are a convenient form of commentary. In this case, the programmer's use of KEY, DATA1, and DATA2 facilitate recognition of the relationship between the surface structure of the code and the underlying plan. The idea of plan recognition is elaborated in Section 5.2.]

PA: Sorry, but I see a few problems:

1. You are building an entry here by doing

```
(CONS (CONS DATA1 DATA2) KEY)
```

But in the lookup routine, the key is the CAR of an entry.

2. The first argument to STORE must be a subscripted array reference, but HASH is not an array. HASH is a function which returns a bucket of the hash table.

[The apprentice has already compiled the internal dependencies of the hashing scheme. For example, the entry builder of insert must agree with the key-extractor of the lookup routine. Also, it has built-in the syntax and semantic restrictions of the basic LISP functions, such as STORE .]

Pgmr: To fix the first problem, reverse the order of the arguments to the outer CONS.

[The apprentice diagnoses an inconsistency. In general, it is the responsibility of the programmer to supply a correction.]

PA: That fixes it.

```
(CONS KEY (CONS DATA1 DATA2))
```

Pgmr: Let me see the second problem again.

PA: In the indicated form, the first argument to STORE must be a subscripted array reference:

```
(STORE (HASH KEY)
       (CONS KEY (CONS DATA1 DATA2)))
```

Pgmr: What is the name of the hash table array?

PA: TBL

Pgmr: Change the first argument of STORE to

```
(TBL (HASH KEY))
```

PA: That won't work either. TBL is an array. The argument to TBL must be an non-negative integer; however HASH returns a bucket, which is a list.

[The apprentice has checked for arguments being of the right type. In this case, TBL and HASH have already been defined. It checks to see if the one produces the right kind of data for the other. Since they don't, it complains.]

Pgmr: Can I change HASH so that it will return the integer index of the bucket rather than the bucket itself?

PA: No. In the lookup routine, HASH is required to return a list which is searched for the desired entry.

[The apprentice has searched through its internal representation of the program structure to see if the suggested change creates an inconsistency elsewhere.]

Pgmr: OK, let me see the code for HASH .

PA:

```
(DEFINE HASH (KEY)
        (TBL
          (ABS (REMAINDER (MAKNUM KEY) TBLSIZE)))))
```

Pgmr: Rename this function BKT-HASH. Also make a function called HASH out of the inner form. HASH will now take a key and return the index of the appropriate bucket.

PA: Is this what you want?

```
(DEFINE HASH (KEY)
        (ABS (REMAINDER (MAKNUM KEY) TBLSIZE)))

(DEFINE BKT-HASH (KEY)
        (TBL
          (ABS (REMAINDER (MAKNUM KEY) TBLSIZE))))
```

[The apprentice performs a simple edit under instructions from the programmer.]

Pgmr: Yes. Now change the inner form in BKT-HASH to call HASH with the key as argument.

```
(DEFINE BKT-HASH (KEY)
        (TBL (HASH KEY)))
```

[This time the programmer recognizes a redundant computation. Now the apprentice reminds the programmer that this change must be propagated to several other routines.]

PA: Should all previous references to HASH be changed to BKT-HASH ?

Pgmr: Yes, except for the one in INSERT. That one should be HASH.

```
(DEFINE INSERT (KEY DATA1 DATA2)
        (STORE (TBL (HASH KEY))
               (CONS KEY (CONS DATA1 DATA2))))
```

PA: There is still a bug. After you do the STORE in INSERT, the array slot won't contain a bucket, but rather an entry. The old entries in that bucket will not be in the table afterwards.

[In attempting to verify that the insert routine satisfies its specifications, it finds a violation, i.e. it is not true that all entries in the table before are in the table after INSERT. Furthermore, there was an inconsistency detected between the types of data objects.]

Pgmr: Change the second argument of STORE to:

```
(CONS (CONS KEY (CONS DATA1 DATA2))
      (BKT-HASH KEY))
```

PA: OK, INSERT now satisfies the specifications I have been given. However, you still have a redundant HASH computation.

Pgmr: Leave it.

.
.
.

[The session continues until the programmer has finished and the apprentice is satisfied there are no problems. The program, together with the knowledge about it are filed away for future use.]

Second Scenario: Maintenance

Pgmr: I'd like to work on the hashing system again.

PA: OK.

Pgmr: What is the part structure of an entry?

```
(KEY . (DATA1 . DATA2 ))
```

[The apprentice displays a "typical" entry in the hash table. This knowledge was recorded during the design and coding of the program.]

Pgmr: What are the restrictions on keys ?

PA: None.

Pgmr: Let an entry whose key is an atom be called a "simple entry".

PA: Simple entries are a subset of entries. The key of a simple entry must be an atom.

[The new information is stored in its knowledge base.]

Pgmr: Where are entries referenced ?

PA: In the lookup and deletion routines, where the key part is checked, and in the insert routine, where the entry is built.

[This information is not found by a syntactic cross-reference on the code. For example, in the insert routine, there was no explicit named module that built an entry. Rather, the apprentice's internal representation of the program includes concepts like building an entry and extracting its parts, and pointers from these concepts to the code that realizes them.]

1.4 Outline of Our Work

Our first step towards building a programmer's apprentice was to look at a variety of programs, together with the written annotation that accompanied them and verbal explanations by programmers of what the programs did and how they did it. This led us to believe that there really is a common conceptual level underlying the various capabilities such as explanation, verification, and debugging exhibited by the apprentice in the preceding scenarios.

This belief is reflected in the order of presentation of this report. The purpose of Chapter Two, the first chapter in the body of the report, is to lay out our system's basic concepts and formalisms in a fairly definitional manner. Chapter Two also provides initial motivation for our descriptive theory by showing its application to the description of an already written program. Then in Chapters Three, Four, and Five we show how the same elements of program description are used for reasoning about programs, evolving designs, and documentation of programs.

The Domain of LISP Programs

The majority of our work on a programmer's apprentice is intended to be independent of particular programming languages. The basic features of data and control flow that we deal with differ only in their syntactic realizations in the major programming languages in common use. Nevertheless we have in fact been looking specifically only at LISP programs. Waters <1976> has proposed a system which works on FORTRAN programs in a similar way.

Our work thus far has also been biased towards a particular genre of programs written in LISP. Our favorite example program, which we use throughout this report, is the implementation of a simple hash table (see Figure 1). More generally, we have been interested in programs in which there is significant internal structure both in the data (such as the hash table) and the procedures that use it (such as lookup, insert, and delete). We have found the hash table example to be very productive so far, but to continue our research we need to more clearly define a class of programming problems we want our apprentice to initially address.

Pgmr: I want to make a design change in the lookup routine. If an entry is a simple entry, then the test for the desired entry should also check that the DATA1 part of the entry is equal to an additional input called PARENT.

PA: Since the only distinguishing feature of a simple entry is that its key is an atom, I assume that the check for simple entries is a test that the key is an atom.

> [The apprentice makes this modification to the code and checks it has no repercussions elsewhere. Notice that the apprentice explains the changes in terms of the design concepts, key, entry, etc., rather than the implementation in terms of car's and cdr's.]
>
> .
> .
> .
>
> [The programmer continues working on further modifications.]

```
(DECLARE (ARRAY TBL 500))                          ;hash table


;;; This is a hashing system in which buckets of the table are
;;; implemented as lists.  Entries in the table are CONS cells
;;; where the key part is the CAR.


(DEFINE LOOKUP (KEY)
       ;; returns entry with given key or nil
       (PROG (BKT)
             (SETQ BKT (TBL (HASH KEY)))           :fetch bucket
         LP  (OR BKT (RETURN NIL))                 ;failure, return nil
             (COND ((EQUAL (CAAR BKT) KEY)         ;match keys
                    (RETURN (CAR BKT))))           ;success, return entry
             (SETQ BKT (CDR BKT))
             (GO LP)))



(DEFINE INSERT (KEY DATA)
       ;; side effect on the table
       ((LAMBDA (INDEX)
                (STORE (TBL INDEX)
                       (CONS (CONS KEY DATA)       ;make new entry
                             (TBL INDEX))))
        (HASH KEY)))
```

Figure 1. Example LISP Program (cont'd next page).

```
(DEFINE DELETE (KEY)
        (PROG (BKT)
              (SETQ BKT (TBL (HASH KEY)))       ;fetch bucket
              ;; special case for entry first in bucket
              (COND ((EQUAL (CAAR BKT) KEY)     ;match keys
                     (STORE (TBL (HASH KEY))    ;store new bucket
                            (CDR BKT))
                     (RETURN NIL)))
              ;; search loop through bucket
           LP (OR BKT (RETURN NIL))
              (COND ((EQUAL (CAADR BKT) KEY)    ;match keys
                     (RPLACD BKT (CDDR BKT)))   ;side effect bucket in place
                    (T (SETQ BKT (CDR BKT))
                       (GO LP)))))


(DEFINE HASH (KEY)
       ;; returns index into table
       (ABS (REMAINDER (MAKNUM KEY) 500)))
```

Figure 1. Example LISP Program (cont'd).

The Elements of Program Description

The conceptual basis of our system is contained in three main ideas:

(i) Structural description of data objects.

(ii) Input-output specification of behavior.

(iii) Teleological description of the internal structure of programs.

Chapter Two of this report describes the formalisms we have developed to represent these ideas. Section 2.1 explains how we describe the structure of a data object in terms of its parts, properties and relationships between them. Section 2.2 defines the specification language we use for the input-output behavior of program segments. Our major effort, however, has been in area (iii) above. We have developed a representation called the plan, which is a coherent, formal way of describing the internal structure of a program in terms of how the behaviors of the parts interact to achieve the overall specifications. Plans are discussed in detail, with examples, in Sections 2.3 and 2.4.

In addition to providing the vocabulary for describing particular programs, the descriptive elements of Chapter Two are also used to represent the apprentice's knowledge base of general programming techniques. Section 2.5 discusses our current ideas about the organization of such a knowledge base. Also, at the end of Chapter Two, there is a brief section which relates the elements of program description to the types of comments that we have observed on LISP programs, and suggests how a program annotation facility might be implemented in the context of an apprentice system.

Implementation Progress Report

We are currently in the process of designing and implementing several key components of the complete apprentice. Chapters Three and Four describe the two components that are the furthest advanced. Chapter Three reports on the initial implementation of a deductive system that, given input-output specifications, can reason about the effect of a program on abstract data objects (this process is often called "symbolic evaluation"). Chapter Four describes a

program that has been written to analyze the control flow and data flow in a LISP program and build up a more schematic representation (called a "surface plan") that will form the framework for further understanding of the program.

Chapter Five discusses three additional components of a complete programmer's apprentice environment that have not yet been implemented: a plan formulation system for use in program design, a program recognizer to aid in catching implementation and coding bugs, and an interactive documentation system.

Further Work

We hope to use our initial implementation of a programmer's apprentice as a vehicle for continued research on program understanding. What we have done so far is to lay the foundations of a representational system intended to meet several criteria. First and foremost, we want our system to be based on concepts which correspond naturally to those used by practicing programmers. Secondly, we require representational adequacy: that is to say, our system must be capable of supporting all the capabilities we desire of an apprentice. Finally, the representation should be simple enough to allow an honest implementation attempt.

We believe that our work to date does satisfy the basic criteria we have established. However, many areas of implementation and theory still remain to be elaborated and clarified. For example, we would like to explore the applicability of our current approach to a larger range of programming domains, such as numerical computation, symbolic manipulation, and data base systems. There are also many desirable functional modes of the apprentice which we have not yet thought about in detail, such as a smart editor, a "code-cleaner", a smart display system, and so on. Undoubtedly, expansion in these two directions will reveal shortcomings in our initial system.

Two significant problems have already surfaced. First of all, we are lacking a good theory of how to organize a large data base of programming knowlege. We have managed thus far in our prototype implementation to avoid facing this issue squarely, but we cannot do so for much longer. Secondly, our current implementation effort has been fragmentary. The deductive system and the surface plan analyzer don't really "talk to each other" at the moment. The communication and interaction between subsystems required to achieve a fluid and robust apprentice environment presents new and as yet untreated design problems.

1.5 Anatomy of the P.A. System

In order to aid in reading of the remainder of this report, Figure 2 names and shows the important relationships between the major knowledge representations and processes in our system.

At a gross level, the anatomy of our P.A. system consists of two major blocks, corresponding to two major phases of writing a program: design and coding. The left side of the figure shows the knowledge structures that are most directly involved in coding activity; the right side shows the deeper knowledge used in program design. All the apprentice's knowledge about a given program taken together is referred to as the "complete program description", which is used for continued maintenance of the program.

At the leftmost of the figure is LISP code with comments. We feel it is very important to have as the base level in our descriptive system actual code that can run in the standard LISP environment at our computer site.

The first level of abstraction above the raw LISP code is called the surface plan. It is obtained from the LISP code by a process called surface flow analysis. In a surface plan, the code has been aggregated into meaningful units called segments, and the implementation details of data flow and control flow (e.g. the use of variables and the control primitives of LISP such as PROG, COND, etc.) have been abstracted away.

Deep plans are the level of description which captures the design intentions of the programmer. A deep plan is structurally similar to a surface plan in the sense that it is made up of segments. However, the segments in a deep plan are related by purpose links, in addition to data flow and possibly control flow. Purpose links express teleological dependencies between parts of a program, such as goal-subgoal and prerequisite relationships. In the ideal case, purpose links form the basis of a formal verification of the program.

In order to support plan recognition and verification, the apprentice requires a reasoning system, and a knowledge base of standard data structures, specification forms, and plans. These components of the P.A. are also shown in Figure 2.

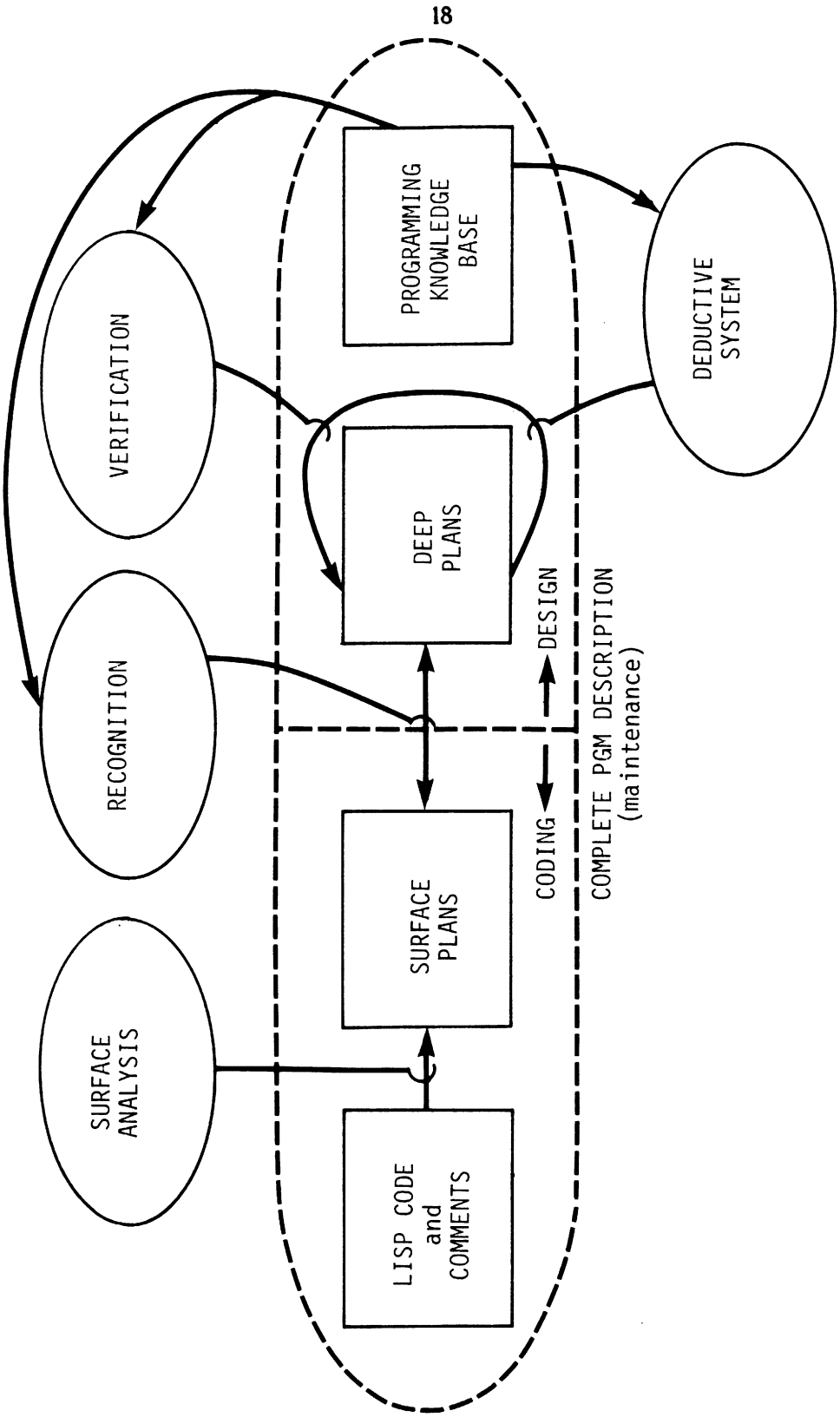


Figure 2. Gross Anatomy of P.A. System

For the apprentice to understand a particular LISP program means that it has connected all three levels of description for the program: code, surface plan, and deep plan. One way this can happen is for the programmer to initially specify a deep plan for his program through interaction with a plan formulation system like that described in Section 5.1. Then the programmer actually implements his plan in LISP code. The apprentice checks to see that the LISP code is compatible with the programmer's intentions by first building a surface plan, and then "recognizing" the correspondence (with the aid of the programmer's comments) between segments in the surface plan and segments in the deep plan.

CHAPTER TWO

THE ELEMENTS OF PROGRAM DESCRIPTION

In this chapter we present the descriptive system which underlies all of our work on program understanding. This includes a description of (i) data objects and their implementation, (ii) the input-output behavior of modules in the program, (iii) the internal data flow and teleological structure of the modules, and finally (iv) the surface features of the programming language used to write the code.

Programming Concepts

Often a group of procedures and the data structures upon which they operate can be thought of as multiple aspects of a single concept. Hashing, for example, is a concept which involves both a table data structure and procedures which implement the rules of associative retrieval. Thus hashing is a conceptual grouping of related objects (e.g. a table, buckets, entries, and keys) and related-operations (e.g. insert, lookup, delete, and hash). Thus the conceptual foundations of our apprentice include both procedural and data abstractions.

The notion of a programming concept as a grouping or "cluster" of objects and operations is similar to the ideas expressed in SIMULA <Dahl, 1970> and CLU <Liskov, 1974>. However our intention is not to enforce these ideas as part of a programming language, as in these systems, but rather to employ the notion of clustering as a basis for describing programs written in LISP.

Before going on to the details of our descriptive system, it is important to stress that our initial treatment of programming concepts is intended to be abstract; for example, we are trying to capture the essential aspects of hash tables in general, not an ad hoc description of a particular hash table in a particular configuration. In later sections, we return to the issues of understanding a particular program in terms of more general concepts.

The reader will notice that our current descriptive system ignores many complex efficiency-related issues such as time-space trade-off. For example, nowhere in our current system have we provided the ability to state that hash tables provide quick access at the cost of excess storage. We have however concerned ourselves with another difficult problem, that of describing and understanding programs with mutable data structures, side effects and sharing of sub-parts. This aspect of our work is a significant advance over most previous systems which have stayed within the domain of pure programs.

2.1 Description of Data Objects

The apprentice's knowledge of data structure takes two forms:

(i) The definition of object types, which are recorded in the programming knowledge base, and thus become part of the apprentice's permanent "vocabulary".

(ii) The description of particular objects (i.e. tokens of object types) that arise in describing and reasoning about the behavior of particular programs. These objects are described using vocabulary appropriate to the type of object involved.

Parts Decomposition

We want our apprentice to use a description of data objects which human programmers will also find natural. One of the most basic data structure notions is that there are object types which are characterized by their decomposition into parts. The basic object type of LISP, for example, is the CONS which can be described in terms of having two parts named CAR and CDR. In order to avoid committing ourselves to the terminology of a particular programming language we refer to this object type as a PAIR and call its parts LEFT and RIGHT:

```
(OBJECT-TYPE pair)
(PART pair left)
(PART pair right)
```

The simplest object type in our system which has parts is called a CELL; it has one part called CONTENTS:

```
(OBJECT-TYPE cell)
(PART cell contents)
```

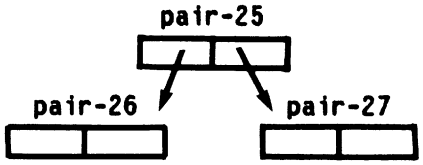
Furthermore, in our current system every object type having parts decomposition is potentially mutable, i.e. its parts may be changed without changing its identity (more on the implications of this point in Section 2.2).

Particular objects are denoted by unique object identifiers, e.g. OBJECT-25. In order to indicate that a particular object is an instance (token) of a given type, e.g. PAIR, we write:

```
(PAIR object-25)
```

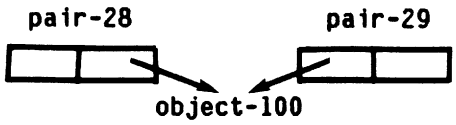
As a mnemonic device, in the remainder of this report we will usually employ unique object identifiers which have the object's type embedded, e.g. PAIR-25. However no semantic significance is attached to the use of upper and lower case type; these are used only to improve readability.

Once the abstract definition of the part structure of an object type has been given, the parts decomposition of a particular object can be written using the vocabulary of defined part names. The parts of an object are other objects. Thus, for example, the following list structure would be represented in our system as shown below:




```
(LEFT  pair-25 pair-26)
(RIGHT pair-25 pair-27)
```

Part names can also be used in a functional notation to refer to a part of an object without specifying an actual identifier for the part. For readability, square brackets (read "the object which is the") are used to denote functional use of a part name. Thus to describe the following shared structure:



either of the three following forms are equivalent:

```
(LEFT  pair-29 [RIGHT pair-28])

(RIGHT pair-28 [LEFT  pair-29])

(LEFT  pair-29 object-100)
(RIGHT pair-28 object-100)
```

In any case, when it comes to reasoning with this information, the square bracket notation is always expanded into the symmetric multiple-statement form above, introducing "anonymous objects" (see Section 3.1) if the actual identity of all the objects involved cannot be determined at the time of expansion.

Generic Part Structure

Some objects, for example arrays, have many parts which are not usually distinguished by distinctive names. We call this generic part structure, in which each part is identified by the generic part name and a numerical index. For example,

```
(OBJECT-TYPE array)
(GENERIC-PART array item)
```

In order to refer to a particular item of a particular array, we write

```
(ITEM array-25 index-2 object-17)
```

meaning that OBJECT-17 is the item in ARRAY-25 indexed by the index number INDEX-2.

Type Restrictions

In general, the definition of an object type involves more than just part structure. Often there are restrictions on the type of object that may fill particular part slots. For example, the key part of an entry in a hash table might be restricted to being an atom, or the items of an integer-array must be integers. We express this through a MUST-BE statement:

```
(OBJECT-TYPE entry)
(OBJECT-TYPE atom)
(PART entry keypart)
(MUST-BE keypart entry atom)


(OBJECT-TYPE integer-array)
(OBJECT-TYPE integer)
(GENERIC-PART integer-array item)
(MUST-BE item integer-array integer)
```

Similarly, a part may be restricted to one of several object types. A linked list structure, for example, may be defined recursively as an object with two parts called FIRST and REST; the REST is required to be either an object of type LIST or the empty list. (In LISP, of course, the empty list is the special atom NIL.) Such ranges of restriction are expressed by MAY-BE statements:

```
(OBJECT-TYPE list)
(OBJECT-TYPE emptylist)
(PART list first)
(PART list rest)

(MAY-BE rest list list)
(MAY-BE rest list emptylist)
```

Furthermore, our current system interprets MAY-BE's for a given part as an exhaustive enumeration of possible types.

Required type and permissible type statements can also be applied to properties, relations, and implementation parts (Section 2.3).

Properties

Besides decomposition into parts, data objects are also characterized by various properties, such as the length of a list, or the depth of a tree. The notation for property description is similar to that for parts, e.g.

```
(PROPERTY list length)
(PROPERTY emptylist length)

(LENGTH list-1 5)
```

Furthermore, since properties of particular objects are themselves objects, type restrictions are also applicable to properties. For example, the length of a list must be a positive integer, and the length of empty lists is zero:

```
(OBJECT-TYPE pos-integer)
(MUST-BE length list pos-integer)
(MUST-BE length emptylist 0)
```

Many properties of programming data objects are defined in terms of the object's internal part structure, and are thus subject to change if a part is changed. For example, in LISP the RPLACD operation changes the REST of a list, thereby changing its length if the new REST has a different length than the old one. Such dependencies are captured in property-definitions such as the following:

```
(PROPERTY-DEFINITION
    (length list pos-integer) <=>
    (length [rest list] [minus1 pos-integer]))
```

The above definition can be used in several ways. When the deductive system (Chapter Three) is attempting to determine the length of a particular list, it can expand the right-hand side of the property definition and try to prove that the length of the REST of the list is one less than the desired length. Conversely, given the known length of the REST of a particular list, the definition can be used in the opposite direction to calculate the length of the enclosing list.

Notice that in this example we have introduced symbolic arithmetic, which brings with it a raft of possible computational difficulties. However, for our present purposes, simple ad hoc techniques for reasoning about arithmetic relationships will be adequate.

A final use the apprentice can make of the above property definition is to conclude that the length of a list depends on the REST, while being independent of the FIRST. This sort of dependency analysis is important for reasoning about side effects (Sec. 3.2).

Relations

Objects may be related to other objects in other ways than being parts or properties. For example, a common relation is membership of an object in a data structure. Unlike properties, relations can be many-to-many mappings. A list has only one length, but it may have many members; similarly, a given object may be a member of several lists:

```
(RELATION member list object)

(MEMBER list-1 object-2)
(MEMBER list-1 object-3)
(MEMBER list-2 object-3)
```

Like properties, however, relations are often defined in terms of the part structure of the objects involved, and are therefore subject to change when the parts of either object are changed. For example, an object is a member of a list if and only if it is the FIRST of the list or a member of the REST of the list. This is expressed in a relation-definition:

```
(RELATION-DEFINITION
    (member list object) <=>
    (or (first list object)
        (member [rest list] object)))
```

In our descriptive system, the same named relation or property may exist for several different object types. For example, membership is also defined for pairs:

```
(RELATION member pair object)
(RELATION-DEFINITION
    (member pair object)  <=>
    (or (left pair object)
        (right pair object)))
```

Thus, for any given objects, the applicable relation or property definition is determined by the types involved. Furthermore, there is inheritance of relation definitions from a given object type to its specializations (see Section 2.5). For example, since association lists (a-lists) are a specialization of lists, they have the same membership relation, unless specified otherwise.

Type restrictions can also be placed on relations. For example, an a-list is a list, all of whose members are pairs:

```
(OBJECT-TYPE a-list)
(MUST-BE member a-list pair)
```

Figure 3 is a summary of the the data description statements presented in this section. The forms enclosed in angle brackets are meta-syntactic in the usual sense. Note that an <object-id> is a unique object identifier, such as OBJECT-75.

Description of Object Types

Parts Decomposition

```
(OBJECT-TYPE <object-type> )
(PART <object-type> <partname> )
(GENERIC-PART <object-type> <partname> )
```

Type Restrictions

```
(MUST-BE <part,property,relation-name> <object-type-1> <object-type-2> )
(MAY-BE <part,property,relation-name> <object-type-1> <object-type-2> )
```

Properties

```
(PROPERTY <object-type> <property-name> )
```

Relations

```
(RELATION <relation-name> <object-type-1> <object-type-2> )
```

Description of Particular Objects

```
(<object-type> <object-id>)
(<partname> <object-id-1> <object-id-2> )
(<partname> <object-id-1> <index> <object-id-2>)
(<property-name> <object-id-1> <object-id-2> )
(<relation-name> <object-id-1> <object-id-2>)
```

Figure 3. Summary of Data Object Descriptions.

2.2 Specification of Behavior

Structural decomposition of objects does not alone provide an adequate description of programming concepts. For example, the concept of a stack is characterized both by its recursive structure (top element and rest of the stack) and by the push and pop operations used to manipulate this structure. Thus in order to describe a stack at this level of abstraction (i.e. without reference to details of implementation) we require a way of describing the behavior of the operations on it.

Programmers understand the behavior of their programs in terms of the effect of important units of behavior (which correspond to "chunks" of code). Depending on focus and level of abstraction the appropriate aggregation of behavior (and therefore code) changes. For example, if one is focussed on a particular stack, the appropriate units of behavior are the push and pop operations. However, if one is thinking at the level of the entire time-sharing system in which the stack is embedded, the appropriate units of behavior are much larger. We refer to units of program behavior (and the aggregations of code which realize that behavior) as segments.

A segment can be characterized by its effect on any set of input objects which satisfy certain restrictions. For example, a push operation acts on two input objects, one of which must be a stack and the other of which might be restricted to the object type appropriate for membership in that stack. The effect of this segment is to make the second input object the new top of the stack, while the old stack becomes the rest of the new stack.

In general, a segment's behavior is specified by its input restrictions, the modification it performs on the input objects, the new objects it creates, and the conditions that are guaranteed to hold true between the input and output objects following execution of the segment. Thus for an implementation-independent level of behavioral description, we use the traditional notion of input-output specifications.

Specs

We have developed a particular notation for input-output specifications called specs. Specs have four sections: a list of input objects (INPUTS), a list of input conditions which are required to be true for the segment to perform its function (EXPECT), a list of output objects including new objects created and input objects which have been modified (OUTPUTS), and finally a list of conditions guaranteed to be true on output (ASSERT). As a simple example, consider the specs for a push segment:

```
(SEGMENT-TYPE push-segment)
(OBJECT-TYPE stack)
(PART stack top)
(PART stack rest)

(SPECS-FOR: push-segment
    (INPUTS: stack-1 object-2)
    (EXPECT: (stack stack-1))
    (OUTPUTS:  stack-3)
    (ASSERT: (stack stack-3)
             (top stack-3 object-2)
             (rest stack-3 stack-1))
```

Specs like these express what we think is the natural, common way that programmers conceptualize the behavior of code they have written, or are planning to write. Each segment's specs describes the intrinsic input-output behavior of the segment, i.e. the conditions that hold before and after execution of the segment, independent of its interaction with surrounding segments. The intrinsic behavior of a segment is determined solely by its internal workings. As a second example, consider the intrinsic description of FIRST-SEGMENT, which takes the FIRST part of a list:

```
(SEGMENT-TYPE first-segment)

(SPECS-FOR: first-segment
    (INPUTS: list-1)
    (OUTPUTS: object-2)
    (EXPECT: (list list-1))
    (ASSERT: (first list-1 object-2)))
```

FIRST-SEGMENT is a segment type in the same sense that LIST is an object type. Each instance (token) of a segment type has a unique name, e.g SEGMENT-75. The type of a particular segment is denoted as follows:

```
(FIRST-SEGMENT segment-75)
```

Each instance of FIRST-SEGMENT has the same intrinsic behavior, i.e. the specs given above. However, various instances of FIRST-SEGMENT may be used for different purposes, e.g. to extract a key from a data entry, get the top of a stack, and so on. The use of a segment in a particular program is called its extrinsic description. We will return to the topic of extrinsic descriptions in Section 2.3 on implementation and plans.

The definition of segment type implicitly defines a new n-ary relation with the same name, which can be used to express the relationship between the input and output objects. For example,

```
(PUSH-SEGMENT stack-10 object-3 stack-11)
```

asserts that STACK-11 is the output object resulting from the application of the specs for PUSH-SEGMENT to inputs STACK-10 and OBJECT-31. Notice that the meaning of the argument positions in the relation implicitly defined by a segment is determined by the order of objects listed in the input and output clauses of the specs. In the case where there is a single output object, is is also convenient to allow use of functional notation for such relations, e.g.

```
[PUSH-SEGMENT stack-10 object-3]
```

refers to the single output object of the segment with the given inputs, without naming it explicitly.

The general format for writing the specs of a segment is shown at the top of Figure 4 (angle brackets are meta-syntactic as usual). Note that each <object-id> in the input and output lists is assumed to be unique only within the scope of the specs in which it appears. However, the <segment-id> is a system-wide unique identifier for the segment. Beneath the written format is the diagrammatic form for representing specs that we will follow throughout this report. (Notice that there are two alternative positions for the <segment-id> in the diagram.)

Input objects are any data objects (e.g. an array, a number, or a list) that the behavior of the segment depends upon in any way. In terms of LISP code, input objects enter a segment either as bindings of a lambda argument or the value of a free variable. The conditions listed under EXPECT are the conditions on, or relationships between individual input objects that are expected to hold just prior to the execution of the segment, and upon which the correct functioning of the code depends. All pre-conditions upon which the segment depends much be listed in EXPECT's.

The other half of a specs is the list of output data objects and conditions on them. Any data structure that is changed as a result of executing the segment, or newly created in the segment, must be listed in the OUTPUTS. In terms of LISP code, output objects are usually the return value of an s-expression, the value of a free variable, or list structure that has been RPLAC'ed or newly CONS'ed. ASSERT conditions are conditions on the output objects, or relations between input and output objects, that will hold immediately following the correct execution of the segment. With the exception of special rules for side-effected objects (which will be specified later), nothing is assumed to be true of output objects except what is explicitly ASSERT'ed.

Our specification language is based on the concept of data flow between operations, similar to Dennis <1975>. Thus, a fundamental difference between input-output specifications in our system and those in the traditional Floyd-Hoare approach <Hoare, 1969> is that variables in our specification assertions denote program data objects rather than literal program variables, as in Floyd-Hoare. An additional minor difference is that, since we are using a specially tailored deductive system, we have modified the standard quantifiers of predicate calculus to more

```
(SPECS-FOR:  <segment-id>
    (INPUTS:   <object-id> <object-id> ... )
    (OUTPUTS:  <object-id> <object-id> ... )
    (EXPECT: <pre-condition>
             <pre-condition> ... )
    (ASSERT: <post-condition>
             <post-condition> ... ))
```

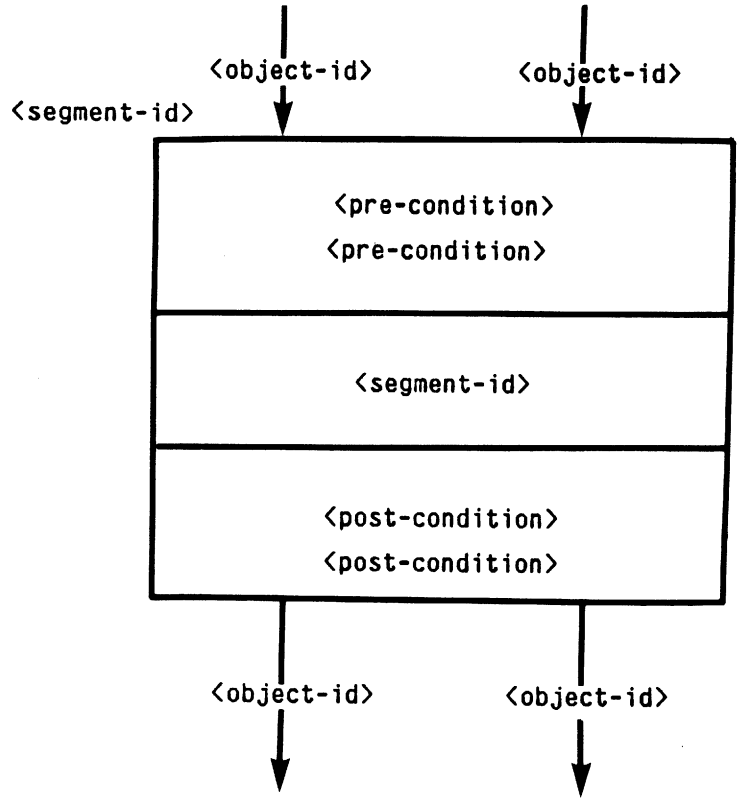

<object-id>
<object-id>
<segment-id>
<pre-condition>
<pre-condition>
<segment-id>
<post-condition>
<post-condition>
<object-id>
<object-id>

Figure 4. Written and Diagrammatic Format for Specs.

convenient forms. (See Section 3.1 for more details on quantifiers in our system.)

One very important form of condition that will appear in almost all specs, is the type restriction. For example, in the specs for FIRST-SEGMENT, given earlier, there is the expectation (LIST LIST-1). This means that the correct functioning of such a segment depends on the input object LIST-1 being of the type LIST. Because type restrictions are so common in specs, for presentation purposes we will often embed the type restriction in the object identifier itself and omit the explicit expectation in the specs. Thus for example, we abbreviate

```
(INPUTS:  object-1)                to      (INPUTS: list-1),
(EXPECT: (list object-1))
```

and similarly for other types.

Case Splitting

Much of the power of computer programs derives from the ability to do conditional branching. This branching behavior is reflected in specs by introducing cases. For example, the specs for LOOKUP-SEGMENT, a segment to look up entries in a hash table, has two cases: either the search is successful, in which case the required entry is returned; or the search is unsuccessful, in which case a failure indication is returned.

```
(SPECS-FOR: lookup-segment
  (INPUTS: key-5 table-6)
  (CASE-1
    (EXPECT: (there-exists-a-unique (member table-6 =entry)
              such-that (keypart entry key-5)))
    (OUTPUTS: entry-7)
    (ASSERT: (keypart entry-7 key-5)
             (member table-6 entry-7)))
  (CASE-2
    (EXPECT: OTHERWISE)
    (OUTPUTS: failure-8)
    (ASSERT: (for-all (member table-6 =entry)
                      (not (keypart entry key-5))))))
```

The part of a segment's specs that is applicable in all cases is listed first. In the above example, for instance, the inputs to a LOOKUP-SEGMENT are always a key and a table. The rest of the specs is divided into case clauses, labelled CASE-1, CASE-2, etc. The expectations of the segment as a whole are met if and only if all non-CASE-embedded EXPECT's are met and the EXPECT's of one case are met. Furthermore, the EXPECT conditions are supposed to be mutually exclusive, though the current version of the apprentice does not actually check for violations of this. The special keyword OTHERWISE is a convenient abbreviation for the conjunction of the negations of the EXPECT conditions of all the other clauses. Obviously only one OTHERWISE may be used in each specs.

In diagrams, case splitting is indicated by dividing the bottom of a specs box into sections, one per case, using vertical lines. Examples of diagrams with case splitting will appear later. More details on the role of case structure in reasoning about programs will be given in Chapter Three.

Side Effects

In Floyd-Hoare type programming semantics, which is based on the concept of program variables as opposed to the flow of data objects, the issue of "side effects" usually refers to programs that change the value of a global free variable. In our descriptive system, however, the side effect problem arises due to the notions of identity and mutability of data objects. A mutable data object is one whose parts (and therefore properties) can be changed without changing its identity. All data objects in our system that have part structure are potentially mutable.

In specs, each output object falls into one of three classes:

(i) The output objects is known to be identical to one of the input objects.

(ii) The output objects is known to be new and uniquely created.

(iii) Neither of the above is known.

Our specification language has special assertions to express these possibilities. The default notation denotes the least specified case (iii) above. For example, the following specs would be satisfied either by a segment that replaces the LEFT part of pair (i.e. RPLACA in LISP) or by one that makes up a new pair with the new LEFT and the old RIGHT parts:

```
(SPECS-FOR: put-left
    (INPUTS: pair-1 object-2)
    (OUTPUTS: pair-3)
    (ASSERT: (left pair-3 object-2)
             (right pair-3 [right pair-1])))
```

Thus in these specs, PAIR-3 may or may not be indentical to PAIR-1. To specify a side effect, an ID (for "identical") assertion is used:

```
(SPECS-FOR: replace-left
    (INPUTS: pair-1 object-2)
    (OUTPUTS: pair-3)
    (ASSERT: (id pair-1 pair-3)
             (left pair-3 object-2)))
```

These specs would be satisfied only by an implementation using RPLACA. For output objects that are ID to input objects, our current system assumes that all parts not explicitly changed in the output situation are the same in the output situation as in the input situation. Thus, for example, in the specs for REPLACE-LEFT it is assumed that the right part of PAIR-3 is unchanged from the input situation, i.e. it was not necessary (though it would not be incorrect) to ASSERT:

```
(right pair-3 [right pair-1])
```

However, all properties and relations involving side-effected objects in the input situation are considered unknown in the output situation unless they are explicitly ASSERT'ed.

Notice that PAIR-1 and PAIR-3 in the above specs are different names for the identical object in the "before" and "after" situations, respectively. This double naming is necessary in order to allow relations to be specified between the parts of the side-effected object in the before situation and in the after situation, as for example in the following segment which exchanges the RIGHT and LEFT parts of a pair:

```
(SPECS-FOR: swap
    (INPUTS: pair-1)
    (OUTPUTS: pair-2)
    (ASSERT: (id pair-1 pair-2)
             (left  pair-2 [right pair-1])
             (right pair-2 [left  pair-1])))
```

A NEW assertion is used to specify a newly created object (e.g. by CONS) which is known to be distinct from all existing objects:

```
(SPECS-FOR: create-left
    (INPUTS: pair-1 object-2)
    (OUTPUTS: pair-3)
    (ASSERT: (new pair-3)
             (left pair-3 object-2)
             (right pair-3 [right pair-1])))
```

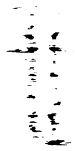
Identity and mutability of data objects introduce another complexity that is not present at all in traditional variable-oriented semantics: sharing of data structures. This is a very common technique in LISP programming, which is achieved by inserting a single data object as part of two different data objects. The specification language presented thus far is adequate to describe segments that have this effect. However, it is important to recognize that the real difficulty with sharing is not the specification per se, but rather being able to reason effectively using the specifications. Thus we will see that a great deal of mechanism in the deductive system is concerned with reasoning about side effects.

Programming Concepts

Now that we have introduced the essentials of program behavior and data object description, we can return more concretely to the notion of a programming concept as a grouping of object types, segment types, and associated relations and properties. For example, the concept of hashing, which runs throughout this report, has as its constituent object types: the hash table, buckets which are part of the table, entries, and keys which are part of entries. Furthermore, in many concepts one of the constitutuent object types seems to be the primary object, i.e. the object whose implementation has the most important effect on the implementation of the other objects and segment types. For example, in hashing the hash table is the primary object. Three important segment types for hashing at this most abstract level of description are lookup, insert, and delete, each with its appropriate specs. Furthermore, the relation of membership of an entry in the table and of the hash formula itself should also be considered as important high-level constituents of the hashing concept. (Unfortunately our present specs for the hash segment are significantly deficient, since we do not yet know how to

represent the efficiency issues that are the crux of why hashing is done in the first place.)

Figure 5 shows how the information associated with the hashing concept is represented by the apprentice. Figure 6 following also shows the information which our current apprentice associates with the concept of arrays, another important concept that we will make use of in examples. Many of the details in these figures, for example the unusual quantifiers used in the specs for lookup, insert, and delete, can be skipped on first reading, since they will be explained in later sections.



```
(CONCEPT hashing)

(CONSTITUENT-OBJECTS hashing (table entry key index))
(CONSTITUENT-SEGMENTS hashing
              (lookup-segment insert-segment delete-segment hash))
(CONSTITUENT-RELATIONS hashing (member index-of))
(CONSTITUENT-PROPERTIES hashing (size))
(PRIMARY-OBJECT hashing table)

(OBJECT-TYPE table)          (OBJECT-TYPE key)
(OBJECT-TYPE bucket)         (OBJECT-TYPE entry)

(GENERIC-PART table bucketpart)
(PART entry keypart)
(PART entry datapart)
(MUST-BE bucketpart table bucket)
(MUST-BE keypart entry key)

(PROPERTY table size)
(MUST-BE size table pos-integer)

(RELATION member table entry)
(RELATION-DEFINITION
    (member table entry) <=>
    (member [bucketpart table [hash [keypart entry]]]
            entry))
(RELATION index-of table index)
(RELATION-DEFINITION
    (index-of table index) <=>
    (and (pos-integer index)
         (less-than-or-equal [size table] index)))
```

Figure 5. Representation of Hashing Concept (cont'd next page).

```
(SPECS-FOR: lookup-segment
  (INPUTS: key-5 table-6)
  (CASE-1
    (EXPECT: (there-exists-a-unique (member table-6 =entry)
              such-that (keypart entry key-5))
    (OUTPUTS: entry-7)
    (ASSERT: (keypart entry-7 key-5)
             (member table-6 entry-7))))
  (CASE-2
    (EXPECT: OTHERWISE)
    (OUTPUTS: failure-8)
    (ASSERT: (for-all (member table-6 =entry)
                      (not (keypart entry key-5))))))


(SPECS-FOR: insert-segment
    (INPUTS: entry-1 table-2)
    (OUTPUTS: table-3)
    (ASSERT: (id table-2 table-3)
             (member table-3 entry-1)
             (for-all (member table-2 =entry)
                      (member table-3 entry))))
```

Figure 5. Representation of Hashing Concept (cont'd next page).

```
(SPECS-FOR: delete-segment
    (INPUTS: key-1 table-2)
    (OUTPUTS: table-3)
    (ASSERT: (id table-2 table-3)
             (for-all (member table-2 =entry)
                      (member table-3 entry)
                 except-for (keypart entry key-1)
                 for-which (not (member table-3 entry)))))
```

```
(SPECS-FOR: hash
    (INPUTS: key-1 table-2)
    (OUTPUTS: index-3)
    (ASSERT: (index-of table-1 index-3)))
```

Figure 5. Representation of Hashing Concept (cont'd).

```
(CONCEPT arrays)

(CONSTITUENT-OBJECTS arrays (array index))
(CONSTITUENT-SEGMENTS arrays (arrayfetch arraystore))
(CONSTITUENT-RELATIONS arrays (index-of))
(CONSTITUENT-PROPERTIES arrays (size))
(PRIMARY-OBJECT arrays array)

(OBJECT-TYPE array)              (OBJECT-TYPE index)

(GENERIC-PART array item)

(PROPERTY array size)
(MUST-BE size array pos-integer)

(RELATION index-of array index)
(RELATION-DEFINITION
    (index-of array index) <=>
    (and (pos-integer index)
         (less-than-or-equal index [size array])))

(SPECS-FOR: arraystore
    (INPUTS: array-1 index-2 object-3)
    (OUTPUTS: array-4)
    (EXPECT: (index-of array-1 index-2))
    (ASSERT: (id array-4 array-1)
             (item array-4 index-2 object-3)))

(SPECS-FOR: arrayfetch
    (INPUTS: array-1 index-2)
    (OUTPUTS: object-3)
    (EXPECT: (index-of array-1 index-2))
    (ASSERT: (item array-1 index-2 object-3)))
```

Figure 6. Representation of Arrays Concept

2.3 Implementation and Deep Plans

Although part of the programmer's task is to choose the appropriate procedural and data abstractions for his problem domain, his work does not stop at that point. The abstract data objects and program specifications must be transformed into the available primitives of the programming environment which is being employed. We call this process program implementation. Many methodologies have been advanced as the "correct" way of undertaking this process: top-down, step wise refinement, bottom-up programming, middle-out programming, and so on. Our intention is not to argue for one or another such methodology, but rather to investigate what conceptual structures are needed by the programmer's apprentice to support implementation in any form.

Implementation of Data Objects

Data objects are implemented by mapping the parts of the abstract data object onto structures of less abstract data objects. For example, a hash table may be conceptualized as having a generic part structure consisting of objects called buckets, each of which may contain a variable number of entries:

```
(OBJECT-TYPE table)
(OBJECT-TYPE bucket)
(GENERIC-PART table bucketpart)
(MUST-BE bucketpart table bucket)
```

A table may be implemented as an array, with the items of the array playing the role of the buckets. Thus,

```
(IMPLEMENTATION-PART table table-array)
(MUST-BE table-array table array)
(IMPLEMENTATION-DEFINITION
   (bucketpart table index bucket) <=>
   (item [table-array table] index bucket))
```

TABLE-ARRAY is the name of an implementation part. It is used syntactically in the same way as a part name. This is a comparatively simple example of implementing a data object, since a single abstract object, the TABLE, is implemented using a single implementation part, the TABLE-ARRAY. Consider the implementation of a queue using an array. This requires three implementation parts, an array plus two pointers:

```
(OBJECT-TYPE queue)
(PART queue front)
(PART queue back)

(IMPLEMENTATION-PART queue queue-array)
(IMPLEMENTATION-PART queue front-pointer)
(IMPLEMENTATION-PART queue back-pointer)

(MUST-BE queue-array queue array)
(MUST-BE front-pointer queue cell)
(MUST-BE back-pointer queue cell)

(IMPLEMENTATION-DEFINITION
    (front queue object) <=>
    (item [queue-array queue] [contents [front-pointer queue]] object))
(IMPLEMENTATION-DEFINITION
    (back queue object) <=>
    (item [queue-array queue] [contents [back-pointer queue]] object))
```

Of course there are usually several different ways to implement any given abstract object. For example, queues can also be implemented using two implementation parts, a list and a back-pointer:

```
(IMPLEMENTATION-PART queue queue-list)
(IMPLEMENTATION-PART queue back-pointer)

(MUST-BE queue-list queue list)
(MUST-BE back-pointer queue cell)

(IMPLEMENTATION-DEFINITION
    (front queue object) <=>
    (first [queue-list queue] object))
(IMPLEMENTATION-DEFINITION
    (back queue object) <=>
    (first [contents [back-pointer queue]] object))
```

The information above does not specify all aspects of implementing the concepts of either hashing or queues. Just as the primary data objects above are implemented in terms of the structure of a more primitive objects, so too segment types are implemented in terms of more primitive behavioral units. Furthermore, the implementation of data objects and segments in terms of more primitive abstractions often proceeds in coordinated steps, so that at each level there is a coherent vocabulary for describing the program. How to organize the representation of alternative implementations and multiple levels of description in a large data base is a problem which we have not yet dealt with in any depth. Section 2.5 discusses our current ideas on this and other difficult issues of data base organization in the apprentice.

Implementation of Specs by Plans

Segments are the programmer's building blocks. To design a program, he puts together a number of segments, arranging data flow and control flow to satisfy the overall specifications, in much the same way as an electronics engineer constructs a circuit from electronic components, or a mechanical engineer constructs a machine from mechanical parts. This purposeful arrangement of building blocks is called a plan, by analogy with construction plans, mechanical plans, and so on. In making a plan, the design engineer chooses from a repertoire of many types of building blocks, according to the requirements of his application. Building blocks may be very simple, such as resistors, capacitors, and diodes for electronics, or plates,

rods, and bolts for mechanical engineering; or they may be large modules with internal plans of their own, such as filters, amplifiers, and modulators, or pulleys, motors, and gearboxes. Similarly in programming there is a repertoire of segment types at different levels of complexity, e.g. FIRST-SEGMENT, PUSH-SEGMENT, INSERT-SEGMENT, etc., which the programmer has to work with.

Thus a program plan is a network of segments (similar to Sacerdoti's <1975a> procedural nets), each having its own intrinsic specs, along with the extrinsic relationships between them. In our present work we also distinguish between two levels of plan: deep plans and surface plans. Deep plans, which we will discuss first, contain only information about the data flow between segments and the goal structure (teleological relationships) between segments. Surface plans add to this the details of actual control flow in the coded program and the syntactic mechanisms used in the code to achieve inter-segment communication.

The statement of a deep plan has three major sections: a list of sub-segments in the plan, a list of data flow between sub-segments, and a list teleological relationships between the sub-segments. We illustrate with GET-BUCKET, a segment used in a typical hashing system to fetch a bucket, given the key. The specs for GET-BUCKET would be:

```
(SPECS-FOR: get-bucket
    (INPUTS: table-6 key-5)
    (EXPECT: (table table-6)
             (key key-5))
    (OUTPUTS: bucket-10)
    (ASSERT: (bucketpart table-6 [hash key-5 table-6] bucket-10)))
```

The typical plan for GET-BUCKET in an array implementation of the hash table uses a HASH segment and an ARRAYFETCH segment. In the apprentice's notation, this becomes:

```
(PLAN-FOR: get-bucket
    (SUB-SEGMENTS: (hash-1 arrayfetch-2)))
```

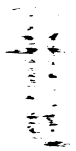
Here HASH-1 and ARRAYFETCH-2 are instances (tokens) of the respective segment types. Diagrammatically, the segments in a plan are drawn as sub-boxes inside the larger box

representing the overall segment being implemented. For example, see Figure 7.

Our plan representation does not exclude overlapping of segments, i.e. sub-segments shared between two different enclosing segments -- though no overlapping will appear in examples in this report. Waters <1976> has reported on overlapping loops and similar plans in a corpus of mathematical FORTRAN programs.

Since each sub-segment in a plan is an instance of a segment type, it has the same specs as given for that segment type:

```
(SPECS-FOR: hash-1
    (INPUTS: key-13 table-14)
    (EXPECT: (key key-13)
             (table table-14))
    (OUTPUTS: index-15)
    (ASSERT: (index-of table-14 index-15))
```



```
(SPECS-FOR: arrayfetch-2
    (INPUTS: array-17 index-18)
    (EXPECT: (array array-17)
             (index-of array-17 index-18))
    (OUTPUTS: object-22)
    (ASSERT: (item array-17 index-18 object-22)))
```

The plan for a segment achieves its specs as a result of the purposeful interaction between its sub-segments. Sub-segments interact in several ways. First of all, there is data flow between segments, e.g. the output object of one segment becomes the input object of another. There are also teleological relationships between segments, called purpose links. The purpose links in a plan describe "why" each segment is used. For example, a segment is often used because its output assertions establish the required input expectations of another, subsequent segment. Let us first consider data flow.

Data Flow

There are three possible forms of data flow between segments in our plan representation: input-input, output-output, and output-input. Input-input and output-output data flow occur between a segment and its sub-segments; output-input data flow occurs only between segments at the same level of plan. Diagrammatically data flow is shown using arrows between the labelled input and output objects of each segment involved, as in Figure 7. (Note that no significance is to be attached to thick vs. thin arrows that will appear in some flow diagrams -- these have been used in alternation simply to increase legibility of crossing arrows.) Figure 7 is encoded in the apprentice's data as follows:

```
(PLAN-FOR: get-bucket
    (SUB-SEGMENTS:  (hash-1 arrayfetch-2))
    (DATAFLOW: (get-bucket INPUT key-5)
               (hash-1 INPUT key-13))
    (DATAFLOW: (get-bucket INPUT table-6)
               (hash-1 INPUT table-14))
    (DATAFLOW: (get-bucket INPUT table-6)
               (arrayfetch-2 INPUT array-17))
    (DATAFLOW: (hash-1 OUTPUT index-15)
               (arrayfetch-2 INPUT index-18))
    (DATAFLOW: (arrayfetch-2 OUTPUT item-22)
               (get-bucket OUTPUT bucket-10)))
```

Let us now consider a particular data flow link:

```
(DATAFLOW: (hash-1 OUTPUT index-15)
           (arrayfetch-2 INPUT index-18))
```

This is an example of an output-input data flow link. HASH-1 and ARRAYFETCH-2 are sub-segments in the plan for GET-BUCKET. The output INDEX-15 of HASH-1 becomes the input object INDEX-18 of ARRAYFETCH-2. This kind of data flow can occur by several mechanisms in LISP. For example, the two sub-segments could use the same free variable for the corresponding data objects, or alternatively, HASH-1 could give its output object as the return value of an s-expression, which is then lambda-bound to an argument of the code which

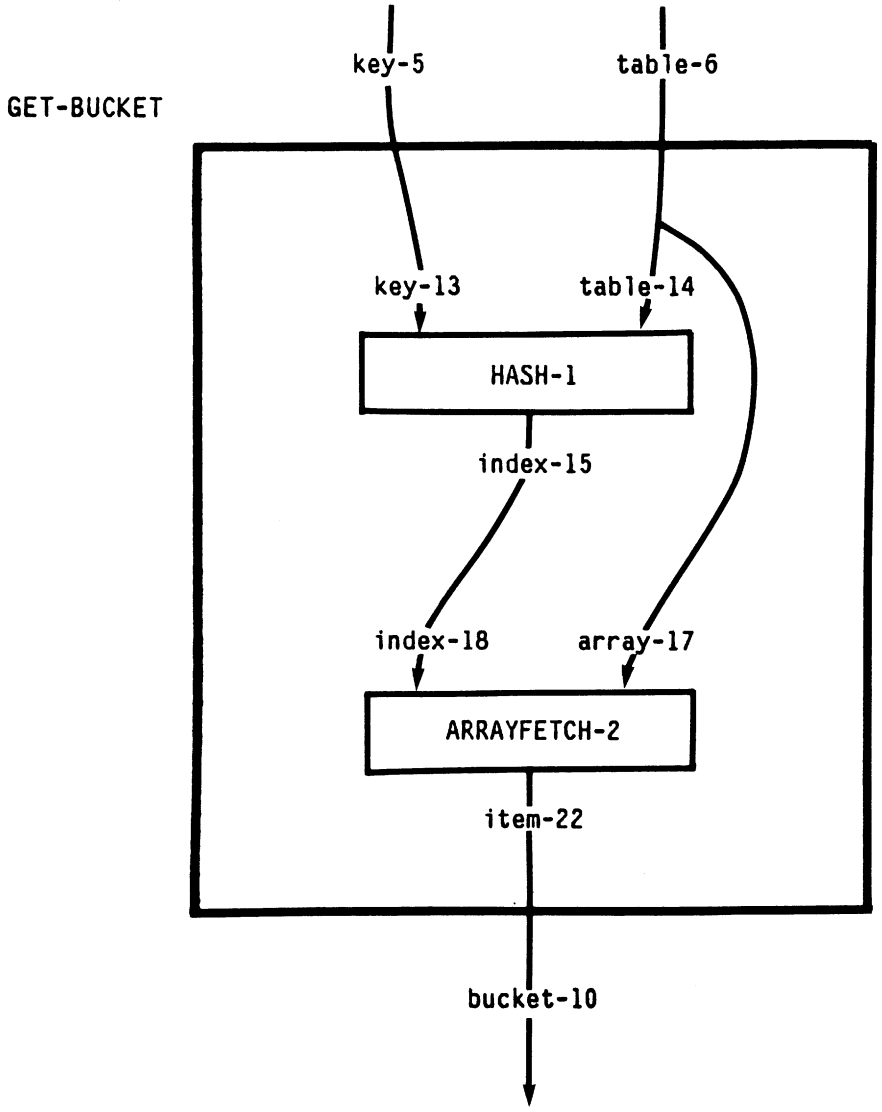


Figure 7. Data Flow Links in Deep Plan for GET-BUCKET.

realizes the ARRAYFETCH-2 segment. Such information on how data flow in the deep plan is realized at the LISP code level is represented only in surface plans, which will be described in Section 2.4.

Sometimes the input data to a segment does not come from the output of other sub-segments in the same plan. For example, the input KEY-13 to HASH-1 in the plan for GET-BUCKET is not the output of any other sub-segment; rather, it is the same as the KEY-5 which is an input object of GET-BUCKET. This is called input-input data flow. Similarly, there are output-output data flow links between a sub-segment and the enclosing segment. For example, the output ITEM-22 of ARRAYFETCH-2 becomes the output BUCKET-10 of GET-BUCKET. These kinds of data flow also have many possible realizations in LISP code.

Purpose Links

Obviously segments are not linked together in an arbitrary manner, randomly running data from one to the other; rather the programmer has in mind a structure of interlocking goals and subgoals, which are embodied by the sub-segments of the deep plan. To achieve the overall specs of the segment he is trying to implement, the programmer often poses intermediate goals which are achieved by sub-segments in the plan. These sub-segments are useful only if their expectations can in turn be satisfied. Thus a network of goal-subgoal and prerequisite relationships is created tying together the sub-segments. This teleological structure is the deepest level of understanding how a program works.

Since the teleological relationships between a given segment and the rest of the plan answer the question of "why" that particular segment is being used, we call these relationships purpose links. In our system a purpose link is represented as a one-to-many relation between an input or output condition of a given segment and the input or output conditions of other segments that support it. Furthermore we categorize purpose links according to whether an EXPECT or an ASSERT condition is being satisfied: the support for an EXPECT condition is called a PREREQ (for "prerequisite") link; the support for an ASSERT condition is called an ACHIEVE link.

Let us first consider PREREQ links. The expectations of a sub-segment in a plan can be satisfied from two directions: either from the output assertions of another (preceding) sub-segment, or from the overall expectations of the super-segment. For example, ARRAYFETCH-2 in the plan for GET-BUCKET expects one of its input objects to be a valid index for the given array. This expectation is guaranteed in the plan by the output assertion of HASH-1:

```
(PLAN-FOR: get-bucket
    ...
    (PREREQ: (arrayfetch-2 EXPECT (index-of array-17 index-18))
             ((hash-1 ASSERT (index-of table-14 index-15))))
    (DATAFLOW: (hash-1 OUTPUT index-15)(arrayfetch-2 INPUT index-18))
    ... )
```

As is quite typical, this purpose link is coupled with a data flow link between the segments. However, it is also possible to have a PREREQ relationship without direct data flow between the two segments involved. For example (see Figure 8), in plans which CDR down lists, it is common to check for the empty list (NIL) before taking the CAR. The specs for a test segment has two cases, asserting respectively the success or failure of the test condition on the input object; but there are no output objects. Nevertheless, one of the cases of the test segment can the PREREQ for some other operation to be performed on the same input object.

The second input expectation of ARRAYFETCH-2, i.e. (ARRAY ARRAY-17) is satisfied by one of the input conditions of the overall plan segment, GET-BUCKET:

```
(PLAN-FOR: get-bucket
    ...
    (PREREQ: (arrayfetch-2 EXPECT (array array-17))
             ((get-bucket EXPECT (table table-6))))
    (DATAFLOW: (get-bucket INPUT table-6)
               (arrayfetch-2 INPUT array-17))
    ... )
```

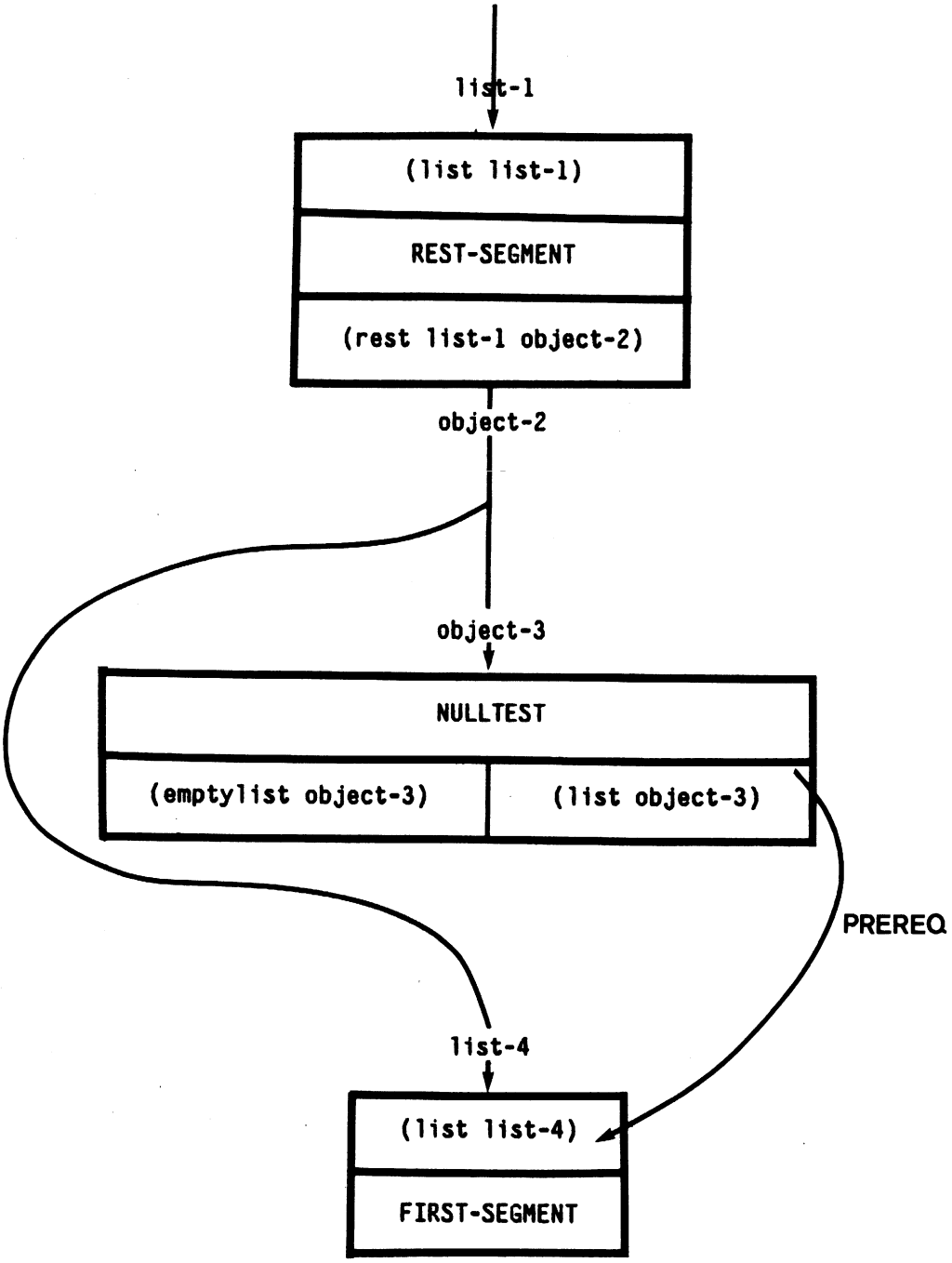


Figure 8. Purpose Link without Direct Data Flow.

Notice that in this case the support relation between the conditions in the purpose link is not literal equality as in the first example. Often a purpose link depends on some intermediate deductions using knowledge about design choices. In this case the validity of the purpose link depends on the fact, recorded in the knowledge base, that the hash table in question has been implemented as an array:

```
(IMPLEMENTATION-PART table-6 table-array)
(TABLE-ARRAY table-6 array-17)
```

The dependence of purpose links on such design and implementation facts is also recorded by the apprentice as part of a complete program description.

ACHIEVE links usually express goal-subgoal relationships between desired output assertions of the overall segment and output assertions of certain sub-segments in the plan. Goldstein <1974> has called these important sub-segments "main-steps". It is also possible in our formalism, though somewhat of a programming oddity, for an output assertion to be satisfied directly by an input expectation of the same segment; however, we have found no natural examples of this kind of purpose link.

The ACHIEVE link in the plan for GET-BUCKET is the most complicated purpose link presented thus far:

```
(PLAN-FOR: get-bucket
    ...
    (ACHIEVE:
(get-bucket ASSERT (bucketpart table-6 [hash key-5 table-6] bucket-10))
((hash-1 ASSERT (hash key-13 table-14 index-15))
 (arrayfetch-2 ASSERT (item array-17 index-18 item-22))))
    ... )
```

The validity of this link depends not only on all the data flow in the plan, but also on the implementation fact that the i'th bucket of the table is implemented as the i'th item of the array:

```
(bucketpart table-6 index bucket) <=>
(item array-17 index bucket)
```

The complete deep plan for GET-BUCKET is shown in Figure 9.

Purpose Links as Summary of a Justification

We have already seen that purpose links often involve a brief deductive step, typically using a fact from the design knowledge base to show that two non-identical conditions correspond. However, the proof structure of purpose link may also be much more complicated than this. In Chapter Three we will show such an example, wherein the verification of an ACHIEVE link requires a proof by cases involving virtually every assertion in the data base.

In general, a purpose link is supposed to capture the programmer's reason for believing that a required condition will be met at a particular point in his program's execution. As an alternative to just "believing" the programmer, the apprentice can attempt to formally verify each of these purpose links using the deductive system described in Chapter Three. The full record of verification for all the purpose links in a deep plan is called the justification of a program. The structure of this justification, though frequently quite simple, can also be arbitrarily complex.

Thus one way of thinking about purpose links is that they summarize the important steps in a program's justification. We can imagine two different scenarios in which purpose links are used. In the first scenario, the deductive system is given just the data flow part of a deep plan, and the purpose links are generated automatically as a by-product of the verification procedure. In this case, the purpose links truly constitute a complete justification of the plan, since the original proof detail may be recreated at will. In a less ambitious scenario, the purpose links are supplied explicitly by the programmer himself as part of his design, in which case they are used by the reasoning system as an outline of what to try proving in order to verify that his intentions are being met. In this scenario, it is possible that the purpose links the programmer supplies constitute a necessary but not sufficient justification of the plan. Both of these proposed scenarios are elaborated in Chapter Five.

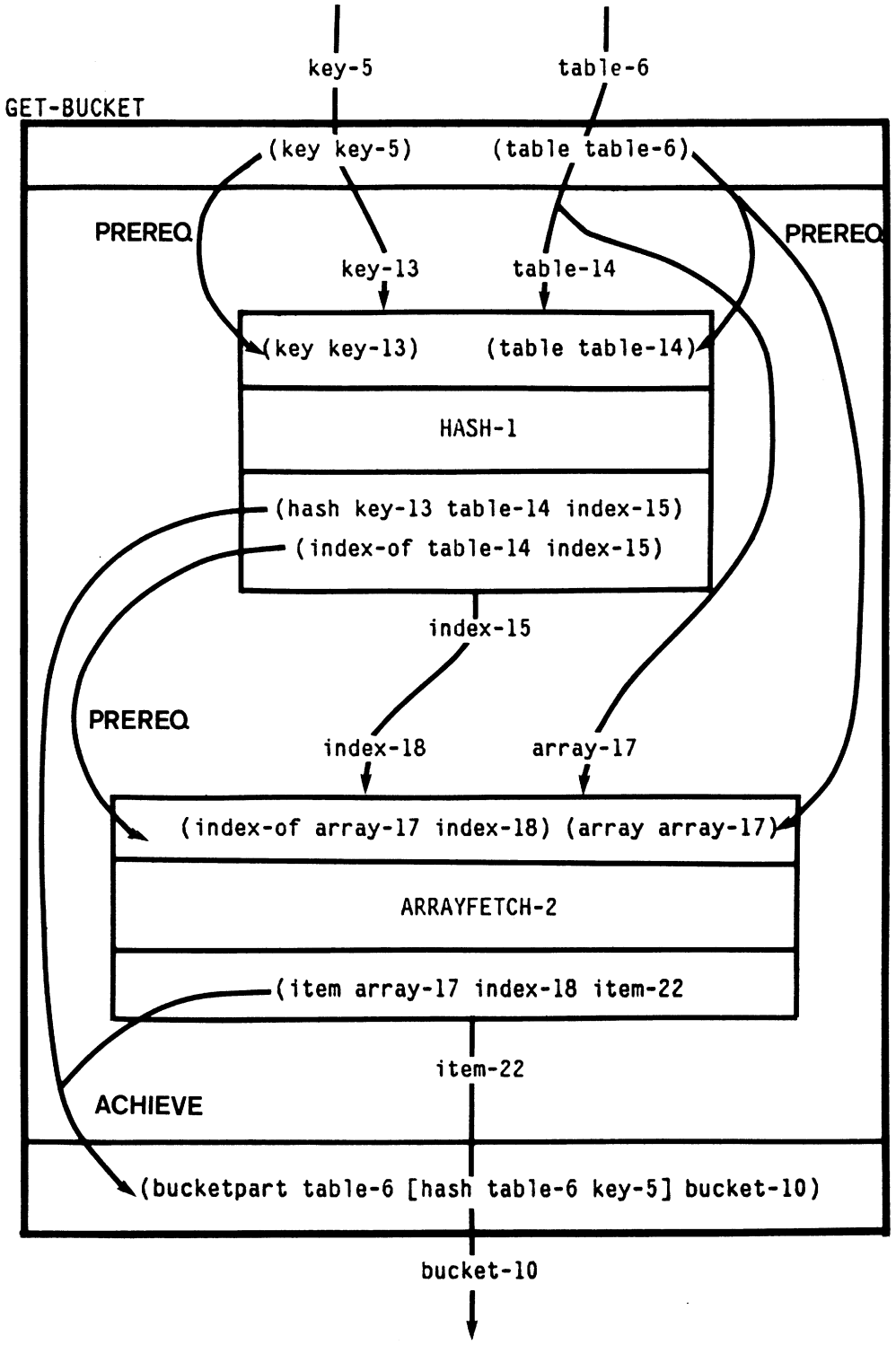


Figure 9. Deep Plan for GET-BUCKET.

Purpose Links and Control Flow

Purpose links form a set of constraints on sub-segment ordering (similar to Sacerdoti's <1975> procedural nets). For example, a PREREQ link between two sub-segments constrains one to precede the other, since a segment may not execute correctly until its prerequisite condition has been established.

A single segment may have several segments which establish prerequisites for it. For example, consider a plan (Figure 10) for intersection which takes two lists, sorts each one and then passes the two sorted lists to a fast-interesect segment which repeatedly compares the first items of each list. The fast-intersect segment has two input objects (the two lists) and two input expectations, namely that each list be sorted. Each of these expectations is satisfied by an output assertion of one of the sort segments.

Notice that although the two sort segments each have a prerequisite link to the fast-intersect segment, there are no purpose or data flow links between them, and thus no constraints on their relative ordering. Viewed from this level of abstraction, these two segments might execute in any order (or in parallel, though we are not for the moment giving any attention to the general issue of parallelism in programs), as long as they both precede the execution of FAST-INTERSECT. Thus purpose and data flow links do not always determine a unique total ordering of sub-segment execution; they form a partial ordering which is necessary (and if the purpose links constitute a complete justification, sufficient) for correct execution of the overall plan. Of course, once a program is actually coded, some totally ordered control flow must be chosen. For this reason we consider control flow in general to be part of the surface plan of a program, which will be described in Section 2.4.

A Relation Between Specs and Plans

The form of specs for a segment often gives a strong hint to the structure of a plan to implement those specs. For example, the main output assertion of GET-BUCKET is formed of a composition of relations. This form of specification is often implemented by plans which have a cascade of data flow, as is the case in the present plan example. Knowledge of such correlations between particular forms of specification and typical plan structures to achieve them can be used by the apprentice to aid in understanding programs and, eventually, to

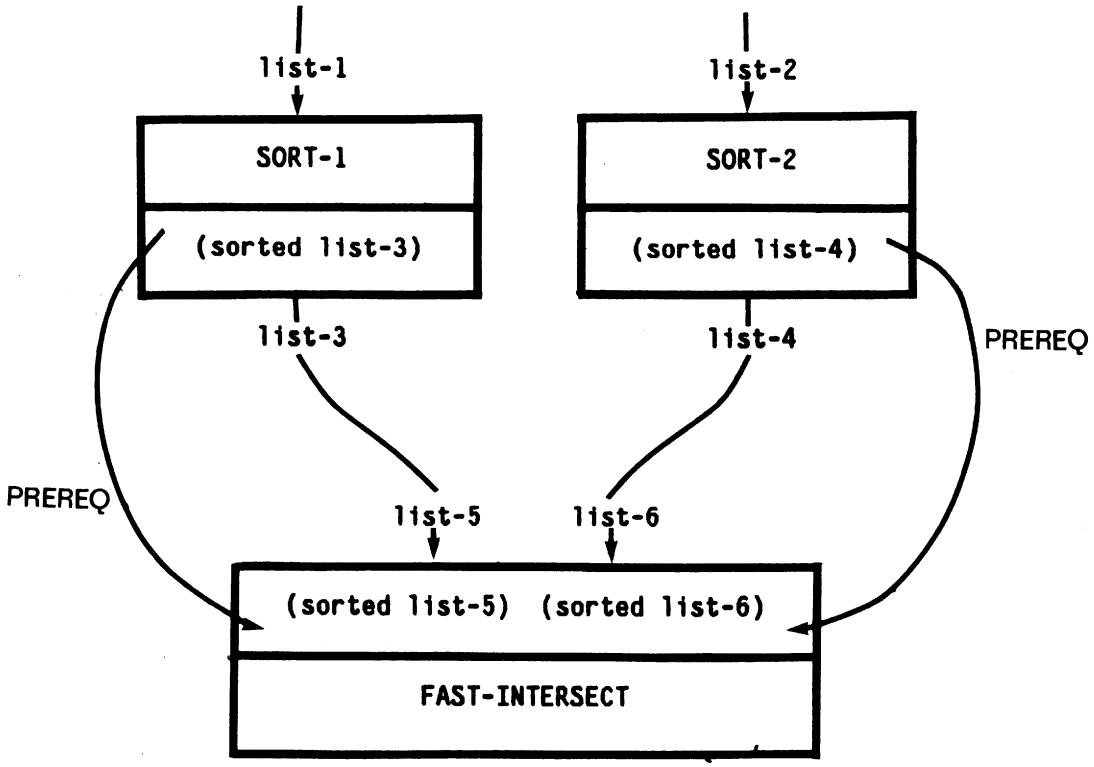


Figure 10. Plan With Multiple Purpose Links

generate plans automatically. For example, the use of certain quantifiers in specs strongly suggests certain types of "loop" plans: existential quantification is often achieved by search loops; universal quantification is achieved by "visit-each" loops. Hewitt <1975a> makes much the same observation in his proposal of a "behavioral semantics for logic". Further discussion of the apprentice's stored knowledge of general plan types appears in Section 2.5.

Deep Plans as Levels of Abstraction

Deep plan structure is an important abstraction which allows a program to be conceptualized independently of certain syntactic and implementation details. For example, the data flow links of a deep plan only state which objects move from one segment to another, not what programming language constructs are employed in realizing the data flow.

The use of specs to summarize the important behavior achieved by a given plan is also a kind of abstraction. When a segment is used in a larger plan, only its specs are relevant; its internal plan can be ignored. Furthermore, when specs are written using very general relations like membership, whose specific definition depends on implementation decisions, additional abstraction is obtained.

While omitting certain information about a program through abstraction, deep plans also contain additional knowledge which is not explicitly present in the fully coded program. One example of such knowledge is the segmentation of the code into conceptual units. (This point will be amplified in Section 2.4.) The teleological structure embodied in purpose links is also not always obvious from the program implementation. Finally, as we have seen, purpose links often refer to background knowledge of data structure, implementation choices, etc.

Thus to summarize, it should be clear that deep plans are not just a different way of representing the information already contained in a program, but rather a signficantly deeper form of knowledge. We believe that plans capture a programmer's underlying conceptualization of program structure -- the skeleton upon which code is eventually hung. A deep plan is realized as a concrete program by choosing a total ordering which is consistent with the constraints of the data flow and purpose links, and by using the available programming language primitives to implement lowest level segments, along with the control

and data flow between them. To emphasize this distinction between the underlying conceptual structure of a program and the more superficial structure influenced by the programming language, we have called the plans of this section *data plans*, in contrast with *surface plans*, which will be described in the following section.

2.4 Surface Plans

In this section we show how the apprentice represents the relationship between the underlying conceptual structure of a program, as expressed by the deep plan, and the structure of actual LISP code that is the realization of the program.

Why Have Code?

Given deep plans, which describe the programmer's intentions in a much more perspicuous form than raw code, one might be tempted to banish traditional programming languages entirely in favor or programming solely in deep plans or something similar. However, this will not be possible in the near future because of serious inadequacies in the current state of the art in specification languages (see Chapter Six). The search for specification techniques that encompass all the important design criteria used by practicing programmers is still going on.

Furthermore, our deep plan representation is intended to be a level of abstraction which ignores many implementation efficiency issues. Until we have a theoretical basis for dealing with time-space trade-offs and such, we cannot give the programmer any better way of expressing his efficiency-determined design choices than letting him actually write the detail of the code in the manner he wishes, as long as it is compatible with the deep plan. We feel this is the only realistic approach to building a usable programmer's apprentice system in the near future.

In order to be a useful LISP programmer's apprentice, our system must have significant knowledge of how LISP programs in particular are crafted. Thus, whereas deep plans are intended to be programming language independent, the surface plan representation in this section is tailored for LISP code. However, the basic approach used here could equally well be applied to constructing surface plan representations for programs written in other languages (see for example Waters' <1976> work on FORTRAN programs).

Building a Surface Plan and a CPD

A surface plan is created by augmenting the deep plan of a program with information on how parts of the deep plan are realized using features of the available programming language (in this case LISP). In this section we are primarily concerned with defining the surface plan representation. Chapter Four describes an algorithm for deriving surface plan information from given LISP code.

In general the building of a surface plan can occur in two ways, depending on the programmer's mode of interaction with the apprentice. With the most primitive apprentice, a programmer must provide both the deep plan and all the actual LISP code. In this mode, the apprentice first analyzes the LISP code bottom-up, as described in Chapter Four, and then attempts to "recognize" the correspondence between the deep structure and the surface structure, as described in Section 5.2. A more advanced apprentice, however, moves closer to automatic programming wherein, for some parts of the program, the programmer supplies only the deep plan and the apprentice generates the LISP code. In this mode, the surface plan is simply a record of how the apprentice made use of the programming language constructs to realize the control flow and data flow in the plan.

The net result of either of the above modes of interaction with the apprentice is what we call the complete program description (CPD). The CPD includes everything the apprentice knows about a program: the deep plan at all levels (which refers to data structure descriptions and other background knowledge), the surface plan, and the actual LISP code (including comments).

Code Segments and Plan Segments

The notion of a segment is as fundamental to the surface description of a LISP program as it is to the description of the underlying conceptual structure. Part of our implicit model of a programmer is that he will write a code segment corresponding to each segment in his deep plan. A code segment is simply a "meaningful" aggregation of program code, such as a function definition, a function invocation expression, or several lines of open code. A particular aggregation of code is "meaningful" in our system if it is considered by the programmer as a unit of behavior, i.e. if it corresponds to a segment at some level in the deep

plan. Since each plan segment has a corresponding code segment which is its realization, we will often use the unmodified term "segment" when the ambiguity is not confusing.

Code segments must be hierarchical in the same way that plan segments are, i.e. there are segments within segments, and sub-segments may be shared between larger segments. The possible sharing of plan segments means that code segments may also overlap. A particular code segment may be large or small, according to the level of detail required. For example, it may be appropriate for some purpose to consider the entire volume of code for a large system to be a single code segment. A more refined description, however, would break the code into its major sub-systems with a plan showing the links between them. This description could be successively refined until individual code segments become very primitive operations, such as CAR and CDR . Furthermore, the code for a segment need not appear contiguously on the standard program listing; for instance, a group of related function definitions, though scattered throughout a code listing, may at some level of description be considered a single code segment.

Since the goal of the apprentice is eventually to integrate its understanding of a program at all levels, surface plan information is represented as extra annotation distributed on the corresponding parts of a deep plan. To begin with, the code correspoinding to each plan segment is recorded as follows:

(<segment-id> SEGMENT { LAMBDA-EXP / FUNCALL / OPENCODE } <name> <code-entry> <code-entry> ...)

The <segment-id> above thus identifies both a deep plan segment and a corresponding code segment. The <code-entry>'s are pointers to actual LISP s-expressions. In examples we will indicate code-entries by enclosing the LISP s-expression in brace brackets thus: { (PROG ...) } .

We distinguish three kinds of code segments in LISP: lambda-expressions (named and unnamed), function invocation segments (uses of named lambda-expressions), and open code groupings. These are indicated respectively by the three alternative keywords LAMBDA-EXP, FUNCALL, and OPENCODE above. For named lambda-expressions (i.e. function definitions), and

for FUNCALL segments, the <name> field in the statement above is filled in with the appropriate atomic function name, to aid cross-referencing.

Instantiations of a particular segment type, e.g. HASH-SEGMENT, may be coded in any of these forms. For example:

(i)

```
(DEFINE HASH (K)
        (ABS (REMAINDER (SIZE TBL) K)))

(hash-88 SEGMENT LAMBDA-EXP hash {(define hash (k) ... )})
```

(ii)

```
( ... (HASH X) ... )

(hash-96 SEGMENT FUNCALL hash {(hash x)})
```

(iii)

```
( ... (ABS (REMAINDER (SIZE TBL) X)) ... )

(hash-90 SEGMENT OPENCODE nil {(abs (remainder ... ))})
```

Each of the above segments has the same behavior at the deep plan level, i.e. the same specs. The differences between them are important only for the surface structure of a particular program. For instance, in a single program there is usually only one DEFINE (i.e. named LAMBDA-EXP) segment for a given segment type; all FUNCALL segments have the same specs as the DEFINE segment with the same function name.

Surface Plans and Connective Tissue

The surface structure of a LISP program has two components: control flow and surface data flow. The control flow specifies which code segments follow one another in sequential execution, which segments are invoked as part of the execution of another (enclosing) segment, and which segments return control to another (enclosing) segment. The surface data flow in a program comprises the use of free variables, argument binding, and return values to achieve the flow of data objects between segments. The particular combinations of control flow and data flow that are possible in a given program are determined by the programming language.

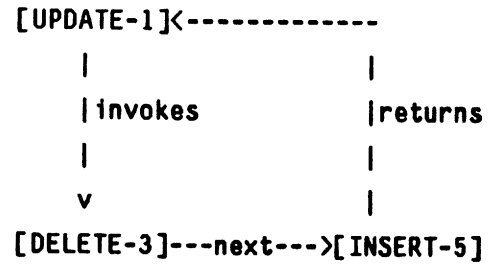
In LISP some control sequencing, such as the left-to-right order of evaluation of arguments, is built into the definition of the basic interpreter. However, other control flow facilities, such as PROG and COND, are implemented so that they appear syntactically as function calls; similarly for surface data flow primitives such as SETQ and RETURN. The P.A. views these special forms as connective tissue between the code segments in the program which actually "do" something (i.e. have specs). Thus, the P.A. never builds a FUNCALL segment for PROG, COND, SETQ, RETURN, etc. As will be shown in the following sections, these special forms serve only to determine the control and data flow relationships between actual program segments.

Surface Control Flow

The apprentice's representation of surface control flow between segments is based on three relationships: next, invokes, and returns. These particular concepts were chosen because they are the ones most human programmmers naturally use to describe control flow in single-process, recursive languages like LISP. The following LISP program illustrates surface control structure.

```
(DEFINE UPDATE (KEY DATA)
   (PROGN
      (DELETE KEY)
      (INSERT KEY DATA)))
```

Surface Control Flow:



In the P.A.'s data base the above information is encoded as part of the plan for UPDATE-1:

```
(update-1 SEGMENT LAMBDA-EXP update {(define update (...) ...)})
(delete-3 SEGMENT FUNCALL delete {(delete ...)})
(insert-5 SEGMENT FUNCALL insert {(insert ...)})

(PLAN-FOR: update-1
    (SUB-SEGMENTS: (delete-3 insert-5))
    ...
    (INVOKES: delete-3)
    (NEXT: delete-3 insert-5)
    (RETURNS: insert-5)
    ... )
```

Connective tissue for control flow generally falls into two categories: sequencers and groupers. A sequencer is a special form such as PROGN or GO which causes sequencing of execution between LISP forms that are not nested. A grouper is a special form, such as LAMBDA or DEFINE, which allows the execution of a number of sub-forms to be grouped together for net effect. Of course, many special forms are both sequencers and groupers.

Figure 11 illustrates more complicated control flow, which is part of the surface plan for the following program. (This particular example will be returned to in Chapters Four and Five).

```
(DEFINE LOOKUP (KEY)
        (PROG (BKT)
              (SETQ BKT (TBL (HASH KEY)))
          LP  (OR BKT (RETURN NIL))
              (COND ((EQ (CAAR BKT) KEY)
                     (RETURN (CAR BKT))))
              (SETQ BKT (CDR BKT))
              (GO LP)))
```

In this example, some of the control flow in the plan depends on cases. Optional case identifiers are added to NEXT and RETURNS statements to signify that the indicated control flow takes place only for a given case of the sub-segment involved. To indicate control flow which pertains only to a given case of the super-segment, the PLAN-FOR expression itself can be broken into cases, just like a SPECS-FOR. For example, from Figure 11:

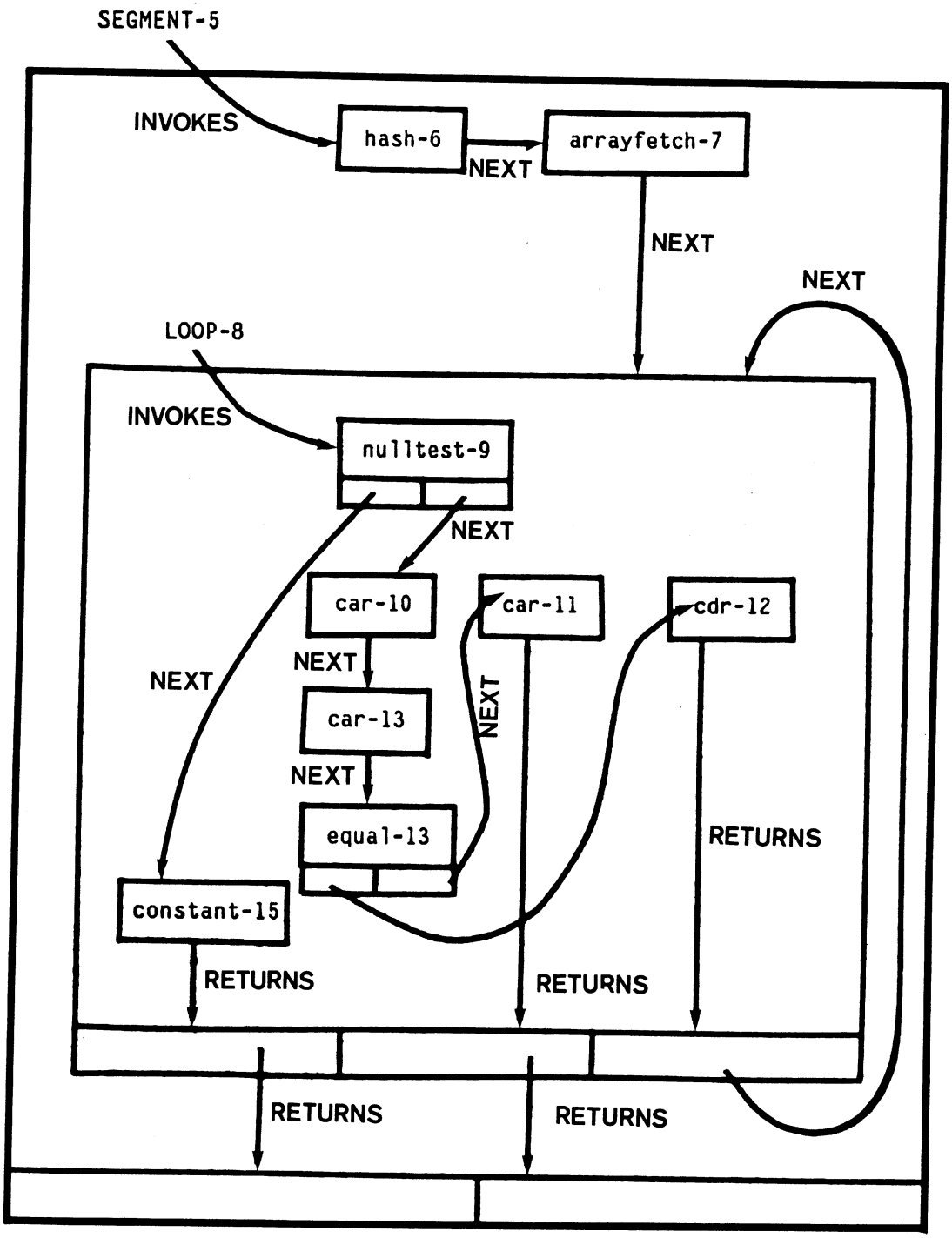


Figure 11. Control Flow for LOOKUP.

```
(PLAN-FOR: loop-8
    ...
    (INVOKES: nulltest-9)
  (CASE-1
    ...
    (NEXT: nulltest-9 constant-15 case-1)
    (RETURNS: constant-15))
  (CASE-2
    ...
    (NEXT: nulltest-9 car-10 case-2)
    (NEXT: car-10 car-13)
    (NEXT: car-13 equal-14)
    (NEXT: equal-14 car-11 case-1)
    (RETURNS: car-11))
  (CASE-3
    ...
    (NEXT: equal-14 cdr-12 case-2)
    (RETURNS: cdr-12)))
```

Surface Data Flow

WE now reconsider the three types of data flow as they are realized in the code for a program: data flow in and out of a single code segment, data flow between code segments at the same level of description, and data flow between sub-segments and the enclosing main segment.

In LISP there are four basic techniques for moving data between code segments: variables (i.e. SETQ), nesting of s-expressions (i.e. return values and argument binding), property lists, and side effects on CONS cells (RPLACA and RPLACD) or arrays (STORE). Of these we have currently fully treated only the use of variables, arguments, return values, and the use of arrays. Our current reasoning system (Chapter Three) is able to deal with side effects on list structures in a quite powerful way, but we are still not sure of the best way to represent the surface plans of

programs using RPLAC's. We have not yet paid any attention to property lists, but it seems to us that they should not be any harder to handle than RPLAC's.

Thus in the subset of LISP we are working with, there are syntactically two ways that a data object may be input to a code segment, and two ways it may be the output. A data object can be input either as the value of a free variable or, if the segment is a FUNCALL or a LAMBDA-EXP, as the bound value of a argument. On output, a data object can either be the value of a free variable after the segment has executed or the return value of an s-expression. Different instances of the same segment type can differ in the surface data flow of their input and output objects. For example, consider the input and output objects of a segment type which computes the hash of a key.

```
(SPECS-FOR:  hash
    (INPUTS: key-13 table-14)
    (OUTPUTS: index-15)
    ... )
```

The following two LISP code segments are alternative instantiations of this segment type which differ in their surface data flow. (Of course, only one of the following definitions would appear in a given program.)

```
(DEFINE HASH (K)                    (DEFINE HASH (K L)
  (ABS                                (ABS
    (REMAINDER (SIZE TBL)               (REMAINDER (SIZE L)
               K)))                                K)))

(hash-88 SEGMENT LAMBDA-EXP hash {(define hash (k) ...)})
(hash-92 SEGMENT LAMBDA-EXP hash {(define hash (k l) ...)})
```

In the code on the left, the hash table is input to the segment as the value of the free variable TBL; in the code on the right, however, the same data object is input as the bound value of the second lambda argument, L. In both of them the index is output as the return value of the lambda-expression. It is important for the apprentice to keep track of this sort of variation in surface data flow, since it is the cause of many careless programming bugs. This information is recorded as extra annotation on the input and output statements of the specs for

a segment. (Note here that each input object is specified in a separate specs clause, a slight variation from the initial definition of specs in Section 2.2.)

```
(SPECS-FOR: hash-88
    (INPUT: key-89 ARG {k})
    (INPUT: table-90 FREE-VAR {tbl})
    (OUTPUT: index-91 RETURN-VAL {(define ...)}))

(SPECS-FOR: hash-92
    (INPUT: key-93 ARG {k})
    (INPUT: table-94 ARG {1})
    (OUTPUT: index-95 RETURN-VAL {(define ...)}))
```

The brace brackets in each line above show the relevant code: for FREE-VAR data flow, the variable involved; for ARG inputs, the argument position; for RETURN-VAL's, the s-expression whose return value is the data object.

FUNCALL code segments derive their surface input and output data flow from the corresponding function definition. Thus in a program which used the left hand definition above , HASH-88, a function call would appear as follows:

```
(HASH ... )

(hash-96 SEGMENT FUNCALL hash {(hash ...)})

(SPECS-FOR: hash-96
    (INPUT: key-97 ARG {...})
    (INPUT: table-98 FREE-VAR {tbl})
    (OUTPUT: index-99 RETURN-VAL {(hash ...)}))
```

Data Flow Between Segments

Surface data flow between segments at the same level of description is achieved by matching data flows in and out of code segments. For example, in LISP two code segments can communicate data by using the same free variable. Nestings of s-expressions (i.e return values and lambda-binding) is the other common way to move data between segments in LISP. Each data flow link in the deep plan for a program will have a corresponding surface data flow mechanism in the surface plan. Let us first consider the two simple cases which do not require extra connective tissue:

(i) Same Free Variable

When the same free variable is used for "both ends" of a data flow link, the two segments communicate directly. To continue the above example, suppose the function call HASH-96 was used in a program together with a function call segment CREATE-TABLE-101 which has the following surface data flow:

```
(SPECS-FOR: create-table-101
    ...
    (OUTPUT: table-102 FREE-VAR (tbl))
    ...)

(create-table-101 SEGMENT FUNCALL crtable ((crtable ...)))
```

In the code then we would see the following:

```
(CRTABLE ...)
 ...
(HASH ...)
```

These are two FUNCALL segments which have an output-input data flow link between them. Just as the surface mechanism for data flow into and out of a single segment is recorded as extra annotation on the corresponding INPUT and OUTPUT statement in the specs, surface data flow between segments is recorded on the corresponding DATAFLOW statement in the plan of which they are sub-segments. For the present example, let SEGMENT-33 be the unspecified

segment which the above code is a part of.

```
(PLAN-FOR: segment-33
    (SUB-SEGMENTS: (create-table-101 hash-96 ... ))
    ...
    (DATAFLOW: (expand-table-101 OUTPUT table-102 FREE-VAR {tbl})
               (hash-96 INPUT table-98 FREE-VAR {tbl})
               SAME-FREE-VAR)
    ... )
```

(ii) Nested S-Expressions

The other simple way to achieve output-input data flow between segments is by nesting the corresponding s-expressions and arranging that the desired data object is output as the return value of one segment and input as an argument to the other. Expanding on the current example:

```
(HASH (KEYPART ...))

(hash-96 SEGMENT FUNCALL hash {(hash ...)})
(keypart-65 SEGMENT FUNCALL keypart {(keypart ...)})

(SPECS-FOR: keypart-65
    ...
    (OUTPUT: key-66 RETURN-VAL {(keypart ...)})
    ...)

(PLAN-FOR:  segment-33
    (SUB-SEGMENTS: (... hash-96 keypart-65 ... ))
    ...
    (DATAFLOW: (keypart-65 OUTPUT key-66 RETURN-VAL {(keypart ...)})
               (hash-96 INPUT key-97 ARG {(keypart ...)})
               NESTED-SEXP)
    ...)
```

Data Flow Coupling

Often when formulating a plan using existing code segments the surface input and output data flows of two segments will not match directly. For example, two code segments might use different free variables for the same data object. In such cases, the programmer will typically use data flow couplers in LISP such as SETQ and RETURN to match the segments. (The term "coupling" is by analogy with the interfacing of electronic modules in circuit design.) Data flow couplers are part of the connective tissue of a program.

To illustrate coupling, suppose the programmer of SEGMENT-33 had already written the function definition for HASH which expects the table as the value of the free variable TBL , and a different definition of CRTABLE which used the free variable TABLE. The surface data flow can be coupled using SETQ as follows:

```
(CRTABLE ...)
 ...
(SETQ TBL TABLE)
(HASH ...)

(PLAN-FOR: segment-33
    ...
    (DATAFLOW: (create-table-201 OUTPUT table-202 FREE-VAR {table})
               (hash-96 INPUT table-98 FREE-VAR {tbl})
               COUPLING {(setq tbl table)})
    ... )
```

Obviously there are numerous other kinds of data flow coupling, corresponding to other possible cases of mismatch between the surface inputs and outputs of two code segments. Examples of a few more of the common cases should suffice.

When an output object is the return value of a code segment and it is not possible to nest s-expressions, a common coupling technique is to use a variable to store the output object. Then if the object is input to the destination segment as an argument, the variable is used in appropriate position. For example, the surface data flow between KEYPART and HASH in the earlier example could have been:

```
(SETQ X (KEYPART ...))
...
(HASH X)

(PLAN-FOR: segment-33
    ...
    (DATAFLOW: (keypart-65 OUTPUT key-66 RETURN-VAL {(keypart ...)})
               (hash-96 INPUT key-97 ARG {x})
               COUPLING {(setq x (keypart ...))})
    ... )
```

All the examples of surface data flow thus far have been between segments at the same plan level. There is also data flow between a segment and its sub-segments. For example, suppose the code for SEGMENT-33 is being used in a larger surface plan in which its return value should be the hash table. The hash table is available inside SEGMENT-33 as the free variable TBL. In such a situation, the programmer typically uses PROG - RETURN as the coupling mechanism:

```
(PROG (...)
      ...
      (CRTABLE ...)
      ...
      (RETURN TBL))

(segment-33 SEGMENT OPENCODE nil {(prog ...)})

(SPECS-FOR: segment-33
    ...
    (OUTPUT: table-34 RETURN-VAL {(prog ...)})
    ... )

(PLAN-FOR: segment-33
    (SUB-SEGMENTS: (create-table-101 ...))
    (DATAFLOW: (create-table-101 OUTPUT table-102 FREE-VAR {tbl})
               (segment-33 OUTPUT table-34 RETURN-VAL {(prog ...)})
               COUPLING {(return tbl)}))
```

Some special forms in LISP have "built-in" coupling, e.g. PROGN returns the value of its last form. Thus an alternative coding of the above might be:

```
(PROGN
   ...
   (CRTABLE ...)
   ...
   TBL)

(PLAN-FOR: segment-33
    ...
    (DATAFLOW: (expand-segment-102 OUTPUT table-103 FREE-VAR {tbl})
               (segment-33 OUTPUT table-34 RETURN-VAL {(progn ... tbl)})
               COUPLING {(progn ... tbl)}))
```

More complicated coupling situations are represented by more complicated constructions following the keyword COUPLING.

Knowledge Specific to LISP

Figure 12 is a summary of our current representation for the surface structure of LISP programs. As mentioned previously, the specs and deep plan formalism developed in Sections 2.2 and 2.3 are intended to be independent of particular programming languages. The basic idea of a surface plan as presented in this section is also quite general. In other languages, such as FORTRAN, ALGOL, and COBOL, we expect to find the same notions of control sequencing, grouping, data flow coupling, and connective tissue recast in different syntactic forms.

In order to understand LISP programs, the apprentice needs two classes of LISP-specific knowledge. First it should be initialized with the specs and surface data flow information for built-in segments like CAR, CDR, CONS, PLUS, EQUAL, and many other functions that are considered a part of basic LISP. Notice that these built-in segments are not treated as primitives; rather they are described using specs in exactly the same way as user-defined segments.

Another large body of LISP-specific knowledge is implicit in the procedures the apprentice uses for building the surface plan for new LISP programs. Knowledge about the operation of the LISP interpreter and the meaning of special (FEXPR) forms such as PROG and COND falls in this category. We currently have no theoretically motivated discipline for encoding this kind of knowledge.

Implementation Note: The Code Table

The P.A.'s use of surface plans requires having the ability both to point at arbitrary s-expressions in LISP code from statements in its knowledge base and, given an s-expression, to retrieve any comments or knowledge assertions referring to it. These requirements, coupled with our decision to make the P.A. transparent to the standard LISP interpreter, dictated the implementation of an auxiliary indexing structure called the code table.

```
(SPECS-FOR: <segment-id>

    (INPUT: <object-id> {FREE-VAR} <code-entry> )
                        {  ARG   }

    (OUTPUT: <object-id> {FREE-VAR  } <code-entry> )
                         {RETURN-VAL}
                                    )


(PLAN-FOR: <segment-id>

    (DATAFLOW: (<segment-id> {INPUT } ... )
                             {OUTPUT}

               (<segment-id> {INPUT } ... )
                             {OUTPUT}

              {SAME-FREE-VAR} <code-entry> )
              {NESTED-SEXP   }
              {COUPLING      }

    (NEXT: <segment-id> <segment-id> <case-id> )

    (INVOKES: <segment-id> )

    (RETURNS: <segment-id> <case-id> ) )
```

Figure 12. Summary of Surface Plan Representation

As a new LISP program is read into our current apprentice, an entry in the code table (a code-entry) is created in a recursive fashion for each s-expression in the code. A code entry has three fields: the first field is a pointer to the actual s-expression, which is used as the key to retrieve the entry from the table; the second field points to the code-entry of the parent s-expression, which facilitates the P.A.'s moving around locally in the LISP code; and the third field is one bit, denoting whether the current s-expression is the CAR or CDR of its parent.

Thus whenever statements in the surface plan refer to something in the code, such as a function name, a lambda-argument, etc., the apprentice always points to the code indirectly using the corresponding code-entry. This indirection has been indicated syntactically in this report by enclosing the code in brace brackets {...}.

2.5 The Organization Of Programming Knowledge

The elements of program description outlined above provide a convenient language for representing the structure and behavior of particular programs. However, a useful programmer's apprentice will also need a large knowledge base of common and general programming concepts. We do not yet completely understand how to build this knowledge base; nonetheless, we feel that the descriptive elements already developed for particular programs will provide an adequate basis for representing more general knowledge as well. In this section we present our current thoughts on the design of such a programming knowledge base.

Our design has two major considerations: first, the knowledge base should contain most of the standard programming concepts, object types, segment types, and plans in common use; second, it should be structured so as to capture the significant generalizations of the domain. Our discussion will concentrate on the issues of overall organization, ignoring many technical problems of large-scale data base design which are currently being researched elsewhere.

Programming Concepts

There seems to exist a general hierarchy of programming concepts which may be used as the basis for structuring part of the knowledge base. Within this hierarchy, as much knowledge as possible is to be captured at the most general level. For example, a very abstract concept is data-structures, which subsumes all object types that contain a varying number of members. This general concept also includes the membership relation and the most general specs for lookup, insert and delete.

Most concepts are specializations of more general concepts. For example, lists is a specialization of data-structures. Some knowledge about lists comes from the fact that lists are data-structures, while some of it is particular to lists. The parts decomposition of lists into FIRST and REST, and a definition of the MEMBER relations is particular to lists, while the specs for insert, lookup and delete are essentially the same for lists as for data-structures in general (see Figure 13).

DATA-STRUCTURES

```
(SPECS-FOR: insert-segment
    (INPUTS: object-1 object-2)
    (EXPECT: (data-structure object-1))
    (OUTPUTS: object-3)
    (ASSERT:
     (data-structure object-3)
     (member object-3 object-2)
     (for-all (member object-1 =object)
              (member object-3 object))))
```

```
(RELATION data-structure member)
```

LISTS

```
(SPECS-FOR: insert-segment
    (INPUTS: object-1 object-2)
    (EXPECT: (list object-1))
    (OUTPUTS: object-3)
    (ASSERT:
     (list object-3)
     (member object-3 object-2)
     (for-all (member object-1 =object)
              (member object-3 object))))
```

```
(PART list first)
(PART list rest)
```

```
(RELATION list member)
(RELATION DEFINITION
 (member list object) <=>
 (or (first list object)
     (member [rest list] object)))
```

Figure 13. Lists as a Specialization of Data-Structures.

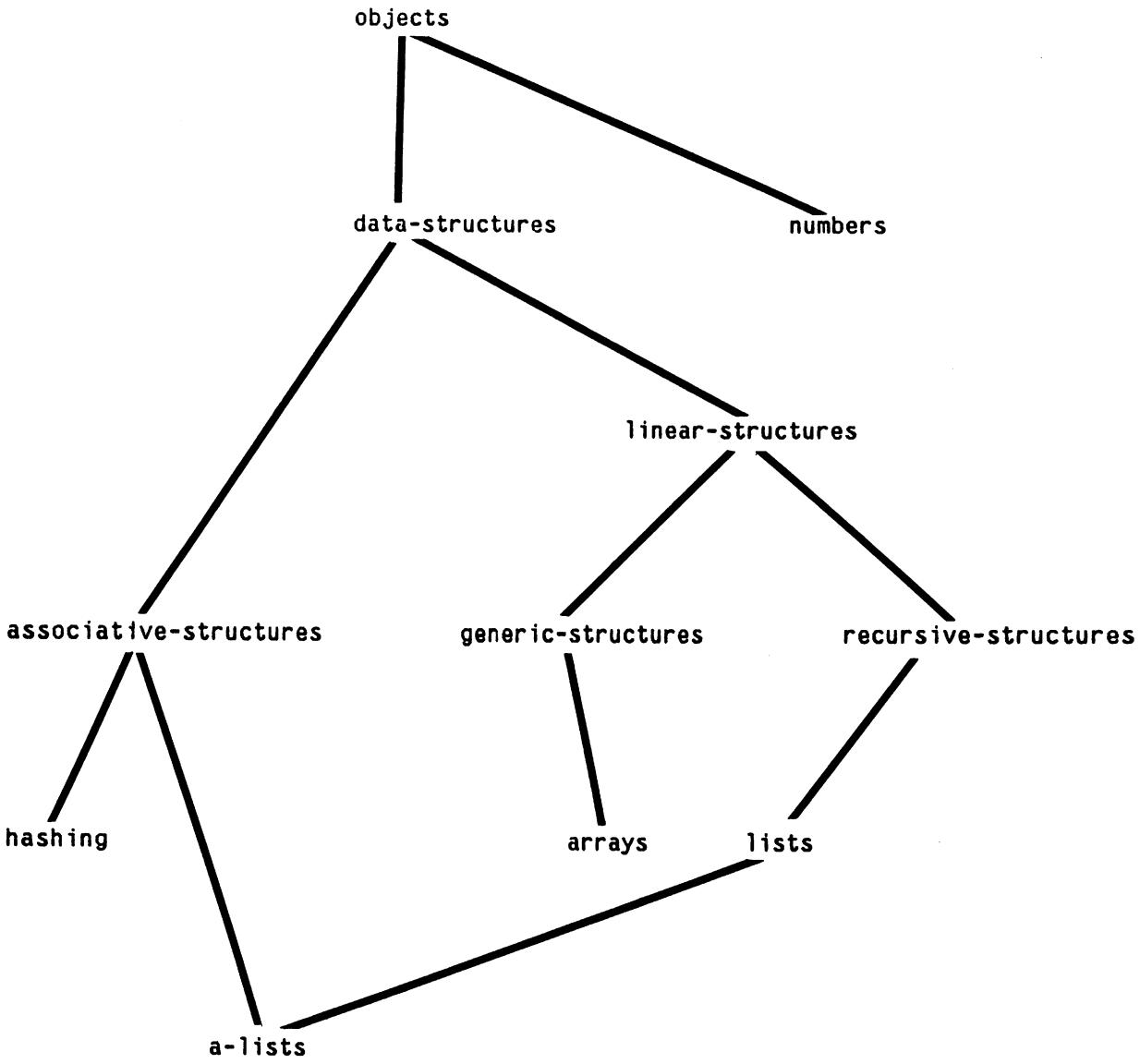


Figure 14. Partial Hierarchy of Programming Concepts

General concepts can be specialized in a multiple-level, tree-like fashion (see Figure 14). For example, one specialization of data-structures is associative-structures. Associative-structures contain members called entries which are decomposed into KEYPART and DATAPART. The specs for lookup and delete operations on associative-structures differ from those on data-structures in general, in that associative retrieval is usually performed given only the KEYPART of an entry. (Hash tables and a-lists are associative-structures.) A second possible specialization of the data-structures concept is linear-structures. Arrays and lists are examples of linear-structures, since they both have natural total orderings of their member objects. Notice in Figure 14 that we expect the hierarchy of programming concepts to be "tangled", i.e. some concepts may have several possible generalizations.

Plan Types

In addition to the hierarchy of concepts there also seems to be a hierarchy of plan types which captures generalizations of procedural structure. An obvious example of a very general plan type is the plan for a loop (see Figure 15). Specific kinds of loops, such as search loops or approximation loops, can be considered to be refinements of this general plan.

In its most abstract, a loop has four essential segments (see Figure 15): a TEST on the current loop object to see if the loop is done; a BUMP which calculates the loop object for the next iteration from the current one; the BODY , which does something to the loop object; and the tail recursion segment, LOOP-1 in the diagram, which achieves the iteration. The actual specs for these sub-segments are not shown, since they are too abstact to be useful -- for example, the BODY has no particular input or output conditions at this level.

However, if this plan is refined to be a search loop, the specs for the body become the test for a sucessful match. Thus the following specs, together with the plan in Figure 16, express the most general structure of iterative searching.

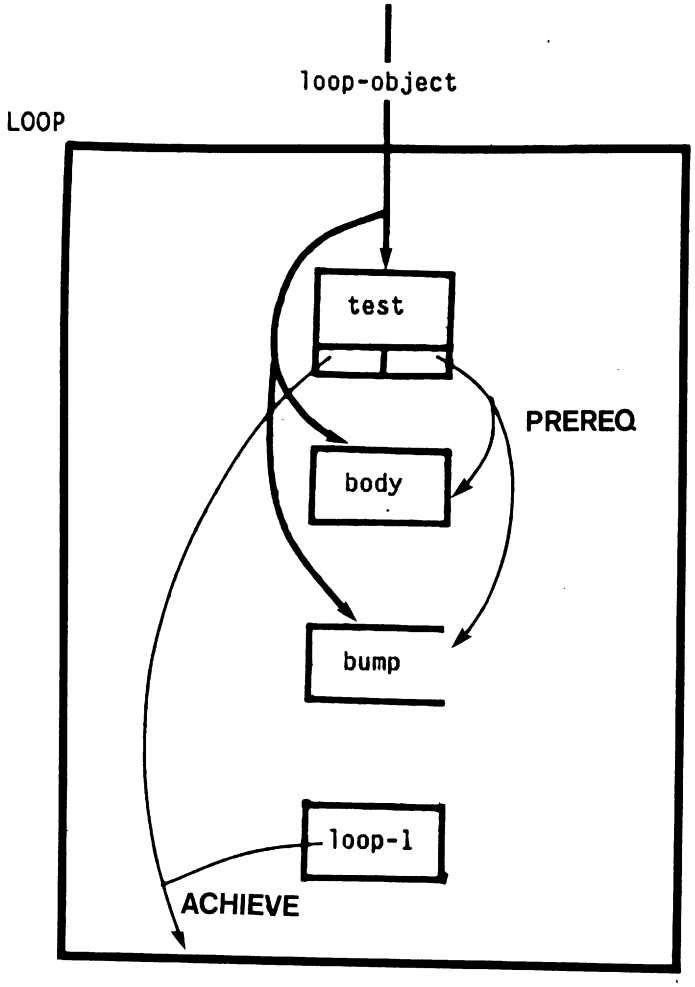


Figure 15. Most General Plan for Loops

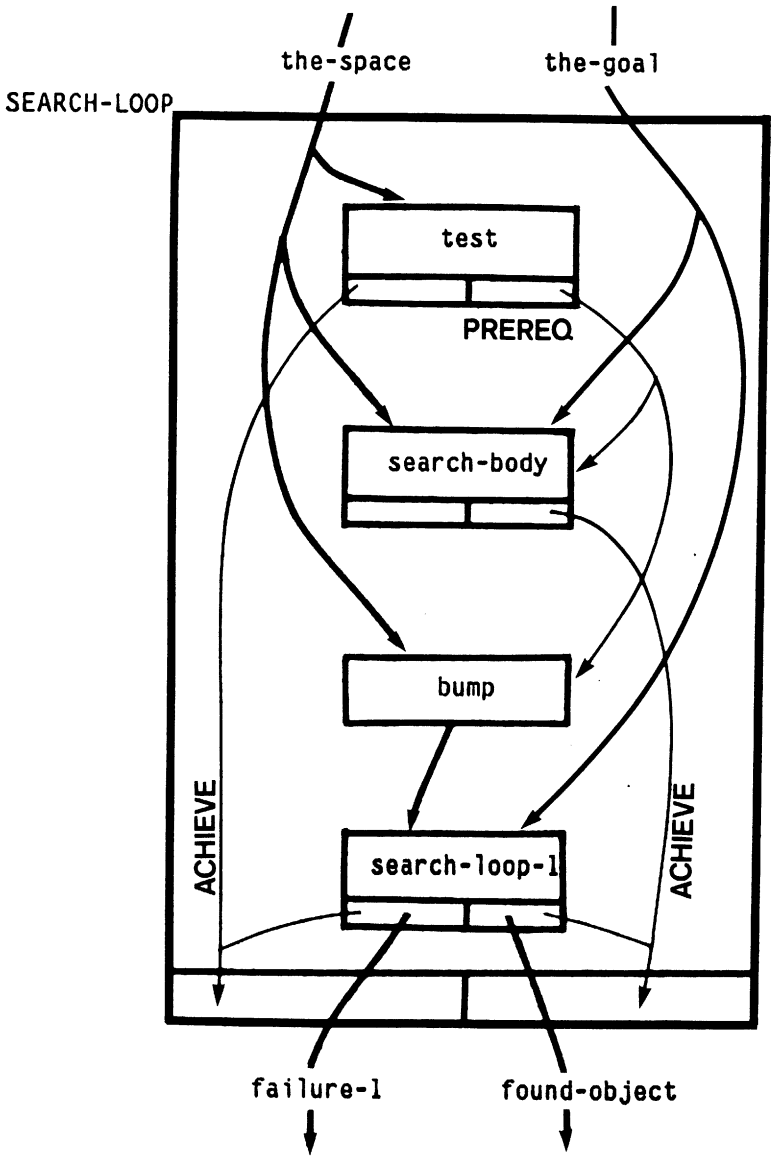


Figure 16. Plan for Search Loop.

```
(SPECS-FOR: search-loop
    (INPUTS: the-space the-goal)
    (EXPECT: (linear-structure the-space))
  (CASE-1
    (EXPECT: (there-is-a (member the-space =object)
              such-that (matchtest the-goal object)))
    (OUTPUTS: found-object)
    (ASSERT: (member the-space found-object)
             (matchtest the-goal found-object)))
  (CASE-2
    (EXPECT: OTHERWISE)
    (OUTPUTS: failure-1)
    (ASSERT: (for-all (member the-space =object)
                      (not (matchtest the-goal object))))))


(SPECS-FOR: search-body
    (INPUTS: the-space the-goal)
    (EXPECT: (linear-structure the-space))
  (CASE-1
    (EXPECT: (matchtest the-goal [front the-space]))
    (OUTPUT: the-object)
    (ASSERT: (front the-space the-object)
             (matchtest the-goal the-object)))
  (CASE-2
    (EXPECT: OTHERWISE)
    (OUTPUTS: failure-1)
    (ASSERT:  (not (matchtest the-goal [front the-space])))))
```

Notice that since the search space is known to be a linear-structure, there must be a total ordering of its members. Therefore, we may talk about its front element, even though we have not yet chosen a particular linear-structure. We will talk about how such design choices interact with plan structure after a few other remarks.

In one sense the plan for a search loop in Figure 16 is fully refined, since all of the purpose links are filled in. Thus the addition of further detail, such as the definition of MATCHTEST, will not change the teleological structure of this plan. However, further refinement may take place within the internal plans of the sub-segments, while the top level purpose links will remain unchanged.

In another sense, there is a great deal more that can be said about search plans. For example, the object to be searched might be a list or an array. In the case of lists, the bump step is implemented by taking the list's CDR, while the CAR of the list is its "front" member. In the case of arrays, the bump is implemented by adding one to the index, while the front member is defined to be the item pointed to by the current index. Similarly, if the search space is an associative-structure such as an a-list, then the MATCHTEST will have an internal plan which extracts the key from the current object and then makes the test for a match. Thus as an implementation choice is made for each object in the general plan, the specs for the sub-segments can be further specialized (in a consistent manner) and sometimes internal plans can be chosen.

For example, the specs for the sub-segments of a plan to search lists are as follows:

```
(SPECS-FOR: search-body
    (INPUTS: list-1 the-goal)
    (EXPECT: (list list-1))
  (CASE-1
    (EXPECT: (matchtest the-goal [first list-1]))
    (OUTPUTS: the-object)
    (ASSERT: (first list-1 the-object)
             (matchtest the-goal the-object)))
  (CASE-2
    (EXPECT: OTHERWISE)
    (OUTPUTS: failure-1)
    (ASSERT:  (not (matchtest the-goal [first list-1])))))
```

```
(SPECS-FOR: bump
    (INPUTS: list-1)
    (EXPECT: (list list-1))
    (OUTPUTS: list-2)
    (ASSERT: (rest list-1 list-2)))

(SPECS-FOR: test
    (INPUTS: object-1)
  (CASE-1
    (EXPECT: (list object-1)))
  (CASE-2
    (EXPECT: (emptylist object-1))
    (OUTPUTS: failure-2))
```

Design Choices

Since we do not yet have a theory of the interaction between the specialization of data objects and the refinement of plans, our system cannot automatically create new specializations and corresponding refinements of plans. However we can compile this information for those cases which are common and already known. In this pre-compiled knowledge, the appropriate plan can be selected by specifying the design choice. Since we intend to implement the programming knowledge base using a CONNIVER-like data base, the design choice can be represented as a context in which the plan is already appropriately refined, inheriting some assertions from more general contexts and adding those new assertions which are necessary.

The context mechanism can also be used to represent the existence of several alternative ways to implement a concept or several different plans for the same specs. For example, stacks can be implemented using arrays or lists. Thus there would be a parent context in which all the general knowledge about stacks is kept and two daughter contexts each containing the statements appropriate to that design choice. Similarly, there would be a daughter context for each of the large number of sorting algorithms. Finally, each of these plans might have daughter contexts representing their refinements under the choice of specific types for the

objects in the plan. Thus the context tree would encode a range of plans from the most abstract, like that for LOOP, to the completely refined plan of an actual program.

Plan Transformations

There is yet another form of variation in plans that is not captured naturally in a refinement hierarchy. This variation arises from the fact that certain plans are strongly equivalent even though their plan representations are different. For example, a summation program which counts up is in many ways equivalent to one which counts down -- the same essential teleology is involved in both plans, although the data flow and segments are slightly different. Such variations can be captured in the knowledge base by considering these plans to be equivalence classes, and choosing a canonical plan for each class. This canonical plan can then be tranformed to any of the other equivalent plans by applying a plan tranformation. (Ruth <1973> has a similar notion.)

For example, consider the plan for searching a list given above. In the specs for the SEARCH-BODY segment there is the clause:

```
(EXPECT: (matchtest the-goal [first list-1]))
```

which implies that the plan for SEARCH-BODY is a cascade of applying FIRST-SEGMENT and then MATCHTEST . However, since the FIRST-SEGMENT in this little sub-plan depends on nothing other than the overall input expectations, it is possible to transform the plan by moving FIRST-SEGMENT out of SEARCH-BODY, thus reducing its internal plan to just MATCHTEST. The transformed plan for SEARCH-LOOP will then have five internal steps (FIRST-SEGMENT, MATCHTEST, BUMP, TEST, LOOP) instead of the four in the original plan.

Another simple transformation is to combine two sub-segments of a plan into one segment whose sub-plan comprises the two original segments. For example, in the plan for a hash table lookup (see Figure 11) the HASH and ARRAYFETCH could be combined to form a GET-BUCKET segment. Other simple transformations under certain circumstances is to combine two instances of the same segment type into a single instance, or to copy a single instance into two.

2.6 Annotation of Programs

In the preceding sections we have outlined a system for describing programs. In later chapters we will show how the representations we have developed are adequate to support question answering, reasoning about program correctness, program formulation and evolution. However, an additional criterion which we desire our descriptive system to meet is naturalness, i.e. correspondence to the way people conceptualize the domain. Our initial beliefs regarding the conceptual foundations of programming (from an anthropocentric perspective) arose out of examination of the programming practices of ourselves and other programmers. In this section we return to the existing practices of human programmers to reaffirm that the commentary they typically write indeed fits into the descriptive system we have developed.

Understanding previously unseen code can be extremely difficult, even for an expert programmer. It is precisely for this reason that programmers attach annotation to their code in the form of mnemonic names and line-by-line commentary, which provide various kinds of extra information necessary to understand the code.

Annotation of code can be divided into two general categories. First, there are comments that assume knowledge shared between the programmer and the reader, i.e. that the code involves some concept the reader is already familiar with. The other category concerns precisely the opposite situation: comments which attempt to describe new concepts. Because this second class of commentary is intrinsically more demanding of the programmer, it is also the most frequently neglected.

Comments That Invoke Shared Knowledge

When a programmer intends a comment to refer to shared background knowledge, it is often done most simply through the use of mnemonic identifiers. Thus, for example, the lookup routine for the hash table would typically be called LOOKUP, or some variant, rather than say FUNC1. The important point here is that such namings do not convey information unless there is shared knowledge within which to resolve their significance. In a later version of the apprentice we expect to use the knowledge base described in Section 2.5 in this lexical role.

Another form of commentary which refers to shared knowledge is comments such as "this is the hash table lookup routine" or "the next five functions make up the hash table". Typically, this kind of comment appears preceding a code segment as an introductory remark. These comments, like mnemonic naming, serve to set context and supply necessary but otherwise unstated information. For example, the comment "this is the hash table lookup routine" would ideally have the effect of retrieving the specs for LOOKUP and invoking the context of the HASHING concept for resolving further name references. Similarly, comments can also invoke a particular design choice from the knowledge base. For example, "using the rehash scheme" has the effect of telling the reader which of several possible implementation schemes is being used.

New Information Comments

The second broad category of annotation occurs in those circumstances when the programmer wants to convey new information that cannot be assumed to be part of the reader's basic knowledge. In this case, he has to completely define a new concept on the page. The comments that programmers use for this purpose fit well into the descriptive system of our P.A.: they answer why, how, and what questions by presenting what are essentially plans, specs, part descriptions, and various definitions.

The following are examples of comments that answer why questions: " X is positive so SQRT won't fail" or "set up for first pass". What typifies these comments is that they establish a link between the behavior of the code upon which they appear and some other segment to which they refer. Thus, they correspond directly to the purpose links in our plans. (These two examples are PREREQ links). PREREQ links can also appear in comments viewed from the opposite direction, e.g. on an invocation of SQRT the comment, "X positive because ABS made it so".

How questions tend, by and large, to be answered by reference to shared background knowledge, such as the standard design choices for implementing a hash table. However when a new concept is involved in a "how" comment, a large explanatory block can appear explicitly on the page; for example, "use the hash routine to get a bucket, then use the a list insert to insert the item". It would be an interesting problem to expand such a description into a complete plan.

What comments typically provide all or part of a structural description or a specs. For example, in an interactive bibliography program, we observed a half-page comment which explained the structure and use of an entity called a "prompt". The details of this are not relevant here, but it was interesting for us to note that the comment included precisely those descriptive elements which our P.A. uses, namely specs, parts decompostions, etc. The data structures defined in such situations often have only limited application, i.e. they are used only in one section of a program and are not likely to be used in future programs.

We have often observed comments that explicitly give the complete specs in a block of commentary immediately preceding the code segment, as in: "when given a list, return its third element if present; otherwise return FOO". Specs are also sometimes broken up into the components pre- and post-conditions and distributed through the body of a code segment; for example, "assume A negative", or "now the item is in the table". The first comment specifies a condition that is assumed or expected to hold just prior to the execution of the segment of code which it annotates, presumably because the correct behavior of the code depends on it, i.e. it is an EXPECT; the second comment asserts that a certain condition will hold immediately following (and usually as a result of) the annotated segment of code, i.e. it is an ASSERT.

Finally, there is a remaining category of very idiosyncratic annotation. We make no theoretical claims vis a vis such material. Nonetheless, there are several recurrent forms that might eventually find some realization in our P.A. system, e.g. "this is a kludge", "missing code to be inserted here", "this needs to be fixed", etc.

Annotation in the P.A. System

Eventually, a programmer's apprentice should be able to read all the of the comments a good programmer puts on code and use them to aid in understanding a new program. As a step towards this, we are currently trying to develop a formal, machine-understandable language in which a programmer can write comments. At the moment this language is extremely primitive, but we hope that it will eventually develop to the point where a human programmer would find it convenient to use in supplying plentiful commentary. Furthermore, because the P.A. has deep expectations of the kinds of things that a programmer is trying to express in comments, the understanding of comments written in natural language may be a realistic (though major) goal.

In the current version of our P.A. we have implemented a few straightforward, fixed-format comments that give important clues to help the P.A. understand a new program. How these comments would actually be used in recognition is suggested in Section 5.2. We currently use percent sign (%) as a special macro character in LISP to attach a comment at the current point in the code. In the P.A. environment, comments are interpreted according to the list structure representation of the code, rather than the text string representation. Thus, the P.A. takes the s-expression immediately following the percent sign as the comment itself, which then applies to the s-expression following that. Figure 17 shows the simple comment forms currently implemented. Using these, the LOOKUP program might be annotated as follows:

```
%(design bucket-lists)

(DEFINE LOOKUP (K)
   (PROG ( %(object bucket) BKT )
      (SETQ BKT (TBL (HASH K)))
   LP (OR BKT (RETURN %(object failure) NIL))
      (COND ((EQ (CAAR BKT) KEY)
             (RETURN (CAR BKT))))
      %(segment bump)
      (SETQ BKT (CDR BKT))
      (GO LP)))
```

A feature planned, but not currently implemented, is for the apprentice to look at the actual LISP atoms in a program to see if they have any mnenmonic content. Thus

```
(DEFINE HASH ...)
```

could be equivalent to

```
%(segment hash)
(DEFINE HASH ... )
```

`%<label>`

The label is a unique atomic identifier scoped within each DEFINE, which can be used in other comments to refer to parts of the code.

`%(SEGMENT <segment-type> <label>)`

The following code-segment is a segment of the indicated type. The label is optional; if included it means the s-expressions from from the here to the label constitute one code segment.

`%(OBJECT <object-type>)`

The following s-expression is the surface realization of a data object of the indicated type. For example, the following variable has the object as its value, or the following s-expression has the object as its return value.

`%(DESIGN <design-switch>)`

Indicating the design choice made, so as to facilitate understanding the code following.

`%(ASSERT <condition>)`

The indicated output condition holds after execution of the following code segment. This can aid in identifying the segment.

`%(EXPECT <condition>)`

The indicated condition is expected to hold previous to the execution of the following code segment.

Figure 17. Comment Forms.

CHAPTER THREE

THE DEDUCTIVE SYSTEM

In this chapter we demonstrate how the elements of program description we have outlined in Chapter Two can be used to reason about the behavior of programs. There are several areas of reasoning which might be chosen as the basis for demonstrating such capabilities. In our present research, we have focussed on program verification.

Given the specs of a segment, we assume the programmer has in mind some deep plan for achieving the goal assertions of this segment. At minimum, this plan is a network of data flow links between sub-segments which he believes will achieve the main segment's goals. His plan may also include purpose links, which give explicit reasons why a particular sub-segment is used and, in cases where there is no other convenient way of expressing the constraint, control flow links which specify a particular order between sub-segments. Verification is the process of showing that this deep plan satisfies its specs.

In the current implementation of the deductive system, the P.A. is given the complete data flow of a plan, and the specs it is intended to achieve. The verification proceeds roughly as follows: first the apprentice asserts the pre-conditions (EXPECT's) of the overall segment in a situational data base similar to that of QA4 <Rulifson 1972> or CONNIVER <McDermott & Sussman, 1972>; then for each sub-segment in the plan, it must be shown that its pre-conditions are met in the situation resulting from the execution of all the sub-segments that provide its inputs, and if so its post-conditions are asserted in its output situation; finally, the goal assertions (ASSERT's) of the main segment must be proven to hold after the execution of all the sub-segments. If all these proofs succeed, the programmer has formulated a correct plan for the segment.

Thus verification in our system is a combination of symbolic evaluation and proof (as in <Hewitt &Smith, 1975>). Proof is required to show that the pre-conditions of each segment are satisfied. Symbolic evaluation, or specs application, as we call it in the deductive system, is the process of asserting a segment's output conditions when its input conditions have been met.

In a more advanced implementation, the deductive system will, as a by-product of successful verification, explicitly generate all the purpose links in the plan, amounting to a complete justification of program correctness. These purpose links, along with the data flow links provided by the programmer, form the complete deep plan of the program, which would be used by other parts of the apprentice.

Structure Of The Deductive System

The deductive system makes use of several data bases. First there is the programming knowledge base, described in Section 2.5, which contains data structure descriptions, definitions, plans, and specs. New information can be added to this data base during design. During verification, however, the information in the programming knowledge base is essentially static; it is used primarily as a reference for definitions, etc. by processes involved in the dynamics of reasoning.

The second main data base is a situational data base, which consists of several smaller data bases called situations each representing the state of the computation at a particular instant. (Our notion of time here is a partial order of moments preceding and following the execution of each segment.) Each situation contains assertions about the data objects of the program and the relationships which hold between them. Situations are causally related by the action of some segment. Causal paths in a program may join as well as split. Thus each situation has (one or more) parent situation(s), from which it was produced by specs application, and one or more daughter situations that are similarly produced from it.

Futhermore, there are procedures in the reasoning system responsible for maintaining the consistency of the situational data base and for the assimilation and propagation of new information. To accomplish this, our system uses demons which are kept in an ancillary demon data base. These demons watch for any changes in the situational data base which might need further expansion or propagation.

Finally, there is a group of procedures in the system which can attempt to prove assertions about data objects in a particular situation or relationships between data objects in different situations. These procedures typically make use of information present in both the situational data base and the programming knowledge base.

Although this is a rather elaborate structure, the reasoning system does not in fact resort to techniques which would make it difficult for the human user to understand its proofs.

3.1 Mechanisms

We have not attempted to produce a general purpose reasoning system; rather we have set the less ambitious goal of developing specialized techniques for the task of program verification. One example of this tailoring is that the system currently allows only six types of assertions in the situational data base. Although an unlimited number of specific facts may be expressed in this manner, each assertion must belong to one of the six categories, each of which is handled in its own special fashion. The six assertion types are shown in Figure 18.

One consequence of this limitation of assertion types is that sophisticated pattern matching, such as in CONNIVER <McDermott & Sussman, 1972>, is unnecessary. The pattern matching used in our deductive system is therefore very rudimentary one-level matching, without pattern variables.

Each instance of one of these assertion types is a fact about a particular object in a particular situation. These particular facts in the situational data base are also related to more general information about objects in the programming knowledge base. For example, if a particular object, LIST-1, is a list and we learn that OBJECT-2 is its FIRST part, the deductive system can use the relation definition for list membership in the programming knowledge base (see Section 2.1) to deduce that OBJECT-2 is a MEMBER of LIST-1. Much of this kind of deduction is organized around knowledge of the object types involved.

Uncertainty and Anonymous Objects

Programmers often attempt to design programs with the greatest possible generality, i.e. requiring as few input expectations and guaranteeing as many output assertions as possible. For example, it is often desirable to allow the input to a segment to be any object of type LIST, without regard to its length or other properties.

This means that the deductive system must be able to reason under conditions of uncertainty. There will be many possible facts about a given object that it does not know. For this reason the apprentice has three possibilities for the truth value of an assertion. Facts that are known to be true are explicitly asserted in the data base. A fact that is known to be false is

Type Assertions: (<object-type> <object-id>)

e.g. (LIST OBJECT-1)
OBJECT-1 is a list.

Part Assertions: (<partname> <object-id-1> <object-id-2>)

e.g. (FIRST LIST-1 OBJECT-2)
The first part of LIST-1 is OBJECT-2.

Generic Part Assertions: (<partname> <object-id-1> <index> <object-id-2>)

e.g. (ITEM ARRAY-2 5 OBJECT-1)
OBJECT-1 is the fifth item of ARRAY-2.

Property Assertions: (<property-name> <object-id-1> <object-id-2>)

e.g. (LENGTH LIST-1 7)
The length of LIST-1 is 7.

Relation Assertions: (<relation-name> <object-id-1> <object-id-2>)

e.g. (MEMBER LIST-1 OBJECT-4)
OBJECT-4 is a member of LIST-1.

Figure 18. Assertion Types in the Situational Data Base.

explicitly (or implicitly) denied; that is, either its negation is asserted or a fact implying its negation is asserted (for example, asserting that the length of a list is 5 denies that its length is 4). Finally, uncertainty is represented by absence from the data base.

A second method of representing uncertainty is by the use of anonymous objects (or Skolem constants) <Hewitt, 1973><Sussman, 1973><Moore, 1976>. An anonymous object is an object whose true identity is unknown. Therefore, given an anonymous object and any other object in the system, it is a priori unknown whether or not they are identical. Assertions can be added to the data base which specify some property of an anonymous object, and it might even be possible in some situation to ascertain the identity of such an object.

Anonymous objects contrast with identified objects. While particular properties of an identified object may be unknown, its identity is always definite and unique, i.e. no two identified objects are ever identical. Furthermore, two anonymous objects may be discovered to be distinct without discovering their individual identities; this fact must be represented by an explicit assertion.

As an example, suppose we wish to represent the situation in which a particular hash table has a bucket which has a member whose data part is the number 75. The following assertions state exactly this much:

```
(TABLE table-1)                          ;an identified table
(BUCKET table-1 index-2 bucket-3)        ;with some anonymous bucket
(MEMBER bucket-3 entry-4)                ;with some anonymous entry
(DATAPART entry-4 75)                    ;whose data part is 75
```

In the above, and in the remainder of this chapter, we use underlining to denote anonymous objects. In the current implementation the property list of the object name carries this information.

The main advantage of anonymous individuals is that they allow us to state relationships between objects without having to commit ourselves to the particular identity of these objects. In addition, this technique can be used effectively in conjunction with hypothesis formation. For example, suppose the system knows of two lists whose lengths are related as follows:

```
(LENGTH list-1 integer-2)
(LENGTH list-3 integer-4)
(EQUAL integer-4 (TIMES 2 integer-2))
```

If it is then learned (or hypothesized) that the length of LIST-1 is, say 7, the system can then deduce that the length of LIST-3 is 14. Thus the use of anonymous objects allows a general statement to be made, from which more specific information may be deduced at a later time.

The deductive system also uses anonymous objects as "place holders" for facts which it does not yet know. For example, whenever a new instance of an object with part structure is created, anonymous objects are also created to represent its parts, or if the data object has generic parts, an anonymous object is created to represent its size.

Similarly, the expansion of the square-bracket functional notation in specs can also lead to the creation of anonymous objects. For example, the length relation between LIST-1 and LIST-3 above could be written in specs as

```
(length list-3 [times 2 [length list-1]])
```

The deductive system resolves bracketed expressions as follows: first the partial assertion in brackets is matched against the current situation to see if the designated object is already known; if no match is found, a new anonymous object is created and used to make an assertion by adding it on the end of the bracketed expression; in either case, the resolved object (anonymous or identified) is then substituted for the bracketed expression to allow interpretation of enclosing expressions.

In general, if an unknown fact is needed for the current reasoning, the system can create an anonymous object of which the fact is true, in the hope that at some future point it may be able to deduce the actual identity of the anonymous object.

Identification

One particularly simple, yet important way of deducing the identity of anonymous objects is by employing the constraints implicit in certain assertion types. In particular, both part and property assertions uniquely identify their second arguments. For example, if we had both:

```
(FIRST list-1 object-2)
(FIRST list-1 object-3)
```

then it must be the case that the anonymous OBJECT-3 is identical to OBJECT-2. The system can now ascribe to OBJECT-2 all the properties and relations which are known to be true of OBJECT-3, and remove the name OBJECT-3 from the data base. This process is termed identification.

Identification can take place either between two anonymous objects, or between an anonymous object and an identified object. If an anonymous object is discovered to be identical to an identified object, as in the example above, all occurrences of the anonymous name (i.e. in all situations) are replaced by the name of the identified object. In the case of two anonymous objects being identified, one of the names is arbitrarily chosen to replace all occurrences of the other.

A check for possible identifications is currently the first operation performed by the deductive system whenever a new assertion is added to the situational data base.

Identification is also possible on the basis of generic-part assertions, where the first two argument uniquely determine the third. For example,

```
(ITEM array-1 index-2 item-3)
(ITEM array-1 index-2 item-4)
```

implies that ITEM-3 and ITEM-4 are identical.

Quantification

Program specs often need to contain various forms of quantification. The deductive system currently uses the three quantifiers shown in Figure 19, which are extensions of the standard quantifiers of predicate calculus. Note that the quantified variables in a <CLASS-DEF> are indicated by a "=" prefix.

Significantly, the P.A.'s quantifiers also include explicit situational dependence. Since all the clauses and situational tags except for the <CLASS-DEF> are optional, and are appropriately defaulted if omitted, the standard forms may be obtained as degenerate cases.

These quantifiers have the intuitive meaning which is suggested by their names. For example, the intuitive meaning of the following form is that in situation S-2, TABLE-2 will contain all entries of TABLE-1 in S-1, except those whose key part was KEY-3, which will be members of LIST-4:

```
FOR-ALL (member table-1 =entry)  IN  s-1
        (member table-2 entry) IN  s-2
    EXCEPT-FOR  (keypart entry key-3)     IN  s-1
    FOR-WHICH  (member list-4 entry) IN  s-2
```

Thus class definitions are constructed from the various assertion types by variabilization of one argument. For example, in part or property assertions variabilization of the first argument defines the class of objects which have a particular part or property. The set of all lists whose FIRST is OBJECT-21 is denoted by:

```
(FIRST =list object-21)
```

There is no other way to form a class definition from a part or property assertion, since the second argument is uniquely determined by the first. Binary relation assertions, however, can be used to form two different class definitions. For example, we could quantify over either the set of all members of TABLE-1, or the set of all tables of which ENTRY-21 is a member, respectively:

```
THERE-IS-A  <class-def>  IN  <sit-1>
    SUCH-THAT  <pred>  IN  <sit-2>.


THERE-IS-A-UNIQUE  <class-def>  IN  <sit-1>
    SUCH-THAT <pred>  IN  <sit-2>.


FOR-ALL <class-def>  IN <sit-1>
        <pred-1>     IN <sit-2>
    EXCEPT-FOR  <except-pred> IN <sit-3>
    FOR-WHICH   <pred-2>     IN <sit-4>
```

where:

<CLASS-DEF> is an assertion with a single quantified variable, which defines a set of objects, such as the members of a table or the entries with a particular key.

<SIT-n> names some situation in the situational data base.

<PRED-n> is a predicate, i.e. an assertion involving the quantified variable from the class definition.

<EXCEPT-PRED> is the exception predicate, i.e. an assertion involving the quantified variable which further restricts the set of objects defined by the class definition.

Figure 19. Quantifier Forms in Deductive System.

```
(MEMBER table-1 =entry)
(MEMBER =table entry-21)
```

Class definitions are typically made out of generic-part assertions in only one form, e.g.

```
(ITEM array-21 =index =object)
```

which defines the set of all indices and corresponding objects in ARRAY-21. For example, the following quantifier specifies the action of a segment which divides the items of an array (which are PAIR's) between two lists according to whether they have a given LEFT part:

```
FOR-ALL (item array-23 =index =object)
        (member list-1 object)
    EXCEPT-FOR (first object special-1)
    FOR-WHICH (member list-2 object)
```

Asserting Quantifiers

For each of the three types of quantification, there is a procedural semantics which specifies how to change the data base when the quantifier is asserted. For example, to assert a THERE-IS-A quantifier the deductive system creates an anonymous object and asserts of it both the class description and the predicate. Thus,

```
THERE-IS-A  (member table-1 =entry)
    SUCH-THAT (keypart entry key-1)
```

results in the creation of a new anonymous object ENTRY-22 and the assertions:

```
(ENTRY entry-22)
(MEMBER table-1 entry-22)
(KEYPART entry-22  key-1).
```

THERE-IS-A-UNIQUE quantifiers have a similar procedural semantics except that, in addition, the deductive system records the fact (in the ancillary demon data base; see implementation note following) that the anonymous object created is uniquely quantified by the class definition and the SUCH-THAT predicate. If another object is ever asserted to satisfy these predicates, an identification is performed between it and the anonymous object created by the quantifier.

The check for identification of uniquely quantified objects is implemented by creating a UNIQUE-DEMON with two triggering patterns constructed by replacing the variabilized argument in the class definition and the SUCH-THAT clause by a "don't care" pattern matching symbol (*). For example, consider the following quantifier:

```
THERE-IS-A-UNIQUE  (member table-1 =entry) IN s-1
    SUCH-THAT  (keypart =entry key-1) IN s-2
```

According to the procedural semantics for THERE-IS-A-UNIQUE, the following assertions are made in the situational data base:

```
(ENTRY entry-1)
(MEMBER table-1 entry-1)
(KEYPART entry-1  key-1)
```

In addition, the following demon is recorded in the demon data base:

```
(UNIQUE-DEMON: entry-1
               ((MEMBER table-1 *) IN s-1)
               ((KEYPART  * key-1) IN s-2))
```

Now suppose the new assertion is added

```
(MEMBER table-1 entry-2)
```

Since this assertion matches one of the trigger patterns for the demon above, the system attempts to verify the other part of the unique quantification (substituting ENTRY-2 for *). If this fact is known to be true, ENTRY-2 is identified with ENTRY-1; otherwise no action is taken. Similarly, the same demon would be triggered by the assertion:

```
ASSERT:      (KEYPART entry-2 key-1)
```

The procedural semantics of FOR-ALL quantification are more complex. It might appear adequate simply to find all objects currently in the data base which satisfy the class definition, determine which of them satisfies the exception predicate, and then assert the corresponding predicate for each of them.

However this ignores the possibility that facts related to the quantification might either be learned or hypothesized at a later time. Thus, the full procedure for asserting FOR-ALL's is as follows (where the meta-identifiers refer to Figure 19). For each assertion matching the <CLASS-DEF> in <SIT-1>, the truth value of the exception predicate is determined in <SIT-3> (substituting the object in the current assertion matching the quantified variable for the quantified variable in the exception predicate). If the exception predicate holds, then the predicate <PRED-2> is asserted in <SIT-4> (again substituting the actual object in the current assertion). If the exception predicate is false, <PRED-1> (appropriately substituted) is asserted in <SIT-2>. Furthermore if the truth value of the exception predicate is unknown, then any assertions which would contradict either <PRED-1> or <PRED-2> are made unknown (erased) in their respective situations, and an EXCEPTION-DEMON is established whose job it is to notice when the truth value of the exception predicate becomes known in <SIT-3>. If the exception predicate becomes known, then the demon asserts either <PRED-1> in <SIT-2> or <PRED-2> in <SIT-4> as appropriate.

Even the above complications are not sufficient procedural semantics for FOR-ALL, since at any later time we might learn or hypothesize that some other object matches the class definition in <SIT-1>. We would then have to proceed just as if the quantifier were still active; i.e. we would have to determine if the object mentioned in this assertion satisfied the exception predicate and then act accordingly. Thus the assertion of a FOR-ALL quantifier also leads to the creation of a FOR-ALL-DEMON, which will watch for new assertions that match the <CLASS-DEF>.

Let us clarify by an example. Consider the following quantification which is the essential part of the specs for a DELETE-SEGMENT.

```
FOR-ALL (member table-1 =entry)          IN s-20
        (member table-1 entry)           IN s-21
    EXCEPT-FOR  (keypart entry key-1)        IN s-20
    FOR-WHICH  (not (member table-1 entry)) IN s-21
```

Suppose there is an entry ENTRY-1 in situation S-20 which is a member of TABLE-1, and whose key part is unknown (or anonymous):

```
(MEMBER table-1 entry-1)        IN s-20
```

When the FOR-ALL quantifier above is asserted, ENTRY-1 will fall into the class definition and, since the truth value of the exception predicate will be unknown, an EXCEPTION-DEMON will be created to watch for new assertions bearing on ENTRY-1:

```
(EXCEPTION-DEMON: ((keypart entry-1 key-1) IN s-20)
                  ((not (member table-1 entry-1)) IN s-21)
                  ((member table-1 entry-1) IN s-21))
```

This demon could be triggered most simply by the assertion of exactly its exception condition, i.e.

```
ASSERT:      (KEYPART entry-1 key-1)         IN s-20
```

If this fact is asserted, the action of the demon will then be:

```
ASSERT:      (NOT (MEMBER table-1 entry-1))  IN s-21
```

The exception demon is also triggered by an explicit denial of the exception demon, i.e.

```
ASSERT:      (NOT (KEYPART entry-1 key-1))   IN s-20
```

in which case it asserts <PRED-1>:

```
ASSERT:      (MEMBER table-1 entry-1)        IN s-21
```

Finally, an exception demon can be invoked by implicitly denying its trigger assertion, as by

```
ASSERT:      (KEYPART entry-1 key-2)         IN s-20
```

which causes the same action as for an explicit denial. Implicit denials can be detected by way of the constraints on certain assertion types mentioned previously with regard to functional notation, e.g. that part and property assertions uniquely determine their second argument.

To complete this example, a demon must also be created which will apply the procedural semantics of the FOR-ALL to any other objects which are later discovered to fall in t the class definition. The trigger pattern of this demon is

```
(FOR-ALL-DEMON: ((member table-1 *) IN s-20)
                ...)
```

while the body just records the entire original FOR-ALL assertion so it can be reapplied.

Thus the invocation of demons in our system may involve specific domain knowledge (as in the case of implicit denial of part assertions) in addition to the syntactic pattern matching used in other systems such as PLANNER. Also, our demons are all instances of pre-defined forms, such as UNIQUE-DEMON, EXCEPTION-DEMON, and FOR-ALL-DEMON, while those in PLANNER-like systems usually may be arbitrary programs.

Proving Quantifiers

A procedural semantics for proving quantifiers is also required. As with any assertion, the possible truth values of a quantified assertion are true, false and unknown. Thus there are three logical steps: first an attempt is made to prove the assertion true; then an attempt is made to refute the assertion; if both of these fail, the truth value of the quantified assertion is deemed to be unknown.

In order to prove a THERE-IS-A quantifier true, it is sufficient to simply find an object which in <SIT-1> satisfies the class definition and which in <SIT-2> satisifes the predicate. A refutation consists of showing that no such object can exist, i.e. of showing that for all objects satisfying the class definition the predicate is false. This is implemented exactly as this suggests, i.e. by proving a FOR-ALL assertion.

To prove a THERE-IS-A-UNIQUE quantifier true it is necessary to: (1) find at least one object which satisfies the class definition in <SIT-1> and the predicate in <SIT-2> and (2) prove that there cannot be another such object, i.e. that for all objects which satisfy the class definition except for the one already found <PRED> is false in <SIT-2>. Again this is implemented as the proof of a FOR-ALL quantification.

There are two possible refutations of a THERE-IS-A-UNIQUE assertion. The first is simply to find two objects which satisfy both the class definition in <SIT-1> and the predicate in <SIT-2>. The second is to prove that no object can satisfy both requirements. This second alternative again amounts to the proof of a FOR-ALL assertion.

FOR-ALL assertions are easily refuted by finding a single exception, i.e. an object satisfying the class definition and either satisfying the exception predicate but not <PRED-2>, or not satisfying the exception description and also not satisfying <PRED-1>.

Proving a FOR-ALL assertion true requires a different approach. Since we are working under conditions of incomplete knowledge, there might well be objects which satisfy the class definition of the quantifier, but for which the relevant assertions are not currently in the data base. Thus searching the data base for all objects satisfying the class definition and checking the other predicates is not adequate.

Instead a new anonymous object is created which is asserted to satisfy the class definition in <SIT-1>, and to hypothetically satisfy the exception predicate in <SIT-3>. If the FOR-ALL is to be true, it must then be possible to prove <PRED-2> of the anonymous object in <SIT-4>. Then the negation of the exception condition for the anonymous object is hypothetically asserted in <SIT-3>. For this hypothesis, it must be possible to prove <PRED-1> in <SIT-2>. Since the anonymous object created has nothing extraneous asserted about it, the proofs must apply to any possible object. Thus the two proofs using the anonymous object amount to a validation of the FOR-ALL.

Implementation Note: Contexts and Demons

The situational data base is implemented as a CONNIVER-like context-layered data base in which each succeeding situation is represented as a pushed-context of its parent. This allows incremental updating rather than reconstruction of each situation. There are however two major differences between our implementation and CONNIVER: the implementation of demons (corresponding to if-added methods in CONNIVER), and the relationship between time and hypothetical extensions of a situation.

In CONNIVER and other PLANNER-like languages, demons are antecedent processes whose presence or absence in various contexts is handled identically to that of normal assertions. For example, if a demon is asserted in situation S-1 then it will be triggered by any assertion matching its pattern which is asserted in S-1 or in any of its descendants. This will have exactly the wrong effect for our purposes.

To illustrate, let us return to the deletion quantifier given earlier:

```
FOR-ALL (member table-1 =entry)          IN s-20
        (member table-1 entry)           IN s-21
    EXCEPT-FOR  (keypart entry key-1)        IN s-20
    FOR-WHICH  (not (member table-1 entry)) IN s-21
```

The intuitive meaning of this assertion is that any entry which was a member of TABLE-1 in situation S-20 will also be a member of TABLE-1 in s-21 , unless its key part was KEY-1, in which case it is not a member of TABLE-1 in S-21. The trigger pattern of the FOR-ALL-DEMON for this quantifier is:

```
((member table-1 *) IN s-20)
```

Now suppose a new entry is created with key part KEY-2 (an identified object distinct from KEY-1), and is added to TABLE-1 in a descendant situation of S-20 (i.e. one which represents a later point in the computation). Certainly this entry will not be a member of TABLE-1 in S-21, since at that stage of computation the entry has not yet been created. Unfortunately, in a CONNIVER implementation of demons, the FOR-ALL-DEMON would trigger on this new fact and incorrectly deduce that the the entry with key KEY-2 was a member of TABLE-1 in S-21.

Alternatively, suppose S-20 is an immediate descendant of situation S-19, and that the segment effecting the transition from S-19 to S-20 does not involve TABLE-1. If at some later time the system should discover the new facts

```
ASSERT:    (KEYPART entry-3 key-1)        IN s-19
ASSERT:    (MEMBER table-1 entry-3)       IN s-19
```

then ENTRY-3 must not be a member of TABLE-1 in S-21. Again unfortunately, in the CONNIVER implementation of if-added methods, the FOR-ALL-DEMON would not be triggered by this new MEMBER assertion in S-19, since it occurs above (before) the demon in the context inheritance tree. Thus (MEMBER TABLE-1 ENTRY-3) will simply be inherited automatically into S-21 , which is incorrect.

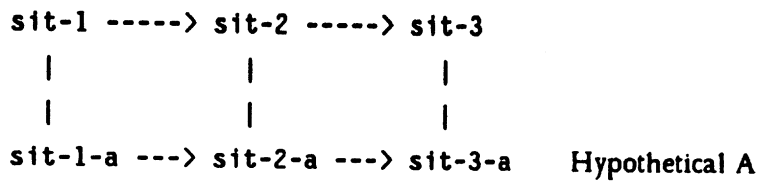
Thus although the concept of an antecedent process, or demon, in our system is the same as in CONNIVER-like languages, our implementation has important differences. Our demons are kept in a separate demon data base with parallel structure to the situational data base. As each assertion is added to the situational data base, a check is made to see if any demons are triggered; if so, they are ordered by situation, most recent firing last. Our demons can be thought of as looking up the context inheritance tree while those of CONNIVER look down.

The situational data base in our deductive system is also a significant modification of the usual CONNIVER context-layer data base. In CONNIVER there is a single context extending mechanism which can be used to represent either time transitions or hypothetical extensions, or with some care both. In our system, we have separated these into a two-dimensional context mechanism, in which one dimension represents time and the other represents hypothetical modifications of situations already present in the time chain.

The two dimensional context facility is exploited by the deductive system in several ways. For one, the proof of FOR-ALL quantifiers postulates the required anonymous object in a hypothetical extension of the initial situation, propagating the new information in hypothetical extensions of the corresponding situations. These extensions can then be ignored, if so desired, once the quantification is proven or refuted. This avoids the danger of destroying information in the primary sequence of situations by the assertion of contradictory facts arising out of the attempted proof.

A second application of the two-dimensional context facility is in interactive design. A programmer may use the deductive system to build a plan out of various segments whose specifications are subject to revision. In particular, he might want to hypothesize additional pre-conditions to see if they would help him achieve his final goal. Again, the use of hypothetical extensions allows the primary time sequence of situations to be kept intact while the effects of a hypothesized change can be examined.

We have implemented two-dimensional contexts by generalizing the layer number in CONNIVER to a 2-tuple, consisting of a time-layer number and a hypothetical-layer identifier. A context in this system is an ordered chain of such layers, where ordering is according to the time dimension as primary key and hypothetical dimension as secondary key. Conceptually at least, the rest of the context mechanism is handled analogous to CONNIVER. Notice that in a two-dimensional system, however, inheritance operates in a grid rather than straightforwardly down a tree.



Demons in this grid are triggered by assertions along the hypothetical dimension in the standard CONNIVER fashion, i.e. information added to any hypothetical extension of a situation which a demon is watching is noticed by the demon. However, new information deduced by the demon is asserted in a hypothetical extension of the appropriate situation corresponding to the hypothetical wherein the triggering took place. If the context system is regarded as a grid with time increasing to the right and hypothetical extensions growing downward (i.e., as we have drawn it), then demons can be thought of as looking downward and to the left, and as acting downward and to the right.

For example, consider again the FOR-ALL-DEMON created for the deletion specs:

```
(FOR-ALL-DEMON: ((member table-1 *) IN s-20)
                ...)
```

Assume further that in a hypothetical extension of S-20, call it S-20-A, the following is asserted:

```
ASSERT:      (MEMBER table-1 entry-4)      IN s-20-a
```

The demon will check if the key part of ENTRY-4 is KEY-1 in S-20-A, and then make the appropriate assertion in situation S-21-A, unless the key part of ENTRY-4 is unknown, in which case an EXCEPTION-DEMON is established to look at S-20-A for information about the key of ENTRY-4.

3.2 Specs Application

Specs application is the most basic operation of the deductive system. It is a kind of symbolic evaluation in which the effect of a segment on particular input objects is simulated in the situational data base. In general, the reasoning involved is simple: first the input expectations are proved in the input situation, and then the output conditions are asserted in the new output situation. If the input expectations cannot be proven, then the segment is termed inapplicable to the present situation, and the programmer is warned of the possible error.

Specs application is denoted by the APPLY command, which has as its argument a segment type followed by an ordered list of input objects to which the segment is to be applied. For example,

```
APPLY:      (HASH key-2 table-3)
```

The system retrieves the specs for the given segment type from the programming knowledge base. These prototypical specs are instantiated for the particular input objects given by building an input association list (a-list) pairing the actual input objects with the corresponding objects in the prototypical INPUT clause. This a-list is then used to systematically substitute the actual input objects into the EXPECT conditions of the prototypical specs. Thus in the above example, if the prototypical specs for HASH were written as follows:

```
(SPECS-FOR hash
    (INPUTS: the-key the-table)
    (EXPECT: (table the-table)
             (key the-key))
    (OUTPUTS: the-index)
    (ASSERT: (index-of the-table the-index)))
```

the input a-list would be:

```
((THE-KEY.key-2) (THE-TABLE.table-3)).
```

The first proof step of specs application is then to prove the expectations, i.e.

```
EXPECT:     (TABLE table-3)                    IN s-1
            (KEY key-2)
```

If this proof succeeds (in a case of type checking like this, typically by simple data base search), then the output clauses are instantiated. This consists of building an output association list in which each output object from the prototypical specs is paired with a new anonymous object (except in the presence of side effects or newly created objects), and doing the appropriate systematic substitution in the ASSERT clauses. For the current example, the output a-list would be:

```
((THE-INDEX.index-4))
```

A new output situation is created as the daughter of the input situation, to which the substituted ASSERT's are added, e.g.

```
ASSERT:     (INDEX-OF table-3 index-4)         IN s-2
```

Side Effects

If the specs being applied involve side effects, the simple procedure described above will not be sufficient. Side effects can take place when the same (mutable) data object is both input to and output of a segment. This is indicated in specs by an ID assertion, e.g.

```
(ID the-array the-new-array)
```

This assertion says that the identifiers THE-ARRAY and THE-NEW-ARRAY, one of which is an input and one of which is an output, refer to the same object. Two different identifiers are required in order to be able to refer to relationships between the object in the input situation and the output situation.

For example, consider the specs for the ARRAYSTORE segment type, which may be written as follows:

```
(SPECS-FOR: arraystore
    (INPUTS: the-array the-index the-object)
    (EXPECT: (array the-array)
             (index-of the-array the-index))
    (OUTPUTS: the-new-array)
    (ASSERT: (id the-new-array the-array)
             (item the-new-array the-index the-object)))
```

Suppose the system were to apply these specs to the following input objects:

```
APPLY:      (ARRAYSTORE array-1 index-2 object-6)
```

The input and output association lists created would be, respectively:

```
((THE-ARRAY.array-1) (THE-INDEX.index-2) (THE-OBJECT.object-6))
((THE-NEW-ARRAY.array-1))
```

Notice that the output a-list binds THE-NEW-ARRAY to ARRAY-1 rather than to a new anonymous object as is done for output objects which are not in ID clauses.

The pre-conditions to be proved for this specs application are then:

```
EXPECT:     (ARRAY array-1)                IN s-3
            (INDEX-OF array-1 index-2)
```

If these proofs are successful, the output assertion is entered into the data base:

```
ASSERT:     (ITEM array-1 index-2 object-6) IN s-4
```

When this last assertion is entered into the data base, the system notices that it is making an assertion about a side-effected object. Therefore, a check is made to see if the current assertion contradicts information already present. For part, property, and generic-part assertions this involves checking to see if the indicated part or property is already known. If so, the old information is erased in the new situation. For example, suppose

```
(ITEM array-1 index-2 object-5)          IN s-3.
```

Since the output situation S-4 is the daughter of S-3 this assertion is also present in S-4 and would contradict the output assertion of the ARRAYSTORE. Thus the system takes the action

```
ERASE:      (item array-1 index-2 object-5)          IN s-4
```

In the application of specs involving side effects, wherein a single real object is referred to by both an input name and an output name, the system must take special care to resolve bracketed expressions [...] in the correct situation. If a bracketed expression refers to a side-effected object by its input name, then the expression must be resolved in the input situation, and vice versa. The necessity of this is illustrated by the following specs:

```
(SPECS-FOR: swap
    (INPUTS: the-pair)
    (OUTPUTS: the-new-pair)
    (ASSERT: (id the-new-pair the-pair)
             (left the-new-pair [right the-pair])
             (right the-new-pair [left the-pair])))
```

Consider the application of these specs to a particular object, say PAIR-75. In the input and output a-lists for this application, both THE-PAIR and THE-NEW-PAIR are bound to PAIR-75. Thus, after substituting the real objects for the local names in the specs, we would need to resolve the brackets in:

```
(left pair-75 [right pair-75])
```

If the bracket is resolved in the output situation, as one might first assume since the expression is in an ASSERT, we would then be saying that the LEFT of PAIR-75 and the RIGHT of PAIR-75 were identical in the output situation. However, the intention of the specs is to say that the LEFT of PAIR-75 in the output situation is what the RIGHT of PAIR-75 had been in the input situation. Thus [LEFT PAIR-75] must be resolved in the input situation because the input name of the object is employed in the specs.

When the specs of a segment include a NEW assertion, only a minor adjustment of the simple specs application procedure is required. Since a new output object may never be identical to any other existing or hypothesized object, this output object should be created as an identified, rather than an anonymous individual.

Side Effects and Gate-keepers

Since most relations and properties are defined in terms of the part structure of the objects involved (see Section 2.1), side effects on the parts of an object may change its properties or the relations it enters into. Therefore when a side effect occurs, it is necessary not only to erase the obviously contradictory part assertions, but also to re-calculate certain properties and relations.

A simple example of this effect is the RPLACD operation in LISP which replaces the REST part of a list. If the REST of a list is changed, obviously the length must be re-calculated. However, this in itself is inadequate because the list might be a sub-list of another list, and so on. Thus the re-calculation of properties and relations must be recursive, propagating up the chain of objects which are parts of other objects.

Furthermore, we cannot assume that all this re-calculation can be done at the time of the specs application. As always, there is the problem of incomplete knowledge. At some later time the system may learn or hypothesize a new property or relation in the input situation, which would be affected by the side effect. The solution to this problem is to create a demon called a gate-keeper, which watches for such assertions.

A gate-keeper contains four pieces of information: the name of the object that was side-effected, the input situation name, the output situation name, and the list of side effects. For example, the gate-keeper demon created by the application of ARRAYSTORE to ARRAY-1 would be:

```
(GATE-KEEPER:  array-1  s-3 s-4
               (item array-1 index-2 object-6))
```

A gate-keeper is triggered whenever a new property, relation, or part is asserted for its trigger object, i.e. the side-effected object. A triggered gate-keeper first checks to see if the new assertion is visible in its input situation; only if so is it actually applicable. There are two cases wherein a gate-keeper is applicable, but does not have to do anything. The first case is if the new assertion is an immutable property or relation, i.e. one that does not depend on part structure. For example the relation INDEX-OF between an index and an array is independent of side effects on the array.

The second case where no special action is required is when the new assertion involves a part that was not changed in the gate-keeper's side effect, or a property or relation that is defined in terms of parts other than the particular part(s) changed in the side effect. The length of a list, for example, is independent of side effects on the FIRST part of the list. The dependency of properties and relations on part structure can be derived from the definitions of the properties and relations. For efficiency in the triggering of gate-keepers, our current system pre-compiles this information in the form of DEPENDS-ON assertions in the programming knowledge base, e.g.

```
(RELATION-DEFINITION
    (member list object)  <=>
    (or (first list object)
        (member [rest list] object)))

(DEPENDS-ON (first list) (member list object))
(DEPENDS-ON (rest list) (member list object))
```

Since relation definitions and DEPENDS-ON assertions in the knowledge base are written in terms of object types, retrieval of this information by a gate-keeper is achieved using the type assertions of its triggering object. Recall that this may require searching up the hierarchy of object and concept types (see Section 2.5) to find a type for which there is a relation definition.

Having determined that it is applicable and that the new assertion depends on its side effect, the gate-keeper now tries to determine the truth value of the new assertion in its output situation. (Remember that demons look "backwards" in time and thus must decide whether new assertions in earlier contexts can pass through.) To do this, it erases (makes unknown) the assertion in the output situation, asserts the expansion of the relation or property according to its definition in the input situation, and then attempts to prove the relation or property assertion in its output situation (also expanding the definition if necessary). If the proof succeeds, the new assertion is "passed through the gate", i.e made true, in the output situation; otherwise a refutation is attempted; if both of these fail, the truth value is left unknown.

An additional complication arises in the case of generic-part assertions. For objects with generic parts, the gate-keeper must determine whether the index in a new generic part assertion is identical to the index in its side effect assertion. If the indices can be proven identical, then the appropriate action is to erase the new assertion in the output situation, since it contradicts the side effect. If the indices can be proven distinct, no action is required. Finally, if the identity of the indices is unknown a exception demon must be created which will take the appropriate action if the unknown fact ever becomes known.

3.3 A Complete Verification

This section shows the complete step-by-step verification of a plan for a hash table insert routine. The heart of the verification process is the specs application procedure described in the preceding section. The example plan in this section is a very clean design with evenly layered conceptual levels, using only the concepts of hash table, hash, bucket, entry, and key. In Section 3.4 we will show how the deductive system has been extended to handle plans which are stated with uneven levels of description.

The specs for a hash table insert routine may be written as:

```
(SPECS-FOR:  insert-segment
    (INPUTS: the-table the-key the-data)
    (EXPECT: (table the-table)
             (key the-key)
    (OUTPUTS: the-new-table the-entry)
    (ASSERT: (table the-new-table)
             (entry the-entry)
             (keypart the-entry the-key)
             (datapart the-entry the-data)
             (id the-new-table the-table)
             (member the-new-table the-entry)
             (for-all (member the-table =entry)
                      (member the-new-table entry))))
```

The programmer's plan for achieving this routine could be stated in words as follows:

> Given a key, data, and the hash table; hash the key to get an index. Fetch the corresponding bucket from the table. Build a new entry from the key and the data, and then build a new bucket from the old one by adding this entry. Insert the new bucket into the table in the position indexed by hashing the key.

```
(SPECS-FOR hash
    (INPUTS: the-key  the-table)
    (EXPECT: (table the-table)
             (key the-key))
    (OUTPUTS: the-index)
    (ASSERT: (index-of the-table the-index)))


(SPECS-FOR: build-entry
    (INPUTS: the-key the-data)
    (EXPECT: (key the-key))
    (OUTPUTS: the-entry)
    (ASSERT: (entry the-entry)
             (new the-entry)
             (keypart the-entry the-key)
             (datapart the-entry the-data)))
```

Figure 20. Segment Types in Plan for Insert (cont'd next page).

```
(SPECS-FOR bucket-fetch
    (INPUTS: the-table the-index)
    (EXPECT: (table the-table)
             (index-of the-table the-index))
    (OUTPUTS: the-bucket)
    (ASSERT: (bucket the-bucket)
             (bucketpart the-table the-index the-bucket)))


(SPECS-FOR: bucket-store
    (INPUTS: the-table the-index the-bucket)
    (EXPECT: (table the-table)
             (bucket the-bucket)
             (index-of the-table the-index))
    (OUTPUTS: the-new-table)
    (ASSERT: (id the-new-table the-table)
             (bucketpart the-new-table the-index the-bucket)))


(SPECS-FOR bucket-insert
    (INPUTS: the-bucket the-entry)
    (EXPECT: (bucket the-bucket)
             (entry the-entry))
    (OUTPUTS: the-new-bucket)
    (ASSERT: (bucket the-new-bucket)
             (member the-new-bucket the-entry)
             (for-all (member the-bucket =entry)
                      (member the-new-bucket entry))))
```

Figure 20. Segments in Plan for Insert (cont'd).

Specs for the segment types mentioned in this plan are shown in Figure 20. The data flow of the plan is given in Figure 21.

```
(PLAN-FOR: insert-segment
    (SUB-SEGMENTS: (hash-1 bucket-fetch-1 build-entry-1
                   bucket-insert-1 bucket-store-1))
    ...)
```

In our current implementation of the deductive system, a plan such as the one above is entered by the programmer one segment at a time (with data flow links). A step of symbolic evaluation occurs after each plan segment by applying its specs. However it is a trivial extension to allow the system to be given an already constructed plan with data flow links (e.g. as output from recognition; see Chapter Five), and have the steps of specs application take place automatically. In terms of the way this example is presented in the following pages, the difference between these two modes amounts essentially to whether the APPLY commands are typed in interactively by the programmer, or generated automatically as the system symbolically evaluates an already constructed data flow plan. Figure 21 shows the data flow plan for this example, with the situation numbers to help follow the verification.

The verification begins in an initial situation S-0 in which the expectations of INSERT-SEGMENT are asserted of newly created anonymous input objects.

SITUATION s-0

(TABLE table-1)
(KEY key-1)

The first step in the verification is to apply the specs of HASH to the initial situation.

APPLY: (HASH key-1 table-1) IN s-0

EXPECT: (TABLE table-1)
(KEY key-1)

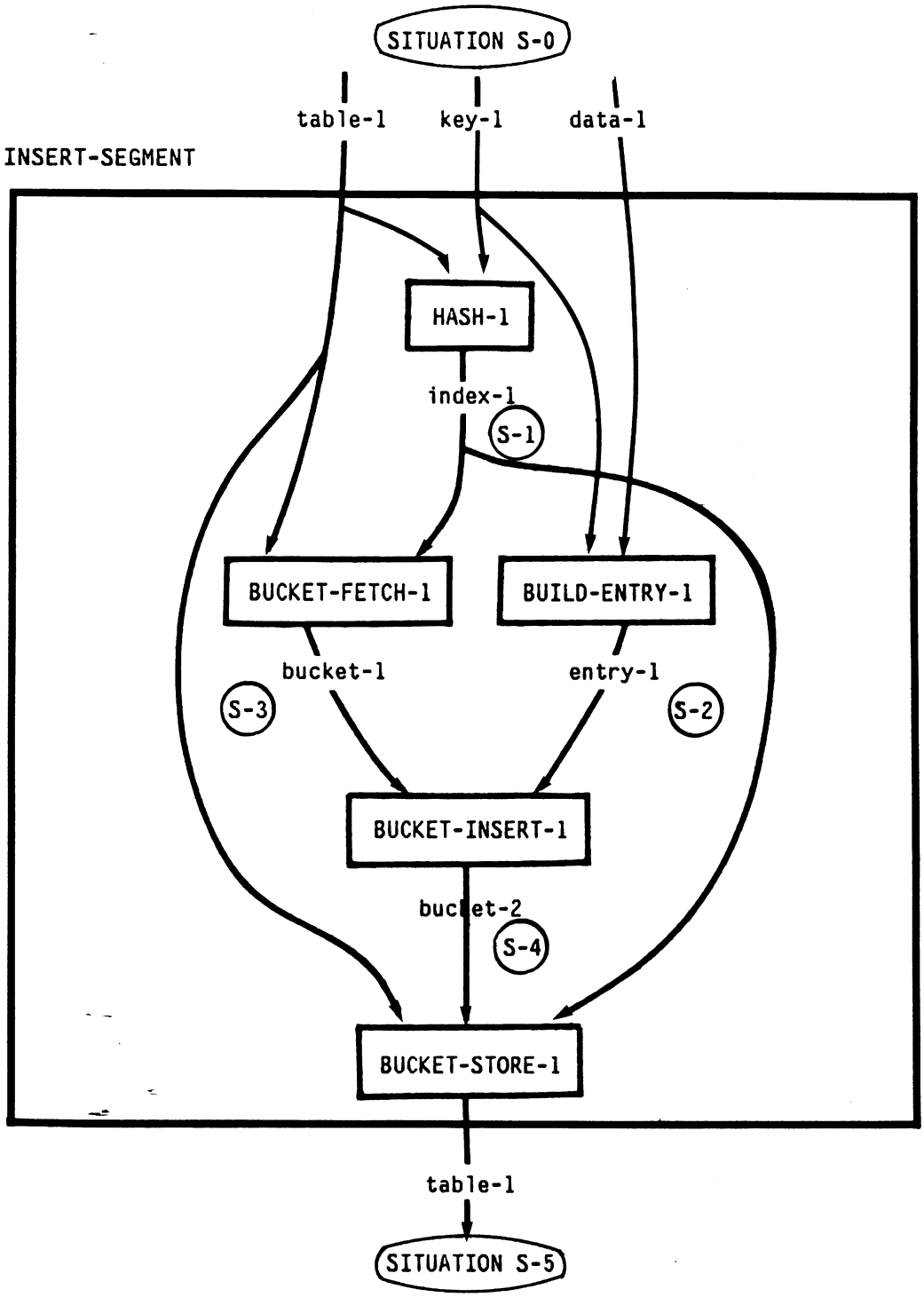


Figure 21. Data Flow for Insert with Situation Numbers

These expectations are satisfied trivially, since the facts are explicitly present in S-0. A new output situation is created, in which the output assertions are added:

SITUATION s-1

ASSERT: (INDEX-OF table-1 index-1)
(HASH table-1 key-1 index-1)

According to the data flow in the plan, the BUILD-ENTRY segment is also applied to the inputs in S-0.

APPLY: (BUILD-ENTRY key-1 data-1) IN s-0

EXPECT: (key key-1)

The input expectation is again satisfied by being present in S-0. A new output situation is created:

SITUATION s-2

ASSERT: (ENTRY entry-1)
(KEYPART entry-1 key-1)
(DATAPART entry-1 data-1)

Notice that since THE-ENTRY is declared to be NEW in the specs of BUILD-ENTRY. ENTRY-1 is created as an identified, rather than as an anonymous object.

Next the system symbolically evaluates the cascade of BUCKET-FETCH, followed by BUCKET-INSERT, followed by BUCKET-STORE. The justification for each EXPECT clause is shown as a comment to the right.

```
APPLY:    (BUCKET-FETCH table-1 index-1)          IN s-2

EXPECT:   (TABLE table-1)                    ;present in s-0
          (INDEX-OF table-1 index-1)         ;present in s-1
```

SITUATION s-3

```
ASSERT:   (BUCKETPART table-1 index-1 bucket-1).
          (BUCKET bucket-1)

APPLY:    (BUCKET-INSERT bucket-1 entry-1)        IN s-3

EXPECT:   (BUCKET bucket-1)                  ;present in s-3
          (ENTRY entry-1)                    ;present in s-2
```

SITUATION s-4

```
ASSERT:   (BUCKET bucket-2)
          (MEMBER bucket-2 entry-1)
          (FOR-ALL (member bucket-1 =entry) IN s-3
                   (member bucket-2 entry)  IN s-4)
```

Recall that the assertion of a FOR-ALL quantifier creates a FOR-ALL-DEMON to watch for new MEMBER assertions in S-3.

```
APPLY:    (BUCKET-STORE table-1 index-1 bucket-2) IN s-4

EXPECT:   (TABLE table-1)                    ;present in s-0
          (INDEX-OF table-1 index-1)         ;present in s-1
          (BUCKET bucket-2)                  ;present in s-4
```

SITUATION s-5

ASSERT: (BUCKET table-1 index-1 bucket-2)

Since there has been a side-effect on TABLE-1, a gate-keeper is created:

(GATE-KEEPER: table-1 s-4 s-5
(bucketpart table-1 index-1 bucket-2))

This completes the symbolic evaluation of all the sub-segments in the plan for INSERT-SEGMENT. It remains now only to prove each of the output assertions of INSERT-SEGMENT. Of these, most are trivial type checks which are satisfied by assertions already present in the data base:

```
PROVE:   (TABLE table-1)              ;present in s-0
         (ENTRY entry-1)              ;present in s-2
         (KEYPART entry-1 key-1)      ;present in s-2
         (DATAPART entry-1 data-1)    ;present in s-2
```

The first non-trivial output assertion to be proven asserts that the newly created entry is a member of the table after the insert:

PROVE: (MEMBER table-1 entry-1) IN s-5

This relation assertion is not explicitly present in any situation in the data base. Therefore the system uses its standard strategy of expanding the relation according to its definition, and trying to show the definition is satisfied. In order to retrieve the appropriate definition from the programming knowledge base the system uses the type assertions of the objects involved. In this example, the relevant membership definition states that entries which are members of tables are members of the bucket part of the table which is hashed to by the key of the entry:

```
(RELATION-DEFINITION
  (member table entry)  <=>
  (member [bucketpart  table
                       [hash table [keypart entry]]]
          entry))
```

The assertion to be proven is expanded into simpler assertions by substituting the specific objects for the object types in the above definition, and resolving the brackets using information in the situational data base:

```
(member table-1 entry-1) <=>
(member [bucketpart table-1
                    [hash table-1 [keypart entry-1]]]
        entry-1)

(KEYPART entry-1 key-1)                  ;resolved in s-2
(HASH table-1 key-1 index-1)             ;resolved in s-1
(BUCKETPART table-1 index-1 bucket-2)    ;resolved in s-5

PROVE:      (MEMBER bucket-2 entry-1)    ;present in s-4
```

The remaining simple assertion to be proven above is present in situation S-4 as an output assertion of BUCKET-INSERT. Q.E.D.

Proving the FOR-ALL Assertion

The second non-trivial output assertion of INSERT-SEGMENT asserts that all entries that were present in the table before are in the table after:

```
PROVE:      (FOR-ALL (member table-1 =entry) IN s-0
                     (member table-1 entry)  IN s-5)
```

In preview, the P.A.'s strategy for proving this assertion can be paraphrased as follows:

> Consider an arbitrary entry that was in the table before the insert was performed. There are two cases: either that entry is in the same bucket as the new entry being inserted, or it is in a different one. In the first case, BUCKET-INSERT guarantees that any entry in the old bucket will be in the new one, and hence in the table after the insert. In the other case, BUCKET-STORE guarantees that all the buckets not involved in the store are not changed. Therefore the entry continues to be a member of the table.

Thus the proof begins by creating an anonymous object ENTRY-2 which is a member of the table in the hypothetical input situation S-0-A (see Figure 22):

```
ASSERT:    (ENTRY entry-2)                IN s-0-a
           (MEMBER table-1 entry-2)
```

The first assertion triggers automatic expansion of anonymous parts:

```
ASSERT:    (KEYPART entry-2 key-2) IN s-0-a
           (DATAPART entry-2 data-2)
```

The assertion of the member relation in S-0-A is visible in S-4 and therefore triggers the gate-keeper which is watching for side-effects on TABLE-1:

```
(GATE-KEEPER:  table-1   s-4  s-5
               (bucketpart table-1 index-1 bucket-2))
```

This gate-keeper checks to see if the new relation being asserted depends in any way on the side effects which it represents. The MEMBER relation for tables does depend on BUCKETPART's; therefore the gate-keeper performs three steps of action. First it erases the relation assertion in the matching hypothetical extension of its output situation:

```
ERASE:     (MEMBER table-1 entry-2)       IN s-5-a
```

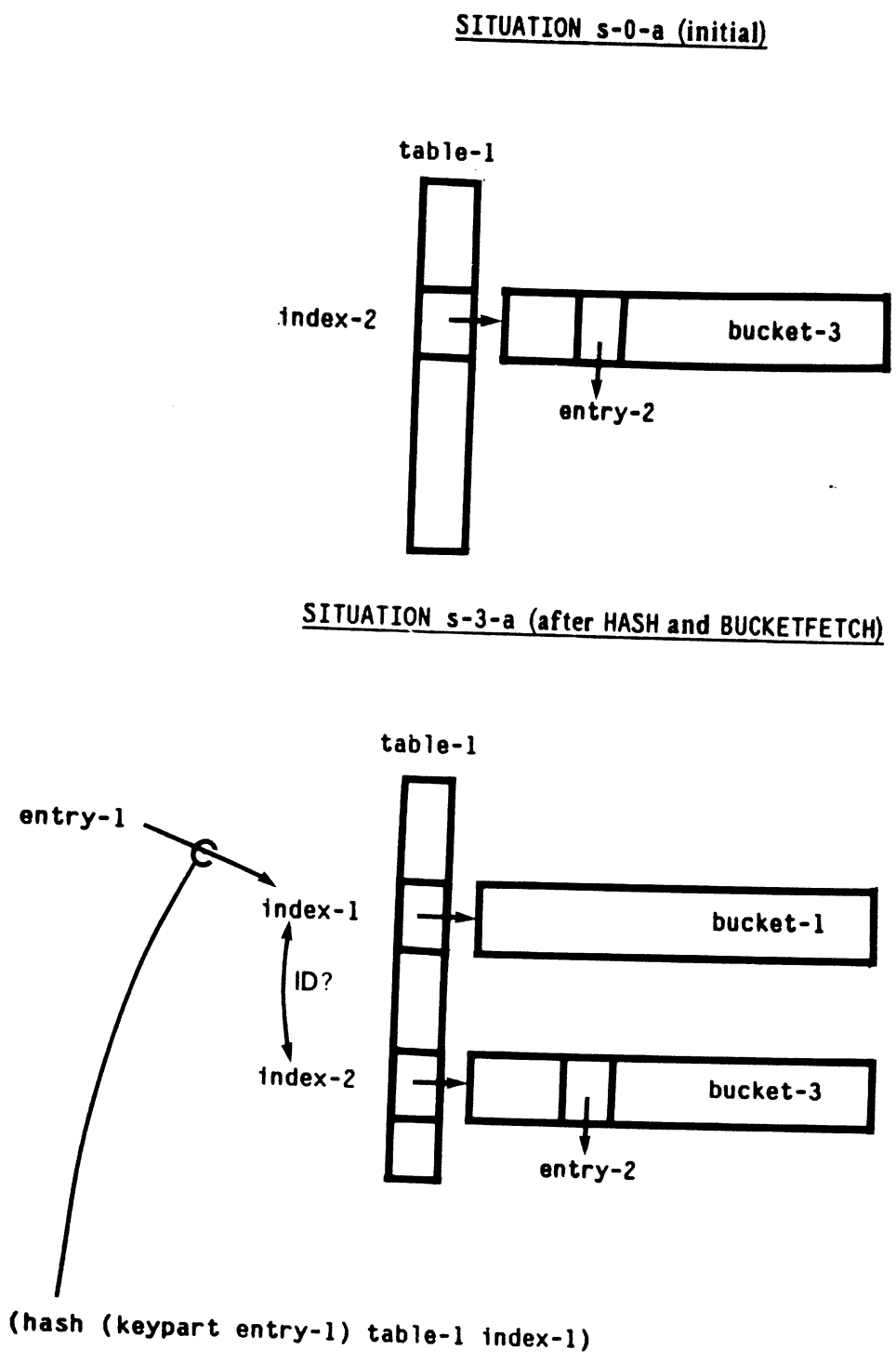


Figure 22. Uncertainty in Identities of INDEX-1 and INDEX-2.

Second, the gate-keeper expands the relation definition in its input situation:

```
ASSERT:     (member [bucketpart table-1
                                [hash table-1 [keypart entry-2]]]
                    entry-2)                          IN s-4-a
```

where the bracketed expressions are resolved as:

```
(KEYPART entry-2 key-2)                  ;present in s-0-a
(HASH table-1 key-2 index-2)             ;generated in s-4-a
(BUCKETPART table-1 index-2 bucket-3)    ;generated in s-4-a
(MEMBER bucket-3 entry-2)                ;generated in s-4-a
```

Notice here that two new anonymous objects have been created, INDEX-2 and BUCKET-3. BUCKET-3 is the bucket in the INDEX-2 position of the table, of which ENTRY-2 is a member. At this point in the proof, it is unknown whether the new entry, ENTRY-1, and the anonymous old entry, ENTRY-2, hash to the same index or not (see Figure 22). Thus BUCKET-2 (see Figure 23) represents the bucket in which ENTRY-1 is inserted, and BUCKET-3 represents the bucket in which ENTRY-2 is inserted. Notice that INDEX-1 being identical to INDEX-2 would make BUCKET-2 identical to BUCKET-3.

When the BUCKETPART assertion above is asserted, the same TABLE-1 gate-keeper is invoked recursively. In this instance, however, the invoking assertion matches the side effect which the gate-keeper represents, i.e. the change of a BUCKETPART of TABLE-1. The problem is that both INDEX-1 and INDEX-2 are still anonymous, so the gate-keeper cannot decide whether to let this assertion through or not. Therefore a exception demon is created to wait for further information to become available, and meanwhile the BUCKETPART assertion is erased in the output situation, since an unknown truth value is better than one that could later be proven wrong:

```
ERASE:      (BUCKETPART table-1 index-2 bucket-3)   IN s-5-a

    (EXCEPTION-DEMON: ((id index-1 index-2) IN s-4-a)
                      nil
                      ((bucketpart table-1 index-2 bucket-3) IN s-5-a))
```

SITUATION s-4-a (after BUCKET-INSERT, before BUCKET-STORE)

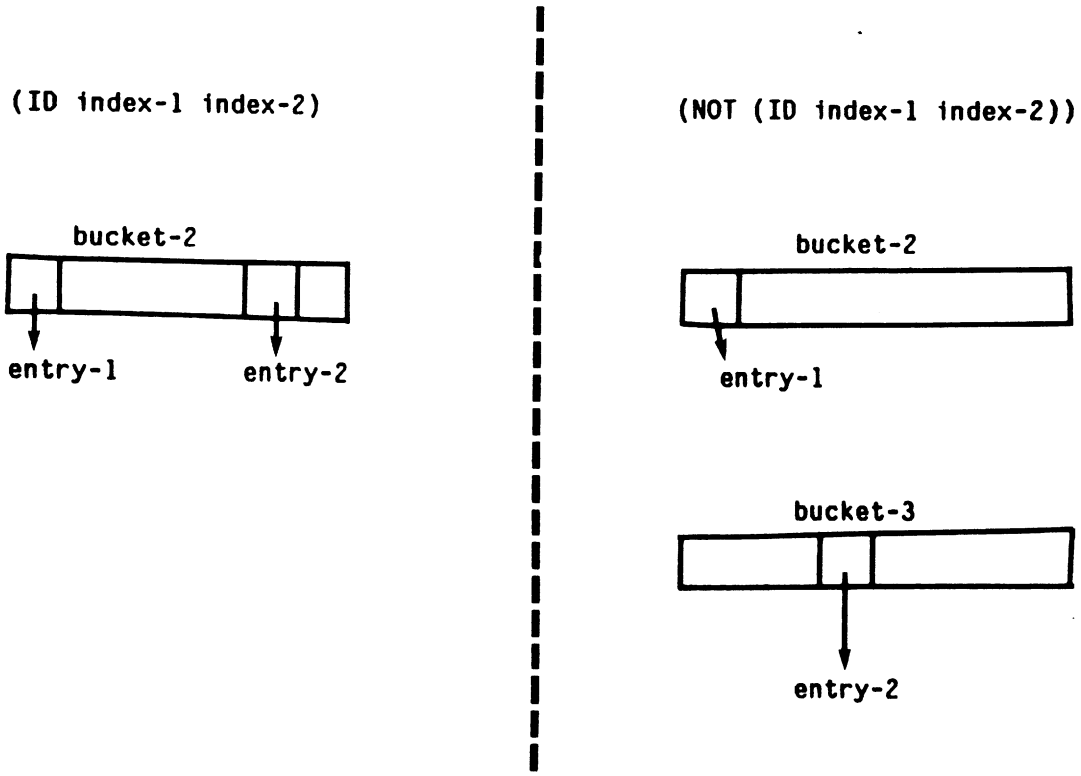


Figure 23. Uncertainty in Identities of BUCKET-2 and BUCKET-3.

This demon says that if INDEX-1 and INDEX-2 are discovered to be identical then no further action is required (since the BUCKETPART assertion for INDEX-1 is already in S-5). However if it is discovered that the indices are distinct, then the demon will assert the BUCKETPART assertion for INDEX-2, which then would not be affected by the side effect.

At this point, control has popped back to the first gate-keeper invoked. As its third and final step, this gate-keeper attempts to prove the MEMBER relation in its output situation by expanding its definition:

```
PROVE:      (MEMBER table-1 entry-2)          IN s-5-a
             <=>
            (member [bucketpart table-1
                                [hash table-1 [keypart entry-2]]]
                    entry-2)

     (KEYPART entry-2 key-2)                  ;resolved in s-0-a
     (HASH table-1 key-2 index-2)             ;resolved in s-4-a
     (BUCKETPART table-1 index-2 bucket-4)    ;generated in s-5-a

PROVE:      (MEMBER bucket-4 entry-2)                IN s-5-a
```

Notice that expanding the MEMBER definition has led to the creation of yet another anonymous bucket, BUCKET-4. BUCKET-4 represents the BUCKETPART of the table indexed by INDEX-2 in situation S-5-A after the BUCKET-STORE has been performed. This anonymous object is necessary since, as noted before, the gate-keeper for the BUCKET-STORE side effect could not tell whether or not INDEX-2 and INDEX-1 were identical. Thus BUCKET-4 represents our uncertainty as to the real identity of the bucket which is hashed to by the keypart of ENTRY-2 (see Figure 24). This turns out to be critical to the proof, since if the system knew the identity of BUCKET-4, then it would know if that bucket was the one involved in the BUCKET-STORE, and then might be able to conclude whether or not ENTRY-2 was still a member.

SITUATION s-5-a (after BUCKET-STORE)

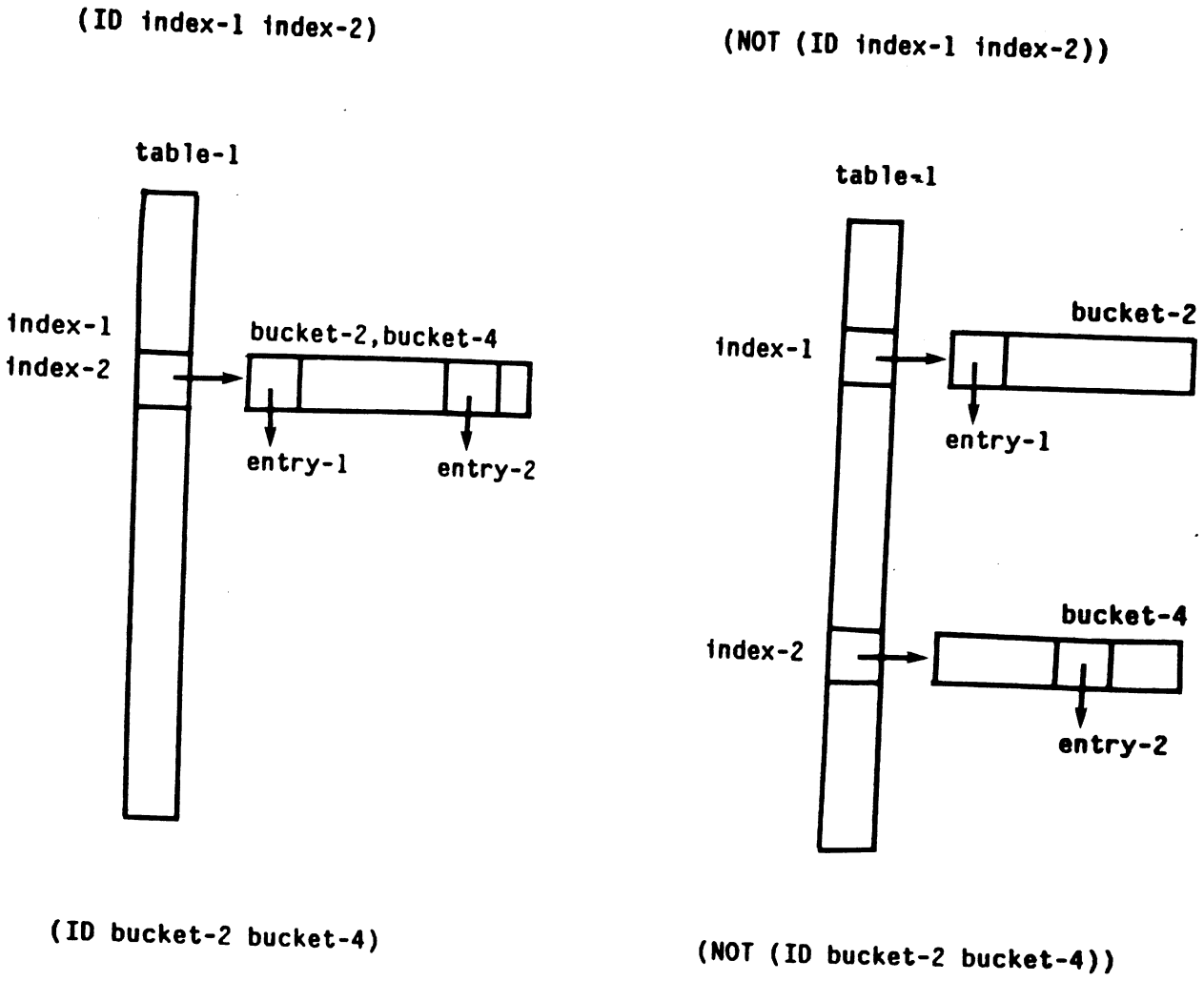


Figure 24. Uncertainty in Identity of BUCKET-4.

The Hypothetical Case Split Heuristic

When the deductive system hits a "road block" such as the above inability to either prove or refute (MEMBER BUCKET-4 ENTRY-2), it can resort to various heuristic strategies. The expansion of a relation according to its definition is an example of such a strategy that we have already seen in operation.

Another important general strategy for unblocking inconclusive proofs is to try to split the world into two hypothetical cases and prove the goal assertion separately in each. For example when the current deductive system is blocked, as in the proof above, it looks for exception demons that were created in the course of the proof attempt. The exception condition of an exception demon provides a good heuristic for splitting the world into two hypothetical cases: one in which the exception is asserted true, and one in which it is denied. The demon can then assert the corresponding result predicate in each hypothetical output situation, providing new information which may advance the proof.

In the current proof of INSERT-SEGMENT, there is such an exception demon:

```
(EXCEPTION-DEMON: ((id index-1 index-2) IN s-4-a)
                  nil
                  ((bucketpart table-1 index-2 bucket-3) IN s-5-a))
```

Accordingly, the system now splits situation S-4-A into two hypothetical extensions S-4-B and S-4-C, in which the exception condition is respectively asserted and denied. As we will see, a flurry of identifications and demon invocations is triggered by these assertions, leading eventually to the assertion of (MEMBER BUCKET-4 ENTRY-2) in both S-5-B and S-5-C.

SITUATION s-4-b

ASSERT: (ID index-2 index-1)

This identification replaces all occurrences of INDEX-2 with INDEX-1:

```
(BUCKETPART table-1 index-1 bucket-4)    ;in s-5-b
(HASH table-1 key-2 index-1)             ;in s-1-b
(BUCKETPART table-1 index-1 bucket-3)    ;in s-1-b
```

Furthermore, the first assertion above triggers another round of identification since

```
(BUCKETPART table-1 index-1 bucket-2)    ;in s-5
```

is visible in s-5-B. Therefore the identification of BUCKET-4 with BUCKET-2 is carried out at the "B" hypothetical level. In particular, this applies to the the FOR=ALL-DEMON in situation S-3, which now becomes:

```
(FOR-ALL-DEMON: ((member bucket-1 *) IN s-3)
                (FOR-ALL (member bucket-1 =entry) IN s-3
                         (member bucket-4 entry)  IN s-4))
```

Yet another round of identification is triggered by the second BUCKETPART assertion above in S-1-B since it is visible in S-3 where

```
(BUCKETPART table-1 index-1 bucket-1)    ;in s-3
```

The identification of BUCKET-3 with BUCKET-1 yields

```
(MEMBER bucket-1 entry-2)                          ;in s-1-b
```

triggering the FOR-ALL-DEMON in S-3 above, which puts its output assertion in S-4-B:

```
ASSERT:      (MEMBER bucket-4 entry-2)                    IN s-4-b
```

which is, of course, what we wanted to prove in situation S-5-B, where it is also visible. (End of proof for case one.)

The second hypothetical case is much simpler:

SITUATION s-4-c

ASSERT: (NOT (ID index-2 index-1))

Since this is the denial of the exception condition of the exception demon, it asserts its second result predicate in the output situation:

ASSERT: (BUCKETPART table-1 index-2 bucket-3) IN s-5-c

This assertion triggers an identificaton of BUCKET-3 with BUCKET-4, since

(BUCKETPART table-1 index-2 bucket-4) ;in s-5-a

yielding the following assertion

(MEMBER bucket-4 entry-2) ;in s-1-c

which is visible in S-5-C. Thus the proof has succeeded in both hypothetical cases, and the plan INSERT-SEGMENT is now verified to satisfy its specs.

3.4 Extensions

In this section we discuss several important extensions to the deductive system described above that have been implemented or are about to be implemented.

Building Purpose Links

As a by-product of verifying a data flow plan, it should be possible to abstract the teleological structure of the program, i.e. the purpose links, which explain "how" this arrangement of segments accomplishes the desired goals. A few simple extensions of the current deductive system (which have not been implemented at this time) would provide the basis for this feature.

The basic idea is to annotate each assertion with an record of how it was entered into the situational data base. For example, an assertion may be an input expectation of the main segment; or it may have been asserted as an output assertion; or finally it may be the result of expanding of another assertion, as in the creation of anonymous objects to represent parts.

Purpose links are just a record of how assertions are used in the verification. Thus for example, if an assertion used to satisfy the input expectations of a segment was entered into the data base as the output assertion of another segment, a PREREQ link should be created between the two. Similary, if an assertion used to prove the output assertions of a segment was entered into the data base as the output assertion of a sub-segment, an ACHIEVE link should be created between the two segments.

Reasoning with Mixed Implementation Levels

In Section 3.3 all the specs in the plan are at a uniform level of description in which many implementation decisions are still unspecified. However it would be unnatural and pedantic for the apprentice to force programmers to always formulate plans at such a uniform level. For example, in the plan for INSERT-SEGMENT, if the programmer had already decided to implement the hash table as an array and the buckets as lists, he might find it more natural to

use ARRAYFETCH, LIST-INSERT, and ARRAYSTORE in his plan, rather than BUCKET-FETCH, BUCKET-INSERT, and BUCKET-STORE.

In general this means that mixed levels of plan description must be allowed, which forces the deductive system to do a little more work. For one thing, it must be able make use of assertions in the programming knowledge base which represent the implementation of the programmer's conceptual objects in terms of implementation parts and the relations between them. Consider the example of a queue implemented as an array with front and back pointers, which is discussed in Section 2.3. (To bring out the important issues here, it is necessary to show an abstract object having several implementation parts, rather than the implementation of a hash table as an array.)

```
(PART queue front)
(PART queue back)

(IMPLEMENTATION-PART queue queue-array)
(IMPLEMENTATION-PART queue front-pointer)
(IMPLEMENTATION-PART queue back-pointer)

(IMPLEMENTATION-DEFINITION
        (front queue object) <=>
        (item [queue-array queue]
              [contents [front-pointer queue]]
              object))

(IMPLEMENTATION-DEFINITION
        (back queue object) <=>
        (item [queue-array queue]
              [contents [back-pointer queue]]
              object))
```

Given the above information in the knowledge base, the deductive system must be able to interpret changes in the part structure of implementation objects in terms of the part structure of the implemented object. For example, changing the CONTENTS of the FRONT-POINTER changes the FRONT part of the queue. This is very similar to the problem with relations and

their definition in terms of part structure. Thus the deductive system compiles dependency information from implementation definitions also:

```
(DEPENDS-ON (contents [front-pointer queue]) (front queue))
(DEPENDS-ON (contents [back-pointer queue]) (back queue))
```

When a new object which has an implementation is asserted in the current deductive system, implementation part assertions are created for it just as part structure is automatically expanded.

```
ASSERT:      (QUEUE object-55)

    (QUEUE-ARRAY queue-55 array-56)          ;generated
    (FRONT-POINTER queue-55 cell-57)         ;generated
    (BACK-POINTER queue-55 cell-58)          ;generated
    (CONTENTS cell-57 index-59)              ;generated
    (CONTENTS cell-58 index-60)              ;generated
```

Now if a change is made to one of the implementation objects, the pre-compiled dependency information is used to trigger the recomputation of affected parts of the implemented object. For example, suppose the contents of the front-pointer above is changed:

```
ASSERT:      (CONTENTS cell-57 index-99)
```

The FRONT of the queue now needs to be recomputed. This is done by retrieving the implementation definition for FRONT parts of queues and expanding in the same way that relation definitions are expanded. In this example:

```
(front queue-55
       [item array-56 [contents cell-57]])

(ITEM array-56 index-99 object-100)      ;in data base

ASSERT:      (FRONT queue-55 object-100)
```

When this last assertion is added to the data base it is processed like any other assertion; i.e. it may cause the creation of a gate-keeper, an identification, or it may trigger a demon.

With this extension to the deductive system, the plan for the insert routine given earlier could also have been given in terms of its implementation structures, while the verification itself would remain virtually unchanged. This is because, as shown above, changes at the implementation level are eventually translated up to the conceptual level. Thus the actual proof proceeds as much as possible at the highest conceptual level, keeping implementation detail hidden.

Case Splitting

In general, segments may have case structure in their specs. The symbolic evaluation of segments with cases requires some extensions to the machinery presented thus far.

Each case of a specs has three possibile applicabilities: (i) definitely applicable, i.e. its input expectations can be proven in the situation of its invocation; (ii) definitely inapplicable, i.e. a refutation of the expectations can be constructed; (iii) unknown applicability, if the expectations can be neither proven nor refuted.

If all the cases of a specs to be applied are definitely inapplicable, obviously there is a bug in the design.

By virtue of the definition of specs, in which the cases must be mutually exclusive, there may not be more than one definitely applicable case; and if one case is definitely applicable, all the rest are definitely inapplicable. In this situation specs application is straightforward -- the specs for the applicable case are applied together with any specs which are shared between all caes.

Finally, suppose all the cases are of unknown applicability. To handle this, the reasoning can be split into pairs of possible execution paths for each unknown case. To represent this, the deductive system creates pairs of daughter situations of the current situation for each case (these are daughters along the time dimension, not hypothetical extensions). In each, the case

of unknown applicability is asserted to to be definitely applicable in one situation and definitely inapplicable in the other. Verification proceeds on each path independently.

All splits eventually join, if only at the top level enclosing segment. Furthermore, the join will usually be manifested as a joining of data flow, with output objects from two (or more) different segments becoming the input or output object of a single segment. In the joining situation, the deductive system will assert of the single object only those facts that are true of both (or all) the joining output objects. From this situation forward, all proofs are first attempted on the single joined object. If a proof attempt fails on the single joined object, then the system must prove the required assertion of both (all) the joining output objects in both (all) the pre-join situations.

Obviously if many segments have many cases of unknown applicability there is a potential for combinatorial explosion of situations. However this problem exists for the human programmer as well as for our programmer's apprentice, which leads both to avoid overly complicated designs.

CHAPTER FOUR

SURFACE ANALYSIS OF LISP PROGRAMS

As a long term goal, we would like our P.A. to behave in a way similar to that of a person when presented with the code for a program it has never seen before. A human programmer, according to our observations, first tries to block out the overall structure of a novel program, using the comments and mnemonics in the code to make connection with his own background knowledge about typical programming techniques. A person can then analyze particular parts of the code to any detail required, guided by his understanding of the overall plan and conceptual basis of the program. We call this process of discovering the plan and conceptual basis of a novel program recognition.

We expect the recognition process to include both algorithmic components, such as indexing and flow analysis, and heuristic components. Our first simple design of an apprentice has separated these components into two distinct phases. The purpose of this chapter is to describe the algorithmic first phase, called surface flow analysis, in which a surface control flow and data flow plan is constructed for "arbitrary" LISP code. Algorithmic techniques for flow analysis are already well understood from a graph-theoretic standpoint (e.g. <Allen and Cooke, 1976>).

Furthermore, recognition may take place incrementally. As each function definition of a large program is encountered, the apprentice may use both surface flow analysis and higher level recognition, if possible.

Our current design philosophy generally opposes the use of algorithms that are fundamentally different from those used by humans working in the programming domain, since this can impede close communication and cooperation between the apprentice and a human programmer. For example, if the P.A. detects an error, it must be able to express the problem to the programmer in terms of his conceptualizations. The use of non-intuitive algorithms and represenations creates an extra problem for the apprentice of translating between its representation and what the programmer finds natural. This principle has been

applied elsewhere in the P.A., for example in the decision not to base the P.A.'s reasoning component on a uniform resolution procedure.

Symbolic Evaluation

The most natural way to view surface flow analysis, i.e. the one which corresponds most closely to human behavior on the same task, is as a kind of symbolic evaluation. A human programmer when presented with novel code, will often trace through the program "with his finger", following the control flow and data flow paths of interest without using real values.

Thus symbolic evaluation has emerged as a useful and powerful concept in both the deductive system (Chapter Three) and the recognition system. In the deductive system, symbolic evaluation takes place in the domain of specs and plans, generating purpose links as a by-product. In surface flow analysis, symbolic evaluation mimics the operation of the usual LISP interpreter (with special provisions for splits, joins, and loops), creating a surface plan as a by-product. In a recently reported work, King <1976> has combined symbolic evaluation of code with some deductive capabilities to form a unified system for program verification and testing.

4.1 Operation of the Symbolic Evaluator

The symbolic evaluator mimics the operation of the usual LISP interpreter. At every level it examines the current s-expression: if the CAR is an atom, it is taken as either a function name or a special form (i.e. a FEXPR in MACLISP); if it is a special form, the evaluator calls the special procedure associated with that form to symbolically evaluate the s-expression, otherwise the remaining top-level items of the s-expression are evaluated and the symbolic values passed as arguments to the lambda form defined as the body of the named function. If the CAR of the current s-expression is non-atomic, it must be a lambda form which is used only in this place and is therefore not given a name.

The major difference between a symbolic evaluation and a "real" evaluation arises with conditional branches. Since it is not possible in general to decide on the basis of a symbolic value which branch to take, a symbolic evaluator must split the control flow and follow both paths, leading to an eventual join. Split paths must always join at some point, if only at the top level. Program loops are handled in our symbolic evaluation by proceeding forward normally until a loop is detected, and then taking special corrective action. Thus loops are really viewed as a special case of split and join, wherein the join occurs at a point in the code which has already been executed.

In the following sections, we describe the operation of the symbolic evaluator in more detail, starting with the simple case of evaluating programs with no splits or joins. The purpose of the symbolic evalution is to build a surface plan. This entails three kinds of action that take place as side-effects of the actual execution: creation of segments, recording of control flow links, and recording of data flow links. A diagram of the full surface plan built during the evaluation of an example program is given at the end of the chapter.

Initial Segmentation of the Code

Surface flow analysis breaks up LISP code into the smallest units of description used by the apprentice, which are the FUNCALL (and certain implicit OPENCODE) segments. The rest of the code, such as PROG's, SETQ's, COND's, etc. are part of the connective tissue that tell the apprentice how the real code segments are connected to each other by control and data flow.

The initial segmentation is also very "flat". There are only two kinds of grouping that occur at this level: the segments in a LAMBDA body, whether a top level DEFINE or an unnamed LAMBDA expression, are grouped into a single segment; and the body of each loop, as it is detected, is grouped into a single segment. However the symbolic evaluation of certain special forms, such as COND, does leaves a record of "suggestions" for likely groupings. Later in the recognition process these suggestions, together with deeper knowledge of programming and plans, are used to further group the initial small segments to form larger, more meaningful segments. The initial segmentation however makes use only of LISP-specific knowledge.

On the basis of surface flow analysis alone, the input and output data objects for each code segment can be determined. However, the rest of the specs for a user-defined function (EXPECT's and ASSERT's) will not in general be known at this stage of processing. Thus the segment and its input and output objects cannot be given identifiers that are more specific than SEGMENT-<n>, and OBJECT-<n>'s, respectively. In the following examples, we will sometimes use more mnemonic identifiers, such as KEYPART-65, KEY-66, etc.; this is cheating a bit, but it will make the examples more understandable.

The system is initialized with dummy DEFINE's for all the built-in functions such as CAR, CDR, etc. The segments for these functions have complete specs attached to them as part of the P.A.'s initial knowledge.

When the evaluator encounters a function call s-expression, a FUNCALL code segment is created, with the surface input and output statements and specs copied from the LAMBDA-EXP segment with the corresponding name. For example, suppose the definition of a KEYPART function has already been seen and fully recognized by the apprentice. The following information would be recorded:

```
(keypart-1 SEGMENT LAMBDA-EXP keypart ((define keypart (entry) ...)))

(SPECS-FOR: keypart-1
    (INPUT: entry-1 ARG (entry))
    (OUTPUT: key-2 RETURN-VAL ((define ...)))
    (ASSERT: (keypart entry-1 key-2)))
```

The symbolic evaluation of a function call to KEYPART results in the creation of another instance of this segment type:

```
(KEYPART ... )

(keypart-65 SEGMENT FUNCALL keypart {(keypart ...)})

(SPECS-FOR: keypart-65
    (INPUT: entry-67 ARG {...})
    (OUTPUT: key-66 RETURN-VAL {(keypart ...)})
    (ASSERT: (keypart entry-67 key-66)))
```

Notice that when specs are copied from a DEFINE segment to a FUNCALL segment the identifiers are systematically changed. Furthermore, the surface inputs and outputs of the FUNCALL segment are also different.

The invocation of a currently undefined function causes the evaluator to build a dummy definition for the function, having input and output objects compatible with the current FUNCALL. Furthermore the function name is added to a global UNDEFINED-FUNS list. When the actual definition of the function is encountered, it is substituted for the dummy definition, after checking that its specs are compatible with the dummy specs (and therefore with the existing FUNCALL's). This same mechanism also takes care of self-referential function definitions.

Building Control Flow Links

The building of control flow links is implemented in the symbolic evaluator as follows. Evaluation always commences on a DEFINE, which is an implicit lambda form. The segment identifier (segment-id) of the lambda expression is recorded in a global variable called SUPERSEG. As evaluation proceeds on the body of the lambda expression, segments are created as described above. Each segment-id created is added to a global SUBSEGS list for later generation of a SUB-SEGMENTS statement. Furthermore, the evaluator keeps another global variable called LASTSEG, whose value is the segment-id(s) (and optionally case-id's) of the last segment(s) executed. (There can be more than one LASTSEG because of splits and joins; see

later section). When the next segment is created (which may not be the next s-expression because of the existence of connective tissue), a NEXT link is recorded between each LASTSEG and the present segment. As a special case, an INVOKES link from SUPERSEG is recorded for the first segment in the body of the lambda form. When evaluation of the body is completed, RETURNS links are recorded for each LASTSEG to SUPERSEG.

Building Data Flow Links

The symbolic evaluation keeps track of only two of the four basic techniques for passing data around in a LISP program: the use of free variables, and the the use of return values. This is because the other techniques involve side effects on data structures (lists, arrays, or property lists) and thus require global reasoning to resolve the actual data flows involved. (See subsequent section on data flow using side effects.)

Each s-expression in the code has a symbolic return value, even though each s-expression is not necessarily a code segment. The special evaluation procedures for the connective tissue forms pass up the appropriate return values from the segments inside, according to their standard LISP semantics. For example, the special procedure for LAMBDA returns as its own symbolic value the value returned from the evaluation of the last s-expression in its body.

The symbolic return value from the current s-expression is always available in a global variable called RETURNVAL. So, after the evaluation of the following form:

```
(KEYPART ... )

RETURNVAL = ((keypart-65 OUTPUT key-66 RETURN-VAL ((keypart ...))))
```

When this symbolic value is used, say as the argument to an enclosing s-expression, the evaluator builds a data flow link, e.g.

```
(HASH (KEYPART ...))

(hash-96 SEGMENT FUNCALL hash {(hash ...)})

(PLAN-FOR: ...
    ...
    (DATAFLOW: (keypart-65 OUTPUT key-66 RETURN-VAL {(keypart ...)})
               (hash-96 INPUT key-97 ARG {(keypart ...)})
               NESTED-SEXP)
    ...)
```

Corresponding to the a-list of a standard LISP interpreter, which records the current (real) value of variables, the symbolic evaluator has a symbolic a-list (ALIST) which records the current symbolic value(s) of variables. A symbolic value is a surface plan statement which identifies the segment-id, object-id, and case-id that last assigned a value to the given variable in the current control path. Multiple symbolic values arise due to the existence of splits and joints in the control flow. A simple example:

```
(DEFINE FOO (...)
        (PROG (X)
              ...
             (SETQ X (KEYPART ... ))
              ... ))
```

The ALIST following symbolic evaluation of the SETQ above would have the following binding for the variable X:

```
(X . ((keypart-65 OUTPUT key-66 RETURN-VAL {(keypart ...)}
                                    (COUPLING {(setq x ...)}))))
```

In this example, the action of updating the ALIST is performed by the special evaluation subprocedure associated with SETQ. Notice that optional information to be used later in building a data flow link (in this case COUPLING information) is also recorded at the trailing end of the symbolic value.

When the variable X is subsequently used, its symbolic value is looked up in the ALIST , just as a real value would be, and a data flow link is built using the information in the symbolic value, e.g.

```
(DEFINE FOO (...)
        (PROG (X)
              ...
             (SETQ X (KEYPART ... ))
              ...
             (HASH X)
              ... ))

(segment-33 SEGMENT LAMBDA-EXP foo {(define foo ...)})

(PLAN-FOR: segment-33
    ...
    (DATAFLOW: (keypart-65 OUTPUT key-66 RETURN-VAL {(keypart ...)})
               (hash-96 INPUT key-97 ARG {...})
               COUPLING {(setq x ...)})
    ...)
```

Evaluation Procedures for Special Forms

Most of the important work of surface flow analysis occurs within the symbolic evaluation procedures for the special forms. This is where the majority of the P.A.'s LISP-specific knowledge is currently embedded. Writing the code for these procedures turned out to be a fairly delicate task, leading us to believe it would be desirable to have a more declarative way of expressing programming language specific knowledge in the apprentice. However, for the moment we can give the following informal description of what happens in the special procedures.

There are four general characteristics that categorize special forms: sequencing, grouping, splitting, and joining. Many special forms combine several of these characteristics, for example COND has all four. A further general characteristic shared by all special forms except GO and QUOTE is that they invoke the symbolic evaluator recursively, usually modifying the contents of its global state variables.

Sequencer forms, such as PROG, PROGN, LAMBDA, and the bodies of COND clauses invoke the evaluator on each consecutive s-expression in their body. This takes care of getting the NEXT links recorded for each sub-segment. The sequencers like PROGN, LAMBDA, and COND, also return as their symbolic value the return value from the evaluation of the last s-expression in their body, in accordance with the standard semantics for LISP. The symbolic evaluation procedures for LAMBDA and PROG push and pop their arguments on the symbolic a-list just as in a real evaluation.

Grouping forms, which include PROG, PROGN, LAMBDA, and COND, bind the SUBSEGS list in order to record all the segments that are created inside their scope. Before they return, they leave a record of this information for use by the heuristic second phase of recognition. A major activity in recognition is trying different ways of grouping small segments into larger segments that correspond to the expected deep plan. The use of a grouper form in the code is taken as a syntactic clue to a potentially meaningful grouping of sub-segments. For example, COND leaves a suggestion for grouping together the segments in each of its test conditions and each of its consequent bodies.

Some special forms create extra segments that are not explicit in the code. For example, COND creates a NULLTEST segment for each clause, which splits control flow on the result of evaluating the test condition. AND and OR do similarly.

The special procedure for SETQ updates the symbolic a-list with the symbolic result of evaluating of its second argument, analogous to its operation in normal evaluation.

There are also special procedures for QUOTE and constants like T, NIL and numbers, which create a segment of type CONSTANT with the appropriate specs.

Splits, Joins, and Loops

A split in evaluation takes place whenever a conditional branch (e.g. in a COND) is encountered. Generally speaking, what the special procedures for "splitting" forms do is make copies of the ALIST, bind the global state variables, and then call separate instantiations of the evaluator for each path. The subsequent "joining" entails merging the ALIST's, LASTSEG's, and RETURNVAL's from the separate paths. Thus the net effect of splits and joins is to cause there to be multiple symbolic values for LASTSEG, RETURNVAL, and variable bindings in the ALIST. To illustrate, Figure 25 gives a word description of the full procedure for COND. Notice in this procedure that "merging" of a-lists is a set union operation that does not result in duplicate symbolic values for any variable.

The last major special procedure to be described in detail is PROG, which is also the most complicated because of its interaction with the procedures for RETURN and GO. Similar to COND, the procedure for PROG binds the global state variables RETURNVAL, SUBSEGS, LASTSEG, and ALIST. The following paragraphs describe two simple forms of using PROG, GO, and RETURN that do not involve loops.

When a RETURN is evaluated inside of a PROG, the effect of the special procedure is to add the return value and LASTSEG resulting from the execution of the argument of RETURN to the RETURNVAL and LASTSEG respectively of the enclosing PROG. Then the RETURN procedure itself returns a special value which causes the current evaluation to terminate immediately.

When the PROG procedure encounters a tag in its body it takes special action against the eventuality that a backward jumping GO may subsequently be encountered with the tag as its target. A special entry is pushed onto the current ALIST which records the tag and the current LASTSEG:

```
((TAG <atomic tag>) . <lastseg>)
```

This information is pushed onto the current ALIST because the important question when evaluating a GO is whether the target tag has been encountered on the current path. As shown in the procedure for COND, path environments are preserved as separate ALIST's.

```
Save old value of global variables: set *RETURNVAL to RETURNVAL
                                        *LASTSEG to LASTSEG
                                        *SUBSEGS to SUBSEGS
Do for each clause:
    Bind SUBSEGS to NIL;
    Evaluate test condition;
    Create NULL-TEST segment, adding data and control flow;
    Record grouping suggestion for SUBSEGS;
    Add SUBSEGS to *SUBSEGS;
    Unbind SUBSEGS;

    Set LASTSEG to CASE-1 (non-null) of NULL-TEST;
    Set *ALIST to copy of ALIST;
    Bind ALIST to *ALIST;
    Bind SUBSEGS to NIL;
    Do for each s-expression in consequent body:
        Evaluate s-expression.
    End;
    Add return value of last s-expression to *RETURNVAL;
    Add LASTSEG to *LASTSEG;
    Record grouping suggestion for SUBSEGS;
    Add SUBSEGS to *SUBSEGS;
    Unbind SUBSEGS;
    Merge ALIST into *ALIST;
    Unbind ALIST;

    Set LASTSEG  to CASE-2 (null) of NULL-TEST;
End.

Add LASTSEG to *LASTSEG.
Set LASTSEG to *LASTSEG.
Set SUBSEGS to *SUBSEGS.
Merge *ALIST into ALIST.
Return *RETURNVAL.
```

Figure 25. Word Description of Evaluation Procedure for COND.

When a GO is encountered, the special procedure checks the ALIST to see if the target tag has already been encountered on the current path. If the tag is not present in the ALIST the current GO is a forward jump, which is the simple case. The GO procedure then returns a special value which causes the evaluator to search the current PROG body for the tag, and continue evaluation at that point.

We have now dealt with all cases except GO's which jump to a tag that has already been encountered on the current evaluation path. This GO forms a loop, and triggers special action based on the following observation: the only thing wrong with the way the loop body has already been evaluated is that potential data flow between the outputs of the loop body and the inputs of the loop body has been missed. Corrective action begins by grouping the segments in the body of the loop into a single segment. The segments that are in the body can be discovered by starting with the segment which is NEXT after the LASTSEG(s) recorded in the ALIST with the target tag.

As will be explained in Section 5.2, part of the effect of a grouping operation is to calculate the net inputs and outputs of the group of segments. Thus, in the case of the loop body segment, all that needs to be done to correct the data flow is to merge the symbolic values of any free variable outputs that are also inputs, adding the corresponding data flow links. To complete the control flow, a NEXT link is added from the appropriate case of the loop to itself.

Notice that this algorithm for analyzing loops results in a plan that is essentially a recursion, i.e. the outputs of the segment feed back into the inputs. This should not be surprising since it is a well known fact that iteration and tail recursion are semantically equivalent. This fact is exploited in both the deductive system and in recognition. In the deductive system, reasoning about "loops" is implemented as reasoning about the corresponding recursions. In recognition, plans can be transformed between the two forms in order to facilitate matching against stored knowledge.

A Complete Example Using LOOKUP

Figures 26 and 27 show the output of surface flow analysis for the LOOKUP program. For ease of presentation, the data flow and control flow are shown in separate figures. Together they constitute the complete surface plan for the following code:

```
(DEFINE LOOKUP (KEY)
        (PROG (BKT)
              (SETQ BKT (TBL (HASH KEY)))
          LP  (OR BKT (RETURN NIL))
              (COND ((EQ (CAAR BKT) KEY)
                     (RETURN (CAR BKT))))
              (SETQ BKT (CDR BKT))
              (GO LP)))
```

The dotted lines in Figure 26 indicate grouping suggestions generated during symbolic evaluation of special forms. HASH-6 and ARRAYFETCH-7 are grouped by a heuristic in the special procedure for PROG which suggests grouping the segments in each sequential step. CAR-10, CAR-13, and EQUAL-14 are grouped by COND as shown in Figure 25.

Data Flow by Side Effects on Data Structures

The evaluation algorithm described above cannot detect data flow between segments achieved by side effects on data structures, except for two special cases. We first describe the special cases, and then indicate why the general case is unsolvable by a local flow analysis procedure such as ours.

The first special case is the use of arrays in LISP. The special evaluation procedure for STORE takes advantage of the fact that the name of the array can be treated as a free variable output which carries the array object as its value. Thus, it builds the following surface plan and updates the ALIST:

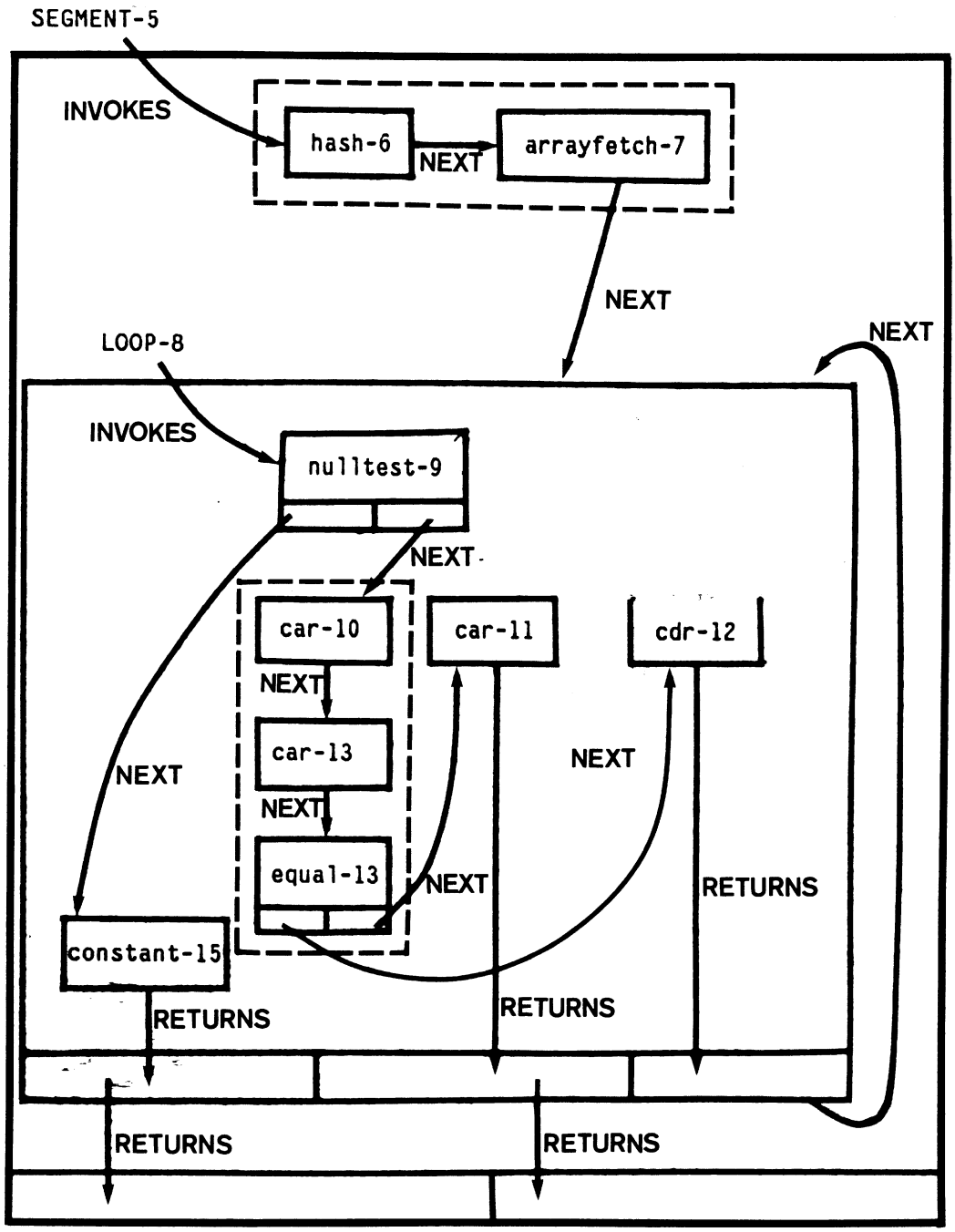


Figure 26. Surface Control Flow for LOOKUP.

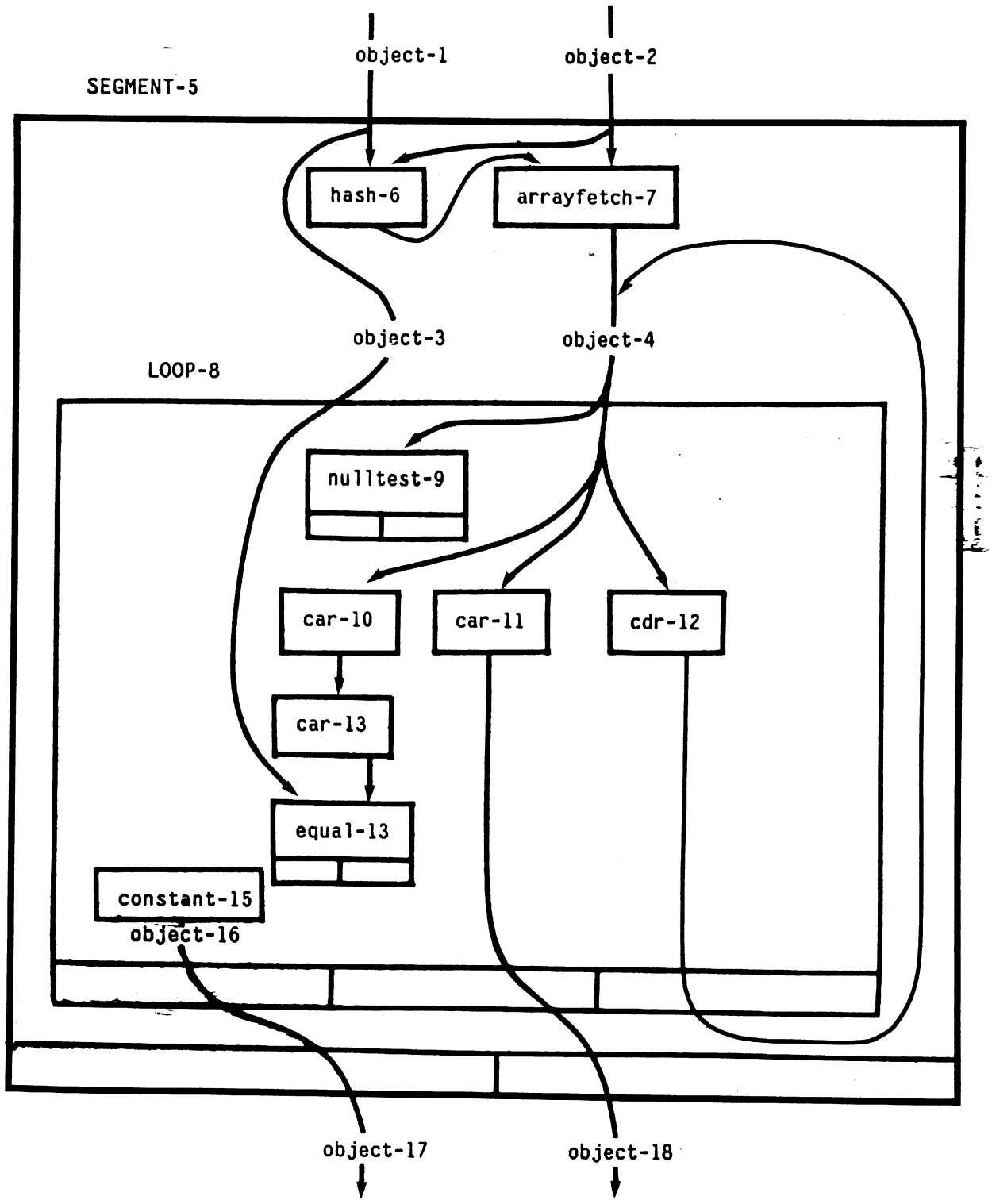


Figure 27. Surface Data Flow for LOOKUP.

```
(STORE (TBL ... ) ... )

(arraystore-77 SEGMENT FUNCALL tbl {(store (tbl ...) ...)})

(SPECS-FOR: arraystore-77
    (INPUT: array-78 FREE-VAR {tbl})
    (INPUT: index-79 ...)
    (INPUT: object-80 ...)
    (OUTPUT: array-81 FREE-VAR {tbl})
    (EXPECT: (index-of array-78 index-79))
    (ASSERT: (item array-78 index-79 object-80)
             (id array-78 array-81)))

ALIST = ((tbl . ((arraystore-77 OUTPUT array-81 FREE-VAR {tbl}))) ... )
```

The evaluator keeps a global ARRAYS list with the names of all arrays that have been declared. When an array access form is encountered, a special evaluation procedure is also called, which simply looks up the array name in the ALIST just as any other free variable. Then, as is usual when a variable is used, a data flow link is built:

```
(TBL ...)

(arrayfetch-88 SEGMENT FUNCALL tbl {(tbl ...)})

(PLAN-FOR: ...
    ...
    (DATAFLOW: (arraystore-77 OUTPUT array-81 FREE-VAR {tbl})
               (arrayfetch-88 INPUT array-89 FREE-VAR {tbl})
               SAME-FREE-VAR)
    ... )
```

The special procedures for RPLACA and RPLACD work analogously to STORE in the case where the first argument to the functions (the list structure to be modified) is a variable. This variable is then also treated as an output of the RPLAC. Subsequent segments using that variable, e.g. CAR's and CDR's, then gratuitously have the correct data flow links built. However, if the programmer is using several free variables to hold pointers to different parts of a single shared data structure, as is quite common in certain types of programming, global reasoning is required to derive all the data flow relationships actually present.

The general case of data flow by side effect which requires global reasoning (i.e. for which none of the above "hacks" works) is illustrated by the following code:

```
(RPLACA (CDR X) (FOO ...))
 ...
(BAR (CADR X) ...)
```

Figure 28 shows the surface data flow plan (solid lines only) that is built for such code by symbolic evaluation. The first clue that there is something wrong is that the conceptual output object of RPLACA, i.e. the modified list, has no surface realization. Furthermore, the data flow link shown by a solid line into BAR-105 is incorrect. The correct data flow link into BAR-105 is the dotted line. Discovering this data flow link entails complicated reasoning, as described in Chapter Three, about the identity of objects and the changing of subparts. Furthermore, it is obvious that this reasoning is not possible until the entire network of data flow links has already been built; thus it does not belong in the initial surface flow analysis. However, it will be necessary to look for such hidden data flows during recognition in order to recognize an expected plan that has been implemented using this side effect technique.

The use of property lists is similarly a kind of data flow that requires global reasoning on the completed surface plan.

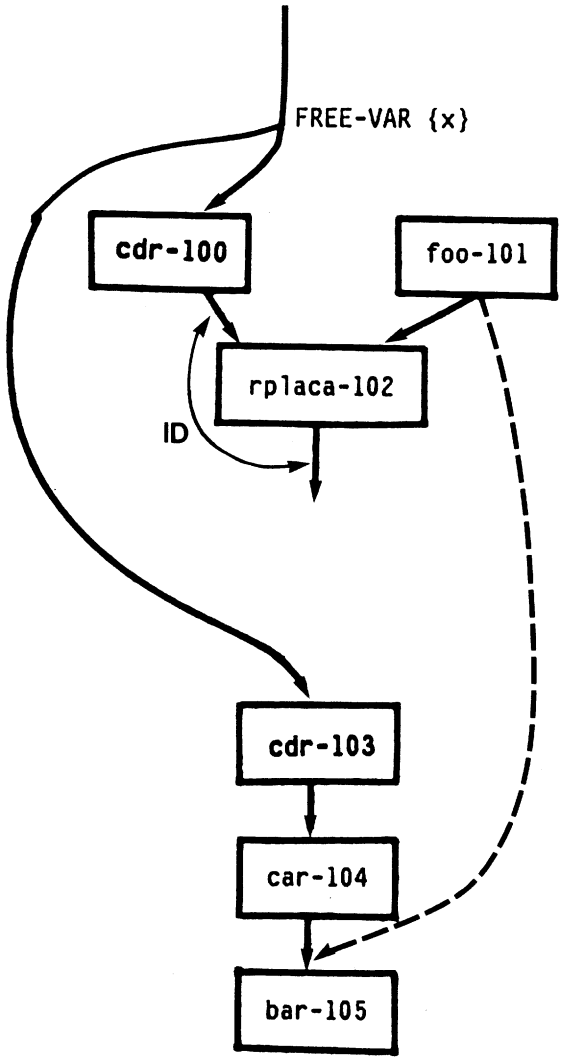


Figure 28. Data Flow by Side Effect on Data Structure.

CHAPTER FIVE

PROPOSED APPRENTICE ENVIRONMENT

The ultimate goal of our work on program description, reasoning, and flow analysis described in the preceding chapters is to build a unified programming environment in which the programmer is aided by the apprentice in the design, coding, and maintenance of LISP programs. The foundations of this cooperation between the programmer and apprentice are the notions of the plan and specs of a program, as defined in Chapter Two. In this chapter we are more explicit about how plans and specs are used in the various phases of programming; however this chapter is essentially speculation, since no implementation or partial implementation exists for any of the facilities described.

The chapter is divided into three sections dedicated to the topics of design, coding, and maintenance. Each section outlines the operation of an apprentice facility in the respective area.

One basic scenario for the combined use of all three of the three sub-systems described in this chapter is the following:

The programmer first engages the design and plan formulation system. The result of this interaction is a verified deep plan for the program. Then the programmer writes LISP code which realizes the segments in the deep plan, annotating them with comments that refer to the design concepts. This code is submitted to surface analysis (Chapter Four), and then to the recognition system. In the recognition, discrepancies between the intended design and the realized program are detected by the apprentice and corrected by the programmer. Successful recognition implies that the LISP program is a valid realization of the deep plan, and thus satisfies the specifications of the design. Furthermore a complete program description (CPD) has been built, which is the final documentation of the program. The CPD supports a question answering facility which helps the programmer maintain consistency of the program and design as changes are made in the future.

5.1 Program Design

In this section we examine how the deductive system of Chapter Three can be built upon to produce an interactive design facility to be used by an expert programmer. Since this chapter is only a guide for future research, we will merely outline several extensions which lead in this direction.

Use of Stored Plans

The basic deductive system described in Chapter Three can derive the teleology of a program given its data flow. However, in some cases describing a design by enumerating all the segments and data flow links is cumbersome. We would like to extend the current system to allow a programmer to use higher level plan descriptions, whose meaning is looked up in the the system's programming knowledge base. In particular, we would like the design system to be able to retrieve a general plan, such as SEARCH-LOOP, from the knowledge base and then refine it according to the programmers choice of data structures.

For example, suppose a programmer is engaged in the design of a hash table lookup routine. Lacking higher level plan concepts, the programmer would express the plan for this routine in terms of the following six segments: HASH, ARRAYFETCH, NULLTEST, FIRST-SEGMENT, MATCHTEST, and REST-SEGMENT. However it would be more natural if he could first specify the plan in terms of the two major steps: GET-BUCKET and SEARCH-BUCKET. The refinement of the first step, GET-BUCKET, is simply to specify its two internal segments, HASH and ARRAYFETCH, and the data flow between them. However the second step, SEARCH-BUCKET, can then very conveniently by specified as a refinement of the general plan for SEARCH-LOOP.

The refinement of the general plan for SEARCH-LOOP depends on the programmer's design choice for the implementation of buckets in the hash table. There are three standard schemes for doing this: hash-rehash, overflow array, and linked lists. The last of these is the most typical for LISP applications, so let us assume the programmer has made that design choice:

```
(IMPLEMENTATION-PART bucket bucket-list)
(MUST-BE bucket-list bucket list)
```

This design choice also implies that the FRONT of the search-space is implemented by the FIRST of the list, that the BUMP step in the search loop is performed by taking the REST of the list, and that the end test is a test for NIL. As explained in section 2.5, this information should be pre-compiled in the knowledge base so that once the programmer states his design choice, the refinements can be made automatically.

To complete the design of the lookup routine, the plan for SEARCH-BUCKET requires one additional refinement: the MATCHTEST inherited from the general SEARCH-LOOP plan needs to be specialized. This requires a choice of implementation for ENTRY's, e.g. as PAIRS

```
(IMPLEMENTATION-PART entry entry-pair)
(MUST-BE entry-pair entry pair)
(IMPLEMENTATION-DEFINITION
    (keypart entry key) <=>
    (left [entry-pair entry] key))
(IMPLEMENTATION-DEFINITION
    (datapart entry object) <=>
    (right [entry-pair entry] object))
```

Given this design choice, the appropriate specs for the MATCHTEST are:

```
(SPECS-FOR: matchtest-39
    (INPUTS: key-1 entry-2)
  (CASE-1
    (EXPECT: (keypart entry-2 key-1))
    (ASSERT: (keypart entry-2 key-1)))
  (CASE-2
    (EXPECT: OTHERWISE)
    (ASSERT: (not (keypart entry-2 key-1)))))
```

At this point the programmer would quite likely want to stop refining the design plan and simply write the code. The final verified deep plan arising out of this design example is shown in Figures 29 and 30. In section 5.2, we will see how a recognition system might bridge the gap between this plan and the actual code.

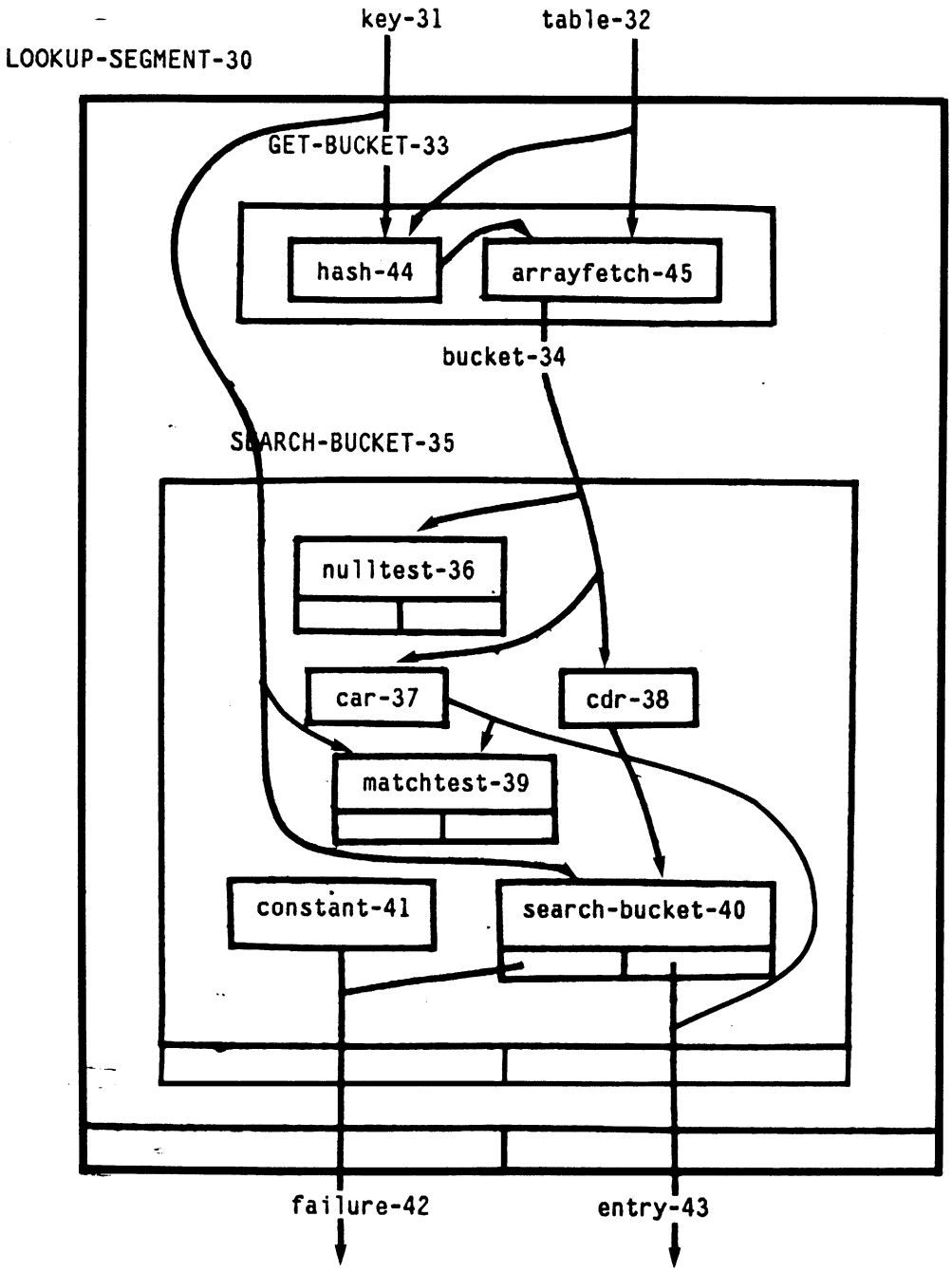


Figure 29. Data Flow in Deep Plan for LOOKUP-SEGMENT.

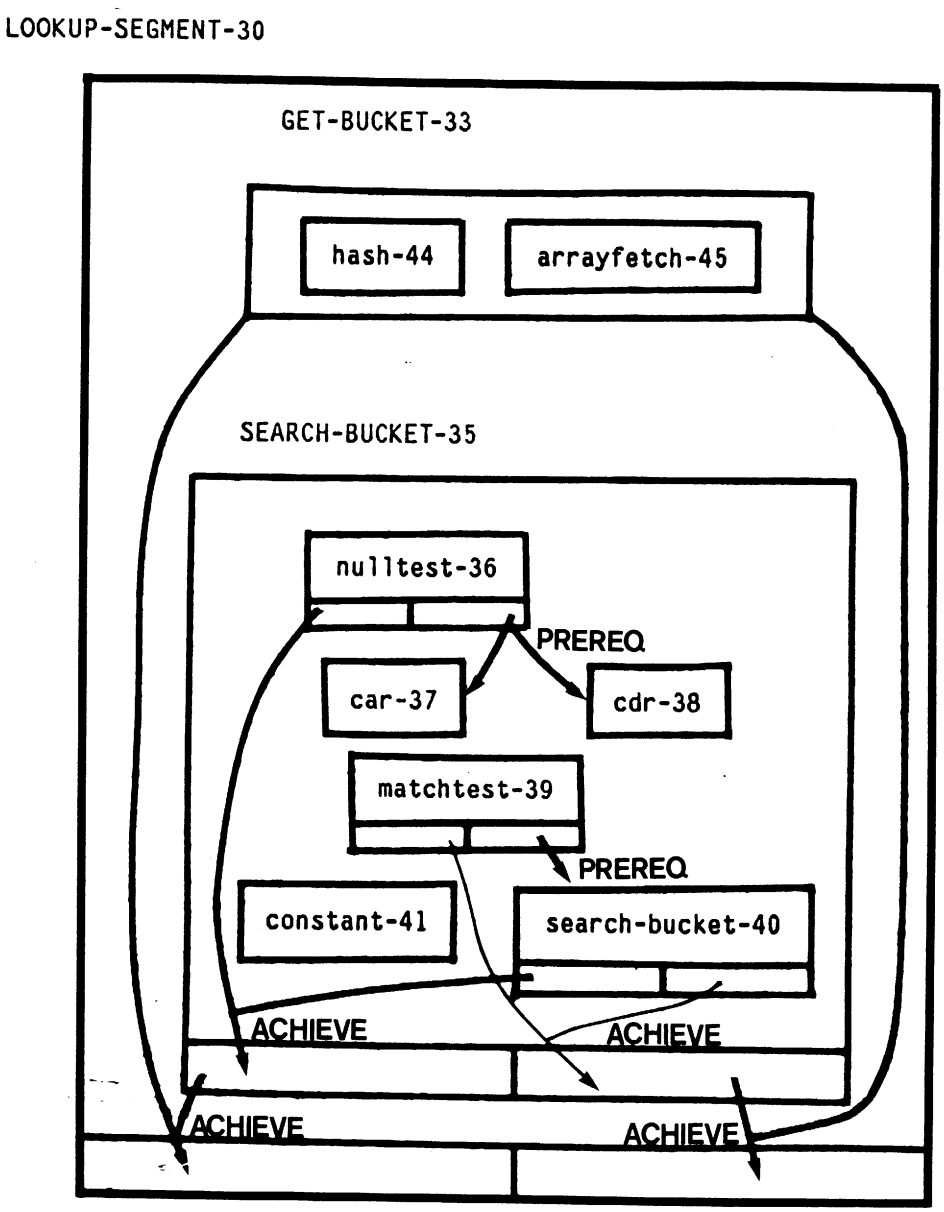


Figure 30. Purpose Links in Deep plan for LOOKUP-SEGMENT.
(Omitting Type Restrictions)

Perturbation Analysis

A second desirable improvement of the deductive system would be greater flexibility in the handling of mistakes. In the current system, when a bug is found in the programmer's design, i.e. when the system is unable to prove an expectation or an output assertion, it can warn the programmer; but there is no facility for recovering from the error. We would like to allow the programmer to intervene at this point by modifying the specs of a sub-segment or of the main segment so as to rectify the problem.

For example, if a main segment's output assertion cannot be proven, there are three possible interventions (other than giving up and starting again). One is to weaken the assertion and see if the less stringent condition can be proven (it may be that this weaker condition is all the rest of the program requires of this segment). A second approach is to strengthen the input expectations of the main segment while leaving the unprovable output assertion unchanged. It is possible that under this stronger input condition the output assertion is provable. Finally, a sub-segment's output assertion might be strengthened so as to make it possible to prove the main-segment's assertion.

Each of these interventions requires a significant amount of fix-up work by the deductive system. The changed expectations or output assertions must be propagated throughout the current plan appropriately. Furthermore any of the segments whose specs have been modified might well be used in other plans that have already been verified.

A similar perturbation capability is required for changes in implementation choice, as when certain data structures in the plan are extended or restructured to deal with new domain requirements. Since purpose links in the deep plan frequently depend on such choices, the system must be able to see which proofs are possibly affected and attempt a new proof using the new design choice. Such perturbation analysis will be a valuable tool during a program's evolution, allowing an orderly and consistent transition from one version to the next.

Plan Transformations and Modularity

A final category of additional design aids has to do with plan transformations and canonical forms for plans. Thus far in our discussion of the programmer's apprentice, we have viewed the drawing of segment boundaries (i.e. modularization) entirely as a matter of the programmer's discretion. However, if the apprentice makes use of stored plans as explained earlier, it becomes advantageous to modularize programs in a way that corresponds with the canonical form of the most general stored plan. For example, the hash table insert routine in Chapter Three takes the key and the data as separate input objects, building the entry in an internal sub-segment. By drawing different segment boundaries it could just as easily take an entry as its input, extracting the key internally for use by the hash routine. (See Figure 31).

Moreover, there is a significant argument that the second form of plan is more natural in terms of the hierarchy of programming knowledge described in Section 2.5. Hashing is a specialization of the associative-structures concept, which is in turn a specialization of the data-structures concept. The specs for a hash table insert are therefore a refinement of the specs for a general data-structure insert. The specs for the most general INSERT-SEGMENT do not take parts of an object as inputs, but rather the whole object to be inserted. (In fact, at this general level the object to be inserted does not necessarily have a parts sub-structure.)

We therefore envision a "cleanup" facility of the apprentice which would apply plan transformations to improve the modularity of programs, in accordance with its built-in criteria, e.g. correspondence with canonical plans in the knowledge base. In the insert example, the plan transformation is called "externalization of initial segments". An initial segment is a sub-segment all of whose EXPECT's are also EXPECTS's of the super-segment, and whose only inputs are also inputs of the super-segment. It is always possible to transform a plan to remove an initial sub-segment. This is done by eliminating all expectations of the super-segment that are used only by the initial sub-segment and by replacing them by the output assertions of the initial segment that other sub-segments depend on. This is equivalent to drawing the super-segment boundary so that the initial sub-segment is outside (see Figure 31).

Of course, the transformation of plans must not occur automatically or without control by the programmer, since the programmer may have intentionally designed an "messy" plan for efficiency reasons that cannot yet be explained to the apprentice.

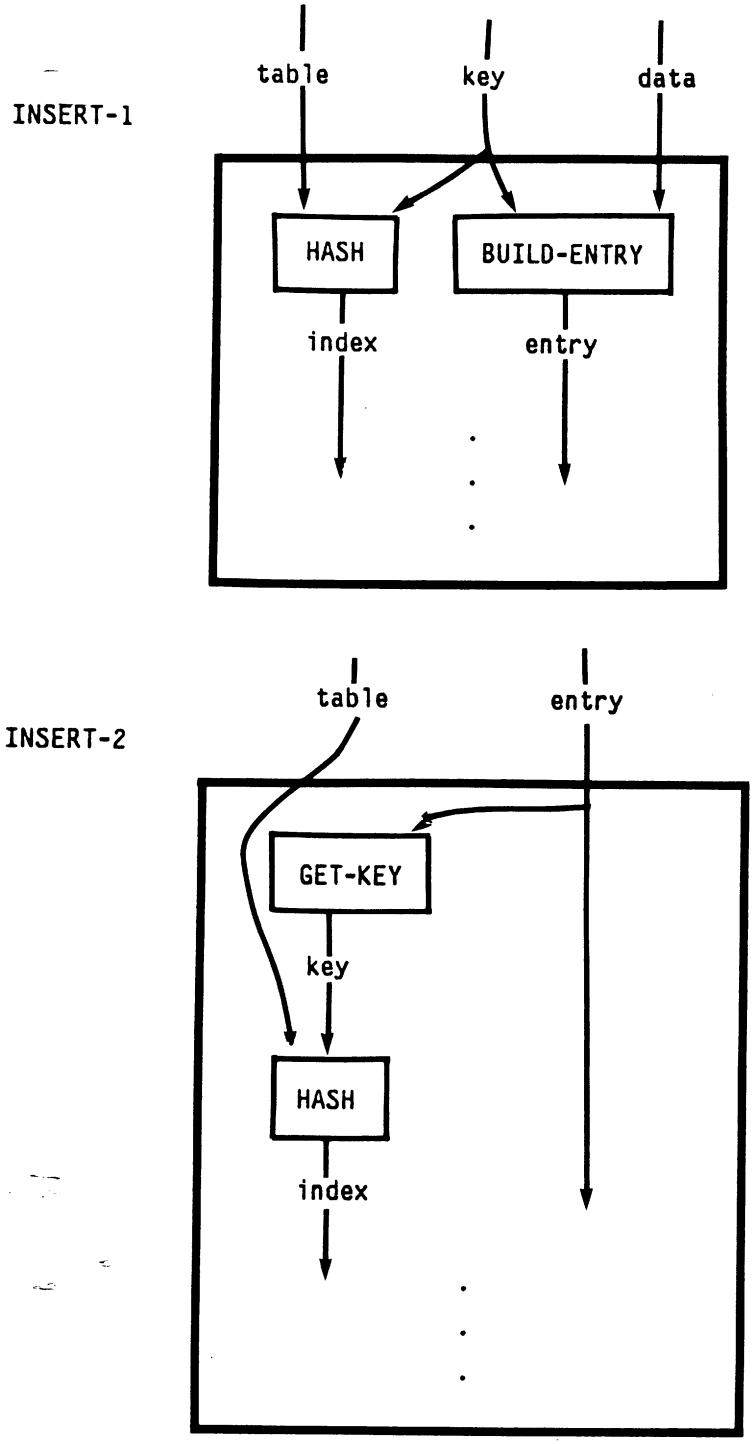


Figure 31. Externalization of an Initial Segment

5.2 Plan Recognition

This section is concerned with how to make connection between a surface plan, which is the output of flow analysis (Chapter Four), and a deep plan. We call this process "recognition" by analogy with visual recognition, to which it bears some similarity.

A surface plan is typically made up of many small segments all at the same level of description (except for grouping of loop bodies and lambda expressions), connected by data flow and control flow links. The segment and object types that appear in a surface plan are very low level concepts such as CAR, CDR, LIST , ARRAY , etc.

In contrast, a deep plan is typically a more hierarchical structure with many levels of segment nesting and grouping. Furthermore, segments and objects in a deep plan are described in terms of higher level programming concepts such as buckets, table, lookup, etc. In addition to data flow (and instead of control flow), deep plans typically have purpose links between segments. These purpose links express teleological dependencies between segments that constrain possible control flows in a corresponding surface plan.

Recognition entails merging these two kinds of plan for the same program into a single CPD. This is achieved by adjusting the structure of the surface plan until its segments correspond one-for-one with the segments of the deep plan, with the same data flow between them and control flow compatible with the purpose links. Adjusting the surface plan involves two basic processes: grouping and plan transformation. Grouping is simply the operation of drawing a segment boundary around a number of segments at the same level in the surface plan, thereby creating a new segment, and calculating the net data flow and control flow between the new segment, its sub-segments and other segments now at the same level. Examples of plan transformations are given in Section 2.5.

Recognition can be thought of as a heuristic matching process with data flow being the major criterion, since it appears in both surface and deep plans. Only when both data flow and segmentation are matched is the control flow checked for compatibility with the purpose links. The two simplest constraints between control flow and purpose links are that a segment must precede all segments to which it has a PREREQ link, and that an ACHIEVE link implies nesting. The constraint relationships between a more complicated network of purpose links and

allowable control flow have been studied (in a different formalism) by Sacerdoti <1975a>.

Since we have not yet implemented a recognition system, the best description of its control structure we can currently give is in the form of a illustrative example.

An Example Using LOOKUP

Consider the programming of a hash table lookup. Suppose the programmer used the design system described in Section 5.1 to formulate the deep plan shown in Figures 29 and 30. Then the following code is written:

```
%(design bucket-list)

(DEFINE LOOKUP (K)
   (PROG ( %(object bucket) BKT )
      (SETQ BKT (TBL (HASH K)))
   LP (OR BKT (RETURN %(object failure) NIL))
      (COND ((EQ (CAAR BKT) KEY)
             (RETURN (CAR BKT))))
      %(segment bump)
      (SETQ BKT (CDR BKT))
      (GO LP)))
```

The result of surface data and control flow analysis of this code is shown in Figures 26 and 27 of Chapter Four. What remains in recognition is to identify the higher level type of each object and segment in the surface plan and group segments to match the deep plan.

A first source of information is the explicit commentary provided by the programmer. From the annotation in this example (including interpretation of the mnemonic function name LOOKUP) the apprentice can immediately conclude:

```
(LOOKUP-SEGMENT segment-5)
(BUCKET object-4)
(FAILURE object-16)
(BUMP cdr-12)
```

Object types propagate along data flow links since the object type at one end must be the same at the other end. Furthermore, once a segment type is known, its specs determine the types of its input and output objects. Applying these propagations to the information recognized above leads to the following:

```
(KEY object-1)
(KEY object-3)
(TABLE object-2)
(FAILURE object-17)
(ENTRY object-18)
```

Notice that by propagation along data flow links it could also be concluded that

```
(ARRAY object-2)
```

since this object becomes the input to ARRAYFETCH-7. Whether this fact or (TABLE OBJECT-2) is discovered first depends on the particular control structure of the recognizer. In any case, the two facts are reconciled by referring to the design knowledge base in which the compatible implementation decision is recorded:

```
(TABLE-ARRAY table-32 array-33)
```

The surface plan for SEGMENT-5 has three segments at the first level of nesting inside of it, whereas the deep plan has two; thus a grouping is required. If there were no grouping clues available from the syntax, the apprentice might have to try each of the three possible groupings until it found a successful one. In the present example, however, there is a grouping suggestion left over from the surface flow analysis, which is shown as a dotted box in Figure 26. A new segment, SEGMENT-19 is created which groups HASH-6 and ARRAYFETCH-7. This allows the immediate recognition of GET-BUCKET and SEARCH-BUCKET, thereby completing

recognition at this level:

```
(PLAN-FOR: segment-19
    (SUB-SEGMENTS: (hash-6 arrayfetch-7))
    ... )

(GET-BUCKET segment-19)

(SEARCH-BUCKET loop-8)
```

Recognition is now applied to the surface plan of LOOP-8 versus the deep plan of SEARCH-BUCKET-35. Since these are iterative plans, the first order of business is to find the corresponding self-references or control jumps to the beginning of the loop. Deep plans, such as that for SEARCH-BUCKET-35, are typically represented self-referentially, whereas LOOP-8 comes out of surface analysis as a control flow loop. For the purposes of recognition it is necessary to transform control flow loop plans into the self-referential form. Applying this transformation to LOOP-8 results in the plan shown in Figure 32. SEGMENT-20 is then another segment with the same specs as LOOP-8.

```
(SEARCH-BUCKET segment-20)
```

Now the main features of the deep plan loop can be matched against the surface plan loop (we envisage a matching expert for loops). Since there is only one occurrence each of NULLTEST, CDR, and CONSTANT in the surface plan, these are easily recognized as the corresponding segment types in the deep plan. This leaves the three CAR's in the surface plan and the EQUAL to be matched against MATCHTEST-39 and CAR-37 in the deep plan. At this point the following syntactic grouping suggestion suggested by COND (indicated by dotted box in Figure 26) might be tried:

```
(PLAN-FOR: segment-21
   (SUB-SEGMENTS: (car-10 car-13 equal-14))
   ...)

(MATCHTEST segment-21)
```

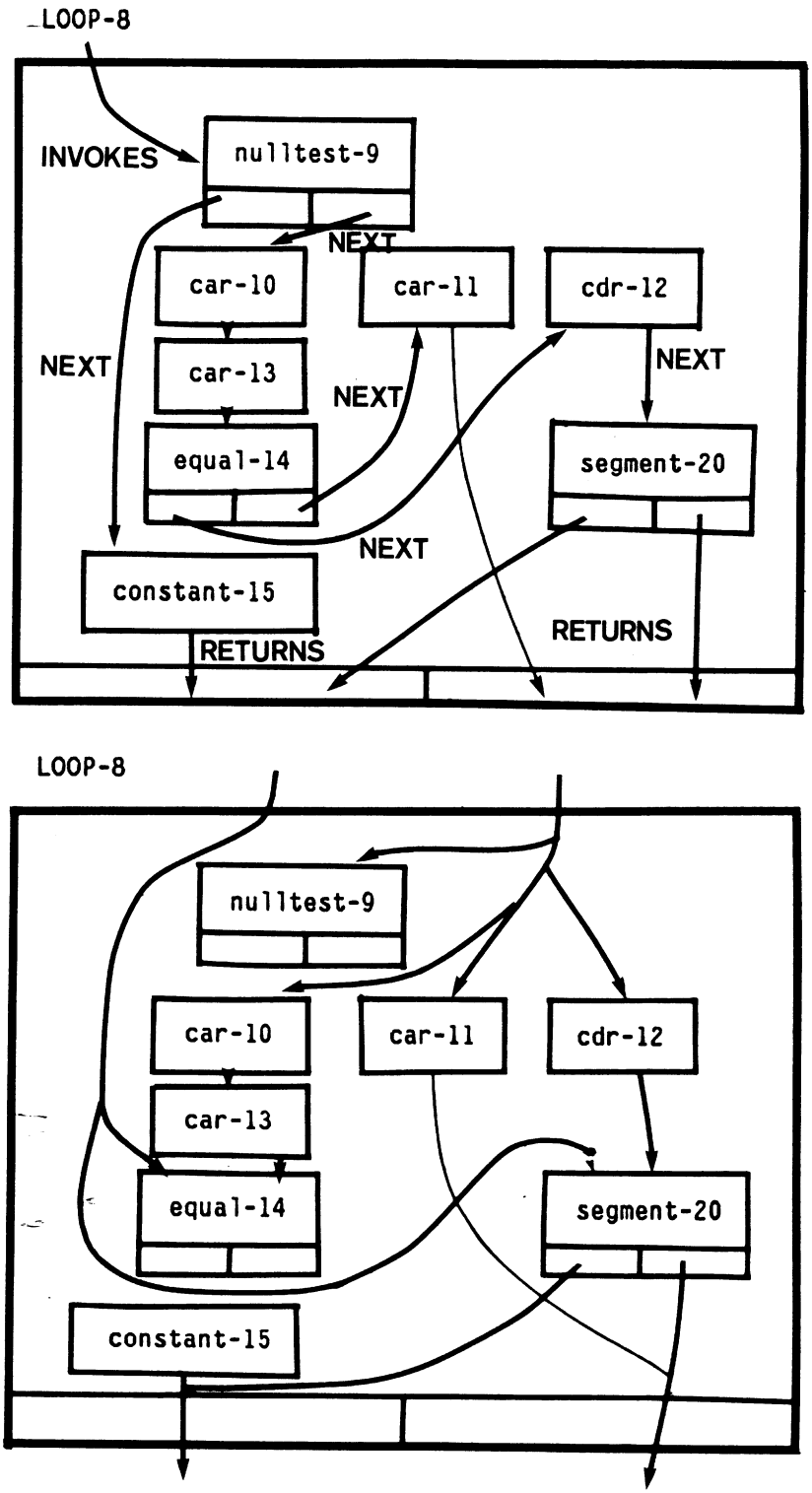


Figure 32. Control and Data Flow in Transformed Plan for LOOP-8

This grouping leaves CAR-11 as the match of CAR-37. Unfortunately this fails, since the data flow out of CAR-11 does not go into SEGMENT-21 (the MATCHTEST). The grouping of SEGMENT-21 is undone.

Another kind of transformation that can be applied to the surface plan in this example is the elimination of redundant segments. By this transformation CAR-11 can be eliminated by adding an extra data flow link out of CAR-10. Given this transformation, the recognition problem is simplified to a matter of deciding which of the two remaining CAR's in the surface plan now corresponds to CAR-37 in the deep plan. The simplest distinguishing constraint in this situation is the input-input data link from the BUCKET-34 object of the loop to CAR-37; only CAR-10 has the corresponding data flow in the surface plan.

With NULLTEST-9, CAR-10, CDR-12, and SEGMENT-21 recognized, the two remaining segments, CAR-13 and EQUAL-14, must be grouped to match MATCHTEST-39:

```
(PLAN-FOR: segment-22
    (SUB-SEGMENTS: (car-13 equal-14))
    ... )
```

Since only the specs, but not the plan, for MATCHTEST-39 was given in the program design, the apprentice must use the reasoning system to verify that SEGMENT-22 is a valid realization of MATCHTEST-39, rather than simply applying recognition recursively. The verification is achieved by the following steps: assert the pre-conditions of MATCHTEST-39; symbolically evaluate the plan for SEGMENT-22, starting in this situation; prove the post-conditions of MATCHTEST-39 in the resulting situation.

Now that segmentation and data flow have been matched, the control flow in the surface plan is checked against the purpose links in the deep plan and they are found to be compatible. As a final step, the purpose links are copied from the deep plan to the now-recognized surface plan, which then becomes part of the CPD for the program.

Control Structure

From this example, several general control structure features are clear:

(i) propagation of discovered information
(ii) bottom-up grouping
(iii) goal-directed grouping and plan transformation
(iv) hypothesis generation and testing

Propagation of object type information takes place along data flow links, and the discovery of a segment type leads to the propagation of further object types. Bottom-up grouping is done on the basis of syntactic clues discovered in surface flow analysis. Goal-directed grouping and plan transformation occur when the deep plan is used as a guide to how to group the surface plan. When there are no successful syntactic grouping clues, and the goal-directed strategy does not indicate a unique grouping, search behavior takes place by the generation and testing of alternative grouping or transformation hypotheses. A mechanism such as contexts will be required to facilitate the undoing of incorrect hypotheses.

Matching of data flow between the deep plan and surface plan may not always be as straightforward as in the example. As explained at the end of Chapter Four, programs can be written in which data flow is achieved by side effects on data structures. Discovering this sort of data flow in the surface plan requires use of the reasoning system.

The recognition procedure described here has a significant advantage over other approaches based on the use of pre-compiled schemata, such as Gerhart <1975a>. The stored schemas in Gerhart, for example, are patterns to be matched directly against program code. Recognition in the P.A., however, takes place with plans, which makes the formulation of the stored information and the matching process itself independent of the syntactic details of the programming language involved.

What About an Incorrect Program?

It was claimed that the apprentice approach to verification results in improved feedback to the programmer in the case of an incorrect program. To illustrate this, let us take two examples of incorrect LOOKUP programs and identify where the recognition breaks down. Some liberty will be taken in the paraphrasing of an error message in each case, based on the knowledge state of the apprentice at the failure point in the recognition.

The first example, shown in Figure 33, is a simple coding bug. We suppose that the programmer typed CAR instead of CAAR in the fifth line. The first phase of data flow and control flow analysis proceeds smoothly on this program, leading to a surface plan identical to Figures 26 and 27, except for the absence of CAR-13 . Surface analysis fails only if there is some violation of LISP language constraints.

The first level grouping of the plan into a GET-BUCKET and a SEARCH-BUCKET also succeeds as before. In the recognition of the SEARCH-BUCKET however, the underlined CAR (CAR-10) is taken as corresponding to CAR-13 in the deep plan, leaving only EQUAL-14 as the realization of MATCHTEST-39. The recognition then breaks down when the P.A. attempts to show that EQUAL-14 achieves the specs of MATCHTEST-39.

```
(SPECS-FOR: matchtest-39
    (INPUTS: key-1 entry-2)
  (CASE-1
    (EXPECT: (keypart entry-2 key-1))
    (ASSERT: (keypart entry-2 key-1)))
  (CASE-2
    (EXPECT: OTHERWISE)
    (ASSERT: (not (keypart entry-2 key-1)))))
```

To paraphrase a plausible error message based on the mismatch of specs, and also using design information from the knowledge base:

> Where you intended to test whether the given key was the key part of each entry, you are testing whether the entry itself is equal to the key. To extract the key part of the entry, take the car.

```
(DEFINE LOOKUP2 (KEY)
        (PROG (BKT)
              (SETQ BKT (TBL (HASH KEY)))
           LP (OR BKT (RETURN NIL))
              (COND ((EQ (CAR BKT) KEY)        <===
                     (RETURN (CAR BKT))))
              (SETQ BKT (CDR BKT))
              (GO LP)))
```

Figure 33. Program with Coding Error.

```
(DEFINE LOOKUP3 (KEY)
        (PROG (BKT)
              (SET BKT (TBL (HASH KEY)))
           LP (COND ((EQ (CAAR BKT) KEY)
                     (RETURN (CAR BKT))))
              (OR BKT (RETURN NIL))
              (SETQ BKT (CDR BKT))
              (GO LP)))
```

Figure 34. Program with Fencepost Error.

Figure 34 shows a program with an error of the type known generically as "fence post bugs". The programmer has put his match test before the end test in the loop, meaning that on the last element of the list his program will attempt to take the CAR of NIL. To see where the recognition breaks down, refer to the purpose links in the deep plan for SEARCH-BUCKET-35 shown in Figure 30. Notice that the data flow for LOOKUP3 is the same as for LOOKUP. Thus the recognition proceeds successfully up to the point where the control flow is checked against the purpose links. It is then discovered that the PREREQ links from the non-EMPTYLIST case of the NULLTEST to the LIST input expectations of the CAR's and CDR's are violated because the NULLTEST segment does not precede the CAR and CDR segments in the surface control flow of LOOKUP3. This could lead directly to the following error message:

> In your loop to search the bucket, the null test should preceed the car's and cdr's because these operations expect their inputs to be non-nil.

5.3 Answering Questions from the CPD

The deep plan and surface plan representations are intended to contain explicit answers to most questions that might be asked about a program. Questions of a strictly design nature can be answered without reference to the surface plan at all. In such cases, the surface plan may be used simply to find a concrete referent in the code, such as a variable name, to facilitate the P.A.'s discussion of a design concept. Descriptions of data structure are also part of the CPD.

This means that the implementation of a basic question answering facility is mostly a matter of information retrieval from an assertional data base containing the specs, plan, and structural descriptions employed in the program under consideration. Of course, it is not trivial to decide how best to communicate the desired information; deciding how much to say, using natural English, and other conventions of discourse are problems we have not studied.

We now consider representative examples from several classes of questions, and show how they are answered directly from the CPD.

Answers to what questions are typically provided by the specs of a segment. For example, if the programmer asks, "what does the insert do?", the apprentice could answer directly from the specs that it makes an entry (composed of a given key and data) member of the table. When the segment in question is also part of a plan, the apprentice could supplement the intrinsic description of the segment (i.e. specs) with a description of its role in the plan. For example, what CDR-12 in the preceding section does intrinsically is take the REST of the given list; its role in the loop plan, however, is the BUMP.

Where and when questions actually ask for similar information, namely an account of which segment performs the questioned action. For example, if the apprentice were asked, "where is the list bumped?", it might answer, "by CDR-12 in the search loop". In order to answer this it would merely have to search the plans in the CPD for a segment whose specs achieved the questioned action. If the question is phrased as a "when", an appropriate answer would provide reference points for the action, e.g. the name of preceding, following, and enclosing segments. In both cases, however, it is also important to determine the appropriate level of abstraction for the answer. For example, an answer like "somewhere in the hashing system" can be either very appropriate or inappropriate, depending on the questioner's current

focus.

Why and how questions are direct retrievals from deep plan structures. A "how" question is a request to produce a segment's plan. For example, suppose the apprentice were asked, "how does an entry get inserted into the table?". An answer to this is the following English paraphrase of the plan for INSERT: "by cons'ing it onto the bucket to which its key hashes, and re-storing the bucket in the table". "Why" questions view plan structure from the opposite perspective. If the apprentice is asked why a particular segment is employed, the answer is typically given by the ACHIEVE and PREREQ links that emerge from it, i.e. a segment is used as a prerequisite for another segment, or as part of achieving the goals of an enclosing segment.

General dependency questions are also answered by reference to purpose links. If the apprentice is asked what routines depend on the definition of a particular segment, it need only examine all the purpose links which emerge from the output half of its specs. Similarly, which segments it depends on is determined by looking at all incoming purpose links.

CHAPTER SIX

RELATIONSHIP TO OTHER WORK

A great deal of work in Computer Science in the past several years has shared our concern for bringing the complexity of large scale programming under control. Many approaches to the problem have been proposed; most, like ours, suggest utilizing the computer itself to help manage the complexity. In Section 6.1 we first present an overview of established research disciplines that are all aimed, more or less, at the same software problem. These disciplines can be contrasted along two dimensions: synthesis approaches vs. analysis approaches, and knowledge-based vs. non-knowledge-based. Following the overview, we will discuss some of the work in more detail and in relation to our apprentice.

6.1 Overview

Synthesis vs. Analysis

One way of comparing the various approaches to the software problem is to consider the decomposition of software into three components: the program specifications, the program code, and the justification (or proof) that the code satisfies the specifications. All three of these are needed for each program. There have traditionally been two approaches to achieving this goal. One branch of research has concentrated primarily on program synthesis, i.e the generation of correct code from high-level specifications. The other branch, program analysis, has tended to be more concerned with how to automatically generate a justification, given a particular program and its specifications.

The purest form of program synthesis is to go directly from a proof to its realization as code. An example of this is Darlington's <1973a> system, which automatically generates SNOBOL programs from formal proofs in the second-order logic. Automatic programming and structured programming are also part of the synthesis solution to the software problem. In an automatic programming system, correct code is produced automatically from specifications.

Although the justification of the program is not usually produced as an explicit output in such systems, it is presumably implicit in their internal operation. Structured programming and its related methodologies, such as "top-down programming", "provable programming" <Good, 1975>, and "stepwise refinement" <Dahl, 1972>, are the weakest kind of synthesis solutions. They increase the reliability of software by providing a more formal framework in which the programmer himself can do the synthesis.

Most of the work in the program analysis direction has been under the rubric of program verification. In this approach, producing the justification or proof for a program is generally viewed as an additional phase, following the design and coding. Early program verification had a strong mathematical flavor. For example, Hoare <1969> was concerned with constructing an axiom system within which desired properties of a program could be proven formally. However, the maturation of this field has been marked by the introduction of heuristics <Katz and Manna, 1973> and compiled schemata <Gerhart, 1975a> into program proofs.

Of course, not all work aimed at improving software economy and reliability is biased towards either the synthesis or the analysis approach. For example, underlying both analysis and synthesis is the need for better formalisms. Much of current research in programming languages and specification languages is directed at this need.

Use of Domain Specific Knowledge

A second fundamental distinction between various approaches to the software problem is whether or not specific knowledge about programming and application domains is incorporated. INTERLISP <Teitelman, 1972, 1974> is an example of a system that is highly developed in the non knowledge-based direction. The code manipulation tools and syntactic bookkeeping facilities of INTERLISP, such as indexers, editors, spelling correctors, and so on, are undoubtedly a boon to orderly programming. However, there is little potential for significant improvement in this direction unless the programming system has deeper levels of program representation which include the programmers's intentions and design strategies. Steiger <1976> has proposed a system which extends syntactic cross-referencing to include simple design dependencies between modules.

Also on the side of non-knowledge-based approaches are the new programming languages which arise periodically in the artificial intelligence and structured programming communities. These new languages usually have particular programming methodologies and special formal techniques associated with them. For example, the PLASMA language <Hewitt, 1973> is based on a methodology for the modularization of knowledge which uses the "actor" concept. Special techniques for symbolic evaluation <Hewitt, 1973a><Yonezawa, 1975>, protection, and synchronization <Hewitt, 1974><Grief, 1975> have been developed using actors. Similarly, the CLU language <Liskov, 1974, 1974a> draws its strength from theoretical work on the specification of abstract data types <Liskov, 1975><Zilles, 1976>.

The non-knowledge-based approaches described above all have potential applicability to any programming problem. In contrast, some researchers have invested a large amount of effort into learning how to represent specialized knowledge about application domains and programming techniques, so that it can be used in automatic systems. Most automatic programming research <Balzer, 1973> falls into this category. Program analysis research has also been moving towards the incorporation of more specific knowledge, as in Green <1975a> for sorting programs. Set in a larger context, questions of how to represent domain specific and procedural knowledge are the major content of most current work in artificial intelligence <Winston, 1974>.

Program Understanding

We use the term "program understanding" to identify a comparatively new approach to the software complexity problem that has its roots in the work of Sussman <1973> and Goldstein <1974>. Their theses emphasized the necessity of program annotation and explicit representation of a program's goal structure in order for there to be a real breakthrough in programming methods. Based on this view, Winograd <1973>, in an influential survey paper, proposed a unified programming environment called "A" in which synthesis and analysis are mixed using significant amounts of specific programming and domain knowledge where necessary. This has been followed in recent years by a spate of projects <Hewitt and Smith, 1975>, <Green, 1975a> <Goldstein and Miller, 1976>, including our own <Rich and Shrobe, 1975>, aimed at trying to actually construct parts of the ideal system described by Winograd.

6.2 Limitations of Other Approaches

Program Verification

An early attempt to solve the problem of expensive and unreliable software came from an approach to program verification based on mathematical logic. The aim of early work in program verification was to develop formal systems in which desired properties of a program could be proven as theorems. By transforming the software problem into these terms, experience and techniques from formal mathematics could be brought to bear on it.

The most significant development of this early mathematical period of program verification was the "axiomatic" approach of Floyd <1967> and Hoare <1969, 1971, 1972>. In this approach, the desired behavior of a program module is specified by pre- and post-conditions, which are restricted to be first order predicate calculus formulae relating the values of the program variables before and after execution. The program code is then written in an axiomatically defined programming language such as PASCAL <Hoare and Wirth, 1973>. The axiomatic definition of the programming language specifies formulae (or formula schemata) that are true before and after each command primitive in the language, and rules for combining these formulae according to the control primitives of the language. For programs without loops, verifying that a program satisfies its specifications is entirely transformed into the problem of proving that the post-conditions are theorems in the theory of the program, taking the pre-conditions as axioms. Thus, under the axiomatic approach, automatic program verification is equivalent to automatic theorem proving. Verifying programs with loops requires additional techniques such as structural induction <Burstall, 1969>, or inductive assertions on each loop <Knuth, 1968, Sec. 1.2.1>. A good example of an automatic verification system based on this approach is King <1971>.

There are major difficulties with the formal axiomatic approach to program verification. First of all, the state of the art in automatic theorem proving is not able to handle the complexity and length of proof arising out of the formalization of non-trivial programs. Even if automatic theorem proving techniques evolve to the point where the computation becomes feasible, there is the the remaining problem that the first order predicate calculus is not a natural (or perhaps even adequate) language in which to express program specifications and describe computations. Furthermore, the type of proof given by formal verification systems

(e.g. using resolution) is not usually understandable by typical human programmers. This problem is especially crucial when a proof fails, i.e. when the code does not agree with the specifications. A useful verification system must be able to give the programmer feedback in a form that will help him make appropriate corrections.

Recent research in program verification has matured in several directions that attempt to overcome the difficulties described with the early mathematical work. For example, in order to help control the computational explosions inherent in automatic theorem proving, heuristics <Katz and Manna, 1973> <Waldinger and Levitt, 1974> <Boyer and Moore, 1975> and interaction with the user <Deutsch, 1973> have been introduced into the proof process. Gerhart <1975a> has worked on the use of pre-compiled program and proof schemata to help overcome proof complexity. In order to make automatic proof systems easier for programmers to use, Wegbreit <1973> has developed some heuristic methods for automatically deriving the inductive assertions required for program loops. All of these elaborations of the axiomatic approach amount in essence to the addition of more specific knowledge about programming in the form of either heuristics or compiled patterns. We believe that this is indeed the only way to significantly improve the performance of verification systems.

There has also been activity towards developing better specification formalisms than the first order predicate calculus. Liskov and Zilles <1975> have developed a formal technique for specifying data abstractions in terms of possible operations on them. Parnas <1972> is working on a particular approach to writing input-output specifications. Scott <1972> has proposed a new fundamental semantics for programming based on lattice theory to replace traditional Floyd-Hoare axiomatics. Hewitt <1973> is also working on a new alternative semantic basis for program description. As this work on the foundations of verification matures, we feel it will also move into the problems of embedding more domain-specific knowledge in the programming semantics.

The approach to verification in our programmer's apprentice shares many features with the later, heuristic-based program verification systems described above, with two major differences. In our research, reasoning about programs and specifications is viewed as part of a complete programmer's apprentice environment, encompassing design, coding, and periodic modification. Thus, the deductive mechanisms we use depend on the existence of a rich and diverse knowledge base. The second distinguishing feature of our apprentice is the use of plans. We have found plans an extremely useful level of program description for organizing and

recording program justifications. From the standpoint of design, plan representations allow us to conveniently reason about partially developed programs. For completely coded programs, maintaining two levels of representation, i.e. code and plan, allows us to express programming and application domain specific knowledge in a more abstract form. Gerhart's <1975a> work, for example, suffers from the fact that her compiled schemata have to match programs at the raw code level.

The specification language and underlying semantic model of programming we have been using in our research thus far are non-innovative. This is in keeping with our desire to stay as close as possible to the natural way programmers think about their programs. Our overall model of a program is one of data flow between operations (as for instance in <Dennis, 1975>). The description of data structures in our current system includes only the traditional technique of naming subparts. Furthermore, the only fundamental difference between input-output specifications for operations in our system and those in the traditional Floyd-Hoare approach is that the variables in our specification conditions denote program data objects rather than literal program variables, as in Floyd-Hoare. An additional minor difference is that, since we are using a specially tailored deductive system, we have modified the standard quantifiers of predicate calculus to more convenient forms.

Automatic Programming

Automatic programming <Balzer, 1973> is another major approach to the software problem that has not fulfilled its initial promise. The goal of automatic programming research has been to create a system which, given "high-level" application oriented specifications for a program, will automatically generate correct and reasonably efficient code that satisfies the specifications. If such a system could perform over a wide range of applications it would be an appealing solution to present problems in programming. Unfortunately, automatic programming in this most ambitious form has foundered. It appears that the only way to succeed in building a totally automatic system is to highly restrict its domain of applicability. However, doing this loses most of the appeal of the automatic programming solution, since a large amount of effort must be repeated every time a new programming domain is encountered.

In addition to a demonstrated lack of technical progress in automatic programming, we believe the approach suffers from important conceptual difficulties that preclude its success in the near future. Most crucially, a totally automatic system depends on the user being able to give a complete specification of his needs. This is unrealistic for two reasons. First of all, it is an observed fact that computer users are seldom, if ever, able to spontaneously generate a complete set of specifications. In non-automatic programming, there is almost always interaction and further specification during the design phase. Secondly, even if a user is able to make all of his needs explicit at one time, the specification languages currently available to express them in are not adequate or convenient. (Letting the user express himself in unconstrained natural language is no answer either -- it only adds one more problem: the translation from natural language into the underlying representation used for specification.)

Another fundamental problem with the automatic programming approach is that it is not well-suited to partial performance. An automatic system that generates eighty or ninety percent correct programs is useless; more so even than a human programmer with the same error rate, since the automatic system's internal operation is usually unintelligible to its users. The way out of this morass is to aim for an appropriate division of labor between man and machine in the task of programming. Our research has begun with the assumption that an apprentice will only perform with partial success. Therefore, the internal operation of our system is based as much as possible on the same principles that human programmers use. Certainly, the descriptive tools we are building will figure into the eventual development of real automatic programming systems. However, in the short run we feel the apprentice approach is much more promising.

6.3 Program Understanding

Program understanding is an approach to the software problem which has its roots in artificial intelligence theories of planning and debugging rather than formal mathematics, and which is committed to incorporating domain specific knowledge wherever necessary in automated systems. In the next section, we briefly trace the origin of the important theoretical ideas on planning and debugging that are used in our apprentice project. Following that, we will review a number of concurrent and similar efforts at program understanding.

Planning and Debugging

The theoretical crux of planning and debugging research is the relationship between actions and their teleology. Two pioneering works in this field were the Ph.D. theses of Sussman <1973> and Goldstein <1974> at M.I.T.

Sussman chose the application domain of moving and stacking toy blocks on a table top (the then popular "blocks world"). His aim was to build a system which could automatically write programs to achieve goals in this simple domain, such as building a tower of three blocks. Sussman's main contribution was his beginning formalization of the debugging approach to program writing. His system first proposes a simple program to achieve the desired goal. It then runs the proposed program in a "careful" mode in which annotation is checked and a complete history is maintained in the form of process snapshots. Typically, the program fails; the history and annotation is then analyzed to find the "bug". Bugs are further classified into bug types, each type suggesting a characteristic fix to the program.

In order to achieve the debugging scenario described above, Sussman had to develop a way of describing the teleological structure of programs. In Sussman's view, each step in a program has some role in relation to other steps and the overall goal of the program. He called these relationships purpose links, which is the origin of our use of the term. Sussman identified two important types of purpose links in his thesis: prerequisite links and main-step (we call them achieve) links.

Closely related to Sussman's thesis is Goldstein's <1974> work on understanding programs in the domain of computer-drawn stick figures. Programs in Goldstein's domain are composed of a sequence of movement commands to a tv-screen drawing cursor. Like Sussman, Goldstein is interested in finding and correcting bugs in programs, though in Goldstein's case the programs are written by school children, rather than as the first pass of an automatic programming system. Goldstein came up with the same theoretical result as Sussman: debugging a program requires knowing the teleological relationship between the temporally ordered actions in a program (e.g. the primitive movements of the cursor) and overall desired result (e.g. the stick drawing of a house). Goldstein introduced the word plan to describe the network of purpose links that relate a program and its goal specification, a usage which we have continued. Furthermore, Goldstein began the important task of classifying plan types. In his thesis he originally identified three very abstract categories: round plans (which we call "loops"), linear plans (which he now calls sequential plans), and insertion plans (formalizing the use of interrupts and state-transparent constructions). This classification has been elaborated and refined in Goldstein and Miller <1976>.

Thus our major debt to Sussman and Goldstein lies in their establishing the correct conceptual foundation for understanding programs. However, we use plans and purpose links in a somewhat different fashion than they originallly proposed. Our apprentice project puts more emphasis on providing a convenient repository for programmer-supplied plan and purpose annotation during design and coding, rather than automatic programming or automatic debugging. Part of the reason for this is that we are aiming towards programming domains which are much more complicated than those that Sussman or Goldstein dealt with. We therefore find ourselves much more often in a situation of partial knowledge, in which interaction with the programmer becomes essential.

An important extension of Sussman's and Goldstein's work has been Sacerdoti's <1975> careful analysis of the interaction between temporal and causal constraints in plans, using a representation called "procedural nets", which are very similar to our plans. Although Sacerdoti's research is not framed in terms of a programmer's apprentice, we hope to make use of his results as our needs develop. Waldinger <1975> has also been working on ways of doing careful reasoning with plan-like structures. Hewitt <1975a> has shown the utility of considering constraints derived from carefully analyzing the goal in the planning process.

Programmer's Apprentices

The earliest definitive description of a programmer's apprentice like the one we are trying to build was by Winograd <1973>. He suggested that the way to overcome "the complexity barrier" was to construct a programming environment unifying editors, debuggers, programming language systems, and a knowledge base. (In an earlier paper, Floyd <1971> also proposed a unified programming environment which, however, did not include a data base for storing implementation and domain-dependent knowledge.) Furthermore, Winograd suggested the use of program annotation to help the system understand the goals, purposes and methods which the programmer is employing. What was lacking in Winograd's description is any idea of how to implement these various features. Our relation to Winograd has been to develop the theoretical and technical foundations from which it will be possible to build the tools he imagined.

A concurrent effort to lay the foundations for building Winograd's vision is the programmer's apprentice project of Hewitt et al. <Hewitt and Smith, 1975>. There is a large amount of common ground between our two projects. For example, we both have settled on use of intrinsic behavioral descriptions (Hewitt's term is "contracts") as the building blocks for representing program structure. Also, the symbolic evaluation in our deductive system is very close to Hewitt's notion of "meta-evaluation".

However, there is a crucial difference between Hewitt's approach and ours. As we see it, Hewitt is attempting a radical solution to the software problem. His apprentice will be built around a new programming language (PLASMA) and a new model of computation ("actors"). Hewitt's approach is based on the belief that this innovation will facilitate the construction of a good programmer's apprentice, particularly for programs involving communication between parallel processes, and that the existence of such an apprentice will then encourage people to convert to the new methodology. In contrast, our project is aimed towards the evolution of a programmer's apprentice within the traditional programming environment of LISP, with innovations such as the use of machine-understandable comments, extended data declaration features, and so on. However these two approaches supplement each other since, as we have seen already, much of the underlying theoretical content is transferable.

Other Program Understanding Systems

In a recent M.I.T. thesis, Ruth <1973> constructed a program understanding system which successfully analyzed correct and near-correct programs from an introductory programming class, giving specific comments about the nature of the bugs detected in the incorrect cases. Like us, Ruth attempts to capture in his system the commonality between programs which do the same thing. He represents a class of such programs as an augmented formal grammar which generates possible programs in a very high level programming language. Then, to analyze a student's program written in, say PL/I, Ruth has written specialized procedures which can recognize the occurrences of the high-level actions as local constructs in a student's program.

Although we agree with Ruth that understanding programs requires significant pre-compiled knowledge, his representation scheme is unsuitable for the type of programmer's apprentice we wish to build. As Ruth himself points out, his representation can only do well when there is a minimum of interaction between the parts of a program; this directly excludes the kind of situation we are most interested in.

A number of researchers are currently working on the problem of how to codify various types of programming knowledge. Green and Barstow <1975> have developed a system of rules which describe how to write sort programs. Their work seems to be thorough with respect to this particular domain of programming; however their rules are still written quite informally. Goldstein and Miller <1976> have accumulated a significant body of knowledge about types of plans in the stick-drawing world and are currently exploring the use of an augmented transition network grammar as a representation. Wilczynski <1975> has developed a production-like formalism, which he shows is natural to the domain of game playing programs. Also, Mikelson <1975> is working on a network data base representation to support automatic consulting on the use of a large subroutine library.

At M.I.T., Waters <1976> has proposed a system for analyzing mathematical FORTRAN programs, which uses a plan representation similar to ours. Genesereth <1976> is also currently using plan representations similar to ours as part of the knowledge base for an online advisor facility for MACSYMA, a large symbolic mathematics system. Finally, in a recent Ph.D. thesis, Brown <1976> has applied the knowledge-based planning and debugging approach to the domain of electronic circuits and has concurrently arrived at very much the same plan

representations as we use.

Two current areas of research potentially related to program understanding that we have not yet developed in our own work are synchronization of parallel processes and specification by examples. Grief <1975> and Owicki <1975> have proposed two dissimilar formal systems for talking about communicating parallel processes. Bauer <1975>, Hardy <1975>, Shaw <1975>, and Siklossky <1975> have been working on various approaches to the specification of data structures and procedures by way of examples.

Reasoning Techniques

The design of the deductive system in our apprentice has been influenced by a number of recent and concurrent works on general reasoning techniques, and the symbolic evaluation of programs. In the category of general reasoning techniques, our use of situational tags can be traced back to McCarthy and Hayes <1969>, and is reiterated with modification in Hewitt <1975a>. The notion of using anonymous and identified individuals for reasoning about quantifications is explained clearly in Hewitt <1973a> and Moore <1976>.

The concurrent work of Yonezawa <1975, 1976> on symbolic evaluation is also attempting to deal directly with the problems of reasoning about side-effects and shared data structures. Thus, there is a great deal of underlying similarity between our two systems despite the lack of superficial resemblence due to the fact that Yonezawa is working within the actor formalism of Hewitt et al. Also, Boyer and Moore <1975> have developed a powerful system for proving properties of simple LISP programs by means of symbolic evaluation. However, their techniques do not seem to be developing in the direction of accepting domain-specific knowledge and user interaction

BIBLIOGRAPHY

Allen, F.E. and Cocke, J. (1976) "A Program Data Flow Analysis Procedure", CACM, Vol. 19, No. 3, pp. 137-147.

Balzer, (1973) "Automatic Programming", Institute Technical Memo, University of Southern California / Information Sciences Institute, Los Angeles, Cal.

Bauer, M. (1975) "A Basis for the Acquisition of Procedures from Protocols", Fourth International Joint Conf. on A.I., U.S.S.R.

Boyer, Robert and Moore, Strother (1975) "Proving Theorms About LISP Programs", Journal of the ACM January 1975, pp. 129-144

Brown, A.L. (1976) "Qualitative Knowledge, Causal Reasoning, and the Localization of Failures", Ph.D. Thesis to be published, M.I.T. A.I. Lab.

Burstall, R. M. (1969) "Proving Properties of Programs by Structural Induction", Comput. Jl. vol. 12, pp. 4-8

Burstall, R. M. (1972) "Some Techniques for Proving Properties of Programs Which Alter Data Structures", Machine Intelligence 7, Edinburgh University Press.

Dahl, O.J., et. al. (1970) "The SIMULA 67 Common Base Language", Publication S-22, Norwegian Computing Center, Oslo.

Dahl, O.J., Dijkstra, E., And Hoare, C. A. R. (1972) Structured Programming, Academic Press, 1972.

Darlington, J. and Burstall, R.M. (1973) "A System Which Automatically Improves Programs", Third International Joint Conf. on A.I., Stanford U.

Darlington, Jared L. (1973a) "Automatic Program Synthesis in Second-Order Logic", Third International Joint Conf. on A.I., Stanford U.

Dennis, J.B. (1975) "First Version of A Data Flow Procedure Language", M.I.T. Project MAC Memo 61.

Deutsch, Peter (1973) "An Interactive Program Verifier", Xerox PARC Report * CSL-73-1. Palo Alto, Ca.

Donzea-Gouge, V., Huet G., Kahn, G., Lang, B., and Levy, J.J. (1975) "A Structure-Oriented Program Editor: A First Step Towards Computer Assisted Programming", Report 114, Institut de Recherche en Informatique et Automatique, France.

Floyd, R. W. (1967) "Assigning Meaning to Programs", Mathematical Aspects of Computer Science. J.T. Schwartz (ed.) vol. 19, Am. Math. Soc. pp. 19-32. Providence Rhode Island.

Floyd, R.W. (1971) "Toward Interactive Design of Correct Programs", IFIP, 1971.

Gerhart, S.L. (1975) "Correctness-Preserving Program Transformations", Proc. of 2nd Symp. on Principles of Programming Languages, Palo Alto.

Gerhart, S.L. (1975a) "Knowledge About Programs: A Model and Case Study", SIGPLAN Notices, Vol. 10, No. 6, Proc. of International Conf. on Reliable Software.

Genesereth, M.R. (1976), "Problem Solving with Imperfect Knowledge", submitted to Conf. on A.I. and Simulation of Behavior.

Genesereth, M.R. (1976a), "An Overview of the MACSYMA Advisor", unpublished progress report, M.I.T. Laboratory for Computer Science.

Goldstein, Ira (1974) "Understanding Simple Picture Programs" PhD. Thesis, M.I.T. A.I. Lab. Technical Report 294.

Goldstein, I. P. and Miller, M.L. (1976), "Intelligent Tutoring Programs: A Proposal for Research", M.I.T. A.I. Lab. Working Paper 122, Logo Lab. Working Paper 50.

Good, D.I. (1975) "Provable Programming", ACM SIGPLAN Notices, Vol. 10, No. 6, Proc. of International Conf. on Reliable Software.

Green, C. and Barstow D. (1975) "Some Rules for the Automatic Synthesis of Program", Fourth International Joint Conf. on A.I., U.S.S.R.

Green, C. and Barstow D. (1975a) "A Hypothetical Dialogue Exhibiting a Knowledge Base for a Program-Understanding System", presented at NATO Advanced Study Institute on Machine Representations of Knowledge, Santa Cruz, Cal.

Greif, I. (1975) "Semantics of Communicating Parallel Processes", M.I.T. Project MAC TR-154.

Hardy, S. (1975) "Synthesis of LISP Functions from Examples", Fourth International Joint Conf. on A.I., U.S.S.R.

Hewitt, Carl.(1971) "Description and Theoretical Analysis (using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot", M.I.T. A.I. Memo No. 251.

Hewitt, C., Bishop, P., And Steiger, R. (1973) "A Universal Modular Actor Formalism for Artificial Intelligence", Proceedings of IJCAI-73, Stanford California.

Hewitt, C., et al (1973a) "Actor Induction and Meta-evaluation", ACM Symposium on Principles of Programming Languages, Boston.

Hewitt, C. (1974) "Protection and Synchronization in Actor Systems", M.I.T. A.I. Lab. Working Paper 83.

Hewitt, C. and Smith, B. (1975), "Towards a Programming Apprentice", Proc. of IEEE Trans. on Software Engineeering, Vol. 1, No. 1, pp. 26-45.

Hewitt, C. (1975a) "How to Use What You Know", Fourth International Joint Conf. on A.I., U.S.S.R.

Hoare, C.A.R. (1969) "An Axiomatic Basis for Computer Programming", CACM Vol. 12, No. 10, pp. 576-583.

Hoare, C.A.R. (1971) "Procedures and Parameters: An Axiomatic Approach", Symposium on the Semantics of Algorithmic Languages, E. Engeler, ed., Springer.

Hoare, C.A.R. (1972) "Proof of Correctness of Data Representations", Acta Informatica, 1,4, pp. 271-281.

Hoare, C.A.R. and Wirth, N. (1973) "An Axiomatic Definition of the Programming Langugae PASCAL", Acta Informatica, 2,4, pp. 335-355.

Katz, S.M., and Manna, Z. (1973), "A Heuristic Approach to Program Verification", Third International Joint Conf. on A.I., Stanford U.

King, J.C. (1971) "Proving Programs to be Correct", IEEE Trans. on Computers, C-20, 11, Nov. 1971.

King, J.C. (1976) "Symbolic Execution and Program Testing", Comm. of the ACM, July, Vol. 19, No. 7, p. 385.

Knuth, D.E. (1968) The Art of Computer Programming, Vol. 1, Addison-Wesely.

Liskov, B. (1974) "A Note on CLU", M.I.T. Computation Structures Group Memo 112.

Liskov, B. and Zilles S. (1974a) "Programming with Abstract Data Types", Proc. of Conf. on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4.

Liskov, B. and Zilles S. (1975) "Specification Techniques for Data Abstractions", IEEE Trans. on Software Engineering, Vol. SE-1, No. 1.

London, R. (1975) "A View of Program Verification", ACM SIGPLAN Notices, Vol. 10, No. 6, Proc. of International Conf. on Reliable Software.

Manna, Z. and Waldinger, R. (1975), "Knowledge and Reasoning in Program Synthesis", Artificial Intelligence 6, pp. 175-208.

McCarthy, J. and Hayes, P. (1969) "Some Philosophical Problems from the Standpoint of Artificial Intelligence", Machine Intelligence 4, American Elsevier, N.Y.

McDermott, D. V., and Sussman, G.J.(1972) "The CONNIVER Reference Manual", M.I.T. Artificial Intelligence Laboratory, Memo 259.

Mikelsons, M. (1975) "Computer Assisted Application Definition", Proc. of 2nd ACM Symp. of Principles of Programmings Languages, Palo Alto.

Miller, M.L. (1976) "Cognitive and Pedagogical Considerations for a Tutorial LOGO Monitor: An Investigations into the Evolution of Procedural Knowledge", unpublished S.M. thesis, M.I.T. Dept. of E.E. and C.S.

Moore, Robert (1976) "Reasoning From Incomplete Knowledge in a Procedural Deduction System", M.I.T. A.I. Lab. Technical Report 347.

Moore, J.S. (1974), "Introducing PROG into the PURE LISP Theorem Prover", Xerox PARC Report CSL-74-3.

Owicki, S. (1975) "Axiomatic Proof Techniques for Parallel Programs", Dept. of Computer Science TR 75-251, Cornell U.

Parnas, D.L. (1972), "A Technique for Software Module Specification with Examples", CACM Vol. 15, No. 5.

Rich, C. and Shrobe H.E. (1975) "Understanding LISP Programs: Towards a Programmer's Apprentice", M.I.T. A.I. Lab Working Paper 82.

Rulifson, J.F., Derksen, J.A., and Waldinger, R.J. (1972) "QA4: A Procedure Calculus for Intuitive Reasoning", Stanford Research Institute, Artificial Intelligence Center, Technical Note 73, Menlo Park, Ca.

Ruth, Gregory (1973) "Analysis of Algorithm Implementations" M.I.T. Ph.d. Thesis, Project MAC Technical Report 130.

Sacerdoti, Earl D. (1975) "The Non-Linear Nature of Plans" Stanford Research Institute A.I. Group Technical Note 101

Sacerdoti, E. D. (1975a) "A Structure for Plans and Behavior", SRI Technical Note 109.

Scott, D. (1972) "Lattice Theory, Data Types and Semantics", in Formal Semantics of Programming Languages, Rustin (ed), Prentice-Hall, p. 65.

Shaw, D., Swartout, W., and Green, C. (1975) "Inferring LISP Programs from Examples", Fourth International Joint Conf. on A.I., U.S.S.R.

Siklossky, L. and Sykes D. (1975) "Automatic Program Synthesis from Example Problems", Fourth International Joint Conf. on A.I., U.S.S.R.

Steiger, R. (1976) "Interdependency Management in a Programming Environment", unpublished paper, M.I.T.

Sussman, G. J., (1973) "A Computational Model of Skill Acquisition", PhD. Thesis, M.I.T. A.I. Lab. Technical Report 297. Cambride Mass.

Sussman, G. J. and Brown, A (1974) "Localization of Failure in Radio Circuits, A Study in Causal and Teleological Reasoning", M.I.T. A.I. Lab. Memo 319.

Teitelman, Warren (1972) "Automated Programmering - The Programmer's Assistant", Proceedings of the 1972 FJCC, pp.917-921.

Teitelman, Warren (1974), INTERLISP Reference Manual, Xerox, Palo Alto, Ca.

Waldinger, R. and Levitt, K.N. (1974) "Reasoning About Programs", Artificial Intelligence 5, pp. 235-316.

Waldinger, Richard (1975), "Achieving Several Goals Simultaneously" Stanford Research Institute A.I. Group Technical Note 107.

Waters, R.C. (1976) "A System for Understanding Mathematical FORTRAN Programs", M.I.T. A.I. Lab Memo 368.

Wegbreit, B. (1973), "Heuristic Methods for Mechanically Deriving Inductive Assertions", Third International Joint Conf. on A.I., Stanford U.

Weinberg, G.M. (1971) The Psychology of Computer Programming, Van Nostrand Reinhold.

Wilczynski, D. (1975) "A Process Elaboration Formalism for Writing and Analyzing Programs", U. of S. Cal. Information Sciences Inst., ISI/RR-75-35.

Winograd, Terry (1973) "Breaking the Complexity Barrier (Again)" Proceedings of the ACM SIGIR-SIGPLAN Interface Meeting, Nov. 1973.

Winston, P. (1974) Ed. "New Progress in Artificial Intelligence", M.I.T. A.I. Lab. TR 310.

Yonezawa, Aki (1975) "Meta-Evaluation of Actors with Side-Effects" M.I.T. A.I. Working Paper 101. June, 1975

Yonezawa, Aki (1976) "Meta-Evaluation for Verification and Analysis of Actor Programs", Draft paper, M.I.T. A.I. Lab.

Zilles, S. (1976) "Data Algebra: A Specification Technique for Data Structures", Ph.D. Thesis forthcoming, M.I.T. Project MAC, Cambridge, Ma

CS-TR Scanning Project
Document Control Form

Date : 4/4/96

Report # AI-TR-354

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 212 (219-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form ☒ Funding Agent Form ☒ Cover Page
☐ Spine ☒ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:

(A) IMAGE MAP (1-212) UNN# TITLE PAGE, UNN# ABSTRACT, UNN# ACKN, UNN# TABLE OF CONTENTS (5), UNN# TABLE OF FIGS, 1-202 (213-219) SCAN CONTROL, COVER, PRINTER'S NOTE, DOD, TRGT'S (3) UNN# FUNDING AGENT

(B) NOTE! PAGES 129-155 ARE XEROXED COPIES OF MISSING PAGES TAKEN FROM ANOTHER COPY OF THIS REPORT.

Scanning Agent Signoff:
Date Received: 4/4/96 Date Scanned: 4/4/96 Date Returned: 4/11/96

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR-354	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Initial Report on a Lisp Programmer's Apprentice		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR*(s)* Charles Rich and Howard E. Shrobe		8. CONTRACT OR GRANT NUMBER*(s)* N00014-75-C-0643 N00014-75-C-0522
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE December 1976 13. NUMBER OF PAGES 217
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Programmer's apprentice	artificial intelligence	side effects
program verification	planning	
automatic programming	debugging	
software engineering	program specifications	
LISP	symbolic evaluation	

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The conceptual basis of the system lies in three forms of program description: (i) definition of structured data objects, their parts, properties, and relations between them, (ii) input-output specification of the behavior of program segments (specs), and (iii) a hierachical representation of the internal structure of programs (plans). The major theoretical work reported here is a representation for program plans which includes data flow, control flow, and also goal-subgoal prerequisite, and other dependency relationships between the segments of the program.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**