This work was presented to the faculty of the Graduate School of Yale University in candidacy for the degree of Doctor of Philosophy. The author is presently at the IBM Watson Research Center, Yorktown Heights, N. Y.

Program Construction from Examples

Phillip Dale Summers

Research Report #51

This work was partially supported by a Resident Study Fellowship from IBM Corporation.

ABSTRACT

Program Construction from Examples

Phillip Dale Summers

Yale University 1975

An automatic programming system, THESYS, for constructing recursive LISP programs from examples of what they do is described. Programs are specified by a finite set of examples. Each xample consists of an input and the desired output. The program construction system is described in terms of a set of algorithms that define a series of transformations from the set of examples to a program satisfying the examples.

The program generation methodology is based on current theories of programming and program language semantics. Theoretical models of data and control structures are presented in proving the validity of some of the synthesis algorithms. The system includes algorithms for 1) deriving the specific computation associated with a specific example, 2) deriving control flow predicates and 3) deriving an equivalent program specification in the form of recurrence relations. A procedure is given for generalizing certain kinds of recurrence relations into recursive programs. A detailed description of the construction of a number of programs is presented to illustrate the application of the various algorithms.

To Linda
Eric and
Michael

ACKNOWLEDGEMENTS

I wish to express my gratitude to Prof. Alan J. Perlis who supervised this work and who has been a source of many insights into the problem. I also wish to thank the other members of my committee, Profs. Richard Lipton and Larry Snyder. Larry Snyder's interest and personal encouragement went far beyond the call of duty.

I am grateful to the IBM Corporation whose Resident Study Fellowship supported me during my residency at Yale and who have continued to support this research by permitting me to work full time at completing this dissertation. The manuscript was typeset by the author using facilities at the IBM Research Center.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The goal of this thesis is to produce a system that will generate programs from examples of what they do. Since this work addresses the problem of automatic generation of programs it may be considered to be a contribution to the field of automatic programming. A history and survey of this field is presented in Chapter 2. In addition to specifying and implementing a system that will produce programs from examples, there has been an attempt to characterize the work in terms of current theories of programming and program semantics. The components and algorithms used in the system are presented as theorems about the nature of programs from the programming domain specified by the thesis. This development is presented in Chapter 3 and is intended to be a precise description and formal justification of the process of programming by example. The capabilities of the system are illustrated by a series of examples in Chapter 4. Chapter 5 concludes the thesis with some observations about future work that might follow from this research. This leaves the remainder of this chapter to give an informal preview of the process of programming by example.

The term, programming by example, will be taken to mean the act of specifying a program by means of examples of the programs behavior. An example will consist of an input and an output. The output is what the program being specified is supposed to produce when it is applied to the example input. A processor that is to produce programs from such a specification is a lot like a compiler for a programming language consisting of examples. The examples convey information about the program to be generated in two ways. First, a single example indicates a specific transformation that must be duplicated by any program defined by the examples. This transformation is precisely that required to produce the example output from its input. The second source of information is that conveyed by the relationships that exist among the examples. Examples are a useful way of describing large or infinite objects only when these objects have repetitive properties that may be demonstrated by means of a finite number of examples. For any set of examples there are an infinite number of objects that have as a partial description the particular finite set of examples. Program specification by example is the same way. For a given set of examples of a program's behavior there are an infinite number of different programs which may satisfy these examples. We will say that a program satisfies or is defined by a set of examples if it produces the example output for each application of the program to the example's input. What the system is to do is to produce the simplest program which satisfies the examples. If the system is able to detect a repetitive phenomenon in the examples it attempts to encode this phenomenon as a recursive program, otherwise it produces a simple non-iterative program that is defined for just the set of examples. Indeed, there is no reasonable way to do more.

Having settled on an abstract syntax for the system, *ie.* that the language will have as statements examples of transformations on data items, the next step is to fix the target language for the programs the system is to generate. For reasons presented in Chapter 3, the target language is a form of basic LISP using the primitives **car**, **cdr**, **cons** and **atom**. Control structures of recursion, function composition and the McCarthy conditional are permitted in the construction of programs. The choice of language thus fixes the concrete form of the examples to be that of LISP S-expressions. Figure 1.1 illustrates the problem of the thesis by giving a set of examples and the program we expect the system to produce.

```
(A . B)->(B . A)
(A . (B . C))->((C . B) . A)
(A . (B . (C . D)))->(((D . C) . B) . A)

f[x]←[atom[cdr[x]]→cons[cdr[x];car[x]];
      T→cons[f[cdr[x]];x]]
```

Figure 1.1: Examples and resulting program.

As an initial approach to the problem, we might try to carry out the following paradigm for program synthesis.

1. For each example, try to find a function that will produce the output from the input using current technology for program synthesis. This function will be called a program fragment or fragment. For example, we might use a theorem proving approach where the example output is proven from the example input and some axioms about the programming language.
2. Assume that each program fragment is a computation sequence for the program we are trying to generate. Call the set of fragments or computation sequences corresponding to the examples, C_E.
3. Have a library of program templates (schemata) that may be used to generate a family of uninterpreted computation sequences, C_S.
4. Using the set of computation sequences associated with the examples (C_E), try to find substitutions in the set of sequences C_S so that they are equal to C_E. (If the substitutions exist we have found values for the function names, which determine a program except for determination of the control flow predicates.)
5. Instantiate the schema with the substitutions found in step 4.
6. Using symbolic execution, run the instantiated schema on the examples. The examples and the associated functions determine the control paths for the schema thereby generating sample values for the predicates. Use these values as examples of the predicates and repeat the synthesis process to determine the control flow predicates.

1. Generate fragments (step 1):
   C_E={ f_1[x]=cons[cdr[x];car[x]],
   f_2[x]=cons[cons[eddr[x];cadr[x]];car[x]],
   f_3[x]=cons[cons[cons[cdddr[x];caddr[x]];cadr[x]];car[x]] }

2. From library of schema try: f[x]←[p[x]→a[x];
   T→b[f[c[x]];x]]
   Generate computation sequence, C_S (step 3):
   C_S={ a[x], b[a[c[x]];x], b[b[a[c[x]]];c[x]];x] }

3. Find substitution for function names in C_S (step 4):
   substitute cons[cdr[x];car[x]] for a[x]
   substitute cons[x;car[x]] for b[x;y]
   substitute cdr[x] for c[x]

4. Form the instantiated schema (step 5): f[x]←[p[x]→cons[cdr[x];car[x]];
   T→cons[f[cdr[x]];car[x]]]

5. Generate examples for the predicates (step 6):
   p[(A . B)]=p[(B . C)]=p[(C . D)]=T
   p[(A . (B . C))]=p[(A . (B . (C . D)))]=p[(B . (C . D))]=F
   Predicate is atom[cdr[x]]

6. Generate program: f[x]←[atom[cdr[x]]→cons[cdr[x];car[x]];
   T→cons[f[cdr[x]];car[x]]]

Figure 1.2: Generation of program of Figure 1.1.

Figure 1.2 illustrates the application of this paradigm to generate the program of Figure 1.1. This approach to program generation, although very general, has a number of drawbacks. The major one being the use of methodologies that are terribly inefficient. For example, implementing the first step of the procedure as a theorem prover based synthesizer would incur all of the combinatorial inefficiencies of the theorem proving system. Figure 1.3 illustrates the complexity of even the simplest of derivations using this approach. Lines 1-4 of the proof in Figure 1.3 are meant to be suggestive of the kinds of axioms needed by a theorem prover based synthesizer. Lines 5-10 are derived from the axioms by a resolution-like rule of inference. Note that Figure 1.3 lists only the steps of a successful proof. No reference is made to terms that might be generated but not used by a theorem prover in a proof, for example the expression (A . C) would follow from steps 1, 5, 7. This coupled with the possible combinatorial search and matching algorithms of step 4 makes this approach very inefficient. A second objection to this approach is that it provides for little insight into the process of program specification by example. By approaching the problem as a series of

complex searches through various solution spaces, the approach fails to isolate the usual issues of program construction: choice of data and control structures.

Prove ((C . B) . A) from

1. ∀xy cons[x;y]=(x . y) Axiom
2. ∀xy car[(x . y)]=x Axiom
3. ∀xy cdr[(x . y)]=y Axiom
4. (A . (B . C)) Axiom (input)

5. A (2,4)
6. (B . C) (3,4)
7. B (2,6)
8. C (3,6)
9. (C . B) (1,8,7)
10. ((C . B) . A) (1,9,5)

Working backward from the theorem we can reconstruct the fragment:

f[input]=cons[step9;step5]
=cons[cons[step8;step7];step5]
=cons[cons[cdr[step6];car[step6]];step5]
=cons[cons[cdr[cdr[input]];car[cdr[input]]];car[input]]

therefore f[x]=cons[cons[cddr[x];cadr[x]];car[x]]

Figure 1.3: Fragment construction from proof.

It became a secondary goal of this thesis to define a system that would reflect an understanding of the issues of program construction from examples. To this end considerable time was spent in analyzing the problem in an attempt to gain an understanding of the problem and its inherent issues in the hope that with such an understanding would come efficient algorithms for the program generation system. This second goal has been met at least in part and has lead to the following improvements in the procedure for synthesizing a LISP programs from examples of its behavior.

1'. Analysis of the LISP primitives and data structures produced an algorithm for the direct generation of the program fragment characterizing each example. Thus eliminating costly theorem proving technology.

2'. By defining a lattice-theoretic characterization of the domain of S-expressions an algorithm was found for generating the control flow predicates for the McCarthy conditional from the example inputs. This eliminates the need for symbolic execution and the re-application of the algorithm in finding the predicates.

3'. It was found that the repetitive nature of the functions being generated could be characterized by recurrence relations derived from the predicates and functions found in steps 1' and 2'. These recurrence relations are analogous to difference equations and are sufficient to derive the control structure for the function being generated. This step replaces the costly step 4 of the previous approach.

4'. The class of programs capable of generation by the system was enlarged by a heuristic based on the logical inference rule of universal generalization.

All of these algorithms and techniques are presented as theorems in Chapter 3. To give an informal introduction to these techniques we will present an example of the synthesis of a program using some of them.

As an example of the program generation procedure we will synthesize the LISP function **unpack**. This function unpacks a list into a list of lists of each element of the original list. A set of examples that might be given to the system is

```
(A)->((A))
(AB)->((A) (B))
(ABC)->((A) (B) (C))
(ABCD)->((A) (B) (C) (D))
```

Each example represents a transformation that is to be encoded as a function formed from an appropriate composition of the primitive functions **car**, **cdr** and **cons**. In order to arrive at this function algorithmically it is necessary to note a particular property of S-expressions. Any non-atomic S-expression is uniquely constructed from the function **cons** and two 'less complex' S-expressions. The functions **car** and **cdr** provide for the unique decomposition of any non-atomic S-expression. By noting that all S-expressions may be constructed and decomposed in a unique way it is possible to define an algorithm that will construct a program fragment from an example. For the examples given above we have the following four functions.

f_1[x]=cons[x;nil]
f_2[x]=cons[cons[car[x];nil];cons[cdr[x];nil]]
f_3[x]=cons[cons[car[x];nil];cons[cons[cadr[x];nil];cons[cddr[x];nil]]]
f_4[x]=cons[cons[car[x];nil];cons[cons[cadr[x];nil];cons[cons[caddr[x];nil];
cons[cdddr[x];nil]]]]

For example, if f_2 was constructed from (A B)->((A) (B)) by first decomposing (A B) into A, (B), B and () and remembering that these values were derived from the input by the functions car[x], cdr[x], cadr[x] and cddr[x] respectively. When the subexpression, (A), was needed for the output it was constructed from the expression cons[car[x];nil]. Such expressions could be combined to form an expression that would evaluate to the output when applied to the input of an example.

The next step in synthesizing a program is to examine the relationships among the examples in an attempt to detect some repetitive pattern that may be used to define a recursive function. The domain of the function to be synthesized is inferred from the example inputs. Consider the four S-expressions (A), (A B), (A B C) and (A B C D) as being examples of the kinds of expressions in the domain of the function. Since these expressions are meant to be representative of S-expressions in general, we will assume there is no inherent meaning in the choice of the atoms used in these expressions. The important attribute of these expressions is their structure, each one being slightly more complicated than the one before. We may derive a series of predicates that characterize the differences between one of the examples and the next more complicated one. A series of such predicates, for the four S-expressions given above, would be $p_1[x]$=atom[cdr[x]], $p_2[x]$=atom[cddr[x]] and $p_3[x]$=atom[cdddr[x]] respectively, *ie.* the difference between (A B) and (A B C) is indicated by the predicate $p_2[x]$ which distinguishes between the two expressions by being true when applied to (A B) and false when applied to (A B C).

Using the predicates and fragments generated above we may construct a non-recursive program which satisfies the examples:

$f[x] \leftarrow [p_1[x] \rightarrow f_1[x];$
$\quad p_2[x] \rightarrow f_2[x];$
$\quad p_3[x] \rightarrow f_3[x];$
$\quad T \rightarrow f_4[x]]$

If the examples have some repetitive property, the property should be reflected in the fragments and the predicates. To see if the computation defined by the examples is repetitive we attempt to find the difference between the fragments and the difference between the predicates. A difference will be said to exist between two functions if one may be expressed in terms of the other. For the program being considered we find the following differences:

$f_2[x]$=cons[car[x];f_1[cdr[x]]]

$f_3[x]$=cons[car[x];f_2[cdr[x]]]

$f_4[x]$=cons[car[x];f_3[cdr[x]]]

$p_2[x]=p_1[cdr[x]]$

$p_3[x]=p_2[cdr[x]]$

Since the differences are of the same form we may encode these relationships as two recurrence relations:

$f_1[x]$=cons[x;nil]

$f_{k+1}[x]$=cons[car[x];f_k[cdr[x]]] $1 \leq k \leq 3$

and

$p_1[x]$=atom[cdr[x]]

$p_{k+1}[x]=p_k[cdr[x]]$ $1 \leq k \leq 2$

By an inductive inference we extend the recurrence relations to all $k \geq 1$, thus increasing the domain of S-expressions over which the transformation indicated by the examples is defined. The recurrence relations define the recursive program:

unpack[x]$\leftarrow$[atom[cdr[x]]$\rightarrow$cons[x;nil];
$\quad$ T$\rightarrow$cons[car[x];unpack[cdr[x]]]]

This function satisfies the examples for the function **unpack** and is the program produced by the system.

This very brief derivation of a program from a set of examples illustrates three of the major components of the system: fragment generation, predicate generation and recurrence relations as a representation of a recursive function. The fourth major system component, the variable addition heuristic, was not needed in this exercise but may be observed in a number of the examples in Chapter 4.

CHAPTER 2

AUTOMATIC PROGRAMMING

Automatic programming is the name given to a variety of computing projects directed towards making a computer easier to use. There appears to be little agreement as to a common definition for such activities. At best one may settle on some broad goals to be aspired to by activities in this field. The goal common to all such efforts is to design systems which shift the effort of program preparation from programmers to a computer system. Progress, then, is measured in terms of decreased human effort in preparing programs with increased function. My own definition of an automatic programming system includes the following properties. It should be a system which 1) helps a user to produce a program which will solve his problem and 2) should not just produce a solution but should produce a program whose execution will produce a solution for the user of the system. The user may be a sophisticated computer scientist or an application person naive in the ways of computing.

It has been observed by [Goldberg 1974] that there are two aspects to research in automatic programming. The first involves the development of interfaces between man and machine and the second consists of the development of efficient algorithms to support these interfaces. Research in automatic programming systems generally falls into one of the two areas depending on what is perceived to be the crucial issues of the research. For example, current efforts in natural language programming generally concentrate on developing algorithms to support an existing English or other natural language interface. On the other hand, two dimensional programming languages, see for example [Wells and Morris 1972] or more recently the description of a language for producing customizable programs [Howe *et al.* 1975] are efforts concerned with developing new interfaces for the user of a computer. The distinction between the two kinds of automatic programming research is useful in helping to evaluate and understand the various efforts in automatic programming.

2.1 *History:*

The first automatic programming systems were the assemblers, compilers and interpreters [Hopper 1952], [Backus 1954], [Backus 1957], [Perlis and Smith 1957] written for the early computers. These systems helped scientists and business people to prepare programs without their having knowledge of the machines on which the programs would be run. By simplifying the user interface to the machine and providing a notation more nearly like the user's own working language, the higher level languages provided the first major step toward automatic programming. High level languages were useful because they assumed some information about the use to which

the language would be put. In assuming that the array was a useful construct for scientists and engineers and then defining such a construct in the original Fortran language the designers of the language permitted a programmer to use the construct without having to program its definition.

The advantages of high level programming languages stimulated numerous groups to propose, design and implement a variety of general and special purpose programming languages during the late 1950's and early 1960's. The developments of new languages were, for the most part, little more than refinements to the basic concepts introduced by the original languages in the early 1950's. A possible exception being the introduction of the language *APL* which defined a large and powerful set of conceptual constructs for a class of users interested in scientific applications. Research at this time consisted of advancing the technology used in the definition and implementation of programming languages. Many significant issues were raised for example, questions of variable bindings, parameter passing, language specification languages, recursion, *etc.*, all of which furthered the state of the art of programming languages but did little toward meeting the goals of automatic programming. By and large, the decade from the mid 1950's to the mid 1960's was the era of programming language technology, and the issues of automatic programming were overlooked as a subject for research.

During this same period, computer scientists began to formulate criteria by which the various languages could be judged. There was one observation that set the tone for much of the work of this period. This was noticing that specifying flow of control in a program was difficult and that programs became easier to write and to understand if the programmer was able to use the language to specify *what* he wanted done rather than *how*. A prediction about the nature of future programming languages that embodied this notion was made by A. E. Glennie [Glennie 1960] in a paper written for the first volume of a series entitled *Automatic Programming*

> 'The major improvement in language will be an increase in the use of *declarations* and these will enable the language to become simpler and more flexible. These declarations correspond to statements of ... the implicit information in a problem.'

By declarations Glennie was probably referring to components of a language that enabled a programmer to say *what* he wanted computed rather than *how*. As was previously noted, declarations or other such constructs represent conceptual units that must some how correspond to elements of the problem being solved if they are to be useful. In order to provide such conceptual constructs in a language, knowledge about the application being programmed must be encoded in the language system. It has only been recently that systematic studies of how to extract such knowledge from problem domain have been undertaken [Miller and Becker 1974].

Through most of the 1960's programming language research (sometimes called automatic programming research) focused on designing languages which would shift the kind of program specification from that of how a calculation is to be carried out to that of specifying what should

be done. Most of this activity was concentrated in two areas: the specification of extensible languages and the specification of non-procedural languages. The advertised advantage of both types of languages was that they permitted programmers to work directly with constructs of his problem domain rather than having to implement the constructs directly in his program. This permitted programs to be defined with more declarative statements and less procedural ones, thus making the programs easier to write. Again much of the work in these areas became directed toward the technology of providing such facilities and almost no real work was spent in trying to define what constructs or extensions would be needed by various groups of computer users.

In 1965 J. A. Robinson published a description of a new method for a mechanized deductive system for the first order predicate calculus [Robinson 1965]. This was to have great impact on automatic programming as it prompted renewed interest in special programming systems based on the new technology for deduction. In declaration oriented or non-procedural programming languages a mechanism is needed to infer explicit procedures from the implicit information of the declarations. With more traditional languages it was left to the programmers to supply the link between statements about *what* problem was being solved to a program that stated how to solve it. With the existence of an improved mechanical theorem prover it became possible to think of automating this.

Another event that motivated the current activities in automatic programming was an optimistic prediction made in the late 1960's about the cost and availability of future computer hardware. It appeared that by the late 1970's hardware and in particular memory costs would become so cheap as to remove most memory and execution time constraints on system development. At the same time it was noted that software costs were not becoming significantly cheaper. These two factors stimulated a renewed interest in automatic programming since it appeared that any shift of work from programmer to machine would result in a net savings. The lure of unlimited hardware promoted interest in the application of technologies previously thought to be too costly for use in application systems. System designs for improving programmer productivity emerged based on methodologies of resolution theorem provers and artificial intelligence techniques. This has produced the current state of automatic programming research that includes activities of conventional programming language design, artificial intelligence, natural language processing systems, *etc.* The specific interest in mechanized deductive systems has enlarged the field further to include systems for the proving and improving of programs.

The remainder of this chapter will survey a number of research activities in automatic programming. The specific work cited should be considered as examples of work in a field that is very large and diverse and not an exhaustive survey of automatic programming.

2.2 *Natural Language Systems:*

We begin the survey of current automatic programming efforts by discussing those systems that many (for example, [Sammet 1966]) feel to be the ultimate goal of automatic programming: natural language program specification. There are a number of research groups concerned with the problems of getting a program to understand English or some other natural language. See for example the work of [Winograd 1972] or [Schank 1973]. However, very little of this work is directly concerned with the production of programs from natural language dialog and therefore does not qualify, under my definition, as work in automatic programming. This is not to say that the knowledge gained from these efforts won't someday be useful in the generation of programs automatically, see for example [Martin 1974], [Baltzer 1972].

A second and more practical area of natural language research has been concerned with investigating the use of natural language as an interface to information retrieval or query systems, see for example [Petrick 1975]. Such work will be excluded from the category of automatic programming because the goal of these systems is not to produce a program but to arrive at an answer to a specific request.

Recent work by [Miller and Becker 1974] has been concerned with an empirical study of the way in which English is used to specify processes in problem domains other than programming. A report from [Baltzer 1974] records his introspective experiments to determine the need of world knowledge to understand English as a language for specifying instructions for carrying out various activities. It is hoped that such studies will lead to the ability to isolate and codify the essential features of natural language as it is used to communicate specific procedural information.

There have been a number of attempts to develop systems which support a reasonable model of English as a specification language for programs. A survey of four such projects may be found in [Heidorn 1975]. There have been two approaches to this problem. The first uses English as a programming language. An example of the first approach, [Heidorn 1972] is a system that generates simulation programs in GPSS from a natural language description of the simulation events. The system is interactive and prompts the user in English for additional information when needed. The system supports a reasonable English grammar and does not use a canned set of phrases or an artificial English like subset for its terminal dialog.

The second approach adds the capabilities for program understanding to the process of specification. In [Mikelsons 1975] and [Winograd 1975] systems are proposed which would interact with the program specifier or potential program user in order to determine how a specific program might be used or modified for a particular application.

2.3 *Formal Deductive Based Systems:*

Representative of systems that automatically construct programs from assertions are those of [Waldinger 1969], [Green 1969], [Waldinger and Lee 1969] and [Manna and Waldinger 1971].

Their systems generate programs and solve problems from descriptions in the first-order predicate calculus. The generated program could, in theory, be in any language capable of description in the predicate calculus. The user of such a system would describe what he wished the program to do by asserting some condition, an input assertion, that would be true upon entry to the program. In addition he would state an output assertion about the conditions that should be true after execution of the program. The system would attempt to prove, constructively, the validity of the output assertion from the input assertion and the description of the programming language. If the theorem prover were successful, it would construct the program during the course of the proof, and the program could be extracted as a final step.

In theory such a system satisfies the objectives of an ideal automatic programming system. All one needs is a theorem prover and a description of some programming language. Writing a program would be done automatically providing the system with the input and output conditions that define the program to be constructed. However, such an approach has a number of drawbacks. First, describing a programming language in the predicate calculus is usually a difficult task. Efforts such as [Darlington 1973] have simplified the description task by implementing a theorem prover for the second-order predicate calculus, thus permitting the language description to closely follow the scheme proposed by [Hoare 1969]. Second, the proofs of theorems for any but the simplest programs are beyond the capabilities of current mechanical theorem provers. A third and more serious problem, from the users point of view, is the difficulty of formulating an assertion in the predicate calculus to accurately and completely describe the function to be performed.

Another synthesis method based on theorem proving technology was proposed by [Siklossy 1974]. Instead of stating input and output assertions about a program, it is proposed that the program be described in terms of assertions about properties of the program to be generated. The program is extracted from the proof, if it exists, of the validity of the program properties. Like other formal systems, a description of the semantics of the programming language in which the program is to be written are used as axioms in the proof.

2.4 *Sophisticated Programming Tools:*

A third approach to automatic programming is that of developing sophisticated programming languages and system tools to be used by programmers in designing, implementing, verifying and improving complex programs and systems.

The ECL system [Cheatham and Wegbreit 1972] was designed to give the sophisticated programmer a powerful system environment or programming laboratory in which he could develop programs. The goal was to implement a system which would automate those tasks that could be identified as a burden to the programmer. [Sussman 1973] discusses a system that will generate programs by starting with a program that almost works and debugging it until it satisfies the user's

requirements. In this way one may develop complex a program from some simpler approximation to it.

Many new programming languages have been proposed and implemented as a solution to programming problems that arise in certain problem areas. For example, artificial intelligence languages such as Micro-PLANNER [Sussman *et al.* 1972], CONNIVER [McDermott and Sussman 1972] and STRIPS [Fikes and Nelson 1971] have relieved the programmer of many tasks associated with the methodology of developing artificial intelligence systems. Every discipline has spawned a variety of languages that help programmers solving problems in these disciplines to specify their problems to a computer.

Automatic verification and debugging systems have been proposed as another means of automating part of the programmer's job. Starting with [King 1969], a number of systems have been proposed and implemented for the automatic verification of programs. Proof methods for a variety of languages exist, *eg.* Algol or PL/I like languages [Sites 1974], LISP [Boyer and Moore 1975] and APL [Gerhart 1972]. A scheme for automated debugging based on program verification techniques has been proposed by [Katz and Manna 1975].

The final topic of this section is concerned with the automatic improvement (optimization) of programs. An excellent bibliography of this field may be found in [Allen 1975]. Optimization, like verification has been investigated for a number of programming languages, *eg.* LISP [Darlington and Burstall 1973], PL/I [Allen and Cocke 1972] and APL [Perlis 1974]. It should be noted that verification and optimization are two sides of the same coin and that techniques for one area are quite often applicable to the other.

2.5 *Programming by Example:*

Finally we turn to the area of automatic programming which includes the work of this thesis. There has been a long standing desire to see a system that would produce programs from examples of what they do. Consider, for a moment, the nature of a recursive or other iterative program whose iteration depends on its input. Such a program is description of a set of different computations, each computation determined by the programs input. Since the set of computations is infinite we might say that the program is a finite description of an infinite object, the set of computations it represents. In conventional programming languages the infinite property of the set is described by use of iterative or recursive control structures. Program specification by assertion relies on the universal quantification of some variable over an infinite domain to convey the infinite description. With both of these specification methods the infinite nature of the program being described may be *deductively* inferred from the use of the language. Program specification by example is different. Such a system must rely on *inductive* inferences of the infinite object from the finite number of examples of it.

One of the earliest reported efforts in programming by example may be found in [Biermann 1971]. He describes a system for inferring Turing machine programs (the finite state portions) from examples of the machine's tape. The approach taken by this work was a refinement of the experimental systems for mechanizing inductive inference, particularly those systems to do grammatical inference, see [Biermann and Feldman 1972] or [Fu 1974]. This approach of enumerating a set of possible solutions and then testing each one to see if it satisfied the examples. The theoretical justification for this approach was discussed by [Blum and Blum 1973].

Other approaches to the mechanization of inductive inference have been suggested [Amarel 1971], [Plotkin 1970] and [Popplestone 1970]. The approach of this thesis is most similar to the approach taken by [Friedkin 1964] and [Pivar 1964] in inferring relationships among numeric data. Inductive inferences are made by setting up and solving recurrence relations derived from the differences among the elements of the example set.

The work at Stanford [Green *et al.* 1974] and [Shaw *et al.* 1974] is closest to the external goals of the system presented in this thesis as both are concerned with the generation of LISP programs from examples. It appears from the reports of this work that we differ in our approach to the solution of the problem. The Stanford system appears to exist as a collection of heuristics that attempt to duplicate a programmer's approach to the problem of LISP program construction. The work of this thesis is based on a mathematical theory of the properties of LISP recursive programs and wherever possible provides algorithms, rather than heuristics, for the various stages of the program generation.

CHAPTER 3

PROGRAMMING BY EXAMPLE

3.1 *Example Language:*

The preceding chapter surveyed a number of systems and methodologies that are considered to be part of the milieu of automatic programming. Each activity proposed solutions to the automatic programming problem by defining a man-machine interface or language for specifying a programming activity and then describing a procedure for translating expressions in the language to some some other language of effective computability.

Programming language interfaces may be classified as one of three types depending on how the interface is used to specify a problem. Traditional programming languages provide an algorithmic interface and may be used by the problem solver to express a problem solution procedurally. The automatic aspect of such a system lies in the translation of the procedure into machine code which will effect the stated solution (the program) on the computer. Further automation is achieved by such systems when the programs generated are further machined to achieve some efficiency goal, such as time or space optimization.

The second kind of programming interface is that supported by systems that process program specifications in the form of assertions. Such languages, sometimes called assertion languages, provide a means for completely specifying a program in a formal assertion language based on some logical calculus. . Programs are extracted from a formal constructive proof of the specification assertions from a set of axioms that describe the target language.

The third category of languages cover that described by this dissertation, that of specifying a program by examples of what it does. It is a thesis of this dissertation that for some class of problems, program specification by example is more natural and efficient than specification either algorithmicly or with a formal assertion language. No claim is made that all programs can or should be specified by example. Indeed, of the three kinds of interfaces only the algorithmic languages are a universal language for describing all computational problems. Consider, for example, the difficulty of describing a random number generator with either an assertion language or by example. Since programming by example and by assertion share the property of being non-procedural it is useful to look at an example that contrasts the two approaches.

Consider, for example, the program to extract the second element from a list. In [Waldinger 1969] there is an assertion which if proven from an appropriate set of axioms will produce the program to do this. The assertion is

(∀a)(∃y)(∃w)(∃zz)equal[a;list[w;y;zz]]

We claim that the following set of examples for the same function is more intuitive, *ie.* easier to understand, than the above assertion.

```
(A B) ->B
(A B C) ->B
(A B C D) ->B
```

Three examples are given to indicate that the implied transformation is to be continued for lists of different length. In addition we can add two more examples to take care of the cases not specified in the assertion

```
() -> ()
(A) -> ()
```

All programming languages or interfaces convey both explicit and implicit information to the systems designed to implement the language. The explicit component is that part of the language that has a direct analog in the machine code, *ie.* it states how something is to be calculated. The implicit information is that information which the supporting system must infer from language statements that convey not how something is to be done but what should be done. All languages, except for machine language, have an implicit component. Procedural languages share with assertion languages the property that their supporting systems deductively infer the meaning of the implicit information from the language statements. Programming by example is different, in that it derives the meaning of the implicit information from inductive inferences.

Whenever inductive inferences are to be made it is best to lay out the ground rules for making them. This will be done formally later in the chapter but it is worth while at this point to informally present the assumptions that are present when a program is inferred from examples of its behavior. To have something specific in mind when discussing the inference assumptions, the example language for the system THESYS will be used. A definition of the terminal language for THESYS is contained in Appendix 1.

After invoking the THESYS system and receiving and responding to the preliminary message requesting the name of the function to be synthesized, the user is asked for an example of the function. Since the program to be generated is a LISP program to be defined over S-expressions the example must be some combination of S-expressions that are representative instances of the program's input and output. Consider the following example written in the terminal language of THESYS:

```
(A B C) (D E F) -> ((A . D) (B . E) (C . F))
```

where the right pointing arrow (->) separates the two S-expressions representing the program's input from the S-expression representing the program's output. (In the formal definition of an EXAMPLE in section 3.3 a slightly different syntax is used.)

The first assumption or inference that is made by the system is that each atom in the example is to stand for a representative instance of a legitimate object from the domain of S-expressions. It

is further assumed that the output is to be derived from the subexpressions of the input, so that the A in the output is to correspond to the first element in the list comprising the first argument. The single example indicates a computation that should be equivalent to the computation performed by any program claiming to be characterized by the example. The computation indicated by the example given above is to build a list of three dotted pairs constructed from the two three element lists given as inputs such that each dotted pair is constructed from corresponding elements of the input lists.

A program should be defined for more than one input. To indicate the range of values over which a program is to operate a number of examples are given. In processing a set of such examples the system assumes that the examples are representative instances of some enumeration of the set of domain values for the program and attempts to infer the complete domain from the examples. We will say that a program satisfies or is defined by a set of examples if it produces an example's output from an application of the program to the corresponding example's input. Since a set of examples is a finite set of instances of a possibly infinite set of program input/output pairs, a given set of examples may be satisfied by more than one program. The procedures for program synthesis given in this chapter represent the formal definition of exactly what program is generated for a given set of examples. The system always attempts to generate a recursive program (such a program corresponds to a infinite set of input/output pairs) from any set of examples. It can do this only if the set of examples possesses a regularity of form that permit inferring additional input/output pairs. In the event that this is not possible, a non-recursive program is generated which is defined for just the set of examples.

3.2 *The Target Language-- LISP:*

There are a number of reasons for choosing LISP as the programming language for constructing programs from examples. LISP is a widely known language that exists in a variety of implementations, all of which are nearly equivalent. The language is simple in form and is defined in terms of few concepts. Basic LISP is usually defined in terms of one data type (S-expressions), five primitive functions and the control structures of conditional expressions and recursion, see [McCarthy 1960] or [McCarthy *et al.* 1962]. Its simplicity of definition has encouraged theoretical study of its properties that has lead to detailed definitions of its semantics. For example, concepts from recursive function theory have been used to define properties of the language.

The properties of the set of primitive operations of the target language are very important to the process of constructing programs from examples. The five primitives, **car**, **cdr**, **cons**, **atom**, and **eq** are sufficiently powerful to define a computationally complete language. It is easy to show that computational completeness may be maintained if the predicate **eq** is removed from the set of primitives. See Appendix 2 for a proof of this assertion. The resulting language may be used to

define programs over the set of data objects that are distinguishable by their cons structure. It is this simpler version of basic LISP that will be used as the language for constructing programs specified by example. To construct programs from any set of primitives requires that the properties of those primitives be encoded into the system. The system must have sufficient knowledge of the operations to be able to derive the result of applying any primitive to a datum in the system. Experience with mechanical theorem provers teaches us that mechanization of the equality predicate is inefficient, see, for example [Sibert 1969], and that using the properties of eq in a synthesis system would be liable to the same kinds of inefficiencies.

A second reason for eliminating eq from the set of primitive functions is that by doing so only one function remains that is not monadic. Monadic systems have nice formal properties with respect to the manipulation of their elements. Computer manipulation of diadic functions is usually complex and inefficient because of the difficulty of encoding and manipulating properties of the function such as associativity and commutativity. The one diadic function cons has none of these structural properties. The algebraic system composed of the operator cons and the set of S-expressions is closed but is not associative and does not contain an identity element. In algebraic terms, it may be considered as a free algebra with one binary operator, cons, and one generator, any atom. Therefore it is always possible, given any S-expression to decode how the function cons had been applied to create the expression. It is this ability to uniquely decompose a data expression into the structure of function calls that created it that permits some of the procedures used the construction of programs from examples to be algorithmic rather than incorporating heuristic devices.

One further property of LISP should be mentioned. There is a certain elegance in systems that do not distinguish, syntactically, between program and data but differentiate between the two by detecting how an item is used. Such a property is particularly fitting for a system that must construct a program from examples of that program's data. The program synthesis system, THESYS, is written in LISP and may be characterized as being a function which maps S-expressions (examples) to S-expressions (programs).

It is assumed that the reader is familiar with LISP as described in [McCarthy *et al.* 1962] or [McCarthy 1960]. The remainder of this section will describe the differences between the language as defined by the references and the language as used in this thesis.

Definition (D1): S-expressions are denoted in this paper by the use of the courier type face, *eg.* (A B C) is the list of three atoms A, B, C corresponding to the dotted pair expression (A . (B . (C . ( )))). All of the examples of program behavior are written as S-expressions. The set of all legal S-expressions over a set of atoms will be denoted as D_S.

Definition (D2): Functions are written as M-expressions (meta-expressions) as defined in [McCarthy *et al.* 1962]. It should be recalled that [McCarthy *et al.* 1962] presents an algorithm for converting any M-expression to a corresponding S-expression. Differences between this paper and the references and some additional conventions and definitions are listed below.

a. Function definitions use a left pointing arrow (←) instead of an equal sign (=) to separate the definiendum from the definien. For example, $f[x] \leftarrow B_x$ will represent the definition of the function f[x] by the definien B_x, called the body of the definition. This is equivalent to defining f to be the lambda expression $\lambda x.B_x$ [Church 1941]. We occasionally make use of the lambda notation to refer to component parts of functions being constructed. We will follow the notation of [McCarthy *et al.* 1962] and write the lambda expression $\lambda x.B_x$ as $\lambda[x;B_x]$. Any legal definien of the form $\lambda[x;B_x]$ will be called a D-FORM. The set of all D-forms will be denoted by DF.

b. Function names when used in text are printed in bold face, *eg.* **car**.

c. It is assumed that the function **apply** exists and that it has the usual meaning, except that arguments of **apply** may be written as M-expressions rather than as S-expressions. We will use **apply** to define functional composition in the following way

$$\text{compose}[\lambda[x;B_x];\lambda[y;C_y]] = \lambda[x;B_x\{\text{apply}[\lambda[y;C_y];x]\}/x]]$$

where $B_x\{\text{apply}[\lambda[y;C_y];x]/x\}$ means to substitute apply[λ[y;C_y];x] for **x** everywhere in B_x.

d. Primitive functions are restricted to **car**, **cdr**, **cons**, and the predicate **atom**.

e. The thesis uses a number of properties of the semantics of the language, see for example [McCarthy 1963a, 1963b] or [Manna *et al.* 1973].

i. car[cons[x;y]]=x

ii. cdr[cons[x;y]]=y

iii. cons[car[x];cdr[x]]=x iff x is not an atom

iv. $f[[p_1[x]\rightarrow e_1[x];\ldots;p_k[x]\rightarrow e_k[x]] = [p_1[x]\rightarrow f[e_1[x]];\ldots;p_k[x]\rightarrow f[e_k[x]]]$ iff f[x] is undefined whenever x is undefined.

f. Tree Notation: It will be convenient, later in the chapter, to represent functional expressions in a tree language. Figure 3.1 gives the tree representations for general functions of any number of arguments and the special shorthand notation for the LISP primitives **car**, **cdr** and **cons**.

car[x] is represented as

cdr[x] is represented as

cons[x;y] is represented as

$f[x_1;\ldots;x_k]$ is represented as

So that cons[caddr[x];cons[cadr[x];cons[car[x];nil]]] is represented as

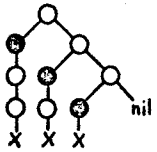


Figure 3.1: Tree notation.

g. LISP programs may be interpreted as a grammar for generating elements of a tree language. A given LISP program, for example

$F[x] \leftarrow [p_1[x] \rightarrow f_1[x];$
$\ldots$
$p_{k-1}[x] \rightarrow f_{k-1}[x];$
$T \rightarrow f_k[x]]$

where the $f_i[x]$, $1 \leq i \leq k$ are expressed in tree notation, may be considered as a grammar for describing a family of trees. Beginning with the tree F[x], the program gives the rules or productions for transforming F[x] into a new tree by substitution of one of the $f_i[x]$ for the symbol F. In such a system the primitive functions and variables act as terminal symbols while the nonprimitive function names act as non-terminal symbols. The predicates control which production is to be applied to the tree at any given time. A COMPUTATION SEQUENCE will be defined to be a sequence of trees, $T=t_0,\ldots,t_k$ where $t_0=F[x]$ and each t_i is the result of applying one of the transformation rules to some subtree of t_{i-1} containing terminal symbols except for the root of the subtree which is a non-terminal. For a more detailed explanation of tree grammars and their relationship to recursive control structures see, for example, [Strong 1971].

Figure 3.2 shows a LISP function, the corresponding tree grammar and an example of a computation sequence for this function.

f[x]←[atom[x]→x;
T→cons[f[cdr[x]];car[x]]]

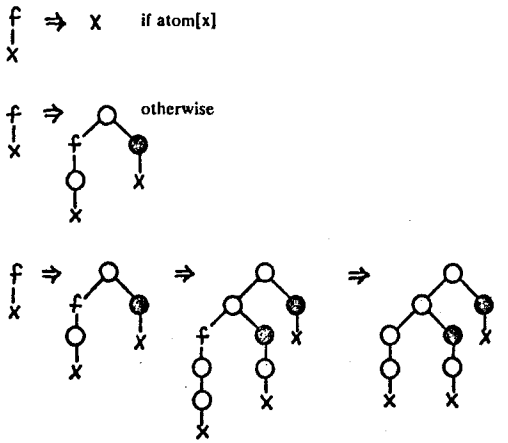


Figure 3.2: Example of a tree grammar.

h. A computation sequence will be called CONSERVATIVE iff there is no occurrence of the subtrees car[cons[x;y]] or cdr[cons[x;y]] in any of the trees in the sequence. A function will be called conservative iff all of its computation sequences are conservative.

3.3 *Examples and Fragment Generation*:

Program synthesis begins with a set of examples that characterize WHAT the program is to do. An example is a doublet consisting of a list of input values and a single output value. As was noted in the section on the target language, our examples are all S-expressions since the programs to be synthesized are LISP programs. Although the synthesized programs are unable to distinguish between atoms, the example language must use unique atom names in order to convey the relationships between component parts of the input and output portion of the examples. The atoms of the example language are used as named meta-variables or Skolem functions [Skolem 1928], [Davis and Putnam 1960], *ie.* the atoms stand for representative instances of S-expressions.

As a first stage in transforming an example to a program the relationship between the input and the output portion of an example must be formalized in a way that is independent of the specific example. The relationship is encoded as a function which if applied to the example's input would yield as a value the example's output. Such a function will be called a program fragment or fragment and is constructed by functional composition from the basic LISP functions, car **cdr**, **cons**, and atom. An algorithm exists for constructing a unique fragment from a given example.

3.3.1 *Examples*:

Definition (D3): An EXAMPLE SET, $E=\{e_i\}$, of a function $F:D_1\times D_2\times\ldots\times D_{k-1}\rightarrow D_k$ is a finite set of pairs, $e_i=<x_i,y_i>$ called EXAMPLES, such that $x_i \in D_1\times D_2\times\ldots D_{k-1}$, $y_i\in D_k$ and $F(x_i)=y_i$. We will call $x_i=input[e_i]$ the EXAMPLE INPUT and $y_i=output[e_i]$ the EXAMPLE OUTPUT.

Since the thesis is restricted to LISP programs that manipulate S-expressions the D_i's will all be restricted to D_S, the set of LISP S-expressions. An alternative way of defining an example e_i, is to say that it is an element of F where F is a subset of $D_1\times\ldots\times D_k$.

The example input and output are specified by writing a tuple of S-expressions for the input and a single S-expression for the output. A k-tuple of S-expressions $S_1,S_2,\ldots,S_k$ is written as $<S_1,S_2,\ldots S_k>$. Note that the formal syntax is different from the syntax of the terminal language as described in Appendix 1.

Exhibit: We wish to write a set of examples, E, for a function that returns the last element of a list given as its input. One possible value for E is

$E=\{\ e_1 = <(A)\ ,A>,$
$\quad e_2 = <(A\ B)\ ,B>,$
$\quad e_3 = <(A\ B\ C)\ ,C>,$
$\quad e_4 = <(A\ B\ C\ D)\ ,D>\ \}.$

Note that each atom of E, (A, B, C and D), has no inherent meaning, rather it is the relationship between the atoms in the example input and the atoms in the output that is important. For example, e_3 might just as well have been written as $<(1\ 2\ 3)\ ,3>$. Also note that the scope of meaning for any atom in the example language is just that of the example, e_i, in which it occurs.

The set of examples

$E'=\{\ e_1 = <(A)\ ,A>,$
$\quad e_2 = <(B\ A)\ ,A>,$
$\quad e_3 = <(C\ B\ A)\ ,A>,$
$\quad e_4 = <(D\ C\ B\ A)\ ,A>\ ].$

represents the same function as that represented by E and is not representative of the constant function that has the value A.

Exhibit: A set of examples, E, for the function which returns a list of dotted pairs of the elements of its arguments.

$E=\{\ e_1=<<(),()>,()>,$
$\quad e_2=<<(A),(1)>,((A\ .\ 1))>,$
$\quad e_3=<<(A\ B),(1\ 2)>,((A\ .\ 1)\ (B\ .\ 2))>,$
$\quad e_4=<<(A\ B\ C),(1\ 2\ 3)>,((A\ .\ 1)\ (B\ .\ 2)\ (C\ .\ 3))>\}$

Definition (D4): The ATOM SET of an S-expression, X, is the set of all atoms contained in X and is denoted by $a[X]$. $a[<S_1,\ldots,S_k>]=a[S_1]\cup\ldots\cup a[S_k]$.

Definition (D5): An example, $e_i=<x_i,y_i>$, will be called SELF-CONTAINED if and only if $a[y_i]$ is a subset of $a[x_i]$.

If an example is not self-contained we mean for the atoms in $a[y_i]-a[x_i]$ to stand for themselves, *ie.* to be quoted.

Exhibit: Let $e=<<(A\ B),(1\ 2)>,((A\ .\ 1)\ (B\ .\ 2))>$ then

$a[input[e]]=a[(A\ B)]\cup a[(1\ 2)]$
$\quad =\{A,B,()\}\cup\{1,2,()\}$
$\quad =\{A,B,1,2,()\}$
$\quad =a[output[e]]$

therefore e is self-contained. The atom ( ) is the LISP atom called nil.

Exhibit: If a function with the constant value XYZ is to be specified we might write the set of examples

$E=\{\ e_1 = <(),XYZ>,$
$\quad e_2 = <(A),XYZ>,$
$\quad e_3 = <(A\ B),XYZ>\ \}.$

The atom, XYZ, is defined as itself and not as a function of the inputs given in the example.

Almost all functions that we wish to synthesize will be self-contained, though occasionally it will be convenient to produce functions which will generate distinguishable atoms, such as TRUE and FALSE. The use of distinguished atoms should be interpreted as a shorthand convenience as the output could just as well have been in the form of distinguishable structures.

3.3.2 *Ambiguity*:

A first step in translating examples to programs is the derivation of the transformation implied by each example. In finding relationships among transforms, it is desirable to have a unique transform for each example. To insure uniqueness, the examples must not be ambiguous. If one specifies the example $<(A\ B\ A\ C),(A\ B\ C)>$ there is ambiguity as to the meaning of the A in the example output. It could be defined as either **car**[(A B A C)] or as **caddr**[(A B A C)]. A

similar situation may arise in multi-argument functions if the atom set of one of the arguments contains an element in the atom set of another argument and that element is in the atom set of the example output. This section will formalize the meaning of ambiguity as used in this thesis.

Definition (D6): The DERIVATION SET, DS_e, of $e \epsilon D_S$ is defined: $DS_e = \{<n,v> \mid n \epsilon DF$, n is defined using only the primitives car and cdr, $v \epsilon D_S$, and apply[n;e]=v}.

Recall that DF is the set of d-forms of the form $\lambda[x;B_x]$ (see Definition D2.a in §3.2).

Exhibit: Let e= (A (B) C) then

DS_e={<λ[x;x] , (A (B) C)>,
<λ[x;car[x]] , A>,
<λ[x;cdr[x]] , ((B) C)>,
<λ[x;cadr[x]] , (B)>,
<λ[x;cddr[x]] , (C)>,
<λ[x;caadr[x]] , B>,
<λ[x;cdadr[x]] , ()>,
<λ[x;caddr[x]] , C>,
<λ[x;cdddr[x]] , ()>].

Note that the second member of each pair is precisely the expression that is created when the first member of the pair is applied to the original expression e.

Procedure (P1):

1. Let $DS_e=\{<n_0,v_0>\}$ where $n_0=\lambda[x;x]$, the identity function, and v_0=e.
2. Create the set D' from the pairs
   <compose[λ[x;car[x]];n_i];apply[λ[x;car[x]];v_i]> and
   <compose[λ[x;cdr[x]];n_i],apply[λ[x;cdr[x]];v_i]> for all $<n_i,v_i> \epsilon DS_e$ for which v_i is not an atom (see Definition D2.c).
3. If D' is contained in DS_e halt, else set DS_e to $DS_e \cup D'$ and return to step 2.

Lemma (L1): The procedure is an algorithm for computing the derivation set.

Proof: First show proof of termination. This is obvious by induction on the complexity of the elements of DS_e. Each pass through the algorithm produces a name-value list with values of the new elements less complex than the values in the previous list.

Next, show that the procedure calculates a derivation set. First show that if $<n,v> \epsilon DS_e$ as defined by the procedure then $n \epsilon DF$ and $v \epsilon D_S$. The function λ[x;x] is a legal function therefore $n_0 \epsilon DF$. The expression v_0=e is an S-expression and by definition is a member of D_S. All other members, <n,v> of DS_e are produced by composing a member of DF, either λ[x;car[x]] or λ[x;cdr[x]], with a member of DF to get a new n, or applying a member of DF to a member of D_S to get a new v. From the definition of the language, Definition D2.c, we have that the new $n \epsilon DF$ and the new $v \epsilon D_S$. For all $<n,v> \epsilon DS_e$, n is conservative because it

is either the identity function, n_0, or else a composition of car's and cdr's.

We prove that apply[n_i;e]=v_i by induction. Assume that $<n_i,v_i>$ is in the set produced by the procedure and that if i<j, $<n_i,v_i>$ was added to the set before $<n_j,v_j>$.

i=0: apply[n_0;e] = apply[λ[x;x];e]
= e by definition of application
= v_0 by definition of procedure

Assume true for i≤k then look at $<n_{k+1},v_{k+1}>$. The notation c*r is either car or cdr.

apply[n_{k+1};e] = apply[λ[xi;c*r[B_{xi}]];e] for some i, by the definition of the procedure
= c*r[B_{xi}[e/xi]] by the definition of apply
= c*r[apply[λ[xi;B_{xi}];e]]
= c*r[v_i] by hypothesis
= v_{k+1} ■

Definition (D7): In the example e=<x,y> where x=[x1;x2;...;xk] we define the DECOMPOSED INPUT SET of e, $DIS_e = DS_{x1} \cup DS_{x2} \cup \ldots \cup DS_{xk}$.

The atom ( ), pronounced nil, has a special function in LISP in that it is used to mark the end of lists. Beyond this, it has no inherent meaning so that multiple occurrences of it in an example is not usually an indication of ambiguity. For this reason we will always assume that any occurrence of ( ) in the output of an example came from the application of the niladic function nil and not from the example's input.

Definition (D8): An example is AMBIGUOUS if there exists $<n_i,v_i>$ and $<n_j,v_j>$ in DIS_e such that $v_i=v_j \neq$ nil and $n_i \neq n_j$.

3.3.3 *Fragments*:

In this section we define a procedure that encodes the formal relationship between an example's input and its output. The encoding is a function defined so as to return the example's output when applied to the example's input. Such a function will be called a fragment.

Definition (D9): Given an unambiguous example $e_i=<x_i,y_i>$, a FRAGMENT or PROGRAM FRAGMENT is defined to be a function $f_i[x]$ such that 1) $f_i[x_i]=y_i$, 2) $f_i[x]$ is conservative and 3) $f_i[x]$ is defined in terms of the primitive functions (Definition D2.d).

Procedure (P2): Given $e_i=<x_i,y_i>$ find the associated fragment $f_i[x]$, *ie.* find $f_i[x]=P2[e_i]=P2[<x_i,y_i>]$.

1. Create the DIS for e_i.
2. If y_i=nil then $f_i[x] \leftarrow$ nil.

3. If $<nr_j,y_i>\epsilon$DIS, nr_j=jth name-value pair for the rth example input then $f_i \leftarrow nr_j$. Since $nr_j=\lambda[x;B_x]$ we define $f_i[x]\leftarrow B_x$. Note that this step indicates a deterministic selection since there is at most one such pair $<nl_j,y_j>$ in the decomposed input list of e_i since e_i is unambiguous.

4. If y_i is an atom and there is no $<nr_j,vr_j>$ such that $vr_j=y_i$ then $f_i[x]\leftarrow$quote$[y_i]$. The quote function is used as the usual LISP meta-linguistic construct to permit the definition of an object that has as its value, itself.

5. In all other cases define $f_i[x]\leftarrow$cons$[a_i[x];d_i[x]]$ where $a_i[x]$ is the fragment associated with $<x_i,$car$[y_i]>$ and $d_i[x]$ is the fragment associated with $<x_i,$cdr$[y_i]>$, *ie.* $f_i[x]\leftarrow$cons$[P2[<x_i;$car$[y_i]>];P2[<x_i;$cdr$[y_i]]$.

Lemma (L2): The procedure is an algorithm for generating a fragment.

Proof: The procedure terminates with some value because of the decreasing complexity of the examples, *ie.* the fragment for $<x,y>$ is defined in terms of the fragments for $<x,$car$[y]>$ and $<x,$cdr$[y]>$, etc. The recursion is terminated for all $<x,y>$ when y satisfies steps 2-4.

1. To prove that $f_i[x_i]=y_i$
   a. if y_i=nil then $f_i[x_i]$=nil from the procedure.
   b. if $<nr_i,vr_i>\epsilon$DIS for x_i and if $y_i=vr_i$ then $f_i[x_i] = nr_i[x_i] = y_i$.
   c. If y_i is an atom and $y_i\neq vr_i$ for any $<nr_i,vr_i>\epsilon$DIS then $f_i[x_i]=y_i$ from the procedure.
   d. Assume that $<x_i,$car$[y_i]>$ and $<x_i,$cdr$[y_i]>$ have associated fragments a_i and d_i respectively, *ie.* $a_i[x_i]=$car$[y_i]$ and $d_i[x_i]=$cdr$[y_i]$, then $f_i[x_i]$ = cons$[a_i[x_i];d_i[x_i]]$ = cons$[$car$[y_i];$cdr$[y_i]] = y_i$ by Definition D2.e.iii.
2. The result f_i would be non-conservative if and only if car or cdr were applied to a cons expression. Since the only time car and cdr are used in the construction of f_i is in the creation of the DIS and since cons is never used in this step the f_i is conservative.
3. It follows immediately that f_i is defined in terms of the primitive functions. ■

Exhibit: Applying procedure P2 to the set of examples for the function **last**, the following set of fragments are generated:

$P2[e_1] = P2[<(A), A>] = f_1[x]\leftarrow$car$[x]$

$P2[e_2] = P2[<(A\ B), B>] = f_2[x]\leftarrow$cadr$[x]$

$P2[e_3] = P2[<(A\ B\ C), C>] = f_3[x]\leftarrow$caddr$[x]$

$P2[e_4] = P2[<(A\ B\ C\ D), D>] = f_4[x]\leftarrow$cadddr$[x]$

It should be noted that each fragment approximates the function to be synthesized in that it is identical to it for at least one element of the function's domain. By collecting all of the fragments together into a single expression with control determined by the McCarthy conditional we are able to make a first attempt at generating a program. If $\{f_i[x]\}$is the set of fragments corresponding to the set of examples, $E=\{e_i\}$, then the program $F_E[x]$ may be defined as follows,

$F_E[x]\leftarrow[p_1[x]\rightarrow f_1[x];$
$p_2[x]\rightarrow f_2[x];$
...
$T\rightarrow f_k[x]]$

where the $p_i[x]$ are true whenever x is 'similar in structure' to the input$[e_i]$.

Formalizing the notion of 'similar in structure to' is equivalent to determining the predicates $p_i[x]$. This, however, is not enough to transform $F_E[x]$ into a well defined program in the target language. Not only must the predicates be generated but the order in which they appear in the conditional expression must be determined because of the sequential nature of the McCarthy conditional. Order of the predicates are important since if $p_j[x]$ subsumes $p_i[x]$, *ie.* $p_j[x]$ implies $p_i[x]$, and if $p_i[x]$ occurs before $p_j[x]$ in the conditional and if x_j=input that is similar to the input of e_j then $F_E[x_j]=f_i[x_j]$ rather than $f_j[x_j]$, the computation indicated by the examples. A second problem of ordering predicates occurs if there is an x_i for which $p_i[x_i]$=T and $p_j[x_i]$ is undefined. In this case $p_j[x]$ should not precede $p_i[x]$ in the conditional of $F_E[x]$. Fortunately the procedure to be described in the next section will generate predicates in the correct order if the examples satisfy certain conditions. Later in the chapter we give conditions for extending programs of this form to a recursive program.

3.4 *Predicate Generation*:

The development of many of the ideas in this section involve the use of definitions and theorems from lattice theory. Before presenting the predicate generation algorithm it is useful to review some basic definitions and theorems from lattice theory. Proofs for the theorems may be found in [Birkhoff 1967].

3.4.1 *Lattice Preliminaries*:

Definition (D10): A POSET is a set with a binary relation $x\leq y$ which for all x, y, z
P1. (Reflexive) $x\leq x$
P2. (Antisymmetry) If $x\leq y$ and $y\leq x$ then $x=y$.
P3. (Transitivity) If $x\leq y$ and $y\leq z$ then $x\leq z$.

For example, the relation of set inclusion is a poset relation.

Definition (D11): A poset for which given an x and y, either $x\leq y$ or $y\leq x$ is called a CHAIN.

Definition (D12): If neither $x \leq y$ nor $y \leq x$, then x and y are said to be INCOMPARABLE.

Theorem (T1): Any subset, S, of a poset, P, is itself a poset under the same partial order relation (restricted to S).

Definition (D13): A LEAST ELEMENT of any subset, X, of a poset, P, is an element $a \epsilon X$ such that $a \leq x$ for all $x \epsilon X$. A GREATEST ELEMENT of X is an element $b \epsilon X$ such that $x \leq b$ for all $x \epsilon X$.

Definition (D14): An UPPER BOUND of a subset, X, of a poset, P, is an element $a \epsilon P$ such that for all $x \epsilon X$ $x \leq a$. The LEAST UPPER BOUND of a subset X of P is an upper bound a such that for any other upper bound $b \epsilon P$, $a \leq b$. The least upper bound is denoted by lub[X]. The notions of LOWER BOUND and GREATEST LOWER BOUND, denoted by glb[X], are similarly defined.

Definition (D15): A LATTICE is a poset, P, any two of whose elements, x and y, have a glb or MEET, denoted by meet[x;y] and a lub or JOIN, denoted by join[x;y].

Theorem (T2): Any system with two binary operators, f and g, which satisfy the following four properties is a lattice and conversely.

1. f[x;x]=x, g[x;x]=x (Idempotent)
2. f[x;y]=f[y;x], g[x;y]=g[y;x] (Commutative)
3. f[x;f[y;z]]=f[f[x;y];z], g[x;g[y;z]]=g[g[x;y];z] (Associative)
4. f[x;g[x;y]]=g[x;f[x;y]]=x (Absorption)

Definition (D16): A SUBLATTICE of a lattice L is a subset X of L such that $a,b \epsilon X$ imply meet[a;b]ϵX and join[a;b]ϵX.

Definition (D17): If $a \leq b$ then an INTERVAL [a,b] of a lattice L is the sublattice $X=\{x \mid a \leq x \leq b\}$.

3.4.2 *Domain of Structures*:

The above review of concepts from lattice theory should be enough for most of the remainder of this section. Returning to the problem of predicate determination, the first step is to try and characterize the domain over which the generated program is to be defined. We have previously said that it was the domain of S-expressions, however since the programs are constructed from primitives which do not permit the identification of individual atoms the domain is really restricted to entities which may only be distinguished on the basis of their cons structure.

Definition (D18): Given an S-expression, x, the FORM of the expression, denoted form[x], is the expression with all atoms replaced by the undefined atom, ω.

Exhibit: If x= (A (B C) D) = (A . ((B . (C . ())) . (D . ()))) then form[x]= (ω . ((ω . (ω . ω)) . (ω . ω)))

Definition (D19): The set, $SF=\{x \mid x=\text{form}[y], y \epsilon D_S\}$ will be called the set of S-FORMS.

The definition of form defines an effective procedure for projecting any S-expression into the set SF. The projection will be denoted by the expression proj[x].

Definition (D20): A relation, $\leq$, over SF defined as (1) $\omega \leq a$, for all $a \epsilon SF$. (2)(a . b)$\leq$(c . d) iff $a \leq c$ and $b \leq d$.

Lemma (L3): $<SF, \leq>$ is a poset.

Proof: Proof consists of proving that $\leq$ is a partial order over SF. The proof will be by induction on the structure of expressions in SF. Define a structural complexity function, $\xi[x]$, to be the number of dotted pairs in the structure x. (We regret the ambiguity of the symbol $\leq$. It is clear from the context whether the symbol denotes the partial order relation on SF or on the integers.)

1. For all $x \epsilon SF$, $x \leq x$.

For complexity 0: $\omega \leq \omega$. Assume true for all x such that $\xi[x] \leq n$ and let $\xi[y]=n+1$. If y=(a . b) then $\xi[a] \leq n$ and $\xi[b] \leq n$ so that $a \leq a$ and $b \leq b$. Therefore, (a . b)$\leq$(a . b) by definition of $\leq$.

2. For all $x,y \epsilon SF$, $x \leq y$ and $y \leq x$ implies x=y.

We will use the obvious fact that $x \leq y$ implies $\xi[x] \leq \xi[y]$.

$\xi[x] \leq \xi[y]$ and $\xi[y] \leq \xi[x]$ therefore $\xi[x]=\xi[y]$.

Proof of basis ($\xi[x]=\xi[y]=0$): $\omega \leq \omega$ implies $\omega=\omega$.

Assume property holds for $\xi[x] \leq \xi[y] \leq n$ and prove for $\xi[x1]=\xi[y1]=n+1$:

Let x1=(a . b) and y1=(c . d) then $\xi[a]$, $\xi[b]$, $\xi[c]$ and $\xi[d] \leq n$.

$x1 \leq y1$ and $y1 \leq x1$ iff (a . b)$\leq$(c . d) and (c . d)$\leq$(a . b)

iff $a \leq c$ and $c \leq a$ and $b \leq d$ and $d \leq b$

implies that a=c and b=d from the induction hypothesis.

implies that (a . b)=(c . d) therefore x1=y1.

3. For all $x,y,z \epsilon SF$ $x \leq y$ and $y \leq z$ implies $x \leq z$

Induction on the complexity of z.

$\xi[z]=0$: $x \leq y$ and $y \leq \omega$ implies $x=y=z=\omega$

Assume the property true for $\xi[z] \leq n$

then if $\xi[z] \leq n+1$, let x=(a . b), y=(c . d) and z=(e . f) and (a . b)$\leq$(c . d) and (c . d)$\leq$(e . f). From this we get $a \leq c$ and $c \leq e$ and $b \leq d$ and $d \leq f$ which implies $a \leq e$ and $b \leq f$ by the induction hypothesis. Therefore (a . b)$\leq$(e . f) or alternatively $x \leq z$. ■

Lemma (L4): $<SF , \leq>$ is a lattice.

Proof: The proof will consist of showing that the two functions, **meet** and **join**, defined below actually calculate the meet and join of any two elements of the set SF.

```
join[x;y]←[x=ω→y;
           y=ω→x;
           T→cons[join[car[x];car[y]];
                  join[cdr[x];cdr[y]]]]

meet[x;y]←[x=ω→ω;
           y=ω→ω;
           T→cons[meet[car[x];car[y]];
                  meet[cdr[x];cdr[y]]]]
```

The proof will consist of proving that the programs have the four properties specified by Theorem T2, *ie.* that they have the properties of idempotency, associativity, commutativity and absorption. Proofs will use the complexity measure, $\xi[x]$, of Lemma L3 for induction.

1. (Idempotency) $\xi[x]=0$: $join[\omega;\omega]=\omega$

Assume for $\xi[x]\leq k$ that $join[x;x]=x$ let $\xi[x]=k+1$

$join[x;x] = cons[join[car[x];car[x]];join[cdr[x];cdr[x]]]$

$= cons[car[x];cdr[x]]$ by hypothesis

$= x$ by Definition D2.e.iv.

A similar proof exists for **meet**.

2. (Commutativity): If $\xi[x]=0$ then $join[\omega;y]=join[y;\omega]=y$.

If $\xi[y]=0$ then $join[x;\omega]=join[\omega;x]=x$.

Assume true for $\xi[x],\xi[y]\leq k$ and prove for $\xi[x]$ or $\xi[y]=k+1$

```
join[x;y] = [x=ω→y;
            y=ω→x;
            T→cons[join[car[x];car[y]];join[cdr[x];cdr[y]]]]

          = [y=ω→x;
            x=ω→y;
            T→cons[join[car[y];car[x]];join[cdr[y];cdr[x]]]]

          = join[y;x]
```

Similar proof for meet.

3. (Associativity) This proof will use the technique of recursion induction [McCarthy 1963]. This will consist of showing that the functions $g[x;y;z]=join[x;join[y;z]]$ and $h[x;y;z]=join[join[x;y];z]$ are both fixpoints of the same functional, $\tau[F]$.

```
τ[F][x;y;z]←[x=ω→join[y;z];
             y=ω→join[x;z];
             z=ω→join[x;y];
             T→cons[F[car[x];car[y];car[z]];F[cdr[x];cdr[y];cdr[z]]]]
```

It is easy to show that $g[x;y;z]$ and $h[x;y;z]$ are fixpoints of the functional if any of the arguments are equal to ω. Assume that none of the arguments are equal to ω. $g[x;y;z] = join[x;join[y;z]]$

$= cons[join[car[x];car[join[y;z]]];join[cdr[x];cdr[join[y;z]]]]$

$= cons[join[car[x];join[car[y];car[z]]];join[cdr[x];join[cdr[y];cdr[z]]]]$

$= \tau[g[x;y;z]]$

$h[x;y;z]=join[join[y;z]]$

$= cons[join[car[join[x;y]];car[z]];join[cdr[join[x;y]];cdr[z]]]$

$= cons[join[join[car[x];car[y]];car[z]];join[join[cdr[x];cdr[y]];cdr[z]]]$

$= \tau[h[x;y;z]]$

This shows that both g and h are fixpoints of the same functional, thus proving that the function join is associative. A similar proof exists for **meet**.

4. (Absorption) The proof will be by induction on the complexity of x, *ie.* $\xi[x]$.

$\xi[x]=\xi[y]=0$: $meet[x;join[x;y]]=x$

Assume true for $\xi[x]\leq k$ then prove for $\xi[x]=k+1$.

```
meet[x;join[x;y]]=[x=ω→ω;
                   T→cons[meet[car[x];join[car[x];car[y]]];meet[cdr[x];join[cdr[x];cdr[y]]]]
                  =[x=ω→ω;
                   T→cons[car[x];cdr[x]]]
                  = x
```

Therefore $<SF,\leq>$ is a lattice ■

Figure 3.3 shows the bottom four levels of the lattice $<SF , \leq>$.

We will use properties of the lattice of S-forms to generate both a set of predicates for selecting the fragments and an ordering for their place in the McCarthy conditional. Before continuing it is necessary to make a simplifying, though restrictive, assumption about the programs to be synthesized. Up to now we have assumed the function to be synthesized to have any number of arguments. From here on we will restrict ourselves to monadic functions. Such a restriction enables us to use the lattice SF as a characterization of the domain of S-expressions. The restriction may be relaxed for some functions through *ad hoc* techniques (see example of §4.10). Chapter 5 discusses the possibility of relaxing the restriction in general.

Each point on the lattice corresponds to all S-expressions having the same **cons** form. If we are given a set of examples and the associated fragments the set of inputs of the examples may be projected onto a finite set of points of the lattice SF. The importance of these points is that for every S-expression that projects to a point in this set there exists a computation, the fragment

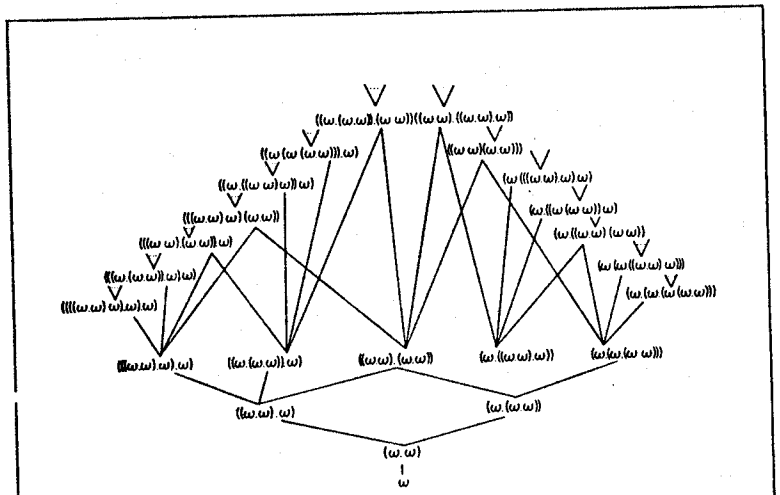


Figure 3.3: Lattice <SF,≤>.

associated with the point, that transforms the S-expression in a way consistent with the set of examples. As will be seen in later sections, inferring a recursive program from a set of examples will mean extending the definition of the computation to include points in the lattice SF other than those corresponding to the input of the examples.

The first use to be made of the lattice characterization of the domain of the function is the specification of a rule for specifying examples. Two examples with inputs of the same form should not be used to specify a function. If the fragments associated with these examples are different the examples are contradictory. If the fragments are the same then the examples are redundant. From this point on we will assume that a set of examples consists of examples with unique input structures.

3.4.3 *Domain Approximations:*

The first method of extending a function's definition may be viewed as defining when, for some arbitrary point on the lattice, a point with an associated fragment may be considered an approximation for it.

Definition (D21): If x,yεSF we will say that x APPROXIMATES y or x is an APPROXIMATION for y iff x is an element of the interval [ω,y] in SF.

Definition (D22): The interval [ω,x], xεSF, will be called an APPROXIMATING INTERVAL of SF.

If we consider the approximating interval of SF that has as a maximal element, y, then any point in this sub-lattice is an approximation to y. Consider a point in SF, for example y=((ω.ω).(ω.(ω.ω))), then the interval, [ω,y] in SF is shown in Figure 3.4.

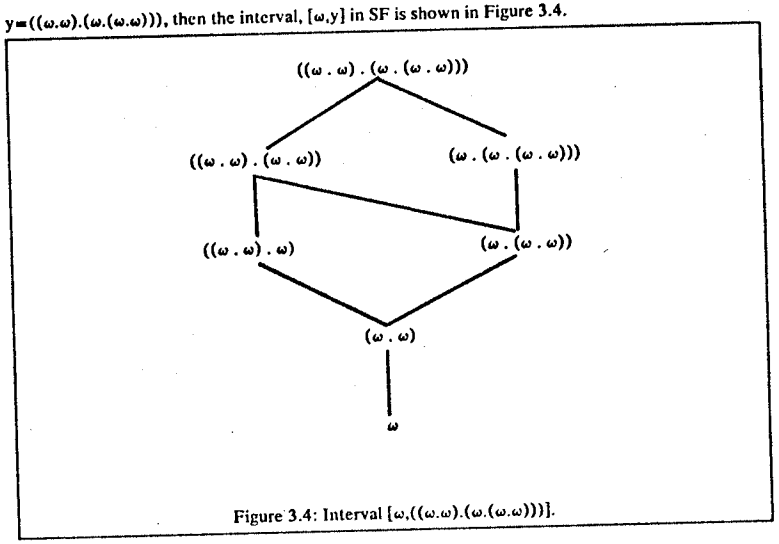


Figure 3.4: Interval [ω,((ω.ω).(ω.(ω.ω)))].

The least element of SF, ω, was earlier given the name of the undefined atom. More appropriately, it might better be called the indeterminate atom, in that it corresponds, in the example language, to any atom. Recall that an atom in the example language is to be representative of any S-expression. Moving up the sub-lattice to (ω.ω) we find an element that corresponds to any non-atomic S-expression. Similarly, any element of the lattice corresponds to an entire set of S-expressions that have at least as much structure as the point in question. If we consider the points on the lattice as standing for these sets then the relation ≤ is equivalent to the set relation includes. If x approximates y then x≤y, or the set representing x includes the set of S-expressions represented by the point y. This then is the sense in which the term approximates is used in the definition, that of a set approximating another set if the first contains the second.

Definition (D23): If X is a subset of the points of SF that approximate the point y, we define xεX to be a BEST APPROXIMATION of y in X iff there is no x'εX such that x<x'.

Note that for an approximating set X and a point y there may be more than one best approximation. If we let **X** be the points in SF corresponding to a set of example inputs then a way of extending the computation to an S-expression having a projection to another point y not in X in SF is to choose the best approximation of y in X, if it exists, and apply its associated computation to y. To do this it is desirable to have a unique best approximation in the approximating set X for y in SF.

Lemma (L5): If X is a chain containing ω then for all y in SF there exists a unique best approximating element, x, in X.

Proof: Existence is achieved since ω is part of every approximating interval in SF. For uniqueness we assume there exists a y such that there are two distinct best approximating elements x_1 and x_2 in X. Since X is a chain either $x_1 \leq x_2$ or $x_2 \leq x_1$. Since x_1 is a best approximation $x_2 \leq x_1$. However x_2 is also a best approximation so that $x_1 \leq x_2$, therefore $x_1 = x_2$, violating the assumption about two best approximating elements in X. ■

Definition (D24): A LINEAR APPROXIMATING SET, X, is defined to be a well ordered subset of SF, *ie.* **X** is a chain.

Theorem (T3): Given a linear approximating set $X=\{x_1,x_2,\ldots,x_k\}$ in SF, such that $x_i \leq x_{i+1}$, there exists a sequence of predicates $p_1[x],p_2[x],\ldots,p_k[x]$ over the set of S-expressions such that for all $y \varepsilon D_S$ if x_i is the best approximation of proj[y] in X then $p_i[y]=T$ and for all $j<i$, $p_j[y]=F$.

Proof: Define $p_i[x] \leftarrow$ join[proj[x];x_{i+1}]$\neq$proj[x] $1 \leq i < k$

Let x_i be the best approximation for some y, first we show that $p_i[y]=T$. Proof by contradiction: Assume that $p_i[y]=F$, then join[proj[y];x_{i+1}]=proj[y]. But this means that $x_{i+1} \leq$ proj[y], *ie.* that x_{i+1} is in the interval [ω,proj[y]] contradicting the assumption that x_i is a best approximation.

We now must show that $p_j[y]=F$ for all $j<i$.

Since x_i is a best approximation to y, $x_i \varepsilon [\omega,\text{proj}[y]]$. Since $x_j \leq x_i$ $j \leq i$ we know that $x_j \varepsilon [\omega,\text{proj}[y]]$. Therefore join[$x_j$;proj[y]]=proj[y] for $j \leq i$. Therefore $p_j[y]=F$ for all $j<i$. ■

The theorem gives a way of constructing a set of predicates for the McCarthy conditional if the projections of the example inputs form a linear approximating set. Note that the ordering used in this construction is precisely that which is required to eliminate the problem of subsumption. This theorem demonstrates the existence of predicates which might be used in the McCarthy conditional to select the appropriate fragments in $F_E[x]$, however the predicates are not defined in terms of the primitives allowed for program construction. The theorem does give an insight into how the predicates might be constructed from the LISP primitives **car, cdr, cons,** and **atom.**

Definition (D25): In a poset, P, y will be said to COVER x in P iff $x<y$ but $x<w<y$ for no w in P.

We note the following fact about x and a covering element y in the lattice SF. The element y covers x in SF if and only if by substituting $(\omega.\omega)$ for some single occurrence of ω in x the new expression is identical to y. Alternatively, y covers x in SF if and only if some single occurrence of $(\omega.\omega)$ in y may be replaced by ω to produce x. From this we may assume, for a given x and covering element y, the existence of the function π which will select the ω in x and the $(\omega.\omega)$ in y, *ie.* $\pi[x]=\omega$ and $\pi[y]=(\omega.\omega)$. The function π is defined as a composition of **car**'s and **cdr**'s to select the appropriate subexpression, *ie.* π finds the position in x and y where the two expressions differ. There is a procedure for generating the function π given $x,y \varepsilon SF$, such that y covers x.

Procedure (P3):

```
πgen[x;y]←πg[x;y;λ[x;x]]
πg[x;y;z]←[and[atom[x];atom[y]]→();
            atom[x]→list[z];
            T→append[πg[car[x];car[y];compose[λ[x;car[x]];z]];
                πg[cdr[x];cdr[y];compose[λ[x;cdr[x]];z]]].
```

Lemma (L6): The procedure is an algorithm for generating π.

Proof: πg and thus πgen obviously terminates as each recursion produces arguments of simpler form than the previous and there are specific values when x is atomic.

Instead of producing a formal proof that πgen produces the function πg as a value, We give a less formal argument that describes the function. The function π recursively traverses the two trees x and y comparing each subtree for equality. The third variable, z, is used to accumulate a function that represents the root of the current subtree being compared. The recursion terminates if both subtrees are atomic, returning a value of the null list, nil, to indicate that there is no difference between the two subtrees. The second form of termination is when x is atomic and y is a dotted pair. In this case the function z is returned to indicate where the two trees differ. ■

Exhibit: Let $x_1=((\omega.\omega).(\omega.(\omega.\omega)))$ and $x_2=((\omega.\omega).((\omega.\omega).(\omega.\omega)))$. Then $\pi[x]=$cadr[x] since $\pi[x_1]=\omega$ and $\pi[x_2]=(\omega.\omega)$. Figure 3.5 shows x_1 and x_2 expressed as cons trees and the value of z at each node during the scan of the trees by πg.

Definition (D26): We define $\pi_i = \pi\text{gen}[x_i;x_{i+1}]$ provided x_{i+1} covers x_i.

Theorem (T4): If $X=\{x_1,\ldots,x_k\}$ is a subset of SF such that x_{i+1} covers x_i then the following

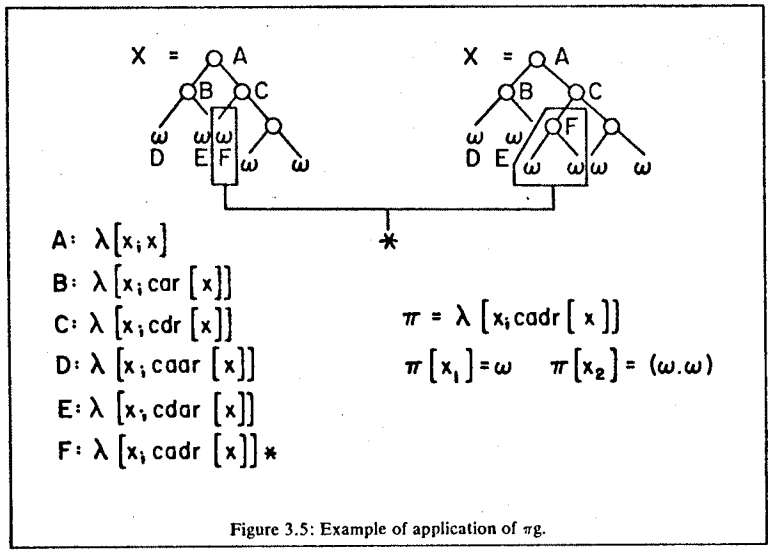


Figure 3.5: Example of application of πg.

sequence of predicates, $p_1[x] \leftarrow atom[\pi_1[x]]$, ... ,$p_{k-1}[x] \leftarrow atom[\pi_{k-1}[x]]$, $p_k[x] \leftarrow T$ over the set of S-expressions is such that $p_i[y]=T$ if x_i is the best approximation to y in X and $p_j[y]=F$, $j<i$.

Proof: If x_i is the best approximation to y then $x_i \leq y$ and x_{i+1} is not less than or equal to y. $\pi_i[x_i] \leq \pi_i[y]$ since π_i is composed of car's and cdr's and both are monotone over the lattice, *ie.* if $x \leq y$ then $car[x] \leq car[y]$ and $cdr[x] \leq cdr[y]$. If $\pi_i[x_i] < \pi_i[y]$ then x_{i+1} would be the best approximation to y since $\pi_i[x_i] < \pi_i[x_{i+1}] = (\omega.\omega) \leq \pi_i[y]$, therefore $\pi_i[x_i] = \pi_i[y]$. And finally $atom[\pi_i[y]] = atom[\pi_i[x_i]] = atom[\omega] = T$.

Now show that $p_j[y]=F$, $j<i$. $\pi_j[x_j] < \pi_j[x_i] \leq \pi_j[y]$ by the definition of π_j and the monotonicity of π_j.

$\pi_j[x_i]=\omega<\pi_j[y]$ therefore $atom[\pi_j[y]]=F$. ■

The theorem gives a constructive way to generate the predicates needed in the McCarthy conditional to select the computations (fragments) associated with the best approximation of a given expression. The restriction that the projection of the example inputs must form a set $X=\{x_1,...,x_k\}$ where x_{i+1} covers x_i is overly restrictive. In particular it says that predicates, and therefore a program, may be generated only if the examples, $\langle x_i, y_i \rangle$, satisfy the following relationship: $proj[x_{i+1}]$ covers $proj[x_i]$ in the lattice SF. An example of inputs that would satisfy

the condition is the sequence (), (A), (A B), ...; whereas the following sequence of inputs would not satisfy the condition: (), (A B), (A B C D), ...

We may generalize the previous theorem by requiring that the set of projections of the example inputs obey the following conditions:

a) The set must be well ordered with respect to the relation $\leq$. We will denote the set of projections of the inputs by $X=\{x_1,...,x_k\}$ where $x_i \leq x_{i+1}$ for $1 \leq i \leq k$.

b) The interval $[x_i, x_{i+1}]$, $1 \leq i < k$, is well ordered.

We may then construct the set of predicates in the following way. Let the interval consist of the points $x_i=y_1<y_2<...<y_{k-1}<y_k=x_{i+1}$ then the predicate that is true if x_i is the best approximation to a point is $p_i[x] \leftarrow or[atom[\pi_{y1}[x]];...;atom[\pi_{y(k-1)}[x]]]$. Note that if $\omega \epsilon X$ then every point in SF has a best approximation.

3.5 *An Approximation of the Function:*

Before continuing let us review the last two sections with an eye toward generating a program that will satisfy a set of examples. In §3.3 we presented an algorithm that will generate a program fragment from an example. The fragment was taken to be the encoding of the relationship between the input and output, in that the fragment would produce the example's output when applied to the example's input. In §3.4 we presented an algorithm that would generate control flow predicates from the set of example inputs. These predicates were to be used in the McCarthy conditional to select appropriate fragments. The predicate was constructed to be true when applied to an S-expression if and only if the example input associated with the predicate was the best approximation to the S-expression. The set of example inputs define a sequence of predicates in the McCarthy conditional that select the fragment associated with the example whose input is the best approximation to the program's input. The instantiation of a McCarthy conditional with the fragments and predicates generated from a set of examples is the most obvious form of a program that is defined by or satisfies a set of examples. Whatever else the system may do, it should at least generate such a program given a set of examples.

Definition (D27): A STRAIGHT LINE PROGRAM $F_L[x]$ of the function, F[x], defined by the set of examples, $E=\{e_k\}$ will be the following LISP program

$F_L[x] \leftarrow [p_1[x] \rightarrow f_1[x];$
$p_2[x] \rightarrow f_2[x];$
...
$p_{k-1}[x] \rightarrow f_{k-1}[x];$
$T \rightarrow f_k[x]].$

where it is assumed that E is well ordered by relation $\leq$ on the input, *ie.*

$proj[input[e_1]] \leq proj[input[e_2]] \leq ... \leq proj[input[e_k]]$.

The function $f_i[x]$ is the fragment associated with the ith example and $p_i[x]$ is a predicate constructed by the method of the previous section.

A number of observations should be made about the program. First, $F_L[x]$ is consistent with **E**, *ie*. $F_L[x_i]=y_i$ for $1\leq i\leq k$. Second, as more examples are given the corresponding program defines unique computations for a larger subset of the domain of S-expressions. Third, for all $y\epsilon D_S$ such that $proj[y]\geq proj[x_k]$ the program computes the same function, namely $f_k[x]$.

Although the first two observations satisfy ones expectations of the program, the third is somewhat contrary to what one would like as the definition of the function characterized by the set of examples **E**. Implicit in most definitions by example is the use of ellipses, a concept inconsistent with this third observation about the program $F_L[x]$. To capture the idea of ellipses we will modify the definition of a straight line program so as to permit an inferential extension of the function to a domain larger than that of the example inputs. Inference is used to extend the domain to elements of D_S that have projections into points of SF that are greater than the projections of the set of example inputs. As will be seen, the rule of inference is not always applicable and therefore it is not always possible to extend the domain of the program being synthesized. The inference process depends on being able to detect a regularity in the structure of the fragments and predicates that are used in the approximation. It is the continuation of a regular pattern that is usually what is meant by ellipses.

If the series of fragments and predicates were numeric series instead of parts of a program one might consider finding a difference equation or recurrence relation that would characterize the difference between successive elements of the two series. It turns out that there is an analogous procedure, that we will call DIFFERENCING, that may be applied to pairs of fragments and predicates in an attempt to reveal the difference between successive elements.

Before presenting the difference algorithm we wish to make a digression concerning the generality of this approach to program synthesis. Up to this point all of the material has depended on the specific kinds of programs being generated. Throughout the sections on fragment and predicate generation we have made extensive use of the LISP programming language and data structures. From this point on the techniques are more general. Given sets of fragments and predicates constructed from any set of primitive functions and an ordering on these sets such that $F_L[x]$ may be generated, one may apply the techniques of the next section in an attempt to find a recursive representation for the program $F_L[x]$. In addition some of the techniques and transformations to be described have potential application to the areas of proving program correctness and the optimization of recursive programs. These topics will be discussed briefly in a later chapter.

3.6 *Differencing*:

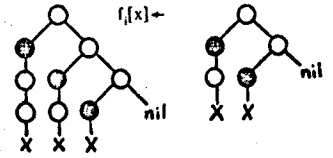
At this point we have constructed a program $F_L[x]$ from fragments generated from each example and from predicates determined from an analysis of the relationship of the example inputs. The idea of approximations was introduced to justify extending the definition of the function to input values not represented by examples. A second way of extending $F_L[x]$ is by inferring new fragments and predicates from the existing ones so that new computations are generated for the extended function.

In determining predicates for the program $F_L[x]$ we required that the set of projections of the inputs characterizing the function be well ordered in the lattice SF. This permitted construction not only of the predicates but ordering them in the program so that for $i<j$ no $p_i[x]$ subsumes $p_j[x]$ and no $p_i[x]=\omega$ when $p_j[x]=T$. If we consider the sequence of fragments and predicates that define an approximation to be analogous to an integer sequence then the problem of extending the function by deriving new fragments is analogous to the problem of finding further elements of the numeric sequence consistent with the given elements. In the numeric example a common approach is to attempt to define a generating function for the sequence by means of a recurrence relation or difference equation. This is analogous to finding an extension to the functional approximation that is a recursive function. In finding a difference equation one uses the given elements of the sequence and attempts to find relationships among them by taking a numeric difference between successive elements of a sequence. Just as a numeric difference is used as a way of understanding relationships among entities of a numeric sequence, so we will define an analogous concept for program fragments and predicates.

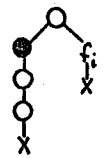
Definition (D28): Given a straight line program $F_L[x]$ defined by a set of examples, we say that there exists a DIFFERENCE between two fragments $f_i[x]$ and $f_j[x]$, $i<j$, if $f_j[x]$ may be expressed in terms of $f_i[x]$ and the LISP primitives. We denote the difference by the expression $f_j[x]\leftarrow\Phi[f_i[x];x]$. A similar definition exists for the difference between two predicates, $p_i[x]$ and $p_j[x]$, $i<j$.

Exhibit: The following examples illustrate the concept of differences.

1. If $f_i[x]\leftarrow car[x]$ and $f_j[x]\leftarrow cadr[x]$ then $f_j[x]=f_i[cdr[x]]$ so that a difference exists.
2. If $f_j[x]\leftarrow$ $f_i[x]\leftarrow$



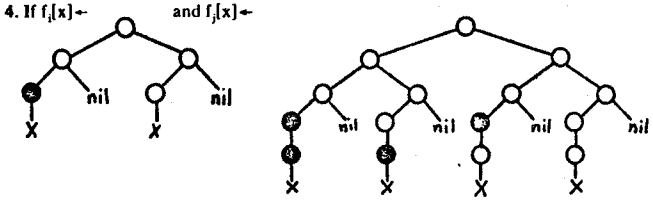
then $f_j[x]=$



so that a difference exists.

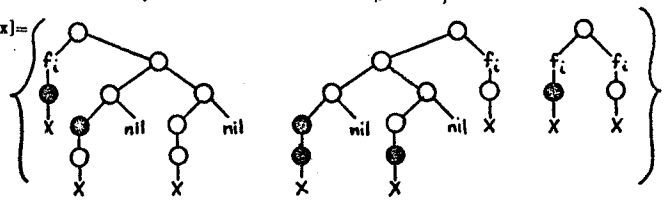
3. If $p_i[x]\leftarrow$atom[cdr[x]] and $p_j[x]\leftarrow$atom[cdddr[x]] then $p_j[x]=p_i$[cddr[x]].

4. If $f_i[x]\leftarrow$ and $f_j[x]\leftarrow$



Then there are three possible differences between $f_i[x]$ and $f_j[x]$.

$f_j[x]=$



5. If f_i=car[x] and $f_j[x]$=cdr[x] then no difference exists between the two fragments.

As may be seen from the example, given any two fragments a difference may or may not exist. If a difference exists it may not be unique. Although the above definition permits $f_j[x]=\Phi[f_i[x];x]$ to contain multiple occurrences of $f_i[x]$, the remainder of this thesis restricts the difference to have at most one occurrence of $f_i[x]$. The restricted form of difference will be called LINEAR DIFFERENCE.

Lemma (L8): There exists an algorithm for finding a set of linear differences between two fragments $f_i[x]$ and $f_j[x]$, $i<j$, from a program $F_L[x]$, if any exist.

Proof:

1. Change the bound variable of $f_j[x]$ to a unique one, say y.

2. Attempt to find a substitution, σ, that replaces all x in $f_i[x]$ by b[y], *ie.* $\sigma=\{b[y]/x\}$, such that $f_i[b[y]]$ is identical to a subexpression of $f_j[y]$.

3. Repeat 2. for all possible subexpressions of $f_j[y]$. There are a finite number.

4. For each substitution that exists there is a difference formed by replacing the matched subexpression of $f_j[y]$ by the expression $f_i[b[y]]$. The difference will be of the form $f_j[y]=a[f_i[b[y]];y]$.

5. If no substitutions exist for x in $f_i[x]$ see if $f_j[y]$ is a subtree of $f_i[x]\{y/x\}$. If so then $f_j[x]=a[f_i[x]]$, where a[x] is the function that selects the subtree from $f_i[x]$ that is equal to $f_j[x]$.

The process of finding the substitution, σ, is identical to the unification algorithm in mechanized deductive systems. Algorithms for unification exist [Robinson 1965, 1970]. Since $f_j[y]$ is finite there are only a finite number of subexpressions, therefore, the procedure will eventually halt, yielding a set of differences, perhaps empty. ■

We use the concept of a difference between two fragments to develop conditions for inferring additional predicates and fragments from those in the straight line program $F_L[x]$. To place the inferred fragments into a meaningful context we need to define a new program $F_k[x]$ whose definition may be extended by including the inferred fragments in its definition. The program $F_k[x]$ will be called an approximating program and will be a partial function over D_S as compared with the total function $F_L[x]$. $F_k[x]$ will be defined for and equal to $F_L[x]$ for all but the last example.

Definition (D29): The kth APPROXIMATING PROGRAM, $F_k[x]$ defined by the set of examples $E=\{e_k\}$ will be the LISP program of the form

$F_k[x]\leftarrow[p_1[x]\rightarrow f_1[x];$
$p_2[x]\rightarrow f_2[x];$
...
$p_{k-1}[x]\rightarrow f_{k-1}[x];$
$T\rightarrow\omega]$

where $f_i[x]$ are fragments corresponding to e_i and the $p_i[x]$ are predicates determined by the algorithm of the previous section. The $p_i[x]$ and $f_i[x]$ for $1\leq i<k$ are the same as for $F_L[x]$.

It should be noted that only the first k-1 fragments are used in the definition of $F_k[x]$. In defining the concept of a kth-approximating program, we wish to avoid letting the kth fragment be the computation associated with the 'else' clause of the conditional. To properly extend the definition of the function it is necessary to have the function's 'else' clause be undefined.

Definition (D30): The DOMAIN, D_k, of the approximating program $F_k[x]$ will be defined as

$D_k=\{x\,|\,p_1[x]\}\cup\{x\,|\,p_2[x]\}\cup\ldots\cup\{x\,|\,p_{k-1}[x]\}$ *ie.*

just those S-expressions for which the program is defined. Note that $D_k=\{x \mid proj[x]=proj[input[e_i]],\ 1\le i<k\}$

The goal at this point is to infer new predicates and associated fragments so as to extend the definition of the function to be a total function over D_S.

Definition (D31): If there exists an initial point j and an interval n such that $1\le j<k$ in a kth-approximation $F_k[x]$ defined by a set of examples such that the following fragment differences exist

$$f_{j+n}[x]=a[f_j[b_1[x]];x]$$
$$f_{j+n+1}[x]=a[f_{j+1}[b_1[x]];x]$$
$$\dots$$
$$f_{k-1}[x]=a[f_{k-n-1}[b_1[x]];x]$$

and such that the following predicate differences exist

$$p_{j+n}[x]=p_j[b_2[x]]$$
$$p_{j+n+1}[x]=p_{j+1}[b_2[x]]$$
$$\dots$$
$$p_{k-1}[x]=p_{k-n-1}[b_2[x]]$$

then we define the FUNCTIONAL RECURRENCE RELATION for the example to be

$$f_1[x],\dots,f_j[x],\dots,f_{j+n-1}[x],f_{i+n}[x]=a[f_i[b_1[x]];x] \quad j\le i\le k-n-1$$

and we define the PREDICATE RECURRENCE RELATION for the examples to be

$$p_1[x],\dots,p_j[x],\dots,p_{j+n-1}[x],p_{i+n}[x]=p_i[b_2[x]] \quad j\le i\le k-n-1.$$

Inference: If a functional recurrence relation and a predicate recurrence relation exist for a set of examples such that $k-j\ge 2n$ then we INDUCTIVELY INFER that these relationships hold for all $i\ge j$. The recurrence relations may be used to produce new examples and corresponding fragments leading to new approximating programs. The mth approximating program, $m\ge j$, is of the form

$$F_m[x]\leftarrow[p_1[x]\rightarrow f_1[x];$$
$$p_2[x]\rightarrow f_2[x];$$
$$\dots$$
$$p_k[x]\rightarrow f_k[x];$$
$$p_{k+1}[x]\rightarrow f_{k+1}[x];$$
$$\dots$$
$$p_m[x]\rightarrow f_m[x];$$
$$T\rightarrow\omega]$$

where the $p_i[x]$ and $f_i[x]$, $j\le i\le n$ are defined in terms of the recurrence relations. We will call the set $\{F_m[x]\}$, $m\ge j$, the set of APPROXIMATING FUNCTIONS for the function defined by the examples. Note that each new predicate fragment pair in $F_m[x]$ provides an effective way of computing other examples for the inferred function.

Lemma (L9): The set of approximating functions, if it exists, is a chain with partial order $\le_F$ where

$F[x]\le_F G[x]$ if it is true that for all x for which F[x] is defined, F[x]=G[x].

Proof: Using the definition of the domain of a function $F_i[x]$, it is obvious that $F_i[x]\le_F F_{i+1}[x]$ for all i since D_i is a subset of D_{i+1}. Since the set $\{F_i[x]\}$ is well ordered it is a chain. ■

The term approximating function set was chosen because each program in the set is an approximation to the function being generated from the set of original examples. The recurrence relations define arbitrarily large sets of examples for which one may derive programs for the approximating function set. In the limit as the size of the example set approaches infinity, the number of computations in the greatest (with respect to the relation $\le_F$) program in the set likewise approaches infinity. This then is the motivation for defining the function defined by the examples, F[x], to be $\sup\{F_m[x]\}$ or the limit as m approaches infinity of $F_m[x]$.

Definition (D32): If a set of examples define the chain $<\{F_m[x]\},\le_F>$ we define F[x], the function defined by the examples, to be the $\sup\{F_m[x]\}$ or limit as m approaches infinity of $F_m[x]$.

3.7 *Program Synthesis*:

The simplest kind of recursive program synthesis is specified by the basic synthesis theorem. The proof of the theorem relies on material from the fixpoint theory of program language semantics. During the early 1970's a number of people, [Scott 1970], [Manna and Vuillemin 1972] proposed and developed a theory of recursive program semantics based on the first recursion theorem of [Kleene 1952]. The purpose of such a development was to try and formalize what partial function was defined by a particular recursive program. The basic synthesis theorem represents the converse of this problem, *ie.* it attempts to find a recursive program from a recurrence relation characterization of a partial function. We will assume that the reader is familiar with the fixpoint theory of program semantics and refer those who are not to [Manna 1974].

Theorem (T5-Basic Synthesis): If a set of examples defines F[x], with recurrence relations

$$f_1[x],\dots,f_n[x],f_{i+n}[x]=a[f_i[b[x]];x]]$$
$$p_1[x],\dots,p_n[x],p_{i+n}[x]=p_i[b[x]] \quad \text{for } i\ge 1$$

then F[x], is equivalent to the following recursive program

$$F[x]\leftarrow[p_1[x]\rightarrow f_1[x];$$
$$\dots$$
$$p_n[x]\rightarrow f_n[x];$$
$$T\rightarrow a[F[b[x]];x]]$$

Proof: To prove the theorem we will show that the function calculated by F[x], *ie.* the least fixpoint of the functional, τ, defined as

$$\tau[F][x]\leftarrow[p_1[x]\rightarrow f_1[x];$$

$p_n[x] \rightarrow f_n[x];$
$T \rightarrow a[F[b[x]];x]$

is identical to $F[x]=\sup\{F_n[x]\}$.

First we will show that $\tau^k[\Omega][x]=F_{kn}[x]$ for all k, where $\tau^k[F][x]=\tau[\tau^{k-1}[F]][x]$ and $\Omega[x]$ is the function that is everywhere undefined, *ie.* $\Omega[x]=\omega$ for all x. From this equality we will conclude that $\sup\{F_i[x]\}$=limit as i approaches infinity $\tau^i[\Omega][x]$= the least fixpoint of $\tau[F][x]$.

Prove the equivalence by induction: The basis (k=1):

$\tau^1[\Omega][x]=[p_1 \rightarrow f_1[x];$
$\quad \cdots$
$\quad p_n[x] \rightarrow f_n[x];$
$\quad T \rightarrow \omega]$
$=F_n[x]$

Induction hypothesis:

$\tau^k[\Omega][x]=[p_1[x] \rightarrow f_1[x];$
$\quad \cdots$
$\quad p_{kn}[x] \rightarrow f_{kn}[x];$
$\quad T \rightarrow \omega]$
$=F_{kn}[x]$

$\tau[\tau^k[\Omega]][x]=[p_1[x] \rightarrow f_1[x];$
$\quad \cdots$
$\quad [p_n[x] \rightarrow f_n[x];$
$\quad T \rightarrow a[[p_1[b[x]] \rightarrow f_1[b[x]];$
$\quad \cdots$
$\quad p_{kn}[b[x]] \rightarrow f_{kn}[b[x]];$
$\quad T \rightarrow \omega];x]]$

$= \quad [p_1[x] \rightarrow f_1[x];$ by D2.e.iv
$\quad \cdots$
$\quad p_n[x] \rightarrow f_n[x];$
$\quad p_{n+1}[b[x]] \rightarrow a[f_{n+1}[b[x]];x];$
$\quad \cdots$
$\quad p_{n+kn}[b[x]] \rightarrow a[f_{n+kn}[b[x]];x];$
$\quad T \rightarrow \omega]$

$= \quad [p_1[x] \rightarrow f_1[x];$ from the recurrence defintion
$\quad \cdots$
$\quad p_{(k+1)n}[x] \rightarrow f_{(k+1)n}[x];$
$\quad T \rightarrow \omega]$

$= \quad F_{(k+1)n}[x]$ ■

The basic synthesis theorem provides the foundation for recursive program synthesis from a finite set of examples. It should be noted that the theorem specifies the generation of a recursive

program based on the the existence of functional and predicate recurrence relations that satisfy a particular set of conditions. The conditions, as stated in the theorem, are far too strict for most applications. Three corollaries will be presented below that will define recursive program equivalents for recurrence relations with simpler conditions than those of the basic synthesis theorem. The first corollary relaxes the condition that the recurrence relations must define the relationship among all the fragments. By exempting some finite set of initial fragments from the set of fragments characterized by a recurrence relation many more functions may be synthesized. The second corollary relaxes the condition that the b[x] in the functional recurrence be the same as the b[x] in the predicate recurrence relation. The third corollary relaxes both conditions.

Corollary (C1): If a set of examples define F[x] with the recurrence relations

$f_1[x],\ldots,f_{j-1}[x],f_j[x],\ldots,f_{j+n-1}[x],f_{i+n}[x]=a[f_i[b[x]];x],\ i \geq j$

$p_1[x],\ldots,p_{j-1}[x],p_j[x],\ldots,p_{j+n-1}[x],p_{i+n}[x]=p_i[b[x]],\ i \geq j$

then the recursive program, F[x] defined below is identical to F[x].

$F[x] \leftarrow [p_1[x] \rightarrow f_1[x];$
$\quad \cdots$
$\quad p_{j-1}[x] \rightarrow f_{j-1}[x];$
$\quad T \rightarrow G[x]]$

where

$G[x] \leftarrow [p_j[x] \rightarrow f_j[x];$
$\quad \cdots$
$\quad p_{j+n-1}[x] \rightarrow f_{j+n-1}[x];$
$\quad T \rightarrow a[G[b[x]];x]]$

Proof: Immediate from the basic synthesis theorem. ■

Corollary (C2): If a set of examples define F[x] with the recurrence relations

$f_1[x],\ldots,f_n[x],f_{i+n}[x]=a[f_i[b_1[x]];x] \quad i \geq 1$

$p_1[x],\ldots,p_n[x],p_{i+n}[x]=p_i[b_2[x]] \quad i \geq 1$

then the of program, F[x], is equivalent to F[x].

$F[x] \leftarrow G[x;x]$

where

$G[x;y] \leftarrow [p_1[y] \rightarrow f_1[x];$
$\quad \cdots$
$\quad p_n[y] \rightarrow f_n[x];$
$\quad T \rightarrow a[G[b_1[x];b_2[y]];x]]$

Proof: Let $F_i[x] \leftarrow [p_1[x] \rightarrow f_1[x];$
$\quad \cdots$
$\quad p_i[x] \rightarrow f_i[x];$
$\quad T \rightarrow \omega]$

be an approximating program for F[x]. The $p_i[x]$ and $f_i[x]$ are defined from the initial fragments and the recurrence relations. We now define a diadic approximating function $G_i[x;y]$ which is identical to $F_i[x]$ whenever y=x.

$G_i[x;y] \leftarrow [p_1[y] \rightarrow f_1[x];$
$\quad \ldots$
$\quad p_i[y] \rightarrow f_i[x];$
$\quad T \rightarrow \omega]$

We can show that $G_{i \times n}[x;y] = \tau^1[G][x;y]$ where G[x;y] is defined to be the recursive function of the corollary. This may be demonstrated by using a proof technique similar to that used for the basic synthesis theorem. ■

Corollary (C3): If a set of examples define F[x] with the recurrence relations

$f_i[x],\ldots,f_{j-1}[x],\ldots,f_{j+n-1}[x],f_{i+n}[x]=a[f_i[b_1[x]];x] \quad i \geq j$

$p_i[x],\ldots,p_{j-1}[x],\ldots,p_{j+n-1}[x],p_{i+n}[x]=p_i[b_2[x]] \quad i \geq j$

then the recursive program, F[x], defined below is identical to F[x].

$F[x] \leftarrow [p_1[x] \rightarrow f_1[x];$
$\quad \ldots$
$\quad p_{j-1}[x] \rightarrow f_{j-1}[x];$
$\quad T \rightarrow G[x;x]]$

where

$G[x;y] \leftarrow [p_j[y] \rightarrow f_j[x];$
$\quad \ldots$
$\quad p_{j+n-1}[y] \rightarrow f_{j+n-1}[x];$
$\quad T \rightarrow a[G[b_1[x];b_2[y]];x]]$

Proof: Immediate from Corollaries C1 and C2. ■

Application of the basic synthesis theorem (T6) and the three corollaries (C1, C2 and C3) in program generation is illustrated in the next chapter (see in particular §4.1, §4.4, and §4.2). All recursive programs generated by the system must use at least one of the four results of this section. These, however, are not sufficient to generate all of the programs presented in the next chapter. To generate the more interesting programs requires use of the results of this section with the technique of variable addition presented in the next section. Variable addition is a procedure for transforming a set of fragments with no characterizing recurrence relation into an equivalent set that may have a characterizing relation. If such an equivalent set exists the results of this section may be applied to the new set of fragments to generate a recursive function that may be used in the definition of the desired function.

3.8 *The Variable Addition Heuristic:*

We recall that the synthesis results of the previous section require the existence of characterizing recurrence relations for the predicate and fragment portion of the program. Existence of these recurrence relations is not always insured, indeed, the ability to find a difference expression among all the elements of a set of fragments is not always possible (see, for example, §4.7). Application of the variable addition heuristic transforms a set of fragments into a more general set for which a recurrence relation may exist. The new set is more general in that each of the fragments of the new set is a function of two variables. Substituting a common expression for the added variable in the new set will produce a a set of fragments identical to the original set.

Heuristic (Variable Addition): Given a set of fragments $\{f_i[x]\}$ $1 \leq i \leq k$

1. Search $\{f_i[x]\}$ for a sub-expression, a, that is common to a subset $\{f_j[x]\}$ $1 \leq j \leq k$.
2. Rewrite these fragments as $g_j[x;a]=f_j[x]$ $1 \leq j \leq k$.
3. Abstract the fragments as $g_j[x;y]$ $1 \leq j \leq k$ from step 2.
4. Derive, if possible, a recurrence relation to describe $\{g_j[x;y]\}$ of the form $g_{j+n}[x;y]=a[g_j[b[x];c[x;y]];x;y]$.
5. If step 4 did not result in a recurrence relation return to step 1 and select another subexpression.
6. If step 4 is successful, use one of the synthesis theorems to generate a recursive program.
7. To generate a definition for the function F[x] associated with the fragments $\{f_i[x]\}$ we instantiate the variable y of the function g[x;y] with the expression a *ie.* perform the inverse of the operation of step 3.

Figure 3.6 is a diagram illustrating application of the heuristic to generate a function F[x] from a set of fragments $\{f_i[x]\}$. The definition of F[x] is in terms of the more general function G[x;y].

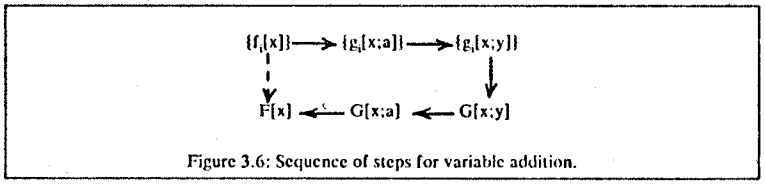


Figure 3.6: Sequence of steps for variable addition.

The technique of solving a particular specific problem by first solving a more general problem whose solution implies the solution of the original problem is called the *insane heuristic* by [Siklossy 1974]. This approach was used by [Boyer and Moore 1975] to prove properties of LISP programs and later by [Siklossy 1974] in a program generation system, and is similar to the idea of *accumulator generalization* in [Moore 1975]. As has been shown by the above authors the concept embodied in this heuristic is a very powerful one, yet lacks intuitive justification for why it

works. Almost all of the interesting (non-trivial) programs that are synthesized by the THESYS system rely on an application of the variable addition heuristic. Intuitively its role is to provide an automatic way of introducing a temporary or local variable to recursive programs. This has long been recognized as a programmer's trick. For example, in [Black 1964] it is recommended that the recursive equivalent for the iterative version of the function **reverse** be implemented in this manner. The THESYS produced program for **reverse**, §4.3, is precisely the one recommended in [Black 1964].

3.9 *A Strategy for Program Synthesis:*

We have presented all of the components of the THESYS system for synthesizing recursive programs from examples. What remains is to organize these techniques into a strategy for the production of programs, *ie.* a system organization for the system THESYS.

Figure 3.7 is a very high level diagram of the general flow of control in the system. Each box references the section or sections where the relevant material may be found for calculating the indicated entities.

A number of issues regarding the implementation need be addressed at this time. First, if at any time in the process of synthesizing a program it becomes apparent that a recursive program is not going to be produced from the examples the user is asked to specify his choice of entering more examples or accepting a default program. The default program is the straight line program that is directly derivable from the examples (§3.5). Addition of more examples restarts the process with the enlarged set of examples.

Next, an issue arises as to user prompting. Given the lattice characterization of the program domain presented in §3.4 it is possible to prompt the user for examples to fill in gaps in a potential chain of example inputs. To prompt the user, the system would select a possible point from the lattice SF. The system would then ask the user to give the appropriate output for an input corresponding to the point. In this way it is possible to guarantee the existence of a set of examples which are well ordered and from which a predicate recurrence relation will exist. Determining when such a mechanism should be employed is a problem. On one extreme it need never be invoked, relying on the shakey assumption that the user will always give a complete specification. At the other extreme is the invoking of the mechanism to acquire examples with inputs forming a chain, particularly a chain containing the element ω, where each element is covered by the next larger element in the chain. This may seem desirable in that all functions synthesized by such a system will be total, however, this imposes a restriction on the class of programs generated. For example, the system has the potential for calculating partial functions, see for example §4.1. The issue of when to prompt has not been settled so that the system defaults to a policy of no prompting.

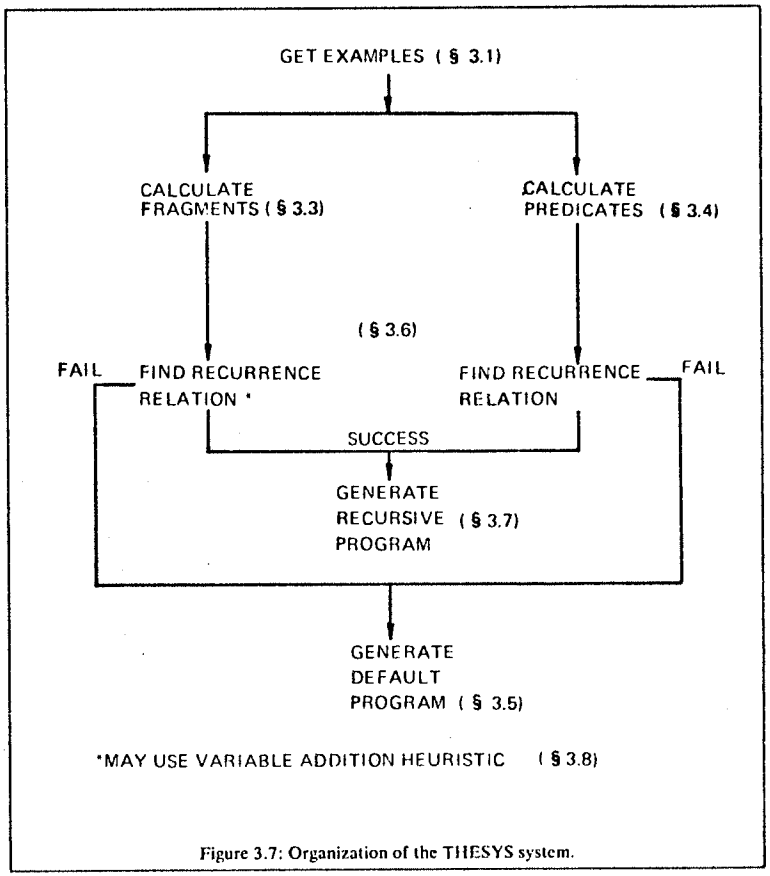


Figure 3.7: Organization of the THESYS system.

With an interactive system, such as the one being described, the issue of verifying the correctness of the system's output arises. Currently the system displays the program generated and relies on the user to judge its correctness. A better interactive solution is for the system to present not only the program but a series of new examples. The generation of new examples is a relatively trivial matter. The chain in SF that characterizes the domain is known so that a next element may be generated. Applying the program to the element produces the desired example. This process may be extended for any number of verifying examples.

The system THESYS was written in LISP. Although the system is complete, it was developed as a research effort and, therefore, has evolved rather than being designed. For this reason no listings are included in this report. Anyone wishing to duplicate the experimental system would be better served by starting with the principles and concepts of §3.3-3.8 than with the listing of the current version of THESYS.

3.10 *The Class of Programs Generated by the System:*

In concluding this chapter on the process of programming by example something needs to be said about the class of programs the system is capable of synthesizing. Unfortunately there are few mechanisms for classifying classes of programs. From recursive function theory there are the categories of primitive recursive, total and partial recursive. It would be nice to say that the class of functions generated by the system fit one of these categories. Two quick examples show this to be impossible. First, the ability to synthesize partial functions (§4.1) means the system generates a subset of partial recursive functions. That it is a proper subset may be seen by the inability to express all computable phenomena as a finite set of examples (recall the example of a random number generator of §3.1). The observation that the class of programs generated by the system is a subset of the partial recursive functions is hardly a useful observation, except to note that the problem of classifying programs is a difficult one and a thorough investigation of this topic is beyond the scope of this thesis. However, there are some observations that may be made about the programs generated by THESYS that may indicate something about the complexity of class of all such programs.

Ignoring for a moment the default program, we note that the programs are monadic and belong to the class of linear recursive program schemata. If this were schemata theory such a description would constitute a classification of the complexity of the programs generated by the system. However, we are dealing with interpreted function symbols and as may be seen by the Turing machine simulation in Appendix 2 any computable function may be written in a form that will satisfy the above description.

Having failed to get an upper bound on the power of the system by the observation above, we make a further observation that reflects the properties of program specification by example. In §3.2 we introduced the notion of conservative programs. It is not quite true that all programs produced by the system are conservative (see §4.6) but the notion of conservative program seems to reflect an essential feature of programming by example.

Consider, for a moment, the following non-conservative program, r,that is similar to the function reverse. Take as a goal the task of synthesizing it from examples. Starting with the program we want to find a set of examples that will lead to the program in its current form.

r[x]←[atom[car[x]]→cdr[x];

T→r[cons[cdar[x];cons[caar[x];cdr[x]]]]

The following set of examples characterize the function

E={ e_1=<((),()>,
e_2=<((A)),(A)>,
e_3=<((A B)),(B A)>,
e_4=<((A B C)),(C B A)>}

As may be seen from the function and the examples r is similar to the two argument definition of reverse (§4.3) but with the two arguments packed (cons ed) together as a single argument. Working backwards from the program we may derive the functional recurrence relation that is needed to generate r. It is of the form

r_{i+1}[x]=r_i[cons[cdar[x];cons[caar[x];cdr[x]]].

Figure 3.8a shows the fragments associated with the first three examples. These are *semantically* equivalent to the first three computation sequences of the function r shown in Figure 3.8b. That the computation sequences for r may be reduced to simpler form by transformations based on the semantics of the language is equivalent to the statement that the program is nonconservative. The process of reduction destroys all evidence as to the structure of the computation that is needed by the system to determine the recurrence relation. It is impossible, therefore, for the system to ever generate the recurrence relation needed for the synthesis of the function r. It is possible to synthesize an equivalent program to r by two applications of the variable addition heuristic. The resulting program will be similar in structure to the example reverse in §4.2. Although the system will never produce a non-conservative program like the function r it has the potential to produce programs that are equivalent to them.

As may be gathered from the above discussion the problem of program characterization is difficult. Before leaving this matter we should point out one further complicating fact. The example at §4.7 is non-conservative, however it is so in a way that does not destroy the information about the recurrence relation needed to characterize the function. No matter how appealing it may be to have a formal characterization for the functions generated by THESYS it must remain for future work to solve this problem.

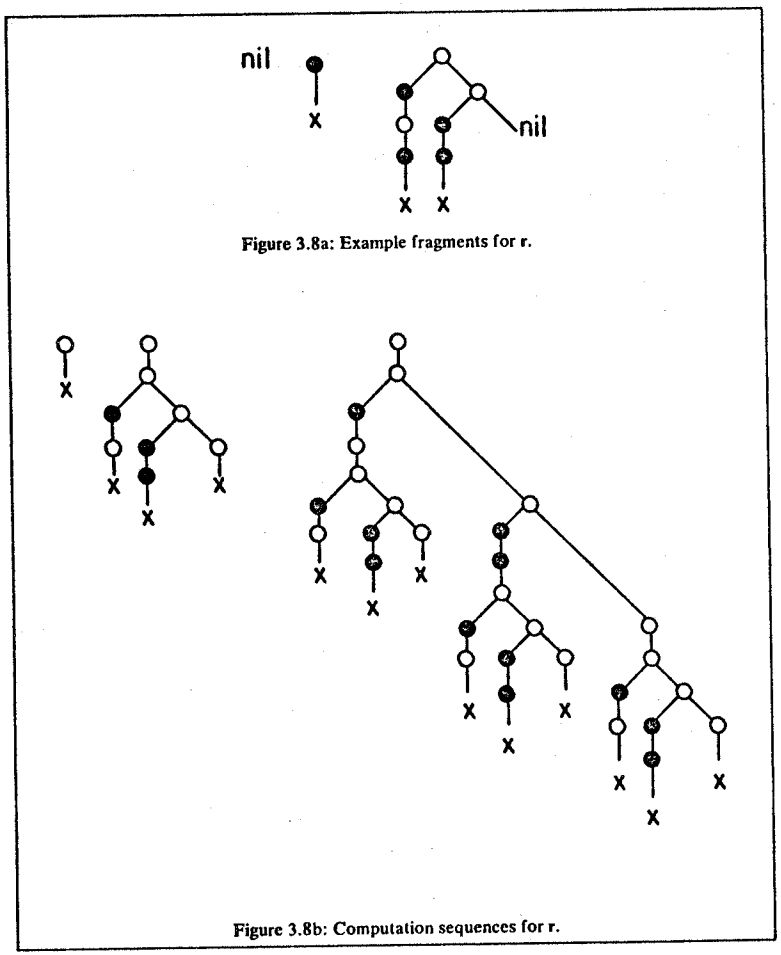


Figure 3.8a: Example fragments for r.

Figure 3.8b: Computation sequences for r.

CHAPTER 4

SAMPLE PROGRAMS

This chapter illustrates, by means of a number of examples, the procedures for synthesizing programs that were presented in the previous chapter. The examples, each chosen to illustrate some facet of the system, are organized as a sequence of steps corresponding to the application of some component of the system. The first two examples include a detailed explanation for each step of the derivation from example set to program. Later sections will include such explanations only for derivations that are new or unique. LISP M-expressions will be used interchangeably with the tree notation introduced in Definition D2.f of §3.2. Tree notation will be used when it is felt to be more concise or illustrative of the control structure being derived. Trees are easier to look at than an expression with a multitude of brackets.

Recall that the subscripts on the examples, fragments and predicates are not arbitrarily chosen but correspond to the ordering implied by the partial order of the inputs projected onto SF. The fragment $f_i[x]$ corresponds to the example e_i and the predicate $p_i[x]$ is derived from the inputs of examples e_i and e_{i+1}.

4.1 *last:*

Description: last[x]=the function that returns the last element of the list **x**.

Examples: E={ e_1=<(A) , A>,
e_2=<(A B) , B>,
e_3=<(A B C) , C>,
e_4=<(A B C D) , D> }

The set E is only one of many such sets that might be given in characterizing the function **last**. Since the set of examples does not include an example for atom input, it is assumed that the function to be synthesized will be undefined for such inputs, *ie.* the function will be defined for arguments that are lists of length greater than zero. From this set of fragments we may use the fragment generation procedure, §3.3.3, to generate the following set of fragments.

Fragments: $f_1[x]$=car[x]
$f_2[x]$=cadr[x]
$f_3[x]$=caddr[x]
$f_4[x]$=cadddr[x]

To find a functional recurrence relation, one must look among the fragments for differences that are the same. Application of the difference procedure and the inference rule, §3.6, produces the following functional recurrence relation.

Functional Recurrence Relation: $f_1[x]=car[x]$
$f_{i+1}[x]=f_i[cdr[x]]\ i\geq 1$

The projections of the inputs of E form a finite chain under the partial order $\leq$ defined in §3.4.2. For each pair of examples (e_i,e_{i+1}) in E a predicate, $p_i[x]$, may be defined using the predicate generation procedure of §3.4.3.

Predicates: $p_1[x]=atom[cdr[x]]$
$p_2[x]=atom[cddr[x]]$
$p_3[x]=atom[cdddr[x]]$

The difference procedure and inference rule may be applied to the set of predicates to yield a predicate recurrence relation.

Predicate Recurrence Relation: $p_1[x]=atom[cdr[x]]$
$p_{i+1}[x]=p_i[cdr[x]]\ i\geq 1$

The functional recurrence relation and predicate recurrence relations satisfy the conditions of the basic synthesis theorem (T5), §3.7, so that a program may be generated. The function last is an example of the simplest kind (in terms of Theorem T5) of recursive program that may be generated by the system. All functions for which the system generates a recursive program will use at least the five procedures in this example, *ie.* fragment generation, predicate generation, functional recurrence derivation, predicate recurrence derivation, and program synthesis.

Program generated:

```
last[x]←[atom[cdr[x]]→car[x];
         T→last[cdr[x]]]
```

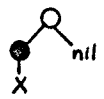
4.2 ***half:***

Description: half[x]=function that returns the first half of the even length list x and is undefined for lists of odd length.

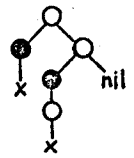
Examples: E={ e_1=<( ) , ( )>,
e_2=<(A B) , (A)>,
e_3=<(A B C D) , (A B)>,
e_4=<(A B C D E F) , (A B C)> }

Again we apply the fragment generation procedure of §3.3.3 to yield a set of fragments corresponding to E.

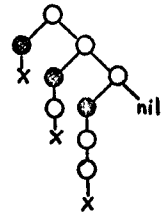
Fragments: $f_1[x]$=nil
$f_2[x]$=



$f_3[x]$=



$f_4[x]$=



The functional recurrence relation is determined as before using the difference procedure and the inference rule.

Functional Recurrence Relation: $f_1[x]=nil$
$f_{i+1}[x]=cons[car[x];f_i[cdr[x]]]\ i\geq 1$

Predicates: $p_1[x]=atom[x]$
$p_2[x]=atom[cddr[x]]$
$p_3[x]=atom[cddddr[x]]$

Predicate Recurrence Relation: $p_1[x]=atom[x]$
$p_{i+1}[x]=p_i[cddr[x]]\ i\geq 1$

The basic synthesis theorem may not be used to produce a program since the functional recurrence relation is of the form $f_{i+1}[x]=a[f_i[b_1[x]];x]$ and the predicate recurrence relation is of the form $p_{i+1}[x]=p_i[b_2[x]]$. The recurrence relations do satisfy Corollary C2 of, §3.7, so that the following program may be generated.

Program Generated:

```
half[x]←h[x;x]
h[x;y]←[atom[y]→nil;
        T→cons[car[x];h[cdr[x];cddr[y]]]]
```

4.3 ***reverse:***

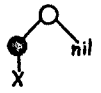
No paper on automatic LISP programming is complete without either the synthesis or verification of the program reverse. Note that unlike other systems [Green *et al.* 1974], [Shaw *et al.* 1974] the procedure described in this thesis does not use the function **append** in the construction of the program for reverse.

Description: reverse[x]=list composed of the top level elements of the list x in reverse order.

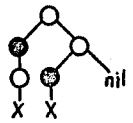
Examples: E={ e_1=<( ) , ( )>,
e_2=<(A) , (A)>,
e_3=<(A B) , (B A)>,
e_4=<(A B C) , (C B A)>}

Fragments: $f_1[x]$=nil

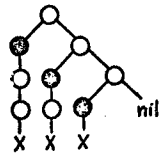
$f_2[x]$=



$f_3[x]$=

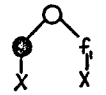


$f_4[x]$=

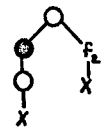


Functional Recurrence Relation: To derive a functional recurrence relation, one must apply the inference rule. The rule may be applied to a set of k fragments only if the differences generated by the difference procedure exists and the differences are all of the same form for some $i \leq k$. In the current example the following differences exist but are not of the same form.

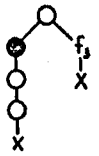
$f_2[x]$=



$f_3[x]$=



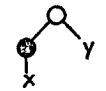
$f_4[x]$=



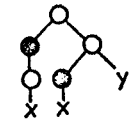
If one defined $cad^nr[x]$ as car[cdr[...cdr[x]...]] where cdr[x] appears n times then there is a pseudo recurrence relation $f_{i+1}[x]$=cons[$cad^ir[x]$;$f_i[x]$], that expresses the difference for the function reverse. Such a recurrence is not useful in synthesizing programs since it does not represent a constant difference, *ie.* successive differences of the same form. Since all of the synthesis algorithms require a functional recurrence relation for the synthesis of a program, the next step is to attempt to transform the fragments for **reverse** into an equivalent set by the variable addition heuristic, §3.8. The smallest subexpression common to all of the fragments is the expression **nil**, therefore we will let y=nil and transform the fragments into a new set.

$g_1[x;y]$=y

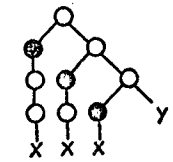
$g_2[x;y]$=



$g_3[x;y]$=

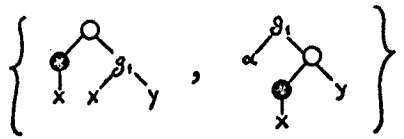


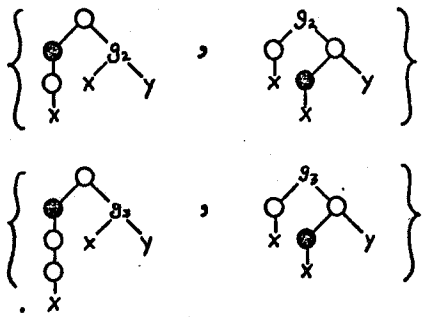
$g_4[x;y]$=



Applying the difference procedure we find that each pair of fragments $g_i[x;y]$ and $g_{i+1}[x;y]$ may be differenced in two ways.

$g_2[x;y]$=



$g_3[x;y]=$ 

$g_4[x;y]=$

The second of these differences are constant, *ie.* are of the same form. Therefore, the inference rule may be applied to produce the following recurrence relations:

$f_i[x]=g_i[x;nil]$ $i\geq 1$
$g_1[x;y]=y$
$g_{i+1}[x;y]=g_i[cdr[x];cons[car[x];y]]$ $i\geq 1$

Predicates: $p_1[x]=atom[x]$
$p_2[x]=atom[cdr[x]]$
$p_3[x]=atom[cddr[x]]$

Predicate Recurrence Relation: $p_1[x]=atom[x]$
$p_{i+1}[x]=p_i[cdr[x]]$ $i\geq 1$

Program Generated:

```
reverse[x]←r[x;nil]
r[x;y]←[atom[x]→y;
        T→r[cdr[x];cons[car[x];y]]]
```

The variable addition heuristic was used during the synthesis of reverse because the differences, though they existed, were not of the proper form for the inference rule. Later we will present examples where the differences do not even exist.

4.4 *Double every second element of a list (dbl2nd):*

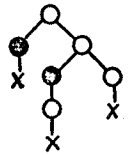
Description: dbl2nd[x]=function that returns the list x with two occurrences of every second element.

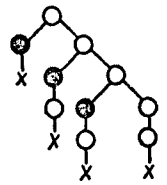
Examples: E={ $e_1=<(),()>$,
$e_2=<(A),(A)>$,
$e_3=<(A\ B),(A\ B\ B)>$,
$e_4=<(A\ B\ C),(A\ B\ B\ C)>$,
$e_5=<(A\ B\ C\ D),(A\ B\ B\ C\ D\ D)>$,

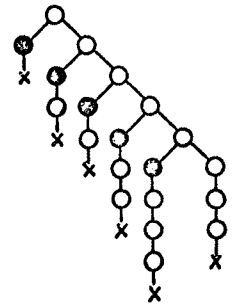
$e_6=<(A\ B\ C\ D\ E),(A\ B\ B\ C\ D\ D\ E)>$}

Fragments: $f_1[x]=nil$

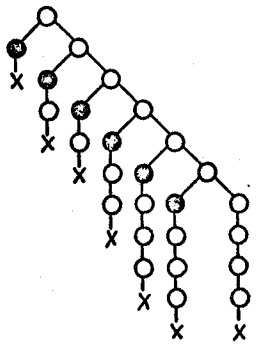
$f_2[x]=$ 

$f_3[x]=$ 

$f_4[x]=$ 

$f_5[x]=$ 

$f_6[x]=$



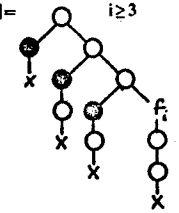
Functional Recurrence Relation: $f_1[x]=nil$

$f_2[x]=cons[car[x];nil]$

$f_3[x]=cons[car[x];cons[cadr[x];cdr[x]]]$

$f_4[x]=cons[car[x];cons[cadr[x];cons[cadr[x];cddr[x]]]]$

$f_{i+2}[x]=$ $i\geq 3$



Note that the recurrence relation might be characterized by stating that it is of order two, *ie.* the recurrence relation defines the i+2nd fragment in terms of the ith fragment.

Predicates: $p_1[x]=atom[x]$

$p_2[x]=atom[cdr[x]]$

$p_3[x]=atom[cddr[x]]$

$p_4[x]=atom[cdddr[x]]$

$p_5[x]=atom[cddddr[x]]$

Predicate Recurrence Relation: $p_1[x]=atom[x]$

$p_{i+1}=p_i[cdr[x]]$

The predicate recurrence relation derived from the examples is of order one. To apply the synthesis theorem (T5) or the corollaries (C1 and C2) it is necessary for the two recurrence

relations to be of the same order. The simplest way of achieving similarity between the orders is to calculate an order two relation from the order one predicate recurrence relation.

$p_{i+2}=p_{i+1}[cdr[x]]$

$=p_i[cddr[x]]$. therefore $p_{i+2}[x]=p_i[cddr[x]]$ $i\geq 1$

Program Generated:

```
dbl2nd[x]←[atom[x]→nil;
           atom[cdr[x]]→cons[car[x];nil];
           T→d[x]]
d[x]←[atom[cddr[x]]→cons[car[x];cons[cadr[x];cdr[x]]];
           atom[cdddr[x]]→cons[car[x];cons[cadr[x];cons[cadr[x];cddr[x]]]];
           T→cons[car[x];cons[cadr[x];cons[cadr[x];d[cddr[x]]]]]]
```

The function to double every second element of a list illustrates the use of Theorem T5 and Corollary C1 to recurrence relations of order greater than one.

4.5 *iota:*

To illustrate a numeric example, we give the steps for generating the *APL* ⍳ function. Integers must be encoded into a structural representation. The scheme used in the examples below encode zero (0) as the null list, nil, and the successor function, S(x)=x+1, as cons[m;x] where m is an arbitrarily chosen atom. This gives an encoding for 0 , 1 , 2 , 3,... as ( ) , (M) , (M M) , (M M M) ,...

Examples: E={ $e_1=<(),()>$,

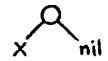
$e_2=<(M),((M))>$,

$e_3=<(M\,M),((M)\ (M\,M))>$,

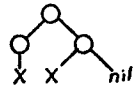
$e_4=<(M\,M\,M),((M)\ (M\,M)\ (M\,M\,M))>$}

Fragments: $f_1[x]=nil$

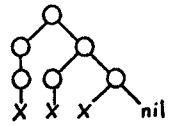
$f_2[x]=$



$f_3[x]=$



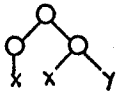
$f_4[x]=$



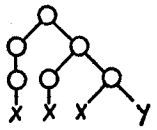
Functional Recurrence Relation: Like reverse, the set of fragments for iota has no functional recurrence relation. Applying the variable addition heuristic with y=nil we get a new set of fragments:

$g_1[x;y]=y$
$g_2[x;y]=$

$g_3[x;y]=$



$g_4[x;y]=$



yielding the functional recurrence relation:

$f_i[x]=g_i[x;nil]$ $i\geq 1$
$g_1[x;y]=y$
$g_{i+1}[x;y]=g_i[cdr[x];cons[x;y]]$ $i\geq 1$

Predicates: $p_1[x]=atom[x]$
$p_2[x]=atom[cdr[x]]$
$p_3[x]=atom[cddr[x]]$

Predicate Recurrence Relation: $p_1[x]=atom[x]$
$p_{i+1}[x]=p_i[cdr[x]]$ $i\geq 1$

Program Generated:

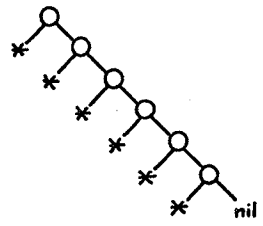
```
iota[x]←i[x;nil]
i[x;y]←[atom[x]→y;
        T→i[cdr[x];cons[x;y]]]
```

4.6 *Inserting a string, right justified, into a fixed field (fixfld):*

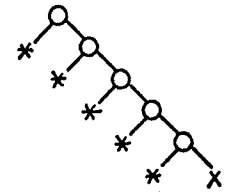
Description: fixfld[x]=function that returns a list of fixed length of the characters of x, right justified in the result, with the character * filling any remaining positions in the list. The function is of interest because it is an example of a function whose computation sequence is not conservative, see §3.2. This is reflected in the fragments characterizing the function by the effect that $f_{i+1}[x]$ is a proper subtree of $f_i[x]$, at least for the first six fragments.

Example: E={ e_1=<( ) , (* * * * * *)>,
e_2=<(A) , (* * * * * A)>,
e_3=<(A B) , (* * * * A B)>,
e_4=<(A B C) , (* * * A B C)>}

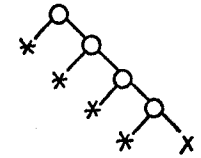
Fragments: $f_1[x]=$



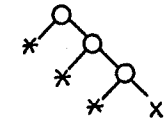
$f_2[x]=$



$f_3[x]=$



$f_4[x]=$



Functional Recurrence Relation: $f_1[x]=cons[*;cons[*;cons[*;cons[*;cons[*;cons[*;nil]]]]]]$
$f_2[x]=cons[*;cons[*;cons[*;cons[*;cons[*;x]]]]]$
$f_{i+1}[x]=cdr[f_i[x]]$ $i\geq 2$

Predicates: $p_1[x]=atom[x]$
$p_2[x]=atom[cdr[x]]$
$p_3[x]=atom[cddr[x]]$

Predicate Recurrence Relation: $p_1[x]=atom[x]$
$p_{i+1}[x]=p_i[cdr[x]]$ $i\geq 1$

Program Generated:

```
fixfld[x]←[atom[x]→cons[*;cons[*;cons[*;cons[*;cons[*;cons[*;nil ]]]]]];
          T→f[x;x]]
f[x;y]←[atom[cdr[y]]→cons[*;cons[*;cons[*;cons[*;cons[*;x ]]]]]
          T→cdr[f[x;cdr[y]]]]
```

Corollary C3 was used in this synthesis. For lists of length greater than six the function returns the right most six elements of the list. For example

fixfld[(A B C D E F G H)] = f[(A...H) ; (A...H)]
= cdr[f[(A...H) ; (B...H)]]
= cddr[f[(A...H) ; (C...H)]]
= cd^3r[f[(A...H) ; (D...H)]]
...

= cd^{7}r[f[(A...H) ; (H)]]
= cd^{7}r[(* * * * * A B C D E F G H)]
= (C D E F G H)

4.7 ***rotate:***

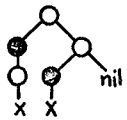
Description: rotate[x] returns the list x with the elements circularly shifted one place to the left.

Examples: E={ e_1=<( ) , ( )>,
e_2=<(A) , (A)>,
e_3=<(A B) , (B A)>,
e_4=<(A B C) , (B C A)>,
e_5=<(A B C D) , (B C D A)>}

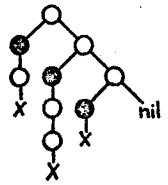
Fragments: $f_1[x]$=nil

$f_2[x]$=

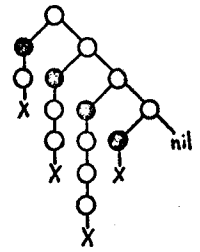
$f_3[x]$=



$f_4[x]$=



$f_5[x]$=

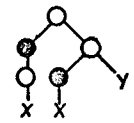


Functional Recurrence Relation: The function rotate is a good example of a more complicated program that requires the variable addition heuristic. Recall that to derive a recurrence relation it is necessary to have differences between pairs of fragments that are of the same form. For the set of five examples for rotate it is impossible to find differences between any two of the fragments f_3, f_4, or f_5. Being unable to produce differences prompts one to use the variable addition heuristic, with y=nil, to produce an equivalent set of fragments. Note that there is no way to difference most of the pairs. So we apply the variable addition heuristic, letting y=nil. This produces the following set of fragments:

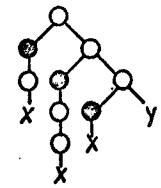
$g_1[x;y]$=y

$g_2[x;y]$=

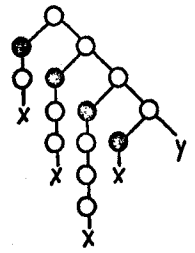
$g_3[x;y]$=



$g_4[x;y]$=



$g_3[x;y]=$

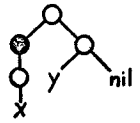


There is still no recurrence relation so we follow the variable addition heuristic and move through the fragments looking for new common subexpressions. We next try y=car[x]. The set of equivalent fragments now becomes:

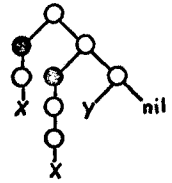
$f_1[x]=nil$

$g_2[x;y]=$

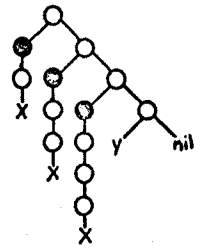
$g_3[x;y]=$



$g_4[x;y]=$



$g_3[x;y]=$



Now we find the recurrence relation $f_1[x]=nil$

$f_i[x]=g_i[x;car[x]]$ $i\geq 2$

$g_2[x;y]=cons[y;nil]$

$g_{i+1}[x;y]=cons[cadr[x];g_i[cdr[x];y]]$ $i\geq 2$.

Predicates: $p_1[x]=atom[x]$

$p_2[x]=atom[cdr[x]]$

$p_3[x]=atom[cddr[x]]$

$p_4[x]=atom[cdddr[x]]$

Predicate Recurrence Relation: $p_1[x]=atom[x]$

$p_{i+1}[x]=p_i[cdr[x]]$ $i\geq 1$

Program Generated:

```
rotate[x]←[atom[x]→nil;
           T→r[x;car[x]]]
r[x;y]←[atom[cdr[x]]→cons[y;nil];
           T→cons[cadr[x];r[cdr[x];y]]]
```

4.8 *pack:*

Description: pack[x] is best described by the examples.

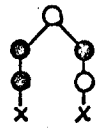
Examples: E={ $e_1=<(),()>$,
$e_2=<(())\ ,\ ()>$,
$e_3=<(\ (A)\ )\ ,\ (A)>$,
$e_4=<(\ (A)\ \ (B)\ )\ ,\ (A\ B)>$,
$e_5=<(\ (A)\ \ (B)\ \ (C)\ )\ ,\ (A\ B\ C)>$,
$e_6=<(\ (A)\ \ (B)\ \ (C)\ \ (D)\ )\ ,\ (A\ B\ C\ D)>$}

Fragments: $f_1[x]=nil$

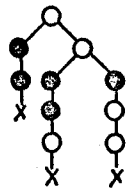
$f_2[x]=nil$

$f_3[x]=car[x]$

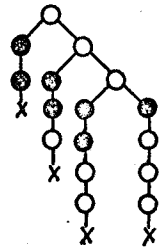
$f_4[x]=$



$f_5[x]=$



$f_6[x]=$



Functional Recurrence Relation: $f_1[x]=nil$
$f_2[x]=nil$
$f_3[x]=car[x]$
$f_{i+1}[x]=cons[caar[x];f_i[cdr[x]]]\ \ i\geq 3$

Predicates: Since the domain of pack is sufficiently different from those of the previous examples we include the chain in SF of the projections of the inputs of E.

proj[input[e_i]]: ω
$(\omega\ .\ \omega)$
$((\omega.\omega).\omega)$
$((\omega.\omega).((\omega.\omega).\omega))$
$((\omega.\omega).((\omega.\omega).((\omega.\omega).\omega)))$
$((\omega.\omega).((\omega.\omega).((\omega.\omega).((\omega.\omega).\omega))))$

yielding the predicates

$p_1[x]=atom[x]$
$p_2[x]=atom[car[x]]$
$p_3[x]=atom[cdr[x]]$
$p_4[x]=atom[cddr[x]]$
$p_5[x]=atom[cdddr[x]]$

Predicate Recurrence Relation: $p_1[x]=atom[x]$
$p_2[x]=atom[car[x]]$
$p_3[x]=atom[cdr[x]]$

$p_{i+1}[x]=p_i[cdr[x]]\ \ i\geq 3$

Program Generated:

```
pack[x]←[atom[x]→nil;
         atom[car[x]]→nil;
         T→p[x]]
p[x]←[atom[cdr[x]]→car[x];
         T→cons[caar[x];p[cdr[x]]]]
```

4.9 *Predicate to test if list is of odd length -- **oddp:***

To demonstrate the applicability of the synthesis technique to programs that produce something other than a rearrangement of the input, we consider a predicate, **oddp**.

Description: oddp[x]=function that returns T if x is a list of odd length and F otherwise.

Examples: E={ e_1=<( ) , F>,
e_2=<(A) , T>,
e_3=<(A B) , F>,
e_4=<(A B C) , T>,
e_5=<(A B C D) , F>}

Fragments: $f_1[x]=F$
$f_2[x]=T$
$f_3[x]=F$
$f_4[x]=T$
$f_5[x]=F$

Functional Recurrence Relation: $f_1[x]=F$
$f_2[x]=T$
$f_{i+2}[x]=f_i[x]\ \ i\geq 1$

Predicates: $p_1[x]=atom[x]$
$p_2[x]=atom[cdr[x]]$
$p_3[x]=atom[cddr[x]]$
$p_4[x]=atom[cdddr[x]]$

Predicate Recurrence Relation: $p_1=atom[x]$
$p_{i+1}[x]=p_i[cdr[x]]$
therefore $p_{i+2}[x]=p_i[cddr[x]]$ (see §4.4)

Program Generated:

```
oddp[x]←op[x;x]
op[x;y]←[atom[y]→F;
         atom[cdr[y]]→T;
         T→op[x;cddr[y]]
```

Note that Corollary C2 generates an unneeded variable resulting in a more complex expression for **oddp** than need be. Postulating a post processing optimizer would remove this inelegance.

4.10 ***pairs:***

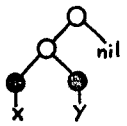
Up to now all functions have been monadic. Although the theoretical justification for polyadic function synthesis is not now complete it is instructive to work through an example of a diadic function to see how the techniques of the previous chapter might be extended.

Description: The function pairs[x;y] returns a list of dotted pairs of the elements of the two lists x and y that must be of equal length.

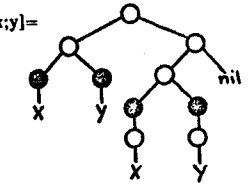
Examples: E={ e_1=<( ) x ( ) , ( )>,
e_2=<(A) x (1) , ((A . 1))>,
e_3=<(A B) x (1 2) , ((A . 1) (B . 2))>
e_4=<(A B C) x (1 2 3) , ((A . 1) (B . 2) (C . 3))> }

Fragments: f_1[x;y]=nil

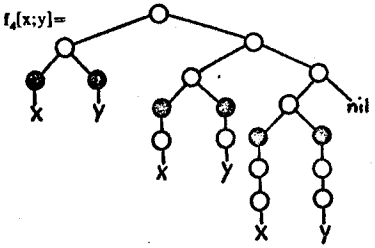
f_2[x;y]=



f_3[x;y]=



f_4[x;y]=



Note that fragment generation is already defined for polyadic functions.

Functional Recurrence Relation: f_1[x;y]=nil
f_{i+1}[x;y]=cons[cons[car[x];car[y]];f_i[cdr[x];cdr[y]]

Predicates: As was noted in the previous chapter the difficulty in synthesizing polyadic functions is in the analysis of the functions domain. In the current example, the analysis should take place in terms of the projection of the example inputs into the lattice SF×SF (the cartesian product of the lattice SF with itself). One way of interpreting this is to derive predicates for each projection and then conjoin them together to get the predicate for the program.

p_1[x;y]=and[atom[x];atom[y]]
p_2[x;y]=and[atom[edr[x]];atom[edr[y]]]
p_3[x;y]=and[atom[cddr[x]];atom[cddr[y]]]

Note that such an assumption about the predicates leaves the function undefined for any pair of arguments x and y that are lists of different length.

Predicate Recurrence Relation: p_1[x;y]=and[atom[x];atom[y]]
p_{i+1}[x;y]=p_i[edr[x];cdr[y]] i≥1

Program Generated: To generate a program for the two recurrence relations we assume a generalization of the basic synthesis program, to two variables. The resulting program is

```
pairs[x;y]←[and[atom[x];atom[y]]→nil;
           T→cons[cons[car[x];car[y]];pairs[cdr[x];cdr[y]]]]
```

4.11 *Comparison to an Existing System:*

The preceding ten examples are in no way an exhaustive list of the list of the programs that may be synthesized by the system. Other programs have been generated but because of structural similarity to some of the above examples were not included in the listing. The only other system, described in the literature, that claims similar capabilities is described in a report by [Green *et al.* 1974] and [Shaw *et al.* 1974]. Their system also generates LISP programs from examples. The Green report lists sixteen programs generated by their system. Of these programs THESYS generated eleven. It failed to generate those programs that required some kind of double iteration. For example, THESYS failed to generate a program that would form a list of all the tails of a given list, *eg.* e=<(A B C D),(A B C D B C D C D D)>. Two iterative steps are required, the first steps through the original list selecting the beginning position of the next tail to be accumulated. The second iteration steps through this tail accumulating its members in the result. Note that this procedure may be reduced to a single iteration if append is available as a primitive since the second iteration may be replaced by appending the selected tail to the result.

CHAPTER 5

CONCLUSION

A number of conclusions may be drawn from the preceding chapters. First, we have the key result of the thesis: it is possible, in a systematic and theoretically justifiable way, to generate a certain class of LISP programs from examples of what they do. Second, we demonstrate an approach that is to a large degree algorithmic, *ie.* it does not rely, for the most part, on heuristic searches or other enumerations that often characterize programs using inductive inference. This means that we have demonstrated not only the existence of a solution but also an effective way of arriving at the solution. A third conclusion concerns approaches to computer science research. It is often easy to demonstrate the existence of something by writing a program to do it. Some areas of computer science, notably artificial intelligence, have, in the past, produced numerous existence proofs. The problem with such 'proofs' is that they seldom contain sufficient analysis to precisely define what is being proved to exist. These existence proofs have often been useful in delineating what could or could not be done with a computer but because they lacked analytic study of the problem being solved, contributed little to the theory of computer science.

This thesis initially approached the problem of program generation from example as an exercise in writing such an 'existence proof'. Once this had been done, *ie.* a paradigm procedure designed, the work shifted to that of trying to define what had been shown to exist. It was the result of this second stage of the thesis work that led to the formal characterization presented in Chapter 3. It is a personal opinion that computer science research that leads to 'existence proofs' is no longer sufficient. It is imperative that such work be carried through an analysis phase in order to understand and codify the computer science concepts that are inherent in the problem being investigated.

More important than the key result of the thesis are the mechanisms of Chapter 3: program fragments, lattice characterization of the data domain, recurrence relations and the variable addition heuristic. Each of these singly or taken together may be considered as identifiable concepts underlying the problem of recursive program synthesis. Extending these concepts may lead to results in the theory of programming. As an indication as to how this work might continue we close this chapter by presenting a number of topics for future work.

5.1 *Addition of power to the system:*

The simplest extension of the system is that of including the ability to generate polyadic functions. The difficulty in doing this was in arriving at a useful characterization of the data domain of an n-ary function. The lattice SF provided a natural characterization for monadic

functions, permitting straight forward derivation of the control flow predicates for the function. When the function has more than one argument the process of predicate determination is less obvious.

The example in §4.10 hinted at a way to extend the class of programs generated from monadic to polyadic functions. The use of the Cartesian product of the lattice, SF, with itself n times for an n-ary function is probably the right model for the data domain of the function. It should be noted that the Cartesian product of a lattices is itself a lattice so that this model for a data domain has at least some of the properties that were found useful in SF in deriving the monadic predicates. One could imagine that a notion similar to 'best approximation' could be extended to the product lattices to describe those functions for which the recursion depends upon a single variable. In the extension to polyadic function indicated in §4.10 logical connectives were introduced. It should be noted that their introduction does not violate the original assumption about the language in that all logical connectives may be defined in terms of the McCarthy conditional [McCarthy 1960].

A harder problem is extending the control scheme to more general forms of recursion. The most obvious problem would be the formulation of a procedure for certain double recursive programs, *eg.* the function which converts all atoms in any S-expression to a list containing that atom. Consider for a moment what such a calculation means in terms of the lattice SF. In the single recursive case, Chapter 3, a point in SF had an associated computation that was determined by either 1) the examples, if the point was on the chain of points inferred from the examples, or 2) the computation associated with the best approximation to the point if the point was not in the chain containing the example's input. In effect all of the points of the lattice were projected onto a chain of points derived from the example's input by the process of finding the best approximation. For the proposed double recursive example, each point in the lattice SF must have its own unique computation.

To achieve a uniqueness of computations for all of the points of the lattice SF we propose a scheme that decomposes the lattice SF into two chains whose Cartesian product is SF. We decompose SF into chains since we require a chain for the ordinary synthesis of programs. The synthesis proceeds using examples on each chain. Two programs are generated, one for each chain. The programs are combined into a single program that is double recursive. This sketch indicates a possible way of getting to the double recursive class of programs. Whether this is a general methodology or is only applicable to a few cases is an open problem.

The final method for enlarging the class of programs generated by the system is that of applying the variable addition heuristic multiple times in the course of a synthesis. Each application would correspond to the introduction of a new local variable. The problem here is to define and organize the heuristic needed to control the multiple applications of the heuristic.Theoretical-

ly, there is the question of how many additional programs may be synthesized with each additional application of the heuristic.

5.2 *Theoretical Questions:*

The following are open theoretical problems related to program generation from examples. Given the procedure presented in Chapter 3, is there an automaton that will recognize the class of programs generated by this procedure? This is a more formal way of stating the open problem of determining what class of functions is generated by the system or alternatively the class of functions that may be characterized by examples. Lacking a complete characterization of this class of programs it would even be nice to have some meaningful upper and lower bounds on the complexity of functions in this class.

Related to the question of the class of programs generated is the notion of a computational hierarchy defined by which components of the system are needed in the synthesis of a program. For example, a function, f, would be more complex than a function, g, if to synthesize f the variable addition heuristic was needed while g could be synthesized without it. At the bottom of such a hierarchy would be the non-recursive programs. Those programs that could be synthesized with only the basic synthesis theorem (T5) might form the next level of the hierarchy. Each level of the hierarchy needs rigorous definition. In addition the question of how to prove the necessity of application of a technique is open. For example, we conjecture but are unable to prove that under the rules of synthesis of Chapter 3 (the function is linear recursive and conservative) that the function reverse does not have an implementation as a program with one variable, *ie.* one must use the variable addition heuristic.

5.3 *Other Application:*

Two of the techniques, recurrence relations and the variable addition heuristic, developed for the system of this thesis show promise of being useful in two other areas of automated programming. Functional and predicate recurrence relations are a concise way of characterizing a recursive program. Some of the preliminary work with this notation has shown it to have some nice properties as a language for proving properties of recursive programs. The formulation of a set of rules of inference is open but it is felt that such a proof procedure would derive its advantage from the use of the recurrence index as a specific indicator of recursion. Proof of properties of recursive programs rely on application of mathematical induction to some facet of the program's execution. Use of the recurrence relations isolate the application of induction to the recurrence index.

Program synthesis systems are very similar to program proving and optimization systems and it is believed that some of the techniques of this thesis may find use in some of these other areas. In particular, it is possible to formulate a method for some kinds of recursion removal in terms of the

algorithms of Chapter 3. Assume we have a recursive function that has a recurrence relation that indicates the need for a stack, *eg.* $f_{k+1}[x]=a[f_k[b[x]];x]$. We first symbolically execute the program to create a set of computations that correspond to the fragments of Chapter 3. Next apply equivalence transforms based on a property of a binary operator, *eg.* associativity. Try to find a new recurrence relation for the transformed set of fragments under the restriction that the relation be of the form $f_{k+1}[x_i;\ldots;x_j]=f_k[a_i[x_i];\ldots;a_j[x_j]]$. Such a recurrence relation has an immediate implementation as a non-recursive program.

APPENDIX 1

THESYS User Interface

This appendix will present a specification for the user/machine interface to the THESYS system for constructing programs from examples. Since the system generates LISP recursive programs, examples of the program's input and output are written as LISP S-expressions. An S-expression is either an atom, a dotted pair of S-expressions or a list expression. A dotted pair is written as a left parenthesis (( ) followed by an S-expression followed by a period (.), followed by a second S-expression followed by a right parenthesis ( )). A list expression is written as a left parenthesis followed by any number of S-expressions, separated by blanks, followed by a right parenthesis. A more formal definition of these entities is found in [McCarthy *et al.* 1962].

THESYS is written in LISP and runs on an IBM 370/168 under VM. After loading the host LISP system THESYS is invoked by evaluating the expression `(thesys)`. Lower face courier type face will denote the user input to the system and upper case will be used to denote the system's response. The system responds with an introductory message

```
(THIS IS A SYSTEM TO SYNTHESIZE PROGRAMS FROM EXAMPLES)
(ALL ENTITIES SHOULD BE ENTERED AS S-EXPRESSIONS)
(INFORMATION MAY BE GOTTEN FROM THE SYSTEM BY TYPING ?)
(ENTER NAME OF FUNCTION TO BE SYNTHESIZED)
```

A response by the user ? at this or any other point will cause the system to print out a instructions as to the appropriate use of the system. A user response of a name will cause the system to continue with the request

```
(ENTER EXAMPLE)
```

An example may now be entered by the user. An example consists of one or more example inputs, written as S-expressions and separated by blanks, followed by a right arrow (->) followed by the example output written as an S-expression. Examples may be continued over more than one line. With the first example the arity of the program is determined and remembered by the system. A check is made on the all remaining inputs to see if they conform to this amount. The system will continue to request examples until a dollar sign ($) is entered at which time it attempts to generate a recursive program. If the system is successful it will print out the program and ask if the user wishes to specify any further functions. An answer of yes restarts the system, otherwise execution of THESYS is terminated. If the system is unable to generate a recursive program it responds with the message:

```
(UNABLE TO SYNTHESIZE A RECURSIVE PROGRAM  WITH THE
EXAMPLES GIVEN)
```



```
(IF YOU WISH TO ENTER MORE EXAMPLES, TYPE A Y)
```

If more examples are to be entered the user types `y`, otherwise the default program is generated.

APPENDIX 2

Computational Completeness of Basic Lisp without *eq*

This appendix contains a proof of the computational completeness of the LISP language used as a target language in the thesis. The proof consists of a procedure for writing a set of LISP functions that will simulate any Turing machine and thus the universal Turing machine. The Turing machine to be simulated is able to change state and either move its tape right or left or write a new symbol on the square being scanned [Nelson 1968].

Tape: The tape will consist of a dotted pair of two lists representing the portion of the tape that is to the left of the head and the portion of the tape that is under and to the right of the head. Each element of the tape alphabet is encoded as a lists of a specific depth. For example, if $A=\{A_1,A_2,\ldots,A_k\}$ is the tape alphabet then the encoding for the simulation would be $A_1=$ (( )), $A_2=$ ((( ))), ... and a blank tape symbol will be represented as ( ).

Finite state portion of the machine: The finite state portion of the Turing machine is simulated by the one argument function tm[tape], where tape is an encoding of the starting configuration of the tape. The function returns a value that may be interpreted as the final state of the tape if the machine and hence the simulation is defined. Each state, Si, of the finite state portion of the machine is encoded as a function, tm_{Si}[tape].

tm[tape]←tm_{S0}[tape] where S0 is the starting state.

tm_{Si}[tape]←[is-sym[tape;A_1]→tm_{Sj}[ϕ[tape]];
is-sym[tape;A_2]→tm_{Sk}[ϕ[tape]]
...
T→tm_{Si}[ϕ[tape]]]

Each conditional should be interpreted as one entry in the state table for the finite state portion of the Turing machine. For example, is-sym[tape;A_i]→tm_{Sj}[ϕ[tape]] in the function tm_{Si} would represent the table entry for the machine in state Si and reading symbol A_i from the tape. The action for this entry would be to go to state Sj and perform the action ϕ where ϕ would be either to write a symbol on the tape or to move the tape right or left one square.

tm_{HALT}[tape]←tape

is-sym[tape;symbol]←same-sym[cadr[tape];symbol]

Note: cadr[tape] is the current symbol being scanned.

same-sym[a1;a2]←[atom[a1]→atom[a2];

atom[a2]→nil;
T→same-sym[car[a1];car[a2]]]

ϕ[tape] is either move-tape-rt[tape], move-tape-lft[tape] or write[tape;symbol]

move-tape-rt[tape]←[atom[car[tape]]→cons[car[tape];cons[nil;cdr[tape]]];
T→cons[cdar[tape];cons[caar[tape];cdr[tape]]]]

move-tape-lft[tape]←[atom[cddr[tape]]→
move-tape-lft[cons[car[tape];cons[cadr[tape];cons[nil;nil]]]];
T→cons[cons[cadr[tape];car[tape]];cddr[tape]]]

write[tape;symbol]←[atom[cdr[tape]]→cons[car[tape];cons[symbol;cdr[tape]]];
T→cons[car[tape];cons[symbol;cddr[tape]]]]

BIBLIOGRAPHY

ALLEN, F. E. Bibliography on program optimization. To appear as an IBM Research Report, IBM Research, Yorktown Hts., N.Y., 1975.

ALLEN, F. E. AND COCKE, J. A catalog of optimizing transformations. In **Design and Optimization of Compilers**, R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N. J., 1972, pp.1-30.

AMAREL, S. Representations and modeling in problems of program formation. In **Machine Intelligence 6**, B. Meltzer and D. Michie, Eds., American Elsevier Publ. Co., Inc. New York, 1971, pp. 411-466.

BACKUS, J. W. The IBM 701 speedcoding system. **J. ACM 1**(1954), 4-6.

BACKUS, J. W. ET. AL. The Fortran automatic coding system. Proc. of the Western Joint Computer Conf., Los Angeles, 1957, pp.188-198.

BALTZER, ROBERT. Automatic programming, technical memo. Information Sciences Institute, Univ. of Southern Calf., Sept. 1972.

BALTZER, ROBERT. Human use of world knowledge. Rep. ISI/RR-73-7, Information Sciences Institute, Univ. of Southern Calf., Marina del Rey, 1974.

BIERMANN, A. W. On the inference of Turing machines from sample computations. Rep. CS-241, Computer Science Dept., Stanford Univ., 1971.

BIERMANN, A. W. AND FELDMAN, J. A. A survey of results in grammatical inference. In **Frontiers of Pattern Recognition**, Watanabe, Ed., Academic Press, New York, 1972, pp. 31-54.

BIRKOFF, GARRETT. **Lattice Theory (Third Edition)**. American Mathematical Society, Providence, 1967.

BLACK, FISCHER. Styles of programming in LISP. In **The Programming Language LISP: Its Operation and Applications**, Berkeley and Bobrow, Eds., The MIT Press, Cambridge, 1964, pp. 96-107.

BLUM, L. AND BLUM, M. Inductive inference: a recursion theoretic approach. Proc. of 14th Annual Symp. on Switching and Automata Theory, IEEE Computer Society, 1973, pp. 200-208.

BOYER, R. S. AND MOORE, J. S. Proving theorems about LISP programs. **J. ACM 22**(1975), 129-144.

CADIOU, J. AND MANNA, Z. Recursive definitions of partial funcitons and their computations. Proc. of an ACM Conf. on Proving Assertions about Programs, **SIGPLAN Notices 7**(1972), 58-65.

CHEATHAM, T. E. AND WEGBREIT, B. On a laboratory for the study of automating programming. Proc. of an ACM Conf. on Proving Assertions about Programs, **SIGPLAN Notices 7**(1972), 208-211.

CHURCH, ALONZO. **The Calculi of Lambda-Conversion.** Princeton Univ. Press, Princeton, N. J., 1941.

DARLINGTON, JARED. Automatic program synthesis in second order logic. Proc. of Third International Conf. on Artificial Intelligence, 1973, pp.537-542.

DARLINGTON, JOHN AND BURSTALL, R. M. A system which automatically improves programs. Proc. of Third International Joint Conf. on Artificial Intelligence, Palo Alto, Calf., 1973, pp.486-493.

DAVIS, MARTIN AND PUTNAM, HILARY. A computing procedure for quantification theory. **J. ACM 7**(1960), 201-215.

FALKOFF, A. D. AND IVERSON, K. E. **APL/360 User's Manual.** IBM Corp., 1968.

FIKES, R. E. AND NILSSON, N. J. STRIPS: A new approach to the application of theorem proving to problem solving. **Artificial Intelligence 2**(1971), 189-208.

FRIEDKIN, EDWARD. Techniques using LISP for automatically discovering interesting relations in data. In **The Programming Language LISP: Its Operation and Applications**, Berkeley and Bobrow,Eds., The MIT Press, Cambridge, 1964, pp. 108-124.

FU, K. S. Grammatical inference for syntactic pattern recognition. In **Syntactic Methods in Pattern Recognition**, Academic Press, New York, 1974, pp. 193-229.

GERHART, S. L. Verification of APL programs. PhD Thesis, Carnegie-Mellon Univ., Pittsburgh, Pa., 1972.

GLENNIE, A. E. Future trends in automatic programming. In **Automatic Programming 1**, Goodman, Ed., Pergamon Press, New York, 1960, pp. 8-15.

GOLDBERG, PATRICIA. Automatic programming. Report RC-5148, IBM Thomas J. Watson Research Center, Yorktown Heights, N. Y., 1974.

GREEN, C. Theorem-proving by resolution as a basis for question-answering systems. In **Machine Intelligence 4**, Melcher and Michie, Eds., American Elsevier Pub. Co., Inc., New York, 1969, pp. 183-205.

GREEN, C. C. ET. AL. Progress report on program understanding systems. Rep. STAN-CS-74-444, Computer Science Dept., Stanford Univ., 1974.

HEIDORN, GEORGE E. Natural language inputs to a simulation programming system. Rep. NPS-55HD72101A, Naval Postgraduate School, Monterey, Calf., 1972.

HEIDORN, GEORGE. Automatic programming through natural language dialogue. To appear as an IBM Research report, IBM Research, Yorktown Hts., NY, 1975.

HOARE, C. A. R. An axiomatic basis of computer programming. **C. ACM 12**(1969), 576-580, 583.

HOPPER, G. M. The education of a computer. Proc. of the Conf. of the ACM, Pittsburgh, 1952.

HOWE, W. G. ET. AL. A new approach for customizing business applications. To appear as an IBM Research report, IBM Research, Yorktown Hts., NY, 1975.

KATZ, S. AND MANNA, Z. Logical analysis of programs. To appear in **C. ACM**(1975).

KING, J. C. A program verifier. PhD Thesis, Carnegie-Mellon Univ., Pittsburgh, Pa., 1969.

KLEENE, S. C. **Introduction to Meta-mathematics.** D. Van Nostrand, Inc., Princeton, N. J.,

1952.

McCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, part I. **C. ACM** 3(1960), 184-195.

McCARTHY, J. A basis for a mathematical theory of computation. In **Computer Programming and Formal Systems**, P. Braffort and D. Hirschberg, Eds. North-Holland, Amsterdam, 1963a, pp. 33-70.

McCARTHY, J. Towards a mathematical science of computation. In **Information Processing: Proceedings of IFIP 1962**, Popplewell, Ed., North Holland Pub. Co., Amsterdam,1963b, pp. 21-28.

McCARTHY, J. ET. AL. **LISP 1.5 Programmer's Manual**. MIT Press, Cambridge, Mass., 1962.

McDERMOTT, D. J. AND SUSSMAN, G. J. The CONNIVER reference manual. Report No. 259, Artificial Intelligence Lab., MIT, Cambridge, Mass., 1972.

MANNA, Z. **Mathematical Theory of Computation**. McGraw-Hill Book Co., New York, 1974.

MANNA, ZOHAR AND WALDINGER, R. J. Towards automatic program synthesis. **C. ACM 14**(1971), 151-165.

MANNA, ZOHAR, ET. AL. Inductive methods for proving properties of programs. **C. ACM 16**(1973), 491-502.

MANNA, Z. AND VUILLEMIN, J. Fixpoint approach to the theory of computation. **C. ACM 15**(1972), 528-536.

MARTIN, W. A. OWL, a system for building expert problem solving systems involving verbal reasoning. Course 6.871 notes, MIT, Cambridge, Mass., spring 1974.

MIKELSONS, MARTIN. Computer assisted application definition. Proc. Second ACM Symp. on Principles of Programming Languages, Palo Alto, 1975, pp. 233-242.

MOORE, J. STRUTHER. Introducting iteration into the pure LISP theorem prover. **IEEE Trans. on Software Engineering SE-1**(1975), 328-338.

MILLER, L. A. AND BECKER, C. A. Programming in natural English. Rep. RC 5137, IBM Research, Yorktown Hts., NY, 1974.

NELSON, R. J. **Introduction to Automata**. John Wiley and Sons, Inc., New York, 1968.

PERLIS, ALAN J. Steps toward an APL compiler. Rep. No. 24, Dept. of Comp. Sci., Yale Univ., New Haven, Conn., 1974.

PERLIS, A. J. AND SMITH, J. W. A mathematical language compiler. Automatic Coding Monograph No. 3., **J. of the Franklin Institute** (1957), 87-102.

PETRICK, STANLEY R. On natural language based query systems. To appear in **IBM J. of Research and Development**.

PIVAR, MALCOM AND FINKELSTEIN, MARK. Automation, using LISP, of inductive inference on sequences. In **The Programming Language LISP: Its Operation and Applications**, Berkeley and Bobrow, Eds., The MIT Press, Cambridge, 1964, pp. 125-136.

PLOTKIN, G. D. A note on inductive generalization. In **Machine Intelligence 5**, B. Meltzer and D. Michie, Eds., American Elsevier Publ. Co., Inc., New York, 1970, pp. 153-163.

POPPLESTONE, R. J. An experiment in automatic induction. In **Machine Intelligence 5**, B. Meltzer and D. Michie, Eds., American Elsevier Publ. Co., Inc., New York, 1970, pp. 203-218.

ROBINSON, J. A. A machine-oriented logic based on the resolution principle. **J. ACM 12**(1965), 23-41.

ROBINSON, J. A. Computational logic: the unification computation. In **Machine Intelligence 6**, B. Meltzer and D. Michie, Eds., American Elsevier Publ. Co., Inc., New York, 1971, pp. 63-72.

SAMMET, J. E. The use of English as a programming language. **C. ACM 9**(1966), 228-230.

SCHANK, ROGER C. Identification of conceptualizations underlying natural language. In **Computer Models of Thought and Language**, Schank and Colby, Eds., W. H. Freeman and Company, San Fransisco, 1973, pp. 187-248.

SCOTT, D. Outline of a mathematic theory of computation. In **4th Annual Princeton Conf. on Information Sciences and Systems**, Princeton, N. J., 1970, pp. 169-176.

SHAW, D. E. ET. AL. Inferring LISP programs from examples. Private communication, 1974.

SIBERT, E. E. A machine-oriented logic incorporating the equality relation. In **Machine Intelligence 4**, B. Meltzer and D. Michie, Eds., American Elsevier Publ. Co., Inc. New York, 1969, pp. 103-134.

SIKLOSSY, L. The synthesis of programs from their properties, and the insane heuristic. Proc. of the Third Texas Conf. on Computer Systems, Austin, Texas, 1974.

SITES, R. L. Proving that computer programs terminate cleanly. PhD thesis, Stanford Univ., Stanford, Calf., 1974.

SKOLEM, THORALF. On mathematical logic (Text of a lecture to the Norwegian Mathematical Assoc., 1928). In **From Frege to Godel**, J. Heijenoort, Ed., Harvard Univ. Press, Cambridge, Mass., 1967, pp. 508-524.

STRONG, H. R. Translating recursion equations into flow charts. **J. of Computer and System Sciences 5**(1971), 254-285.

SUSSMAN, G. J. A computational model of skill acquisition. PhD Thesis, MIT, Cambridge, Mass., 1973.

SUSSMAN, G. J. ET. AL. Micro-PLANNER reference manual. Report No. 255a, Artificial Intelligence Lab., MIT, Cambridge, Mass., 1972.

WALDINGER, RICHARD J. Constructing programs automatically using theorem proving. PhD thesis, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1969.

WALDINGER, R. J. AND LEE, R. C. T. PROW: A step toward automatic program writing. Proc. of International Joint Conference of Artificial Intelligence, 1969, pp. 241-252.

WELLS, M. B. AND MORRIS, J. B., EDS. Proc. of a Symp. on Two-Dimensional Man-Machine Communication. **SIGPLAN Notices 7**(1972).

WINOGRAD, TERRY. **Understanding Natural Language**. Academic Press, New York, 1972.

WINOGRAD, TERRY. Breaking the complexity barrier again. Proc. of ACM SIGPLAN-SIGIR Interface meeting, Programming Languages - Information Retrieval, Gaitersburg, 1973. **SIGPLAN Notices** 10(1975), 13-30.