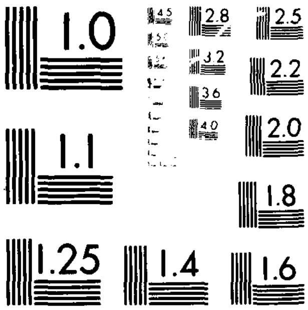
AD-A110 115   MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTE--ETC F/G 9/2
AN ACCOUNTABLE SOURCE-TO-SOURCE TRANSFORMATION SYSTEM.(U)
JUN 81 B S STEELE   N00014-80-C-0505

UNCLASSIFIED AI-TR-636   NL

1 OF 2
AD
A110115



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

AI-TR-636

AN ACCOUNTABLE SOURCE-TO-SOURCE TRANSFORMATION SYSTEM

BARBARA SUE KERNS STEELE

June 1981

ARTIFICIAL INTELLIGENCE LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR-636	2. GOVT ACCESSION NO. AD-A110 116	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* An Accountable Source-to-Source Transformation System		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Barbara Sue Kerns Steele		8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0505
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory. 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE June 1981 13. NUMBER OF PAGES 99
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Transformation System	Symbolic Execution
Code Optimization	Programmers Assistant
Interactive Programming	Program Analysis
	Snapshots

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Though one is led to believe that program transformation systems which perform source-to-source transformations enable the user to understand and appreciate the resulting source program, this is not always the case. Transformations are capable of behaving and/or interacting in unexpected ways. The user who is interested in understanding the whats, whys, wheres, and hows of the transformation process is left without tools for discovering them. I provide an initial step towards the solution of this problem in the form of an accountable

20.
source-to-source transformation system. It carefully records the information necessary to answer such questions, and provides mechanisms for the retrieval of this information. It is observed that though this accountable system allows the user access to relevant facts from which he may draw conclusions, further study is necessary to make the system capable of analysing these facts itself.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505, and in part by National Science Foundation grant MCS-7912179.

The views and conclusions contained in this paper are those of the author, and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Department of Defense, the National Science Foundation, or the United States Government.

An Accountable Source-to-Source Transformation System

by

Barbara Sue Kerns Steele

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUNE 1981

Revised version of a dissertation submitted on July 25, 1980 to the Department of Electrical Engineering and Computer Science of the Massachusetts Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science.

ABSTRACT

Though one is led to believe that program transformation systems which perform *source-to-source* transformations enable the user to understand and appreciate the resulting source program, this is not always the case. Transformations are capable of behaving and/or interacting in unexpected ways. The user who is interested in understanding the whats, whys, wheres, and hows of the transformation process is left without tools for discovering them.

I provide an initial step towards the solution of this problem in the form of an *accountable* source-to-source transformation system. It carefully records the information necessary to answer such questions, and provides mechanisms for the retrieval of this information. It is observed that though this accountable system allows the user access to relevant facts from which he may draw conclusions, further study is necessary to make the system capable of analyzing these facts itself.

Thesis Supervisor: Howard E. Shrobe

Acknowledgements

My sincere thanks to:

Dr. Howard E. Shrobe, my thesis advisor, who was always willing to assure me that I was progressing well.

Professor Gerald J. Sussman, who first suggested I develop an accountable system.

Cordon R. Kerns, my father, who taught me more mathematics and science than any teacher I ever had, and then made me teach *him* math when I got beyond what he knew.

Professor Galen R. Peters, who worked hard to provide me with the education I wanted, and pointed me to the right places to get it.

Kent M. Pitman, who read an initial draft of the dissertation and offered numerous suggestions for its improvement.

James B. Tate, who introduced me to computer science.

Robert G. Jacobsen, who introduced me to LISP.

Dr. James M. Boyle, who introduced me to program transformations.

Ruth J. Kerns, my mother, who managed to keep me a reasonably well-rounded individual despite my growing interest in the world of technology.

The Reverend Doctor Guy L. Steele Sr. and Nalora Steele, my other parents, who followed my progress with interest, and shared both my frustrations and excitement over various developments in the work.

And finally, to Guy L. Steele Jr., my husband, who during the past year dropped anything and everything without fail and without hesitation whenever I needed advice. He patiently answered hundreds of questions and engaged in dozens of conversations with me about my work, though he was in the midst of writing his own dissertation. I owe him much; happily, I have the rest of our lifetime together in which to repay him.

Contents

Figures

Tables

ac-count-a-ble adj. 1 : subject to giving an account
2 : capable of being accounted for

ac-count vb. vt 1 : to give a report on
vi 1 : to furnish a justifying analysis or explanation – used with for

—Webster's Seventh New Collegiate Dictionary

We might suppose that by the combined action of appropriate forces any material form could be transformed into any other: just as out of a 'shapeless' mass of clay the potter or the sculptor models his artistic product; or just as we attribute to Nature herself the power to effect the gradual and successive transformation of the simple germ into the complex organism. ... So the living and the dead, things animate and inanimate, we dwellers in the world and this world in which we dwell ... are bound alike by physical and mathematical law.

—D'Arcy Thompson

Chapter One

Source-To-Source Transformation Systems

PROGRAM TRANSFORMATION SYSTEMS have been produced which claim to ameliorate the problems of inefficient programs caused by modular programming strategies, initial lack of attention to efficiency (for indeed, during the programming process, construction and verification should come first), or perhaps just bad programming style. These transformation systems may be implemented automatically or manually, and may do their work at compile time or before. For example, an optimizing compiler is an automatic transformation system which performs its work at compile time. In contrast, a source-to-source transformation system is an automatic pre-processor whose output may then be compiled on the desired machine. Alternatively, the programmer himself may be responsible for producing efficient code, and, following the construction and verification of his program, must rewrite any part which does not meet its operational standards. In this chapter I offer my opinions and suggestions regarding the current state of program transformation.

1.1. Introduction

Program transformation should be a process which is independent of compilation, and which, though performed automatically, can be guided by the programmer. A source-to-source transformation system provides a programmer with canned sets of transformations which he may invoke and afterwards observe the effect of. I am interested in a more interactive approach in which the programmer is able to intelligently select those transformations which are relevant and effective for the particular situation he presents. Furthermore, the programmer may designate only certain parts of the program for transformation (for example, he may choose to apply optimizing transformations to code which forms a bottleneck at execution time). In order to intelligently make these decisions, the programmer must have the freedom to experiment with various sets of transformations to his program, be able to observe their outcomes, and verify that indeed they do the job he requires. The advantages of this approach are discussed in [Loveman 1977].

Although source-to-source transformation systems allow programmers to perform an analysis of the results, they provide no mechanisms for such analyses; the programmer must study the output himself to discover what transformations, if any, occurred. Furthermore, the effort needed to understand why a transformation applied (or more often, why one did not) is significant and prohibitive. Without mechanisms to aid the programmer in his study of the result of the transformation process, many of the advantages of a source-to-source transformation system are lost.

Since I had to develop a source-to-source transformation system one summer without the aid of any sort of mechanism for evaluating the effectiveness of transformations, I have a great many suggestions to offer as to what sorts of tools would be valuable for doing so. It was often necessary first to learn if there had been any change in the argument program at all, that is, if any transformation had succeeded in applying. A transformation system which would report whether or not it has done anything would be capable of providing this information. Then, once it had been determined that something had happened, it was necessary to discover *what*. Then *where*, then *when*, then *why*. And there was always *why not?* I wished I had a transformation system which would automatically account for its decisions: an *accountable source-to-source transformation system.*

1.1.1. Statement of Thesis

- Source-to-source transformation systems which produce changes in the user's program without any explanation or justification of these changes in some sense "violate" the user's code. To verify that the transformations performed were those expected or desired, the user must manually examine the resultant code. Furthermore, understanding why such transformations were or were not performed requires manual examination of the transformation system itself.
- A source-to-source transformation system which can be queried as to what, where, when, and why it did whatever it did can aid the interactive transformation of a program.
- The development of such a tool is feasible.

1.1.2. Organization of Dissertation

So far in this chapter I have given an overview of the problem I wish to address, and only a hint of the proposed solution. In the remainder of this chapter I will discuss the uses for and advantages of source-to-source transformation systems, then go over again in some detail (and with examples) why many of those advantages are not realized. Chapter Two describes a method for regaining these advantages: making the system accountable for its actions. In that chapter I will define exactly what I mean by an accountable transformation system, then present the design of such a system. Chapter Three begins our discussion of the implementation of the system by introducing the source-to-source transformation system which will be made accountable. Finally, Chapter Four follows the decisions and revisions involved in implementing an accountable source-to-source transformation system. The bulk of the thesis lies in this chapter, since only in attempting to implement an accountable system did many of the subtle difficulties in doing so present themselves.

In this dissertation I am particularly interested in discussing *source-to-source* transformation systems, although I admit that if someone wished to argue the point, I would have a hard time convincing him there was any difference in transformation systems which are source-to-source and those that are not. What is object code to one compiler may be source code to another; however, I will assume the popular definition of source code as a high-level language, and make no claims as to whether or not it is object code as well. In this discussion, then, it will be assumed that when I speak of transformation systems, I mean *source-to-source* transformation systems, unless explicitly stated otherwise.

1.2. Motivation And Uses For Transformation Systems

Transformations allow one to automate almost any kind of systematic change to a program. While some systems are written with a particular transformation set "built-in" because they are meant to peiform a specific task, others are written more modularly and may be used with any of a number of different canned transformation sets. General-purpose transformation systems exist as well; these come with a special language in which the programmer himself can write the transformations he wishes to be applied to his program.

Source-to-source transformation systems provide access to the resultant code in a way that other transformation systems do not. The output from such systems may be studied, modified, or run through the transformation system again before any compilation is done. Thus, whether the programmer uses a set of canned transformations or "rolls his own," he is able to afterwards observe the effects of each on the input code.

Source-to-source transformations may be applied to a program independently of the machine the program is going to be compiled on. Since the system takes source code and produces source code, transforming the program precedes any compiler activity. Thus the programmer has the security of knowing that he can produce a more efficient program for any machine. Of course, certain transformations may be more or less effective depending on the target machine. The programmer who is aware of these differences may tailor his program to the appropriate machine, or to several by generating all of the various versions that are required.

Currently available source-to-source transformation systems are capable of performing optimization, realization, and translation.

1.2.1. Optimization

By far the most common use of transformation systems is for the optimization of code. A number of papers which describe standard optimizing transformations have been published [Standish, et al 1976] [Allen and Cocke 1972], and some successful systems are in use [Kerns 1977] [Boyle and Matz 1977] [Atkinson 1976]. No one will question the need for program optimization, but what are the advantages of performing optimizations by means of a transformation system?

1.2.1.1. *Opens the Way For Interactive Optimization*

Code generated by a source-to-source transformation system may be analyzed by the programmer in order to determine what optimizations were performed. He is therefore able to understand to some extent exactly what transformations to the code are causing any difference in execution time. He can see which parts of the code are being modified, and which are not. On the basis of these observations, he may write or use other transformations, or choose other pieces of code to optimize. Because he has a better understanding of the optimization process, he can therefore better control that process.

1.2.1.2. *Simplifies the Compilation Task*

By moving the optimization process out of the compilation process, a compiler writer is free to concentrate on code generation. In modularizing the entire program development process and allowing compilation to be independent of optimization, we move closer to the possibility of automatic generation of compilers.

As transformation systems evolve, one can imagine canned sets of transformations becoming available which are guaranteed to cure all programming ills. Each set of transformations may be keyed for use with a particular language, depending on the target machine. For example, all Fortran programs to be run on an IBM 370/165 may use transformation set "IEH370FIXIT", while Lisp code to be run on a DEC-10 system will use tranformation set "BAZOLA". Each set would know about the quirks specific to its target machine, and be prepared to put the source code into the form best suited for efficient compilation on that machine.

Or, better yet, let there be only one transformation set for each language. The Fortran set is responsible for optimizing Fortran code, the Lisp set for Lisp code, et cetera. Each set of transformations puts its source code into a form suitable for efficient compilation. That is, there should be a canonical set of transformations that every compiler expects to have already been performed by the time the compiler gets the source code. Since every high-level language is not equally capable of expressing all the possible optimizations of a piece of code, I will instead say: for each high-level language, there should be a canonical set of transformations that every compiler for that language expects to have already been performed. Then, it is the responsibility of the compiler to know the quirks and idiosyncrasies of its machine, and to generate efficient object code for that machine. The compiler may assume that the source code it receives has been optimized into the canonical form agreed upon beforehand.

1.2.2. Realization

Although transformations which perform optimizations to a program have received more attention than those which do not, other uses for transformation systems should not be overlooked. One transformation system currently in use at Argonne National Laboratory [Boyle and Matz 1977] has a library of transformation sets which generate different versions or "realizations" of a single prototype program. For example, given a Fortran program to perform some algorithm upon complex numbers in single precision, it can generate an analogous program which performs the same algorithm on real numbers in double precision. This is done by first applying a set of complex-to-real transformations to the program, and then, using the output of that run as input to the next, applying a set of single-to-double precision transformations.

This method may be used to generate realizations of a program which are geared to run more efficiently (or perhaps correctly!) on a particular machine. The notion of providing portability and reliability by program transformation is discussed in [Boyle 1976]. A particular realization can be thought of as a refinement of the prototype program: transformations may be written to refine high-level programs into underlying representations [Standish, et al 1976].

1.2.3. Translation

Transformations sets may also be written to translate code from one source language to another [Pitman 1979]. These transformations are clearly non-trivial, and the set of programs to which they can successfully apply may be very restricted if the languages are not similar. Syntax transformations, however, should be simple and fairly straightforward.

1.3. Inadequacies of Transformation Systems

Although in theory a source-to-source transformation system provides all the advantages given above, in practice things turn out a little differently. Consider the following example:

A set of optimizing transformations was applied to a simple LISP program to compute two lists and append them.

```
(APPEND (MYMAPCAR (FUNCTION FOO) '(A B C)) (BAZ Z Z))
```

The optimized code returned by the transformations was:

```
(CONS (FUNCALL (FUNCTION FOO) 'A) (MAPCAR (FUNCTION FOO) '(B C)))
```

Understanding why this code replaced the input code takes a bit of study. First off, the programmer would need to remind himself of the definition of MYMAPCAR and BAZ. He goes to his file and looks them up:

```
(DEFUN MYMAPCAR (FUNARG ARG)
       (COND ((NULL ARG) NIL)
             ((ATOM ARG) (FUNCALL FUNARG ARG))
             (T (CONS (FUNCALL FUNARG (CAR ARG))
                      (MAPCAR FUNARG (CDR ARG))))))

(DEFUN BAZ (X Y)
       (COND ((EQ X Y) NIL)
             (T (LIST X Y))))
```

To figure out how these definitions together with the transformations in the set he applied worked together to form the replacement code, his analysis might proceed like this:

"Oh no, I've been QUUXED! What happened to the call to APPEND? It's turned into a call to CONS! How can that be?

"Hmmm. Well, procedure integration has occurred, I guess. The definition of MYMAPCAR has been expanded in-line, but since the second argument is known to be a list, the tests for nullness and atomicity were eliminated, leaving only the CONS. But what happened to my BAZ function? I certainly didn't give it any constants, so what makes it think it can go away? Let's see... If I expand its definition in-line, I get a conditional whose first test is (EQ Z Z). Oh, of course! That must be true, so it returns NIL. Then APPEND of something and NIL leaves that something, which explains why only the CONS is left. Whew!"

That was only a simple example. Imagine trying to verify output from a transformation system in which there are several levels of procedure integration, hairy function definitions, obscure test eliminations, and nested lambda expressions that produce several layers of bindings one can't possibly remember and translate between all at once. If one must go through that sort of analysis for each piece of code in order to follow the activities of a source-to-source transformation system, the advantages of being able to do so lose some of their appeal. The transformations applied to the input program "do violence" to the user's code; it is often difficult or even impossible to understand why a transformation did or did not apply, and how the various transformations interacted to produce the output program.

To understand what occurred during the transformation process requires a great deal of effort on the part of the user. He must compare the input source with the output source to see where changes were made. He must look up function definitions to see how procedure integration took place. To understand which transformations applied, he must have a good idea of the transformations attempted and what their prerequisites for application were. It may be impossible for him to discover what transformations, if any, *almost* applied, and why they didn't. Furthermore, without careful study of the system itself, he has no clue as to the order in which the transformations applied, and how this may have affected the final result.

In case the reader is not convinced, I include one last example (without exposition) for him to ponder over. The original program function..

```
(DEFUN EXAMPLE (FOO BAR BAZ)
       (COND ((ATOM FOO) NIL)
             ((ISINDEXED FOO) (PROCESS-INDEX FOO BAR BAZ))
             (   ...    (GET-PART FOO)    ...   )
             (T     ...      )))
```

The resulting code:

```
(DEFUN EXAMPLE (FOO BAR BAZ)
       (COND ((ATOM FOO) NIL)
             ((AND (ATOM (CAR FOO))
                   (NUMBERP (CAR FOO)))
              (PROCESS-INDEX FOO BAR BAZ))
             (   ...   (CAR FOO)    ...   )
             (T     ...      )))
```

Relevant function definitions:

```
(DEFUN ISINDEXED (X)
       (AND (NOT (ATOM X))
            (ATOM (CAR X))
            (NUMBERP (CAR X))))

(DEFUN GET-PART (Y)
       (COND ((ATOM Y) NIL)
             ((AND (ATOM (CAR Y))
                   (NUMBERP (CAR Y)))
              (CADR Y))
```

```
          (T (CAR Y))))
```

With some study and comparison of the two definitions of EXAMPLE given here, we can eventually see that they are equivalent, and determine what transformations were applied to get the resulting code. However, this can be a tedious process for the entire set of program functions, and one that could be automated. I propose to aid the user in understanding the transformations performed above (for example) by having available to him information about the changes made. The user might ask where the (CAR FOO) in the resulting program came from, and the system would answer that it came from the simplification of (COND (T (CAR FOO))), which was the result of test elimination from the call to GET-PART which was expanded in-line.

Chapter Two

Design of An Accountable System

JUST WHAT IS an "accountable" transformation system? We need to understand what is meant by the term and how the characteristic of accountability might be used to solve the problems presented in the previous chapter. Then we may design and build such a system.

2.1. A Definition

An accountable source-to-source transformation system is one which records in some accessible fashion the circumstances of application of any transformations it performs. That is, not only does such a system return a transformed program, but it leaves behind a history of the transformation process as well. Mechanisms are provided which will allow the user to easily obtain information from this history. Thus, a programmer might well receive direct answers to such questions as:

- "What initial transformation (if any) occurred to start the chain of transformations performed on this section of code?"
- "What did this piece of code look like right before this transformation applied?"
- "What would this piece of code look like if this transformation had been turned off?"
- "Which transformations applied at this point in the program? Why?"
- "What transformations did not apply? Why not?"
- "Which transformations always applied?"

One might wonder if this ability is possible only in a *source-to-source* transformation system, or whether optimizing compilers and other transformation systems could be made accountable as well. The point of an accountable system is that it aids the user in understanding why certain decisions were made, where they were made, and what those decisions were. If we assume that such information is useful, it is natural to then assume that the user is interested in studying the code of the resultant program as well. I wish to make no claims about the amount of interest the typical programmer has in object code, but I will admit that in theory, any transformation system can be made accountable. In practice, however, a source-to-source transformation system seems to be the best candidate for such an improvement.

2.2. Motivation and Uses for an Accountable System

An accountable source-to-source transformation system makes all the advantages of a transformation system listed in Chapter One realistic. The output of an accountable system is not only accessible but manageable, and the transformation process is rendered truly interactive.

2.2.1. Interactive Transformation

As mentioned in the section on source-to-source transformation systems, although such systems provide the programmer with access to the transformed code, for him to understand exactly what took place requires that he invest a significant amount of time and effort. He must not only compare input and output, but have a good understanding of the transformation system itself. By automatically recording and providing access to the information that a programmer needs to decide if the transformation process accomplished all that he desired, we supply yet another tool for the program development process. With the knowledge that such a tool is available, a programmer is more apt to concentrate on construction and verification of a program first, and leave the problem of efficiency until later. When its time comes, the user is able with the use of an accountable system to direct the transformation process. It is the accumulation of such tools which smooths the way for development of automatic program synthesis systems.

2.2.2. Debugging

An accountable system is a debugging aid in the sense that where a transformation may not have had the intended effect, or the previous application of some other transformation caused this one to fail to apply, the system could be queried as to why a given transformation did not apply, or what the state of the program was at the point the user expected the transformation to apply. The answers would help the programmer to locate errors in the transformation set.

2.3. Related Work

Other systems have done some work in the line of accountability [Davis 1978], but most of them consist of little more than a simple trace-back of the computations performed in the program. A full or even partial trace of what happened has limited usefulness, however, and the thought of having to wade through one is enough to deter all but the most desperate. What is really needed is a mechanism whereby the available information about a computation is sifted through and sorted, and then only the information which is relevant is passed on to the user. But even a fancy trace system may do that much. I know of no extant transformation system which makes any serious attempt at understanding its own process and accounting for that process to the user. Instead, developers of current transformation systems have concentrated their efforts on ordering the application of transformations, proving their transformations to be equivalence preserving, and experimenting with various types of control for the systems [Loveman and Faneuf 1975] [Wegbreit 1976] [Gerhart 1975] [Nievergelt 1965] [Schwartz 1974] [Scheifler 1977].

2.4. Understanding the User's Needs

What information does a user need to understand the transformation process? We already have some idea from working through the two examples in Chapter One. Certainly the information needed to answer the questions listed at the beginning of this chapter would be useful. Notice that some of the questions asked information about a particular piece of code, while others requested information about a transformation. That is, one might ask "What are all the transformations that applied to this piece of code?" or "Where are all the places in the code that this transformation applied?" (or "was attempted", or "didn't apply"). The types of questions a person would ask depend on that person's goals and reasons for using the accountable system in the first place.

Let us then consider possible goals: Someone who was developing the transformational component of this system might wish to determine the effectiveness of a particular transformation or be interested in following the interaction of various transformations. That person would tend to have more questions regarding particular transformations. For example, the type of information that would have been helpful to me in debugging would have been the ability to ask what transformations did not apply, or perhaps, which transformations always applied. On the other hand, a programmer using this system to optimize his program would be more likely to ask questions relative to specific pieces of code. For example, he might ask if a certain function was changed at all, or why a certain transformation was able to apply to the piece of code of interest.

Either of these are valid approaches, and each of them was listed as a reason for using an accountable system. Thus, we will attempt to record information which can be used to answer questions that any of these potential users might have. In fact, I claim that the same information is necessary to answer any of them, and that only their viewpoint differs. The information needed is:

(a) that a transformation was attempted at some point in the code, and whether or not it successfully applied;

(b) what any piece of code looked like following any transformation;

(c) things known to be true (or false) at any given location in code;

(d) how the informatioa in (c) came to be known;

(e) for each individual transformation: what must be true to enable successful application (prerequisites), and what is true following a successful application.

From this information, I claim we can derive the answers to any of the questions that a user might ask (which have been suggested so far).

2.5. The Design

I believe that any mechanisms for recording what goes on during the transformation process should be independent of those mechanisms which perform the transformations. That is, an accountable transformation system can be viewed as two separate processes: one behaves just as current transformation systems do in that it decides when and where to perform the transformations, then does so; the other process stands by and quietly watches, taking note of all such decisions. The advantage of this approach is that already extant transformation systems can be made accountable with less hassle, and furthermore, the division of labor simplifies the task of constructing an accountable system. I will occasionally refer to the first process as the "transformational component", and to the second as the "accountable component". As we will see in more detail later on, the accountable component will be further divided into two sub-components: a recording element and a query element.

The two processes viewed together appear to the user as a transformation system that is aware of its own actions, and which can then report to the user what it did, step by step. An interesting variation of this approach might be a system that reported what it did as it did it, rather than waiting until the end to provide the user with information. Although this method has some merit and deserves to be studied, its one clear disadvantage is that it gives the user less choice about what information to receive. Though he might have the option before the process began to specify what types of decisions to be told about, it would be difficult for him to ask for other information based on what he saw during the process. Furthermore, if the program to be transformed is very large, or the accountable system is very slow, the user may not wish to "wait around" for the information he requested to be displayed. Instead, while the transformation system is cooking, the programmer can go off and have a cup of tea. Upon their mutual return, he may ask for whatever information he desires, decide on the basis of that information what else he wishes to see, then immediately study that information as well. One reason to favor having a "real-time" report would be if on the basis of information reported, the user could then opt to change the course of the transformation process. This feature is beyond the scope of our discussion of an accountable system, though I will venture to suggest that such decisions perhaps should be automated; that is, they should be part of the transformation system itself.

For a system to account for its actions (and those actions are the transformations it does or does not successfully perform), it must keep some sort of record of what it is doing as it executes, and be able to later access that record in a meaningful way. We must therefore

(1) collect and record the relevant information in some reasonable fashion, and

(2) access, process, and return that information to the user.

These tasks will be performed respectively by the recording and query elements of the accountable component.

2.5.1. The Recording Element

Let's again go over the information we decided in section 2.4 to collect:

(a) **We must record that a transformation was attempted at some point in the code, and whether or not it successfully applied.**

The time to collect this information is clearly at the time of transformation application. I have not yet discussed exactly what constitutes a transformation, nor what its inputs and outputs are. It seems reasonable to suggest, however, that one thing a transformation might be responsible for returning is a flag as to whether or not it applied. This information can then be recorded by the recording element.

(b) **We must record what any piece of code looked like following any transformation.**

This simply means we must keep track of all the intermediate results of transformation. We can't "let go" of an old expression when a transformation applies which replaces that expression with some equivalent one. The time to update this information is, again, at the time of transformation application, but only when a transformation applies *successfully*, since only at such times will expressions change. We will assume that each successful transformation returns not only a flag signifying that it applied, but the results of its application as well.

(c) **We must record things known to be true (or false) at any given location in code.**

By this I mean information relating to context. For example, code which is executed following the true branch of a test for `x=0` may assume that `x=0` is true. Recording this information is complicated by the fact that, for any given point in code (and there are many!), what is true at one time may not be true at another, because the results of transformation application may change the context of an expression. We will discuss this issue in more detail in Chapter Three. For now, suffice it to say that there will be an initial context to record, and then some updating necessary as each transformation applies.

(d) **We must record how the information in (c) came to be known.**

This consitutes a justification. Let us say some transformation `t` applies to a piece of code. Its application depends on some truth `p`. Although upon being queried as to why `t` applied, the system might return `p`, we might also wish to understand why `p` is true. Thus each truth must also have a justification. Recalling the example given above in which `x=0` was true, a transformation using that information would be responsible for reporting that it did. If we then needed to be reminded why `x=0` was true, we should be able somehow to obtain the reply "`This code is part of the true branch following a test for x=0`", and then perhaps even a pointer to that test. The time to collect this information is at the time that `p`is recorded. That is, every truth must carry with it a justification; one can not be recorded without the other.

(e) **For each individual transformation we must record (1) what must be true to enable successful application (prerequisites), and (2) what is true following a successful application.**

This is static information and only needs to be recorded once. It is associated with a transformation (as opposed to a section of code), and should be accessible given only the name of a transformation. It is in some sense part of the description of a tranformation, though discussion of how the information is represented will be postponed until Chapters Three and Four.

We may now summarize the task of the recording element of the accountable system. Transformations must include in their description a set of input and output assertions, available upon request. Before any transformations apply, a set of truths for each piece of code (dependent on its context) must be collected and recorded along with their justifications. After each transformation is attempted, it must provide to the recording element the following information:

(1) what code it attempted to transform;

(2) whether or not it was successful;

(3) if it was successful, the resultant code.

The recording element must represent this information in an efficient and accessible manner.

Notice that, in all our discussion, no mention has been made of how a transformation comes to apply. Obviously one succeeds if and only if all of its input prerequisites are met, but even before that can occur a transformation must be attempted in the first place. The mechanism which controls this process is independent not only of the recording element, but of the accountable component as a whole. It constitutes the tranformational component of the system, and need not be aware at all that its actions are being monitored. Similarly, recall that I consider the accountable component merely a passive observer of the transformation process implemented by the transformational component; it is not concerned with nor responsible for decisions made by that component. This is not the contradiction it appears to be. *For although, when viewed as two components, one works and the other only blindly records, when they are viewed as one system, as far as the user is concerned that system accounts for its actions.*

2.5.2. The Query Element

As a result of recording all of the above information, we have the ability to provide not just a fancy trace of all the transformations performed, but an historical "slice" of information about any subtree of the program. That is, if we imagine the set of intermediate program trees which represent the entire program after each transformation has applied, the accountable component (via the query element) can "cut across time" (so to speak) and return only the information relevant to changes made in a particular subtree of the program. For example, suppose a programmer expected a certain function call to be replaced by its argument. He should be able to somehow "point" to the spot in code and inquire as to whether this substitution occurred and why or why not.

From grammar school days, we remember that all complete sentences must have both a subject and a verb. The subject may be implied, as in the command "Go" which expands to "You go". Or the verb may be implied, as "I" in answer to the question "Who did it?", which expands to "I did it". Similarly, when one asks a question, both a subject and a verb must be present, expressed, or implied. In the query element, we need a mechanism for "pointing" to the subject of our query, and a mechanism for specifying the action or verb we wish to invoke.

2.5.2.1. *Pointing to the Subject*

If the subject is a transformation, we may refer to that transformation by its name. A transformation never changes (if it does, it is no longer the same transformation) and is therefore easy to point to.

But now consider the problem if the subject is a piece of code. We have no easy way to point to it, unless the code is a function definition, in which case we can refer to it by name. To point to a piece of a function definition, we must describe that piece: "The first predicate of the conditional clause which is the body of the function definition FOO" or "The second argument to the function call BAR where that call occurs in the call to BAZ in the value returned from the second clause of the conditional expression which is bound to the dummy variable TREE in the LET statement which forms the body of the function definition of QUUX." That sounds pretty awful; would you like to see it?

```
(DEFUN QUUX (A B C)
(LET ((TREE (COND ((NULL A) NIL)
                  (T (BAZ (BAR B (CAR A)))))))
     (COND ((NULL TREE) (LIST A B C))
           (T (CONS C TREE)))))
```

That's really not so bad, is it? The problem is that even to point to the code (CAR A) in the trivial definition above requires a lot of blabbering. Another method might be simply to say "The first (or *n*th') occurrence of the call (CAR A) in the definition of QUUX", but there are cases when having to count the occurrences of the desired expression (or even giving the desired expression) would be more complicated than referring to the structure of the context. For example, consider a lengthy function definition in which you wished to point to a certain conditional expression. There very well might be a dozen conditional expressions in the definition, and you would have to look for and find them all until you came to the one you wanted in order to know the value of n. Not that the first way is easier; both methods are painful. A simple solution to the problem of pointing would be to use a graphics terminal equipped with a mouse; however, I don't want this system to have to rely on those.

Another complicating factor arises if we decide to ask about some piece of code that only occurs in an intermediate version of the program. That means that to specify the code we wish to use as the subject of our query, we must not only describe the context of the parse tree it occurs in, but specify the time at which that tree occurred. What constitutes *time* in our transformation system?

Since the passage of time around some object can only be detected by the observation of change in that object, we must consider what might bring forth changes in the argument program. The successful application of a transformation certainly does, though an unsuccessful attempt does not change the argument program. Assuming that the system will operate on a sequential machine (as opposed to the complexities involved in dealing with parallel processes or parallel machines), then defining a unit of time to be the application of a transformation seems appropriate. Later we will see whether this definition is appropriate when dealing with unsuccessful transformation attempts.

We can now label each unit of time. The input program is labelled version 0, and if there were n successful transformation applications, then the resultant output program is version n. For any k such that $0 \leq k \leq n$, version k is the program produced by the first k transformation applications. Or, said another way, version k of the program is the result of applying the kth transformation to version $k - 1$ of the program.

This description of time is useful not only in the the query element, but in the recording element as well. That element will need a means of distinguishing the record of one version from the record of another.

But now back to the problem at hand: pointing to a particular piece of code. We know we can specify the version of the tree we wish used as a search domain by giving the correct numerical label. That can be a big problem if we don't know at what time(s) the tree was extant!

To limit the possibilities, let's constrain them. First, we will assume that the programmer is most familiar with the version of the program he wrote. Thus, if at any time he wishes to ask about a particular piece of code, most likely he mentally sees the subject code as it occurs in version 0 of the program and is able to describe the location of the code he wishes to ask about as it appears in that version. Therefore we state that the initial search for a piece of code to point at will be made using version 0 of the program. Secondly, given the tree to use as a search domain, in order to specify that subtree of it to which one wishes to point, one first specifies the relevant function definition name (QUUX in the example above). Then the programmer must somehow direct the query element to the desired piece. Since LISP lends itself so readily to the use of tree structures, I have decided (for now) that the most painless way of referring to a piece of code is to say "down", "over", or "up" until the correct piece is pointed at. Thus, to point at (CAR A) in the example above, I would give the function name QUUX, then the string of commands "down, over, over, over (now it's pointing to the LET statement), down, over, down, down, over (now it's pointing to the first conditional expression), down, over, over, down, over (now it's pointing to the call to BAZ), down, over, down, over, over (and now it's pointing to the call (CAR A))." Now that also sounds painful, but if a cursor responds immediately to each command by jumping in front of the currently pointed to expression as displayed on the terminal screen, then it's not so bad. Once we discuss the implementation of our system, things may get easier, but for now we will settle for this method of code specification.

2.5.2.2. *Specifying the Action*

Specifying the action to perform or some piece of information to return about the subject is considerably easier than pointing to the code. If we imagine a menu of available information, all one must do is select an item from that menu, and the desired dish of information should be served immediately. Designing the menu to meet the needs of the user is, however, an important and non-trivial task. I have discussed already (in section 2.4) the information we have decided to provide the user. Let's now categorize this information further into some sort of draft menu. Whether or not providing this information is actually feasible will be discussed in Chapter Four.

Given a transformation as the subject of a query, we can ask the following:

- (n) number of times attempted — Returns a number which represents the number of times this transformation was attempted.
- (+) positive attempts — Returns a number which represents the number of times this transformation successfully applied.
- (-) negative attempts — Returns a number which represents the number of times this transfor-

mation failed to apply.

- (<a number>) some number n — Prompts for either a + or a -, then returns the piece of code which was the argument to the transformation the nth time it succeeded or failed, respectively.
- (i) input assertions — Returns those truths which are prerequisites for application of this transformation.
- (o) output assertions — Returns those truths which will be changed as a consequence of the successful application of this transformation.

Given a piece of code as the subject of a query, we may ask the following questions:

- (+) next change to subject tree — Returns the piece of code as it appears following the transformation which next applied to it at the current level (as opposed to one of its subtrees). If no transformation applied at that level, it says so and returns the same piece of code.
- (-) previous change to subject tree — Returns the piece of code as it appears previous to the transformation which last applied to it at the current level. If no transformation has applied at that level, it says so and returns the same piece of code.
- (n) next version of subject tree — Returns the same tree as it appears following the transformation which next applied to any subtree of the subject tree. If there is no change, it says so and returns the same piece of code.
- (p) previous version of subject tree — Returns the same tree as it appears previous to the transformation which last applied to any subtree of the subject tree. If there was no change, it says so and returns the same piece of code.
- (l) last version of subject tree — Returns the final version of the subject tree, as it appears in the output of the transformational component.
- (v) current version of subject tree — Returns a number.
- (x) transformation which created current version of subject tree — Returns the name of the transformation.
- (t) truths known — Returns a list of the things known to be true for subject tree at the current point in time, and for each truth, its justification.
- (s) set version — Prompts for some number n and returns the subject tree as it appears in version n.

2.5.2.3. *An Example*

Given this menu of information, let's dredge up the function call used back in Chapter One and see if our new accountable component with its recording and query elements gets us anywhere. Our input to the transformation system (used with some canned set of optimizing transformations) is the following function call:

```
(APPEND (MYMAPCAR (FUNCTION FOO) '(A B C)) (BAZ Z Z))
```

For now we will assume that the system already knows the definitions of MYMAPCAR and BAZ. We set the query element to point to version 0 of the call to APPEND. Now what? We can ask to see what this tree looked like at any point in time (version 0 through version *n* where there were *n* total transformations applied). The system when it is through transforming will tell us how many transformations applied, that is, the "time" at the end of the run, if we enter the "1" command. Let's say there were 10 applications. Then we ask to see version 10. The fully optimized code returned by the system looks like:

```
(CONS (MYFUNCALL (FUNCTION FOO) 'A)
      (MYMAPCAR (FUNCTION FOO) '(B C)))
```

Now we start scratching our respective heads. First, we are curious to learn where the CONS came from, so we enter "-" to see what was here before.

```
(APPEND (CONS (MYFUNCALL (FUNCTION FOO) 'A)
              (MYMAPCAR (FUNCTION FOO) '(B C)))
        NIL)
```

"Ahhh..." we say, "the call to APPEND was simplified; it's second argument was NIL, so the first argument was returned." And in fact, we can get just that answer if we ask for the name of the transformation that applied, and then, using that as our subject, inquire as to its input assertions.

We're pretty sure we understand how the first argument to APPEND came to be, but we can't figure out the NIL. So we give the correct combination of "downs" and "overs" to arrive at the NIL, which in this case would be "down, over, over."

```
NIL
```

Now we enter "-" to learn what this tree was before it was NIL.

```
(COND (T NIL))
```

That wasn't incredibly helpful; let's try entering "-" again.

```
(COND (T NIL)
      (T (LIST Z Z)))
```

The code begins to look familiar, but obviously what we want is to discover how it comes to believe the first test will always return T. Let's "down, over, down" our way to the first predicate and enter "-".

```
(EQ Z Z)
```

Now it's all clear! Since we called BAZ with the arguments Z and Z, the first test in its definition will always be true. But let's say we didn't remember the call to BAZ (maybe you don't!). We can back up to the conditional expression ("up, up") and ask to see what it was in version 0.

```
(BAZ Z Z)
```

Now we enter "n" to see what happened to this tree next.

```
((LAMBDA (X-0 Y-0)
         (COND ((EQ X-0 Y-0) NIL)
               (T (LIST X-0 Y-0))))
 Z Z)
```

If we ask for the transformation which produced that, it would say Procedure integration. We can live with that. The strange dummy argument names were created (that's "gensymed" for LISP hackers) to avoid variable name conflicts. So now what? If the programmer still doesn't understand, he can continue to ask for information until he does. We can see at this point, however, that when the actual arguments are substituted for the dummy arguments (Z for X-0, and Z for Y-0), that the equality test will change to (EQ Z Z), which explains the NIL we had way back up there.

Although I wish to concentrate on the issues involved in implementing an *accountable* source-to-source transformation system, we must first discuss the transformation system it will work with. As you may recall from section 2.5, the transformational component and the accountable component are to be independent of each other; the transformational component is not aware of the existence of the accountable component, and though the accountable component is aware of the transformational component, it is not responsible for the decisions that the transformational component makes. If I were interested in discussing issues of the implementation of transformation systems, then I could ignore the issue of accountability (and indeed, so far everyone else has ignored it). However, since the accountable component both observes the actions of the other and expects certain information to be made available by the transformations, before describing its implementation I will first discuss the transformational component of the accountable system.

Chapter Three

The Transformational Component

THE TRANSFORMATIONAL COMPONENT of the system will be implemented as simply and straightforwardly as possible so that we may concentrate on the more interesting issue of accountability. Though source-to-source transformation systems conceptually operate on the text of a high-level language source program, internally they typically manipulate a non-textual representation, similar to (or often identical to) a parse tree of the program. The use of this internal form makes the system's access to pieces of the structure more efficient (as opposed to having to re-parse the program every time the system needs to refer to a part of it). In order to avoid having to deal with parsing and internal representation, I made several decisions. First, I chose to transform LISP programs only. Since the LISP parser is made available to the user via READ, I didn't have to write my own. Second, I chose to implement the transformation system in LISP. And third, I decided to hide decision number one from decision number two. Which leads us to the subject of *information hiding.*

3.1. Information Hiding

Information hiding is the separation of data definition from program definition. In the case of the transformation system, it means not letting the implementation realize that it is transforming LISP code. Instead, whenever it wishes to access a piece of data, rather than obtaining that piece directly it calls a procedure to do it. That is, there are specific data selection and data constructing procedures which are called whenever data is manipulated. For example, suppose that we have

```
(DEFUN FCALL!FUNC (FUNCTION-CALL)
       (CAR FUNCTION-CALL))

(DEFUN FCALL!ARGLIST (FUNCTION-CALL)
       (CDR FUNCTION-CALL))

(DEFUN FCALL!FIRST-ARG (FUNCTION-CALL)
       (ARGS!ARG (FCALL!ARGLIST FUNCTION-CALL)))

(DEFUN FCALL!SECOND-ARG (FUNCTION-CALL)
       (ARGS!ARG (ARGS!RESTARGS (FCALL!ARGLIST FUNCTION-CALL))))
```

TABLE 3-1. Selector functions for the Function Call structure type.

a function call which is bound to the atom FCALL, and that we wish to select its first argument. In LISP, a function call is represented as a list, with the operator being the first element and the operands following. Thus, the first argument of a function call would be the CADR of the list. However, instead of writing (CADR FCALL) to obtain the first argument though, we call the procedure FCALL!FIRST-ARG, defined as:

```
(DEFUN FCALL!FIRST-ARG (FCALL)
       (CADR FCALL))
```

If the representation of a function call ever changes, we only need to rewrite the relevant data selecting and constructing functions. The program itself should remain unaffected. We can think of a function call as a particular data type, and require that no direct reference be made to any instance of that type except via specific operations defined for that type only. Similarly, we define a list of arguments to be a separate type also, hiding the representation of argument lists from the system. We write the function FCALL!ARGLIST which takes a function call and returns the list of arguments, and the function ARGS!ARG, which takes an argument list and returns the first argument. The definition of FCALL!FIRST-ARG which uses these two new functions is given in Table 3-1. In addition, we can define predicates on these data types which test their state. For example, by calling the function ARGS!NULL?, we can ask whether a list of arguments is empty or not. An extended version of ARGS!ARG (shown in Table 3-2) uses this predicate to make sure that no argument list is empty.

The advantages of information hiding are that it:

- allows greater flexibility in choice of data representation,
- enhances self-documentation,
- encourages one to consider the data at an abstract level, apart from program specification, and
- facilitates proving program correctness.

```
;;; Selectors

(DEFUN ARGS!ARG (ARGLIST)
       (COND ((ARGS!NULL? ARGLIST)
              (ERROR '|Arglist is null -- ARGS!ARG| ARGLIST))
             (T (CAR ARGLIST))))

(DEFUN ARGS!RESTARGS (ARGLIST)
       (COND ((ARGS!NULL? ARGLIST)
              (ERROR '|Arglist is null -- ARGS!RESTARGS| ARGLIST))
             (T (CDR ARGLIST))))

;;; Predicates

(DEFUN ARGS!NULL? (ARGLIST) (NULL ARGLIST))

;;; Constructors

(DEFUN ARGS!ADD-ARG (ARG ARGLIST) (CONS ARG ARGLIST))

(DEFUN ARGS!ADD-ARGS (ARGS ARGLIST) (APPEND ARGS ARGLIST))
```

TABLE 3-2. Operations for the Argument List structure type.

The *disadvantage* of this strategy is that there is extensive overhead in the additional procedure calls, causing execution time to shoot way up (by about 300 per cent in one of my programs). However, this can be overcome through use of a set of optimizing transformations. Transformations can be written which search for calls to the selector and constructor functions and replace these calls by the appropriate code. We can have our tea and drink it too. In addition, transformations can be written to eliminate duplicate tests that may arise as a result of such procedure integration. What if the call (FCALL!FIRST-ARG FCALL) was made at a point in code where it was already known that the argument list of FCALL was not empty? The call to ARGS!NULL? in the body of the FCALL!FIRST-ARG would be redundant; a set of optimizing transformations could realize this and eliminate the redundant test. Clearly, information hiding (data abstraction) and program transformation go hand in hand. For more of my views on the subject of information hiding, see [Kerns 1977] and [Steele 1980].

3.2. Internal Form

By using this information hiding technique, I have essentially hidden the internal form of the argument program from the transformation system. This will be important later, when I augment the internal form of the argument program, using it to store more than just pieces of program text. At that point having special functions to access pieces of it will simplify the task. For now, however, it is safe to think of the internal form of the LISP program as the LISP program itself. Some sample accessing and constructing functions are shown in Table 3-1 and Table 3-2.

3.3. Control

The "control" of the transformational component is that process which models a human programmer attempting to optimize his code. He may have a catalog of transformations at his side, but without the knowledge of how to apply them they do him no good. He must recognize that a transformation is applicable to his code, understand the possible interactions between multiple *transformations to a single piece of code, and be able to verify that a transformation is valid.* Two of the most difficult problems in automating this process are (1) transformation ordering and (2) information gathering [Loveman 1977]. By "transformation ordering," I mean the decision of when to apply which transformation where. "Information gathering" refers to the ascertaining of various truths about pieces of code.

3.3.1. Transformation Ordering

*Given the parse tree of an argument program and a set of transformations to apply to that pro*gram, the order in which those transformations are applied may determine the resultant program tree. For example, suppose I have the following set of two transformations:

```
(1)  (APPEND <X> (APPEND <Y> <Z>))  =>  (APPEND <X> <Y> <Z>)
(2)  (APPEND NIL <X>)  =>  <X>
```

I wish to transform the program:

```
(APPEND FOO (APPEND NIL BAR))
```

If I apply transformation (1) first, the resultant program is:

```
(APPEND FOO NIL BAR)
```

Then transformation (2) will fail to apply. If, however, I apply transformation (2) first, the result is:

```
(APPEND FOO BAR)
```

Then transformation (1) fails to apply. In the first case (APPEND FOO NIL BAR) is the result, and in the other, (APPEND FOO BAR). Hence, the order of transformation affects not only which transformations will apply, but the resultant code as well.

A previous system of mine dealt with this problem by tightly coupling the transformations with the control. The system knew exactly when a transformation was likely to apply because the successful application of one transformation would suggest others that were likely to apply next. The transformation system made only one pass over the program tree; it took the tree apart on the way down, gathering information as it went, then put it back together "better" on the way up One could not really point to any piece of system code and say "This is a transformation"; the so-called transformations had lost their identity by being so tightly bound up in the system. The consequences of this approach were that transformations could not be added, deleted, changed, or temporarily turned on or off, without much difficulty and groveling around in the system. Indeed, there is some question as to whether this sort of system could correctly be labelled a transformation system. It was rather merely an "optimizing pre-processor."

The transformations in our system must be identifiable; in order to record the information discussed in section 2.4, we have to know when a transformation is attempted, what its prerequisites for application are, and whether it fails or succeeds. Therefore, the current system instead separates the control structure from the transformations. Furthermore, the notion of interactive optimization involves giving the programmer the ability to turn certain transformations on or off and to choose the set or sets of transformations he wishes applied in the first place. Ideally, he should be able even to write his own transformations. Clearly he can do this only if this separation of control and transformation is maintained. Hence, we must severely limit the knowledge the control structure has of the transformations it will use.

Given a set of identifiable transformations, independent of the control structure, how are we then to control their application? We must order not only the transformations to be applied, but the subtrees to which they will apply as well. For those two orderings, we must then choose a method of application. We will discuss each of these three decisions in *order*.

3.3.1.1. *Transformations Within a Set*

Obviously we want to choose an ordering for the transformations within the transformation set which will produce the most desirable result. Consider applying the transformation set

```
(1) (<arithmetic op> <args-1> 0 <args-2>)  =>
    (<arithmetic op> 0 <args-1> <args-2>)
(2) (* 0 <args>) => 0
(3) (+ 0 <args>) => (+ <args>)
(4) (+ <arg>) => <arg>
```

to the program:

```
(+ 7 (* 5 3 0))
```

Input	Transformation	Result
`(+ 7 (* 5 3 0))`	(1)	`(+ 7 (* 0 3 5))`
`(+ 7 (* 0 3 5))`	(1)	fails
`(+ 7 (* 0 3 5))`	(2)	`(+ 7 0)`
`(+ 7 0)`	(1)	`(+ 0 7)`
`(+ 0 7)`	(1)	fails
`(+ 0 7)`	(2)	fails
`(+ 0 7)`	(3)	`(+ 7)`
`(+ 7)`	(1)	fails
`(+ 7)`	(2)	fails
`(+ 7)`	(3)	fails
`(+ 7)`	(4)	7
7	(1)	fails
7	(2)	fails
7	(3)	fails
7	(4)	fails
	FINISH	

TABLE 3-3. Markov algorithm ordering rules demonstrated.

Then neither (2) nor (3) will apply until (1) does, but if (1) applies followed by (2), (3) will not apply unless (1) applies again.[1] One way to ensure that every transformation is applied as often as necessary is to use the ordering rules of Markov algorithms. If these rules are used in the above example, we obtain the results given in Table 3-3. The number of attempts can be increased by changing the order of the transformations in the set, but for this example, the end results would be the same. We of course want to minimize the attempts for reasons of efficiency, but we can derive some comfort from the fact that iterating over the transformations solves some of our problems. Markov algorithm rules are only one method of iteration. We could specify that a transformation is to be applied over and over until it fails, without returning to the top of the list of transformations between. For example, the program `(+ 3 0 4 0 5 0)` would be changed to `(+ 0 0 0 3 4 5)` after three successive applications of (1), then to `(+ 3 4 5)` after three successive applications of (3). Or, instead of requiring the control structure to direct the iteration of transformation application, we could place the burden on the transformations themselves. That is, those transformations which need to be iterated would be responsible for iterating within themselves.

[1] Note that arithmetic optimization like this is very tricky. Round-off error or even overflow may be affected by order of evaluation and/or association. Furthermore, in FORTRAN, *I* + 0.0 may be used to change the type of *I*. Similarly, in MacLISP, `(PLUS (FIXNUM-IDENTITY I) 0.0)` is the same as `(FLOAT I)`.

Iteration of transformations solves the problem of the application of a transformation resulting in code which can be changed by a transformation that has already applied. There is still the problem in which the successful application of one transformation prevents the successful application of another, as was illustrated in the APPEND example given earlier. One solution to this problem is to supply enough transformations to cover all the cases which might arise as a result of previous transformation applications. That is, we would supply a third transformation:

```
(APPEND <X> NIL <Y>) => (APPEND <X> <Y>)
```

This solution can be carried to extreme though, supplying many special purpose transformations with no general utility at all. (Why not simply write the one transformation which will apply to the entire program and produce the desired result?) Clearly, some care needs to be taken when ordering transformations within a set; there is no system of application that is guaranteed to compensate for thoughtless ordering on the part of the programmer.

Let me just comment briefly on one other ordering strategy, where the transformation which applies is responsible for suggesting the transformation(s) to try next. That is, the order of application is determined dynamically. There certainly is much to be said for this method, but unless it is done in such a way as to guarantee that all those transformations which should be applied are reachable, it should only supplement the methods we have discussed above. Whether one specifies the transformation which is to be tried next by ordering them within a set or by linking them within themselves, human thought must be used. In this system, I have chosen to simplify the control structure by making the transformations responsible for their own iteration, and ordering them by means of their placement in a transformation set.

3.3.1.2. *Subtrees of the Parse Tree*

As well as ordering the transformations within the set, we must choose an order in which to transform pieces of the argument program. Several possible orderings are: the lexical order of the input text, the order of evaluation of the input program, or a left-most, depth-first tree order [Geschke 1972]. The lexical order of the input text is not a good choice. In LISP, the order of evaluation is with minor exceptions, a left-most, depth-first tree order. There are a number of good reasons to select the order of evaluation of the program as the order of transformation of the program:

- Optimizing a program can be thought of as partial evaluation of the program.
- If it can be determined that a particular subtree would never be evaluated then similarly there is no need to transform it.
- Since LISP recursively evaluates the arguments of an expression before applying the operator to those arguments, a tranformation which applies to some subtree can assume that its subtrees

```
(DEFUN SEXPR!EXPAND (SEXPR ENVIRON FTLIST)
      (COND ((ATOM? SEXPR) SEXPR)
            ((COND? SEXPR) (COND!EXPAND SEXPR ENVIRON FTLIST))
            ((DEFETTE? SEXPR) (DEFETTE!EXPAND SEXPR ENVIRON FTLIST))
            ((FCALL? SEXPR) (FCALL!EXPAND SEXPR ENVIRON FTLIST))
            ((ANDEXPR? SEXPR) (AND!EXPAND SEXPR ENVIRON FTLIST))
            ((OREXPR? SEXPR) (OR!EXPAND SEXPR ENVIRON FTLIST))
            ((QUOTE? SEXPR) SEXPR)
            (T (ERROR '|Unparsable s-expression  - EXPAND| SEXPR))))
```

TABLE 3-4. Main control function for the transformational component.

have already been transformed.

To better illustrate this last point, consider again the APPEND example. If we transform subtrees in the order in which they will be evaluated, then the transformations will apply to the outer APPEND only after they have applied to the inner one. So regardless of the order in which those transformations occur within the transformation set, the transformation which simplifies the inner call, (APPEND NIL BAR), will always apply before the transformation which simplifies the outer call, (APPEND FOO (APPEND NIL BAR)).

3.3.1.3. *Methods of Application*

Though we have discussed the ordering of transformations within a set and selected an order for application to subtrees of the parse tree, we must still decide on one of the following two methods of application:

(1) Given a set of transformations and a program tree, apply each transformation in its turn to all the subtrees in the tree. First apply one transformation everywhere, then eliminate it, then apply the second transformation everywhere, eliminate it, and continue until the set of transformations is empty.

(2) Given a set of transformations and a program tree, apply all the transformations to each subtree of the tree in its turn. First apply all the transformations to the first subtree in the tree, then move on to the next subtree and apply all the transformations possible there, and continue until every subtree has been transformed.

Our transformation system uses method (2), for the following reasons:

Since we have decided to apply the transformations to the subtrees of the program's parse tree in the order that those subtrees will be evaluated, we can model the control structure after the design of a LISP interpreter. Given some s-expression to evaluate, the interpreter will first determine the type of structure it has hold of. LISP has a number of special forms [Pitman 1980], and all of the arguments in these forms are not necessarily evaluated. For example, the arguments to a conditional expression are evaluated in a special way, as are the arguments to AND, OR, QUOTE,

ATOM	an atom
COND	a conditional expression
CLAUSE	"argument" to a conditional expression
DEFETTE	a function definition
	(may be an internal lambda expression, or a MacLISP DEFUN , or whatever)
FCALL	a function call
	(may be built-in or user defined)
FUNCNAME	the name of a function
AND	a conjunct
OR	a disjunct
QUOTE	a quoted expression

TABLE 3-5. Structure types.

```
(DEFUN AND!XFORM (ANDCELL ENVIRON FTLIST)
       (XFORM 'AND!TRIM ANDCELL (LIST))
       (XFORM 'AND!SIMPLIFY ANDCELL (LIST))
       (XFORM 'SEXPR!FORM-BVAL ANDCELL (LIST ENVIRON FTLIST)))

(DEFUN ATOM!XFORM (ATOMIC-EXPR ENVIRON FTLIST)
       (XFORM 'SEXPR!FORM-BVAL ATOMIC-EXPR (LIST ENVIRON FTLIST)))

(DEFUN COND!XFORM (CONDCELL ENVIRON FTLIST)
       (XFORM 'COND!TRIM CONDCELL (LIST ENVIRON FTLIST))
       (XFORM 'COND!SIMPLIFY CONDCELL (LIST))
       (XFORM 'SEXPR!FORM-BVAL CONDCELL (LIST ENVIRON FTLIST)))

(DEFUN DEFETTE!XFORM (DEFETTE ENVIRON FTLIST)
       (XFORM 'SEXPR!FORM-BVAL DEFETTE (LIST ENVIRON FTLIST)))
```

TABLE 3-7. Transformation sets for some structure types.

etc. Just as the interpreter checks the type of expression it is to evaluate, the transformation system looks at the type of the subtree it is to transform. Based on that type, the system will see that the arguments are correctly transformed (that is, the subtrees of the subtree it is dealing with), then will call *those transformations which can apply to the type of the current subtree.* I have categorized the possible s-expressions into different structure types, and divided the transformations into a separate set for each LISP structure type. That way, once the subtrees of an expression have been transformed, the expression itself is transformed by the set of tranformations which apply to its type. The control mechanism for the transformational component has limited knowledge of the tranformations it applies: it only knows that for each structure type there is a different set of tranformations, and will apply the correct set. The main control mechanism is implemented by the function SEXPR!EXPAND, listed in Table 3-4.

```
(DEFUN AND!EXPAND (ANDCELL ENVIRON FTLIST)
       (AND!PREDICATE ANDCELL)
       (ANDARGS!EXPAND (AND!ARGLIST ANDCELL) ENVIRON FTLIST)
       (AND!XFORM ANDCELL ENVIRON FTLIST))

(DEFUN ANDARGS!EXPAND (LOGARGS ENVIRON FTLIST) ; Slave function to AND!EXPAND
       (COND
        ((ARGS!NULL? LOGARGS) NIL)
        (T (LET ((FIRST-ARG (ARGS!ARG LOGARGS)))
                (SEXPR!EXPAND FIRST-ARG ENVIRON FTLIST)
                (COND
                 ((TRUE? FIRST-ARG)
                  (ANDARGS!EXPAND (ARGS!RESTARGS LOGARGS) ENVIRON FTLIST))
                 ((NULL? FIRST-ARG)
                  NIL)  ;Rest logargs not expanded since won't ever evaluate
                 (T (ANDARGS!EXPAND (ARGS!RESTARGS LOGARGS)
                                    (ENV!RECORD-TEST ENVIRON FIRST-ARG T FTLIST)
                                    FTLIST)))))))

(DEFUN ATOM!EXPAND (ATOMIC-EXPR ENVIRON FTLIST)
       (ATOM!XFORM ATOMIC-EXPR ENVIRON FTLIST))

(DEFUN COND!EXPAND (CONDCELL ENVIRON FTLIST)
       (CLAUSES!EXPAND (COND!CLAUSES CONDCELL) ENVIRON FTLIST)
       (COND!XFORM CONDCELL ENVIRON FTLIST))

(DEFUN DEFETTE!EXPAND (DEFETTE ENVIRON FTLIST)
       (DEFETTE!PREDICATE DEFETTE)
       (SEXPR!EXPAND (DEFETTE!BODY DEFETTE) ENVIRON FTLIST)
       (DEFETTE!XFORM DEFETTE ENVIRON FTLIST))
```

TABLE 3-6. Control functions for some structure types.

Every LISP structure that might be transformed must be given a transformation set and an opportunity to apply it. Though it is possible that someone might write a transformation meant to apply to the clause of a conditional expression, for example, one usually thinks of such a clause as part of the larger *COND* structure, rather than a structure type itself. Rather than put an limitations on transformations, however, I chose to give all such "substructures" structure types of their own. The structure types I have defined can be seen in Table 3-5. Not all of them are recognized by SEXPR!EXPAND, since some of them are not evaluable independently of their superstructure. Each structure type has its own control function. Some of these control functions simply call the appropriate control function for its structure's subtree, followed by a call to the transformation set (see Table 3-6 and Table 3-7), but a few types need slightly more processing. The *FCALL* structure type, for example, consists of a function and an argument list. The function may be either be the name of a function (and thus a special atom of type *FUNCNAME*), in which case it is a candidate for procedure integration, or it may be an internal lambda definition. Controlling the transformation of internal lambda definitions can be a bit tricky, especially when it comes to keeping the information which has been gathered straight concerning the changes in variable names brought about by lambda binding. But that is a subject for the next section.

3.3.2. Information Gathering

Information gathering refers to the ascertaining of those truths about a piece of code which form its context. Local transformations such as the rules (CAR (CONS <X> <Y>)) => <X> and (EQUAL <X> <X>) => T don't need to know anything about the context of the code they are transforming [Bagwell 1970]. But global transformations do; their validity depends on the nature of the surrounding code [Schaefer 1973]. Since we have decided to transform expressions in the order in which they are evaluated, the transformation system will have walked down the parse tree as far as the expression it is currently transforming. That means it can be aware of the context of an expression by the time it gets to the expression, and merely needs a method for recording that context.

3.3.2.1. *Clarifying the Need*

The substitution rule (OR <X> T) => T is not always correct. In MacLISP, at least, OR returns the value of the first argument which is true. Thus, (OR 3 T) returns 3, not T. However, if this test occurs in a predicate position, such as the predicate of a conditional clause, only the boolean value of the expression is needed and the transformation of (OR 3 T) to T would be correct. Knowing whether or not an expression is in predicate position is a type of what I call "special" contextual information, as opposed to "common" contextual information. Special contextual information is true only of a particular instance of an expression. Just because (OR <X> T) occurs in predicate position at one point in code, doesn't imply that other instances of that expression do also. Common contextual information refers to truths pertaining to every occurrence of a particular expression. Consider optimizing the code:

```
(DEFUN EXAMPLE (FOO BAR BAZ)
       (COND ((ATOM FOO) NIL)
             ((ISINDEXED FOO) (PROCESS-INDEX FOO BAR BAZ))
             (   ...    (GET-PART FOO)   ...    )
             (T     ...     )))

(DEFUN ISINDEXED (X)
       (AND (NOT (ATOM X))
            (ATOM (CAR X))
            (NUMBERP (CAR X))))

(DEFUN GET-PART (Y)
       (COND ((ATOM Y) NIL)
             ((AND (ATOM (CAR Y))
                   (NUMBERP (CAR Y)))
```

```
                 (CADR Y))
                (T (CAR Y))))
```

When ISINDEXED is expanded in-line, the definition of EXAMPLE will appear as:

```
(DEFUN EXAMPLE-PROGRAM (FOO BAR BAZ)
       (COND ((ATOM FOO) NIL)
             ((AND (NOT (ATOM FOO))
                   (ATOM (CAR FOO))
                   (NUMBERP (CAR FOO)))
              (PROCESS-INDEX-PROGRAM FOO BAR BAZ))
             (    ...    (GET-PART FOO)     ...    )
             (T      ...      )))
```

The test (ATOM FOO) unnecessarily appears twice. The second clause of the conditional expression can only be evaluated if the predicate of the first clause returned NIL. Thus, at the point where ISINDEXED is expanded, it is known that (ATOM FOO) is NIL. Similarly, at the time that PROCESS-INDEX-PROGRAM is evaluated, it will be known that FOO is not an atom, (CAR FOO) *is* an atom, and (CAR FOO) is a number. And, when (GET-PART FOO) is expanded, it is known that FOO is not an atom, and that (CAR FOO) is not both an atom and a number. If at any time a test is performed whose outcome is already known, we would like to have the ability to eliminate that duplicate test. Thus, after transformation, EXAMPLE should be:

```
(DEFUN EXAMPLE-PROGRAM (FOO BAR BAZ)
       (COND ((ATOM FOO) NIL)
             ((AND (ATOM (CAR FOO))
                   (NUMBERP (CAR FOO)))
              (PROCESS-INDEX-PROGRAM FOO BAR BAZ))
             (    ...    (CAR FOO)     ...    )
             (T      ...      )))
```

I should mention here quickly one caution about global transformations. Most high-level languages are impotent without the use of side-effects. But side-effects wreak havoc for global transformations, which must then constantly be on the lookout for changes in the values of current variables. Just because FOO is not an atom at one point in the code doesn't guarantee that it is later, since (SETQ FOO <whatever>) (or something similar) could occur in the meantime. I have currently made no provisions for the transformation of code which may contain side-effects. The transformational component of this system assumes that the LISP code it operates on is totally applicative. I can defend this seeming lack of power on two counts:

- LISP is one high-level language that manages to accomplish a great deal even when limited to non-side-effecting functions. So it is not completely unreasonable to limit the transformation system in this way.

```
(DEFUN AND!PREDICATE (ANDCELL)
       (SEXPR!MAPCAR (FUNCTION PREDIFY) (AND!ARGLIST ANDCELL))
       (COND ((NOT (PRED? ANDCELL))
              (DEPREDIFY (ARGS!LAST-ARG (AND!ARGLIST ANDCELL))))))

(DEFUN DEFETTE!PREDICATE (DEFETTE)
       (COND ((PRED? DEFETTE) (PREDIFY (DEFETTE!BODY DEFETTE)))))
```

TABLE 3-8. Propagation of predicate position information.

- The accountable component of our system is in no way dependent on this constraint. It merely records the transformations made by the transformational component, and couldn't care less about the restrictions on or justification for those transformations. The emphasis of this thesis is on the accountable component; the transformational component I am describing has been implemented merely to demonstrate the use of the other.

3.3.2.2. *The Environment*

In order to allow global transformations in our system, we record both special and common contextual information. Special contextual information does not present much of a problem; we simply augment the internal form of the program by storing properties of expressions along with the expression itself. When a conditional expression is encountered, the predicate position property may be added to all the predicate expressions within the conditional. If an AND expression is in predicate position, than all its arguments are also. If not, all but its last argument is in predicate position, and so on. The code in Table 3-6 contains function calls of the form <type>!PREDICATE; their definitions are given in Table 3-8. Common contextual information, on the other hand, needs to be recorded in such a way that code lower in the tree can access it. We will do this by maintaining an environment, much like a LISP interpreter does. Then whenever a transformation requests information about some piece of code, the environment is queried.

Each member of the environment consists of some LISP expression whose truth value is known, its truth value, and a justification for that value. The environment acquires more knowledge every time it encounters:

- a conditional expression,
- an AND expression,
- an OR expression, or
- explicit assertions in the code.

A conditional of the form:

```
(cond ((<pred-1> <exitform-1>)
      ((<pred-2> <exitform-2>)
      ((<pred-3> <exitform-3>)
           :           :
)
```

can assume at each location the truths given:

```
(cond (?                                  [<pred-1> true]                                )
      ([<pred-1> false]                   [<pred-2> true, <pred-1> false]                )
      ([<pred-1> false, <pred-2> false]   [<pred-3> true, <pred-2> false, <pred-1> false])
           :                                   :
)
```

TABLE 3-9. Propagation of information within a conditional expression.

As we saw in the above section, tests performed within a conditional expression result in information which can then be distributed within that expression. Table 3-9 illustrates the information which can be distributed. CLAUSES!BRANCH is the function which knows how to transform conditional expressions. Conditionals are a special form in LISP because their "arguments" are not necessarily all evaluated. Thus, instead of transforming all the clauses right away, CLAUSES!BRANCH transforms only the first predicate of the first clause in the list of clauses. If that predicate is determined to be true, CLAUSES!BRANCH transforms the rest of the clause and quits, just like the LISP interpreter would do. If the predicate is false, it ignores the rest of the clause and calls itself on the remainder of the clauses. If the predicate can not be determined true or false, then it must transform both the rest of the clause and the remaining clauses, but it does this with an updated environment. Table 3-10 contains a listing of CLAUSES!BRANCH. Notice the two calls to ENV!RECORD-TEST in the definition of CLAUSES!BRANCH which occur when the current predicate cannot be determined to be true or false. In that case, the environment used for the transformation of the rest of the clause should contain the information that the current predicate is true, and the environment used by the rest of the clauses in the conditional expression should know that the current predicate is false. ENV!RECORD-TEST takes an environment, the predicate expression, and the predicate's outcome, and returns a new environment with the additional information recorded.

```
(DEFUN CLAUSES!EXPAND (CONDLIST *CLAUSES-ENVIRON* *CLAUSES-FTLIST*)
      (DECLARE (SPECIAL *CLAUSES-ENVIRON* *CLAUSES-FTLIST*))
      (COND
       ((CLAUSES!NULL? CONDLIST) NIL)
       (T (LET ((XTST (CLAUSE!PRED (CLAUSES!CLAUSE CONDLIST))))
               (PREDIFY XTST)
               (SEXPR!EXPAND XTST *CLAUSES-ENVIRON* *CLAUSES-FTLIST*)
               (COND
                ((TRUE? XTST)
                 ;; Rest of clauses not expanded since evaluation stops here.
                 (SEXPR!MAPCAR (FUNCTION
                                (LAMBDA (SEXPR)
                                        (SEXPR!EXPAND SEXPR
                                                      *CLAUSES-ENVIRON*
                                                      *CLAUSES-FTLIST*)))
                               (CLAUSE!EXITFORM (CLAUSES!CLAUSE CONDLIST))))
                ((NULL? XTST)
                 ;; Rest of clause not expanded since it will never be returned
                 (CLAUSES!EXPAND (CLAUSES!CLAUSES CONDLIST)
                                 *CLAUSES-ENVIRON*
                                 *CLAUSES-FTLIST*))
                (T (LET
                    ((*LET-TLIST*
                      (ENV!RECORD-TEST *CLAUSES-ENVIRON* XTST T *CLAUSES-FTLIST*)))
                    (DECLARE (SPECIAL *LET-TLIST*))
                    (SEXPR!MAPCAR (FUNCTION
                                   (LAMBDA (SEXPR)
                                           (SEXPR!EXPAND SEXPR
                                                         *LET-TLIST*
                                                         *CLAUSES-FTLIST*)))
                                  (CLAUSE!EXITFORM (CLAUSES!CLAUSE CONDLIST))))
                   (CLAUSES!EXPAND (CLAUSES!CLAUSES CONDLIST)
                                   (ENV!RECORD-TEST *CLAUSES-ENVIRON*
                                                    XTST
                                                    NIL
                                                    *CLAUSES-FTLIST*)
                                   *CLAUSES-FTLIST*)))))))
```

TABLE 3-10. Controlling function for conditional clauses.

After transforming any argument of an AND expression, transformation of the remaining arguments takes place with the assumption that the preceding argument is true. Similarly for an OR expression, the transformation of any argument may assume that all preceding arguments are false. The environment used for the transformation of these arguments is updated and maintained using the same functions described above.

Assertions consist of an s-expression and a list of truths to be assumed by that expression. An assertion may be hand coded by the programmer or generated automatically by a transformation. When an assertion is encountered by the main control function, SEXPR!EXPAND, it records the truths on the current enviroment and proceeds to transform the enveloped text. Thus an assertion constitutes a new structure type; we must provide for it by redefining SEXPR!EXPAND, as seen in Table 3-11.

```
(DEFUN SEXPR!EXPAND (SEXPR ENVIRON FTLIST)
      (COND ((ATOM? SEXPR) SEXPR)
            ((COND? SEXPR) (COND!EXPAND SEXPR ENVIRON FTLIST))
            ((DEFETTE? SEXPR) (DEFETTE!EXPAND SEXPR ENVIRON FTLIST))
            ((FCALL? SEXPR) (FCALL!EXPAND SEXPR ENVIRON FTLIST))
            ((ANDEXPR? SEXPR) (AND!EXPAND SEXPR ENVIRON FTLIST))
            ((OREXPR? SEXPR) (OR!EXPAND SEXPR ENVIRON FTLIST))
            ((QUOTE? SEXPR) SEXPR)
*           ((ASSERT? SEXPR) (ASSERT!EXPAND SEXPR ENVIRON FTLIST))
            (T (ERROR '|Unparsable s-expression  - EXPAND| SEXPR))))

(DEFUN ASSERT!EXPAND (ASSERTION ENVIRON FTLIST)
     (SEXPR!EXPAND (ASSERT!SEXPR ASSERTION)
                   (ENV!RECORD-TEST
                        ENVIRON
                        (SEXPR!EXPAND (ASSERT!TRUTHS ASSERTION) ENVIRON FTLIST)
                        T
                        FTLIST)
                    FTLIST))
```

TABLE 3-11. Providing for assertions.

The function ENV!RECORD-TEST has the responsibility of recording a test on the environment. Rather than blindly recording the test given it, the function attempts to break the test down into as many of its parts as possible. For example, if *ENV!RECORD-TEST is told to record as true* the test (AND P Q R), it will instead construct and record the individual entries: P is true, Q is true, and R is true. If it is told to record the fact that (AND A B C) is false, it first looks on the current environment to see if any of A, B, or C are known to be true. If so, ENV!RECORD-TEST can simplify the given test and record that. Suppose it sees that B is true. Then either A or C must have made (AND A B C) false. Thus it records as false the expression (AND A C). A predicate of the form (NULL <x>) or (NOT <x>) is stripped of its negation and the outcome toggled before the expression is recorded on the environment. Hence, if it is known that (NOT FOO) is true, the entry FOO is false is made on the environment.

These rules for recording information enforce a canonical form for entries in the environment that most efficiently provides information. Suppose we wished to determine whether or not (AND Q R) was true. We can search the environment and learn that Q is true and that R is true, thus (AND Q R) must be true. If, when (AND P Q R) was recorded as true above, we had simply entered the entire predicate, we would have had a harder time verifying (AND Q R). Instead of having to perform the logic every time something is looked up, we record the smallest pieces possible in the first place. A listing of ENV!RECORD-TST is given in Table 3-12.

I should mention one thing more about recording information on the environment: occasionally there are predicates recorded whose truth or falsity may imply other truths. For example, if a certain expression is known to be false (that is, null), then it is necessarily an atom, and conversely

```
(DEFUN ENV!RECORD-TEST (ENVIRON TST OUTCOME FTLIST)
       (COND
        ((NULL TST) ENVIRON)
        ((ATOM? TST)
         (ENV!ADD-TEST ENVIRON
                      (ENV-EL!CREATE (FALIST!SUBLIS FTLIST TST) OUTCOME NIL)))
        ((ANDEXPR? TST)
         (COND (OUTCOME         ; recording a true AND test
                (ENV!RECORD-TESTS ENVIRON (AND!ARGLIST TST) OUTCOME FTLIST))
               (T (ENV!ADD-TEST ; recording a false AND test
                   ENVIRON
                   (ENV-EL!CREATE
                     (FALIST!SUBLIS
                       FTLIST
                       (LET
                        ((REFINED
                          (ARGS!REFINE (AND!ARGLIST TST) OUTCOME ENVIRON FTLIST)))
                        (COND ((ARGS!ONE? REFINED) (ARGS!ARG REFINED))
                              (T (AND!CREATE REFINED)))))
                     OUTCOME
                     NIL)))))
        ((OREXPR? TST)
         (COND (OUTCOME        ; recording a true OR test
                (ENV!ADD-TEST
                 ENVIRON
                 (ENV-EL!CREATE
                   (FALIST!SUBLIST
                     FTLIST
                     (LET
                      ((REFINED
                        (ARGS!REFINE (OR!ARGLIST TST) OUTCOME ENVIRON FTLIST)))
                      (COND ((ARGS!ONE? REFINED) (ARGS!ARG REFINED))
                            (T (OR!CREATE REFINED)))))
                   OUTCOME
                   NIL)))
               (T              ; recording a false OR test
                (ENV!RECORD-TESTS ENVIRON (OR!ARGLIST TST) OUTCOME FTLIST))))
       ((NOTEXPR? TST)         ; strip off NOT and toggle outcome
         (ENV!RECORD-TEST ENVIRON
                          (ARGS!ARG (FCALL!ARGLIST TST))
                          (NOT OUTCOME)
                          FTLIST))
        (T (ENV!ADD-TEST ENVIRON
                        (ENV-EL!CREATE (FALIST!SUBLIS FTLIST TST) OUTCOME NIL)))))
```

TABLE 3-12. Recording information on the environment.

so. Thus, whenever such a predicate is recorded, any implications are recorded as well. This is done by the function ENV!ADD-TEST.

3.3.2.3. *The Index of Bindings*

The current index of bindings is maintained in the variable FTLIST. This index associates with each current internal lambda variable its value in terms of top level variables. More informally, it is an association list of dummy arguments and actual arguments. It provides for the virtual substitution of actual arguments for formal arguments within any number of nested lambda expressions, so that information recorded at one level is accessible by transformations performed at deeper levels. Let's look at an example.

```
(DEFUN MYSEARCH (ENV TST)
       (COND ((EMPTY? ENV) NIL)
             ((EQUAL TST (GET-TST (GET-ENTRY ENV))) (GET-ENTRY ENV))
             (T (MYSEARCH (GET-REST ENV) TST))))

(DEFUN EMPTY? (ENVIRON)
       (NULL ENVIRON))

(DEFUN GET-ENTRY (ENVIRONMENT)
       (COND ((NULL ENVIRONMENT)
              (ERROR '|Can't ask for element of empty environment|))
             (T (CAR ENVIRONMENT))))
```

Suppose that a set of optimizing transformations is applied to the code above. When MYSEARCH is optimized, procedure integration of EMPTY? will occur in the first clause of the conditional. While transforming the remaining clauses, then, it is known that (NULL ENV) is false. (We always record the transformed version of the predicate, not the untransformed version, as the former may be simpler; if it is, no information will be lost by recording the simpler version since that version will have been derived from information already on the environment.) During transformation of the second clause, procedure integration of GET-ENTRY will take place. The definition of MYSEARCH will then look like:

```
(DEFUN MYSEARCH (ENV TST)
       (COND
        ((NULL ENV) NIL)
        ((EQUAL
            TST
            (GET-TST
               ((LAMBDA (ENVIRONMENT)
                        (COND
                         ((NULL ENVIRONMENT)
                          (ERROR
                           '|Can't ask for element of empty environment|))
                         (T (CAR ENVIRONMENT))))
                ENV)))
         (GET-ENTRY ENV))
```

```
(DEFUN FCALL!EXPAND (FCALL ENVIRON FTLIST)
       (FCALL!PREDICATE FCALL)
       (ARGLIST!EXPAND (FCALL!ARGLIST FCALL) ENVIRON FTLIST) ; Expand the arguments
       (COND ((FUNCNAME? (FCALL!FUNC FCALL))
              (FUNCNAME!XFORM (FCALL!FUNC FCALL) ENVIRON FTLIST)))
       ;; note that because the above transformer side-effects, a given fcall
       ;; may pass both the previous and following tests. (i.e. FUNCNAME!SUB-DEF
       ;; may make a defcell out of the funchead of the fcall.)
       (COND ((DEFETTE? (FCALL!FUNC FCALL))
              (DEFETTE!EXPAND (FCALL!FUNC FCALL)
                              ENVIRON
                              (FALIST!ADD-PAIRS  ; the new ftlist
                                 (FALIST!CREATE
                                    (BINDING!DARGLIST
                                       (DEFETTE!BINDING (FCALL!FUNC FCALL)))
                                    (FALIST!SUBLIS FTLIST (FCALL!ARGLIST FCALL)))
                                 FTLIST))
              (DEFCALL!XFORM FCALL ENVIRON FTLIST)))
       (COND ((NOT (FCALL? FCALL)) FCALL)  ; FCALL may no longer be type *FCALL*
             (T (FCALL!XFORM FCALL ENVIRON FTLIST))))
```

TABLE 3-13. Control function for the Function Call structure type.

```
          (T (MYSEARCH (GET-REST ENV) TST))))
```

While tranformation of the body of the internal lambda expression is taking place, information about the variable ENVIRONMENT may be gathered or requested. But only expressions within the lambda function can contribute to or benefit from that information, since ENVIRONMENT is a local variable. We need some means of communicating with variables outside the lambda function. That is, some means of translating between ENVIRONMENT and ENV. The index of bindings provides that for us. Before anything is recorded or searched for in the environment, it is translated into the equivalent expression using top level variable names, by means of the index of bindings. This index is updated whenever an internal lambda function is entered. Thus, for example, the index is empty during transformation of the outer conditional above, but associates the dummy variable ENVIRONMENT with the actual argument ENV during transformation of the inner conditional (which occurs inside the lambda function).

When an internal lambda expression is encountered, the index of bindings is updated as follows:

(1) Using the current index of bindings, translate the actual arguments of the new lambda expression into strictly top level variable references (since they may contain references to earlier internal lambda expression variables).

(2) Create an association list of the dummy arguments used in the new lambda expression, and the translated actual arguments from (1).

(3) Append the association list created in (2) onto the *front* of the current index of bindings, to create the new index of bindings. (The association list from (2) must go on the front so that if any lambda variable names are reused, the new definitions will shadow the old ones on the index of bindings.)

The code which performs these steps is part of the definition of FCALL!EXPAND, which can be seen in Table 3-13.

Why are we limited to the *virtual* substitution made possible by the index of bindings, as opposed to just always substituting actual arguments for formals and therefore avoiding the communication problem? Because only transformations have the prerogative of performing substitutions in the code. The environment and its index is a compendium of information provided as a service to those transformations which require global knowledge of the expression they are operating on. A transformation may or may not perform a substitution, but that's none of the business of the control mechanism.

3.4. The Transformations

To complete our discussion of the transformational component, I need to discuss several aspects of transformations. In order for someone to write their own transformation for the system, they must know

(a) the syntactic form in which the system expects to get the transformation,

(b) in which transformation set to put the transformation, and

(c) the expected inputs and outputs of the transformation.

3.4.1. Form

What constitutes a transformation? In previous sections I have used a substitution rule form for transformations, such as (NULL NIL) => T. While it is possible to devise a system in which transformations are actually given in that form, it involves some complex pattern matching capabilities. Consider the software necessary to find an application of the transformation:

```
(COND <clauses-1> (NIL <exitform>) <clauses-2>)
                         =>    (COND <clauses-1> <clauses-2>)
```

where <clauses-1> and <clauses-2> can match any number of (or zero) clauses. While such pattern matching is possible and certainly has been done before [Boyle 1970], it requires sophisticated control mechanisms (and we're trying to keep our control simple, remember?). Furthermore, not all transformations are easily expressed as substitutions. In particular, global

```
(DEFUN FCALL!SIMPLIFY (FCALL)
       (COND ((AND (CAREXPR? FCALL)
                   (CONSEXPR? (FCALL!FIRST-ARG FCALL)))
              (RESPOND T (FCALL!FIRST-ARG (FCALL!FIRST-ARG FCALL))))
             (T (RESPOND NIL FCALL))))
```

TABLE 3-14. Implementation of `(CAR (CONS <X> <Y>)) =><X>`.

transformations cannot be expressed in that way since their applicability conditions may be more complex than a simple pattern match.

One might consider beefing up a substitution rule form to include other information necessary to describe the transformation. Pre- and post- assertions could be tacked on, as well as predicates to determine whether or not this transformation will really buy anything (an evaluation of its "goodness" in this situation, called "win" predicates in [Loveman 1977]), and perhaps even a list of transformations to try next. There are all sorts of possibilities. However, for my current purposes I want to concentrate on accountability and keep the control for the transformational component very simple. Therefore, I have opted for transformations that know how to apply themselves (otherwise known as procedures).

In our system, once a transformation has been called it is responsible for figuring out whether it applies or fails, and for returning a result to the control function which called it. In this thesis, I refer to the code which implements a transformation as a "transformer". Table 3-14 shows the transformer which implements the transformation `(CAR (CONS <X> <Y>)) => <X>`. The control function passes to the transformer the expression to be transformed, and the transformer responds with either `T` or `NIL`, indicating whether or not it succeeded, followed by the resultant expression.

One of the benefits of implementing transformations as procedures is that a single transformer may implement more than one transformation rule. If similar rules are grouped together in one transformer they can share some of the work needed to determine if they apply. The transformer `FCALL!SIMPLIFY` given in Table 3-14 is expanded in Table 3-15 to include the following transformation rules:

```
(CAR (CONS <A> <B>))  =>  <A>
(CAR (LIST <A> ...))  =>  <A>
(CDR (CONS <A> <B>))  =>  <B>
(CDR (LIST <A> <B> ...))  =>  (LIST <B> ...)
(APPEND NIL <X>)  =>  <X>
(APPEND <X> NIL)  =>  <X>
```

```
(DEFUN FCALL!SIMPLIFY (FCALL)
       (COND
        ;; (CAR (CONS <A> <B>))  ->  <A>
        ;; (CAR (LIST <A> ...))  ->  <A>
        ((AND (CAREXPR? FCALL)
              (NOT (ATOM? (FCALL!FIRST-ARG FCALL))))
         (COND ((CONSEXPR? (FCALL!FIRST-ARG FCALL))
                (RESPOND T
                         (FCALL!FIRST-ARG (FCALL!FIRST-ARG FCALL))
                         (JUST!CREATE '|(CAR (CONS <A> <B>))  ->  <A>| NIL)))
               ((LISTEXPR? (FCALL!FIRST-ARG FCALL))
                (RESPOND T
                         (FCALL!FIRST-ARG (FCALL!FIRST-ARG FCALL))
                         (JUST!CREATE '|(CAR (LIST <A> ...))  ->  <A>| NIL)))
               (T (RESPOND NIL
                           FCALL
                           (JUST!CREATE '|it isn't CAR of CONS or LIST.| NIL)))))
        ;; (CDR (CONS <A> <B>))  ->  <B>
        ;; (CDR (LIST <A> <B> ...))  ->  (LIST <B> ...)
        ((AND (CDREXPR? FCALL)
              (NOT (ATOM? (FCALL!FIRST-ARG FCALL))))
         (COND ((CONSEXPR? (FCALL!FIRST-ARG FCALL))
                (RESPOND T
                         (FCALL!SECOND-ARG (FCALL!FIRST-ARG FCALL))
                         (JUST!CREATE '|(CDR (CONS <A> <B>))  ->  <B>| NIL)))
               ((LISTEXPR? (FCALL!FIRST-ARG FCALL))
                (XFORM-SLAVE 'SEXPR!NOTHING
                             (FCALL!FIRST-ARG (FCALL!FIRST-ARG FCALL))
                             (LIST (JUST!CREATE '|it was CDR'd over.| NIL))
                             FCALL)
                (RESPOND
                 T
                 (FCALL!FIRST-ARG FCALL)
                 (JUST!CREATE '|(CDR (LIST <A> <B> ...))  ->  (LIST <B> ...)| NIL)))
               (T (RESPOND NIL
                           FCALL
                           (JUST!CREATE '|it isn't CDR of CONS or LIST.| NIL)))))
        ;; (APPEND NIL <X>)  ->  <X>
        ((AND (APPENDEXPR? FCALL)
              (NULL? (FCALL!FIRST-ARG FCALL)))
         (RESPOND T
                  (FCALL!SECOND-ARG FCALL)
                  (JUST!CREATE '|(APPEND NIL <X>)  ->  <X>| NIL)))
        ;; (APPEND <X> NIL)  ->  <X>
        ((AND (APPENDEXPR? FCALL)
              (NULL? (FCALL!SECOND-ARG FCALL)))
         (RESPOND T
                  (FCALL!FIRST-ARG FCALL)
                  (JUST!CREATE '|(APPEND <X> NIL)  ->  <X>| NIL)))
       (T (RESPOND NIL FCALL (JUST!CREATE '|no patterns matched.| NIL)))))
```

TABLE 3-15. A transformer may implement several transformations.

Transformers are not allowed to call other transformers. Each transformer is expected to be independent of any other. The reason for this is that confusion would reign if a user removed a transformer from a transformation set (as he should be free to do) which was called by another he left in. Furthermore, the function XFORM, which calls a transformer and receives its response, is dependent on the fact that no other transformer is called in the meantime.

3.4.2. Selecting a Transformation Set

For the most part, selecting the proper transformation set in which to put a transformer should be a simple task. A transformer which replaces an AND expression of no arguments with the atom T should of course be put in the *AND* set, a transformer to simplify the expression (APPEND FOO NIL) clearly belongs in the *FCALL* set.

There are a few subtleties in the decision brought on by the fact that some structures betray their context by their type. The type name *CLAUSE*, for example, gives away the fact that it represents a subtree found only as an argument to a *COND*. A *CLAUSE* structure will always have a *COND* structure as a father. Should we allow transformations to determine the context of a structure by checking its type? Consider the following transformation rule:

```
(1)  (COND <clauses-1> (T (COND <clauses-2>)))
                         => (COND <clauses-1> <clauses-2>)
```

The final clause of the outer COND is being replaced by the clauses of the inner COND. That is, one *CLAUSE* subtree is being replaced by many. Should the transformer which implements this rule be a member of the *COND* transformation set, or could we include the transformer in the *CLAUSE* set by rewriting it?

```
     (T (COND <clauses>)) => <clauses>
```

Before I answer the question, consider another similar example:

```
(2)  (APPEND <X> (APPEND <Y> <Z>)) => (APPEND <X> <Y> <Z>)
```

If we put the first transformer in the *CLAUSE* transformation set, shouldn't we also rewrite this transformer as:

```
     (APPEND <Y> <Z>) => <Y> <Z>
```

and put it in the *FCALL* transformation set? Obviously we can't, since the fact that (APPEND <Y> <Z>) is of type *FCALL* does not tell us its context; we have no way of knowing whether it appears as the argument to another APPEND or not.

A second reason for keeping the first transformer (and others like it) in the higher level *COND* transformation set is for the sake of the person who writes the transformations. Why should that person need to be aware that non-evaluable structure types exist? I have therefore decided that transformations which require a specific context should appear in the set corresponding to that level of context.

3.4.3. Interfacing to the Control Functions

Transformers hook into the control functions by means of transformation sets. Recall from section 3.3.1.3 that we have a different set of transformations for each structure type. Though it would have been simple to have each set be a LISP list of the names of the appropriate transformers (and then map over that list), I saw no reason why I shouldn't just implement each set as an implied PROGN. Thus, the function definitions shown in Table 3-7 are actually transformation sets. A transformer may be added or removed simply by inserting or deleting the call from the set, remembering that the order in which the calls occur in a set is the order in which those transformers will be applied.

Notice that all transformers are called via the function XFORM, and all return information via the function RESPOND. Currently these two functions do nothing more than a FUNCALL and return a LIST of their arguments, respectively (and then upon receipt of the response, XFORM causes the result to be substituted for the original code by side-effecting the internal form), but in the future we will cause more complex things to happen. In any case, I here establish the convention that:

(1) No transformer may be called directly, but via XFORM only.

(2) No transformer may respond directly, but via RESPOND only.

The significance of this is that a transformation which iterates within itself must break one of two rules. If it calls itself again via XFORM, it is breaking the rule that no transformer may call another (remember that XFORM is dependent on the fact that no transformer is called between the time XFORM calls one and receives its response). But if it calls itself without the use of XFORM, it breaks the rule stated above. The solution is in the way we view the problem. I shall define a transformation which is implemented via an iterating transformer to be incomplete until the iteration ends. Thus, while a transformer may call itself directly, it does so only with the understanding that when it finishes, *one* transformation has been applied, regardless of the number of iterations the transformer made.

To call a tranformer which requires global information, the programmer simply includes in the call as additional arguments the environment and the index of bindings. These are usually bound to ENVIRON and FTLIST, respectively, within each transformation set. Compare the calls to AND!TRIM and AND!SIMPLIFY in the transformation set for the *AND* expression structure type given in Table 3-7.

3.4.4. Slave Transformers

There are a number of cases which arise when writing transformations in which changes must be made only to a subtree of the input expression, even though the entire expression was needed in order to determine whether the transformation should apply. Consider the transformation in which actual arguments are substituted for dummy arguments in a lambda expression. The code to be transformed might look like:

```
((LAMBDA (FOO BAR)
         (COND ((NULL FOO) BAR)
               (T (CONS FOO BAR))))
  (QUUX S) S)
```

The transformer decides after looking at the complexity of the actual arguments ((QUUX S) and S) and the use of the dummy arguments in the lambda expression, that S should be substituted for every occurrence of BAR. The resultant expression to be returned will appear as:

```
((LAMBDA (FOO)
         (COND ((NULL FOO) S)
               (T (CONS FOO S))))
   (QUUX S))
```

Only the following expressions are affected:

- the top level *FCALL* expression which now calls a function of only one argument.
- each of the *ATOM* structures representing an occurrence of BAR, both in the dummy argument list of the lambda expression and in its body.
- the *ATOM* structure representing the original occurrence of S as one of the arguments to the top level *FCALL*.

We wish to localize the transformation so that only those sub-expressions affected by the transformation are transformed. Thus, I have invented the notion of slave transformations, which may be called by a transformer to change some subtree of the input expression. Slave transformers have their own XFORM-SLAVE and SLAVE-RESPOND functions which operate much the same as their counterparts, with the exception that the successful application of a slave transformer does not increment the clock. This allows the above described transformation to appear to happen all at once as far as transformation time is concerned; that is, it constitutes *one* transformation.

Another use for slave transformers is for the recording of success or failure for transformers which iterate. Recall from section 3.4.3 that the application of an iterating transformer is said to constitute only one transformation. That means that regardless of the number of iterations, only one history may be updated with success or failure. Slave transformers allow success or failure to be recorded for each iteration. This use (and perhaps other uses as well) of slave transformers may result in no change being made to the input expression other than transformations upon its sub-expressions.

A slave transformer should be used to perform the following transformation:

```
(COND <clauses-1> (T (COND <clauses-2>)))
                      => (COND <clauses-1> <clauses-2>)
```

The final clause of the outer conditional is being replaced by the clauses of the inner conditional. That is, one *CLAUSE* structure is being replaced by many. As discussed in section 3.4.2, this transformer appears in the *COND* transformation set. Rather than create and return a new list of the correct clauses, the transformer should call a slave transformer on the final clause of the outer conditional above which returns the list of *CLAUSE* structures <clauses-2>. This strategy assures that transformations occur as locally as possible, so that expressions which are not changed remain untouched by the system (a principle we shall understand better in the next chapter).

As another example of the locality of transformations, consider the rule:

```
(COND <clauses-1> (NIL <exitform>) <clauses-2>)
                      =>    (COND <clauses-1> <clauses-2>)
```

This transformation deletes a clause which will never be evaluated, because its predicate is known to be false. No other clause is affected by the transformation. The transformer which implements this transformation should appear in the *COND* transformation set because it requires the specific context of a conditional expression to apply. In order to preserve the locality of the transformation, the transformer calls a slave transformer on the clause with the false predicate. The slave transformer returns a "nothing" flag, causing the deletion of the clause from the parse tree of the conditional expression.

3.5. Summary

I have kept the transformational component of the system as simple as possible. It is written in LISP, and is able to control the transformation of applicative LISP code only. SEXPR!EXPAND is the driver control function for the system, and will accept as input any evaluable LISP expression, along with the current environment and list of bindings (both of which are usually empty at the very beginning). This function parses the expression and hands it over to the specific control function for its structure type. The expression is then transformed in the same order as it would be evaluated by a LISP interpreter: an ordered set of transformations is applied to the top level of the expression only after its parts have been transformed. Whenever a conditional expression, AND, OR, or an ASSERT is encountered, the environment is updated; whenever an internal lambda function is encountered, the index of bindings is updated. These two sources of information are maintained as a service to allow global transformations to apply. All transformation occurs via the function XFORM only, and transformations return information via the function RESPOND only. Slave transformations should be used to maintain the local transformation of LISP structures.

Chapter Four

The Accountable Component

HAVING A COMPLETE IMPLEMENTATION of the transformational component we can now implement an accountable component to accompany it. I purposely discussed the design of an accountable system *before* the implementation of the transformational component so that I could not be accused of limiting my plans for the system. Many of the implementation decisions made in the last chapter in the name of simplicity were made not to avoid issues in the implementation of an accompanying accountable component, but to save time and effort in implementing the transformational component. As will be demonstrated, the accountable component is largely independent of the implementation of the transformational component.

The issues we must resolve in the remainder of the thesis include:

- acquisition of information,
- storage of information, and
- retrieval of information.

The first two points fall under the jurisdiction of the recording element. In particular, after reviewing the nature of the information we have decided to collect as discussed in section 2.4, we must rewrite the definitions of the functions XFORM and RESPOND to obtain the information. Then we will discuss a means of storing this information by augmenting the internal form of the argument program.

The last item on the agenda is the responsibility of the query element. Here we must finally retrieve the information at the command of the user. We will see that merely offering to "replay" specified sections of the transformation process can be fairly useful, though trying to outguess the user by providing the answers to more general questions might be more interesting.

4.1. Acquisition of Information

Recall from section 2.5.1 our summarization of the recording element's task:

- **Transformations must include in their description a set of input and output assertions, available upon request.**

  The motivation for this requirement was to couple this information with the truths for a particular piece of code in order to determine why a transformation failed or succeeded. For example, if a transformation T requires properties a, b, and c, but only a and b are true at some point, we may assume that T failed because either c was false or unknown.

  Since we have chosen to implement transformations as procedures, they can be made to return the reason they failed or succeeded themselves. We will therefore let the procedure definition stand as the description of input and output assertions for a transformation, and require that transformers return justifications for their actions.

- **Before any transformations apply, a set of truths for each piece of code (dependent on its context) must be collected and recorded along with their justifications.** That is, before any transformation applies to *a particular piece of code*, truths for that code must be recorded.

  This is the environment, and we have discussed it in the last chapter. We do need to add the justifications; we will require that ENV!RECORD-TEST be given a justification for the truth it is to record, and record that justification along with the test. If ENV!RECORD-TEST does any fancy logic it should augment the justification *it is given with a note to that effect. Similarly,* ENV!ADD-TEST should justify any truth it adds to the environment.

  Though the environment and its index of bindings could have been stored as part of the internal form of the program during the implementation of the transformational component, they were instead created and maintained as separate variables which were passed from control function to control function, and finally to the transformation set to be used by the transformers. The accountable component, however, must record the environment and index used for the transformation of each expression so that its information is not lost once the transformation set is exited, as is the case now. It will do this immediately after a transformation set is entered.

- **After each transformation is attempted, it must provide to the recording element the following information:**

  (1) what code it attempted to transform;

  (2) whether or not it was successful;

  (3) if it was successful, the resultant code.

Since all transformations are called via the function XFORM, and XFORM is responded to via RESPOND, XFORM must already know (1), since XFORM obviously passes the transformation the

```
;;;Transformer
(DEFUN OR!SIMPLIFY (ORTST)
       (COND ((ARGS!NULL? (OR!ARGLIST ORTST))
               (RESPOND T (SEXPR!COPY FALSE) (JUST!CREATE '|(OR) => NIL| NIL)))
              ((ARGS!ONE? (OR!ARGLIST ORTST))
               (RESPOND T
                        (ARGS!ARG (OR!ARGLIST ORTST))
                        (JUST!CREATE '|(OR <x>) => <x>| NIL)))
              (T (RESPOND NIL ORTST (JUST!CREATE '|no patterns match.| NIL)))))

;;;Transformer
(DEFUN FCALL!SIMP-NOT (NULLTST)
       (COND ((NOTEXPR? NULLTST)
               (LET ((LOGARG (ARGS!ARG (FCALL!ARGLIST NULLTST))))
                    (COND ((NULL? LOGARG)
                           (RESPOND T
                                    (SEXPR!COPY TRUE)
                                    (JUST!CREATE '|(NOT NIL) => T| LOGARG)))
                          ((TRUE? LOGARG)
                           (RESPOND T
                                    (SEXPR!COPY FALSE)
                                    (JUST!CREATE '|(NOT T) => NIL| LOGARG)))
                          (T (RESPOND NIL
                                      NULLTST
                                      (JUST!CREATE
                                        '|the arg to NOT or NULL is unknown.|
                                        NIL))))))
              (T (RESPOND NIL NULLTST (JUST!CREATE '|no patterns match.| NIL)))))
```

TABLE 4-1. Transformations must return justifications.

```
(DEFUN RESPOND (APPLIED? OBJECT JUST)
       (COND ((NOT APPLIED?) (LIST **RESPONSE** **FAILED** JUST))
              (T (LIST **RESPONSE** OBJECT JUST))))
```

TABLE 4-2. New definition of RESPOND.

code to transform. So there is no need for a transformation to explicitly return that information. Currently a transformer already supplies the information required by (2) and (3): if it fails it executes the form (RESPOND NIL <input expression>), and if it succeeds, it executes the form (RESPOND T <resultant expression>).

Transformations are required to return a three-valued response, then: a flag signifying whether or not they succeeded (T or NIL, respectively), the resultant expression, and a justification for the success or failure. The new definition of RESPOND may be seen in Table 4-2.

4.1.1. Justifications

A justification consists of an explanation, and optionally, an expression. It may either justify a transformer which succeeds or fails, or justify the addition of a piece of common contextual information to the environment. In the first case, its explanation is a brief phrase which completes the sentence "This expression was transformed because..." or the sentence "The transformer <transformer> failed to transform this expression because...". In the second case, its explanation should complete the sentence "This expression's value is <value> because...". If the explanation refers to some other expression (such as "...it is in the true branch of the conditional clause:"), then that expression may be included in the justification and will be printed when the justification is displayed.

The expression component of a justification may also serve as a link to previous justifications. Suppose `ENV!RECORD-TEST` is asked to record the fact that `(AND S T)` is true because the system is about to transform expressions which are "within the true branch of the conditional clause" whose predicate is that expression. Then `ENV!RECORD-TEST` will make two separate entries: that `S` is true and that `T` is true, because each of them is "an argument to a true `AND` expression." Later, if a transformation succeeds because `S` is true, it may return the justification associated with `S` as its justification. The justification of `S` as recorded by `ENV!RECORD-TEST` is "an argument to a true `AND` expression"; there is no clue left as to why the `AND` expression is true! I solved this problem in the following manner:

- `ENV!RECORD-TEST` will now record the predicate and justification exactly as it receives them in addition to recording any pieces of the predicate. This leaves a record as to why the `AND` expression is true in the above example.
- When pieces of a predicate are recorded, they are given a justification which consists of an explanation specifying the nature of the predicate piece ("argument of a true `NOT` expression", "argument of a false `OR` expression", etc), and the expression representing the predicate the piece came from.

This will allow the query element to perform some simple dependency "backlooking": When a user wishes to know why `S` is true, it can print the explanation component of the justification of `S`, followed by the expression component, then the justification recorded on the environment for *that*, and so on, until the originally asserted expression is found. The new definition of `ENV!RECORD-TEST` is shown in Table 4-3.

The function `XFORM` now has all the information relating to the transformation of an expression, and only needs to store that information so that (1) future transformations will see the correct expression, and (2) the query element can obtain any intermediate version of the expression it desires At the same time, the recording element must represent this information "in an efficient and accessible manner."

```
(DEFUN ENV!RECORD-TEST (ENVIRON TST OUTCOME JUST FTLIST)
       (COND
        ((NULL TST) ENVIRON)
        ((ATOM? TST)
         (ENV!ADD-TEST ENVIRON
                       (ENV-EL!CREATE (FALIST!SUBLIS FTLIST TST) OUTCOME JUST)))
        ((ANDEXPR? TST)
         (COND (OUTCOME         ; recording a true AND test
                (ENV!RECORD-TESTS
•                 (ENV!ADD-TEST ENVIRON
•                               (ENV-EL!CREATE (FALIST!SUBLIS FTLIST TST) T JUST))
                  (AND!ARGLIST TST)
                  T
•                 (JUST!CREATE '|Argument to a true AND expr.| TST)
                  FTLIST))
               (T (ENV!ADD-TEST ; recording a false AND test
                    ENVIRON
                     (ENV-EL!CREATE
                       (FALIST!SUBLIS
                         FTLIST
                         (LET
                          ((REFINED
                            (ARGS!REFINE (AND!ARGLIST TST) OUTCOME ENVIRON FTLIST)))
                          (COND ((ARGS!ONE? REFINED) (ARGS!ARG REFINED))
                                (T (AND!CREATE REFINED)))))
                       OUTCOME
•                      JUST)))))
        ((OREXPR? TST)
         (COND (OUTCOME        ; recording a true OR test
                (ENV!ADD-TEST
                  ENVIRON
                  (ENV-EL!CREATE
                    (FALIST!SUBLIS
                      FTLIST
                      (LET
                       ((REFINED
                         (ARGS!REFINE (OR!ARGLIST TST) OUTCOME ENVIRON FTLIST)))
                       (COND ((ARGS!ONE? REFINED) (ARGS!ARG REFINED))
                             (T (OR!CREATE REFINED)))))
                    OUTCOME
•                   JUST)))
               (T              ; recording a false OR test
                (ENV!RECORD-TESTS
•                (ENV!ADD-TEST ENVIRON
•                              (ENV-EL!CREATE (FALIST!SUBLIS FTLIST TST) NIL JUST))
                 (OR!ARGLIST TST)
                 OUTCOME
•                (JUST!CREATE '|Argument to a false OR expr.| TST)
                 FTLIST))))
        ((NOTEXPR? TST)         ; strip off NOT and toggle outcome
         (ENV!RECORD-TEST
•          (ENV!ADD-TEST ENVIRON
•                        (ENV-EL!CREATE (FALIST!SUBLIS FTLIST TST) OUTCOME JUST))
           (ARGS!ARG (FCALL!ARGLIST TST))
           (NOT OUTCOME)
•          (JUST!CREATE '|Argument to a NOT expression.| TST)
           FTLIST))
        (T (ENV!ADD-TEST ENVIRON (ENV-EL!CREATE
                                  (FALIST!SUBLIS FTLIST TST) OUTCOME JUST)))))
```

TABLE 4-3. The environment must now maintain justifications for its entries.

4.2. Storage of Information

Before we consider a means of storage for the information in hand, let's think quickly about the nature of its retrieval. A programmer who submits his code to the accountable system will be interested in observing its effect on his program. Thus, he will ask questions about specific sections of code. One who has written a new transformation may test it with a random program, and then ask questions pertaining to the actions of that transformation. And someone who is concerned with the order of transformation application will ask questions about what happened at what time. There are then three possible indicies: the code transformed, the name of the transformer, or the time of transformation. We will use them all.

4.2.1. By Code

If the transformational component of the system were to be run without any accountable capabilities, it could simply replace any subtree of the parse tree with the new one whenever a transformation applied. But the accountable component will remember every intermediate version of the program. If there are fifty different versions (because there were fifty successful transformation applications), storing fifty different parse trees is clearly not efficient, whether the program is large or small. Instead, I have chosen to augment the internal form of the parse tree by allowing *multiple versions* of each subtree. When a transformation applies, the resulting subtree is added to the parse tree as version $n + 1$. We can write a special printer which, given the entire tree and a version number, will print the appropriate subtree when it arrives at a node in the parse tree which has more than one. This printer really constitutes a translator or conversion routine between the internal form and the executable source code.

At each node of the parse tree is a structure called a CELL. Cells contain (as well as other information which we will mention later) a list of the different subtrees for this node; initially this list contains only one subtree. Each subtree is represented by a structure called a **MONK**. Every successful transformation causes a **MONK** to be created. Because we want to record unsuccessful transformations as well, we will create a structure called a HERMIT whenever one fails. Besides the subtree it represents, each **MONK** contains a version number, the name of the transformer which created it, and the justification for that transformation. Each HERMIT contains the version number of the subtree which wasn't transformed, the name of the transformer which didn't transform it, and a justification for the failure.

```
(DEFUN XFORMER!RECORD (XFORMER CELL VERSION)
       (PUTPROP XFORMER (CONS (CONS VERSION CELL)
                              (GET XFORMER 'APPLIED)) 'APPLIED))

(DEFUN XFORMER!RECORDF (XFORMER CELL VERSION)
       (LET ((FAILS (GET XFORMER 'FAILED)))
            (COND ((NULL FAILS)
                   (PUTPROP XFORMER (LIST (CONS VERSION CELL)) 'FAILED))
                  ((> (CAAR FAILS) VERSION)
                   (ERROR '|Fails out of order.| XFORMER))
                  ((< (CAAR FAILS) VERSION)
                   (PUTPROP XFORMER (CONS (CONS VERSION CELL) FAILS) 'FAILED))
                  ((MEMQ CELL (CDAR FAILS)) NIL)
                  (T (PUTPROP XFORMER
                              (CONS (CONS VERSION (CONS CELL (CDAR FAILS)))
                                    (CDR FAILS))
                              'FAILED)))))
```

TABLE 4-4. Functions which record information by transformer name.

4.2.2. By Transformer

If someone were interested in the activities of transformer T, pointing them to each CELL transformed by T and giving them the time of each transformation would be enough to allow them to then ask for any other information that might be stored in the cells. XFORMER!RECORD takes the name of a transformer, the argument cell, and the current time, and records the fact that that transformer affected the said cell at the given time. XFORMER!RECORDF records the same information for transformations that fail. (The definitions of these two functions are given in Table 4-4.) Note that this is a record of whether or not a *transformer* succeeds. Recall that a transformer can implement more than one transformation rule. If any rule succeeds, the transformer is said to succeed. Only if *all* the rules implemented by a transformer fail does the transformer fail.

4.2.3. By Version

We can index the information by version[1] in much the same way as by transformer. CLOAKRACK!HANG-UP maintains an association list of transformation times and cells transformed at those times (called the "cloakrack", because a monk hangs up its cloak before entering the cell). RAGRACK!HANG-UP maintains a similar association list for transformations which fail. Its definition differs in that many transformations may fail during the same transformation time since the clock is not incremented until a transformation succeeds. I claim that it is relatively unimportant to keep track of which transformation fails before another, because no subtree changes. If

[1] I will use the terms "version" and "transformation time" interchangeably, since version *n* of a program is that parse tree of the program which was created at transformation time *n*, with one exception: version 0 is created not by transformation but by conversion to the internal form.

```
(DEFUN CLOAKRACK!HANG-UP (CELL VERSION)
       (SETQ CLOAKRACK (CONS (LIST VERSION CELL) CLOAKRACK)))

(DEFUN RAGRACK!HANG-UP (CELL VERSION)
       (COND ((NULL RAGRACK) (SETQ RAGRACK (LIST (CONS VERSION CELL))))
             (T (LET ((V (CAAR RAGRACK)))
                     (COND
                       ((> V VERSION)
                        (ERROR '|Ragrack out of order.|))
                       ((< V VERSION)
                        (SETQ RAGRACK (CONS (LIST VERSION CELL) RAGRACK)))
                       ((MEMQ CELL (CDAR RAGRACK)) NIL)
                       (T (SETQ RAGRACK (CONS (CONS V (CONS CELL (CDAR RAGRACK)))
                                              (CDR RAGRACK)))))))))
```

TABLE 4-5. Definitions of functions which record information by version.

```
(DEFUN XFORM (XFORMER CELL ARGS)
       (LET ((RESPONSE (APPLY XFORMER (CONS CELL ARGS))))
            (COND ((APPLIED? RESPONSE)
                   (CELL!STORE-MONK CELL (RESPONSE!HACK RESPONSE CELL XFORMER))
                   (XFORMER!RECORD XFORMER CELL XFORMTIME)
                   (CLOAKRACK!HANG-UP CELL XFORMTIME)
                   (XFORMTIME!INCR))
                  (T (CELL!STORE-HERMIT CELL (RESPONSE!HACKF RESPONSE XFORMER))
                     (XFORMER!RECORDF XFORMER CELL XFORMTIME)
                     (RAGRACK!HANG-UP CELL XFORMTIME)))))
```

TABLE 4-6. Final definition of XFORM.

transformers T-1 and T-2 both attempt to transform expression E and both fail, it is immaterial which failed first. Expression E remains the same before, after, and during both attempts. The definitions of CLOAKRACK!HANG-UP and RAGRACK!HANG-UP are shown in Table 4-5.

4.2.4. The Final Version of XFORM

We can now complete our definition of XFORM by adding calls to functions which will store the collected information. Look at Table 4-6. RESPONSE!HACK takes the response, the cell which was transformed, and the name of the transformer, and creates a MONK which is then stored in the transformed cell. After creating cross-references to this information by transformer and transformation time, the "clock" is incremented in order to be ready to provide the time (version) for the next transformation. If a transformation fails, first RESPONSE!HACKF creates a HERMIT with information pertaining to the attempt, and returns the HERMIT for storage in the cell. Then the information is again cross-referenced by transformer and transformation time, but the clock is *not* incremented. The definitions of RESPONSE!HACK and RESPONSE!HACKF will be discussed later.

```
(DEFUN SEXPR!CONVERT-TO-X (SEXPR)
       (COND ((CELL? SEXPR) SEXPR)
             ((ATOM? SEXPR) (ATOM!CONVERT-TO-X SEXPR))
             ((DEFETTE? SEXPR) (DEFETTE!CONVERT-TO-X SEXPR))
             ((QUOTE? SEXPR) (QUOTE!CONVERT-TO-X SEXPR))
             ((COND? SEXPR) (COND!CONVERT-TO-X SEXPR))
             ((ANDEXPR? SEXPR) (AND!CONVERT-TO-X SEXPR))
             ((OREXPR? SEXPR) (OR!CONVERT-TO-X SEXPR))
             (T (FCALL!CONVERT-TO-X SEXPR))))

(DEFUN ARGS!CONVERT-TO-X (ARGLIST)
       (COND ((NULL ARGLIST) (LIST (NOTHING!CREATE)))
             ((CELL? (ARGS!ARG ARGLIST)) ARGLIST)
             (T (MAPCAR (FUNCTION SEXPR!CONVERT-TO-X) ARGLIST))))

(DEFUN ATOM!CONVERT-TO-X (AT)
       (COND ((CELL? AT) AT)
             (T (CELL!CREATE AT **ATOM** 'ATOM!CONVERT-TO-X))))

(DEFUN BINDING!CONVERT-TO-X (DARGS)
       (COND ((CELL? DARGS) DARGS)
             (T (BINDING!CREATE (ARGS!CONVERT-TO-X DARGS)))))

(DEFUN DEFETTE!CONVERT-TO-X (DEF)
       (COND ((CELL? DEF) DEF)
             (T (DEFETTE!CREATE (ATOM!CONVERT-TO-X (DEFETTE!NAME DEF))
                                (BINDING!CONVERT-TO-X (DEFETTE!BINDING DEF))
                                (SEXPR!CONVERT-TO-X (DEFETTE!BODY DEF))))))
```

TABLE 4-7. Functions for converting *to* the internal form.

4.3. A Look at Cells

A cell contains the "history" of a subtree of a parse tree. It charts the transformation of that subtree via its list of monks, each of which points to a different version. The subtree itself constitutes a parse tree though, so each of *its* subtrees are kept track of by cells and monks. The use of cells as the internal form of the argument program raises a number of issues regarding conversion to and the transformation of that form.

4.3.1. Conversion to the Internal Form

Transformations may operate only on cells, since XFORM expects to get a cell into which it can store a new monk. Thus, every subtree which might be a candidate for transformation must be converted to a cell. We have already decided which LISP structures to provide with transformation sets (in section 3.3.1.3). We will represent each of the structures whose types are defined by the

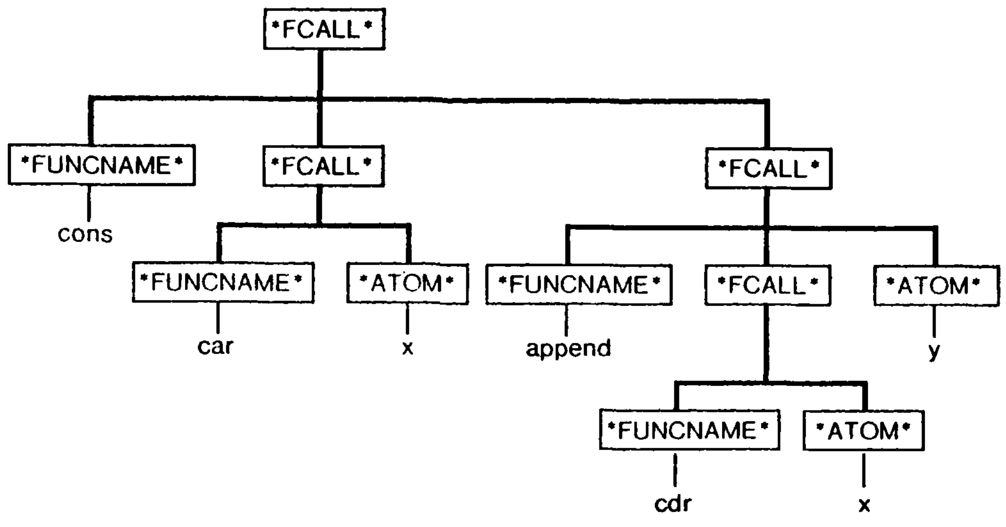


FIGURE 4-1. Cell representation of (CONS (CAR X) (APPEND (CDR X) Y)).

transformational component (and given in Table 3-5 (page 39)) as cells of the same type. Cells are used not only to represent LISP code, but to contain some useful information about that code as well.

4.3.1.1. *Cell Types*

The cell representation of the expression (CONS (CAR X) (APPEND (CDR X) Y)) is illustrated in Figure 4-1. Each atomic symbol of the expression is a leaf of the tree, and may be either of type *ATOM* or *FUNCNAME*.

What about the few remaining pieces of LISP code which are not typed? Consider the dummy argument list of a function definition, or the special form headers COND, AND, OR, QUOTE, etc. My decisions on these issues were made as follows:

(1) The dummy argument list will be represented as a cell of type *BINDING* even though transformations are unlikely to apply to the cell. Representing the list as a cell will cause a function definition (a *DEFETTE* cell) to consist of a list of three cells: *FUNCNAME*, *BINDING*, and a body which may be any evaluable structure type, and thus a cell itself.

(2) Special forms each have their own cell type, and thus the header itself need not be made a cell. Instead, each cell's monk contains a list of the arguments to the special form. Thus, a *COND* cell's monk contains a list of *CLAUSE* cells, an *AND* cell's monk contains a list

```
(DEFUN SEXPR!CONVERT-FROM-X (SEXPR &optional VERSION)
       (COND
        ((NOT (CELL? SEXPR)) SEXPR)
        (T (LET ((TYPE (SEXPR!TYPE SEXPR VERSION)))
                (COND
                  ((EQ **ATOM** TYPE) (ATOM!CONVERT-FROM-X SEXPR VERSION))
                  ((EQ **FUNCNAME** TYPE) (FUNCNAME!CONVERT-FROM-X SEXPR VERSION))
                  ((EQ **BINDING** TYPE) (BINDING!CONVERT-FROM-X SEXPR VERSION))
                  ((EQ **DEFETTE** TYPE) (DEFETTE!CONVERT-FROM-X SEXPR VERSION))
                  ((EQ **QUOTE** TYPE) (QUOTE!CONVERT-FROM-X SEXPR VERSION))
                  ((EQ **COND** TYPE) (COND!CONVERT-FROM-X SEXPR VERSION))
                  ((EQ **CLAUSE** TYPE) (CLAUSE!CONVERT-FROM-X SEXPR VERSION))
                  ((EQ **AND** TYPE) (AND!CONVERT-FROM-X SEXPR VERSION))
                  ((EQ **OR** TYPE) (OR!CONVERT-FROM-X SEXPR VERSION))
                  ((EQ **FCALL** TYPE) (FCALL!CONVERT-FROM-X SEXPR VERSION))
                  (T (ERROR '|Unknown type  --  SEXPR!CONVERT-FROM-X| SEXPR))))))

(DEFUN ARGS!CONVERT-FROM-X (ARGLIST *ARGLISTVERSION*)
       (DECLARE (SPECIAL *ARGLISTVERSION*))
       (COND ((ARGS!NULL? ARGLIST *ARGLISTVERSION*) NIL)
             ((CELL? (ARGS!ARG ARGLIST *ARGLISTVERSION*))
              (SEXPR!MAPCAR
                (FUNCTION (LAMBDA (EXPR)
                                  (SEXPR!CONVERT-FROM-X EXPR *ARGLISTVERSION*)))
                                  ARGLIST
                                  *ARGLISTVERSION*))
             (T ARGLIST)))

(DEFUN ATOM!CONVERT-FROM-X (AT VERSION)
       (COND ((CELL? AT) (SEXPR!TREE AT VERSION))
             (T AT)))

(DEFUN BINDING!CONVERT-FROM-X (BINDING VERSION)
       (COND ((NOT (CELL? BINDING)) BINDING)
             (T (BINDING!CREATE
                   (ARGS!CONVERT-FROM-X
                      (BINDING!DARGLIST BINDING VERSION) VERSION)))))

(DEFUN DEFETTE!CONVERT-FROM-X (DEF VERSION)
       (COND ((CELL? DEF)
              (DEFETTE!CREATE
                 (ATOM!CONVERT-FROM-X (DEFETTE!NAME DEF VERSION) VERSION)
                 (BINDING!CONVERT-FROM-X (DEFETTE!BINDING DEF VERSION) VERSION)
                 (SEXPR!CONVERT-FROM-X (DEFETTE!BODY DEF VERSION) VERSION)))
             (T DEF)))
```

TABLE 4-8. Functions for converting *from* the internal form.

of evaluable cells, etc. We are therefore allowing cell types to take the place of actual code. That is, COND, AND, OR, etc, will never appear as leaves of the parse tree.

The rather pleasant results of these two decisions is that the subtree contained in any cell's monk is either an atom, a cell, or a list of cells, with the exception of the *QUOTE* cell, whose monk may contain any bare s-expression. Whatever that s-expression is, the conversion

```
(DEFUN CELL!CREATE (SEXPR TYPE VERSION CREATOR)
       (LET ((ID (INTERN (MYGENSYM 'CELL))))
            (LET ((NEWCELL
                   (LIST **CELL**
                         ID
                         (LIST (BPLIST!CREATE)
                               (RANGE!CREATE VERSION VERSION)
                               (BVAL!CREATE NIL NIL NIL)
                               NIL  ; slot for environment
                               NIL) ; slot for index of bindings
                         (MONK!CREATE TYPE VERSION SEXPR CREATOR
                                      (JUST!CREATE '|First monk| NIL) NIL))))
                 (SET ID NEWCELL)
                 (CELL!INSTALL-BACKPOINTEES NEWCELL))))

(DEFUN MONK!CREATE (TYPE VERSION SEXPR XFORMER JUSTIFICATION PTCPOINTER)
       (LIST **MONK** TYPE VERSION SEXPR XFORMER JUSTIFICATION PTCPOINTER))
```

TABLE 4-9. The functions which create cells and monks.

routines won't touch it. Table 4-7 and Table 4-8 show some of the conversion functions. SEXPR!CONVERT-TO-X and SEXPR!CONVERT-FROM-X are driver functions which convert any s-expression to or from the internal form, respectively.

The type of a cell is stored in the *monk* which represents it, since transformation of a subtree may result in its being of a different type than before. For example, the rule (COND (T <X>)) => <X> changes the parse tree of (COND (T (CAR FOO))) from type COND to type FCALL. The type of a cell is defined to be the type of the most recent monk.

4.3.1.2. *Cell Creation*

The function CELL!CREATE creates a cell. It requests the subtree and type to be installed in the first monk, and the name of the function requesting the creation. The first monk of each cell is the one given at creation, and always has version 0, whether it was created during the transformation process or during the initial conversion of the LISP program to the internal form. This will not raise ambiguities when converting back to executable LISP code, since even though a cell may have a 0 version monk, it can not be reached if its parent cell doesn't point to it at time 0. Consider, for example, the transformation that replaces a function call by its definition. Suppose *FUNCNAME* cell CELL-33 is FOO at time 3, and at time 4 is replaced by the lambda expression (LAMBDA (X) (CAR X)). Then every subcell containing a subtree of the lambda expression will be at version 0 since the expression is newly copied before replacement. Printing CELL-33 at time 3 will produce FOO though, not the lambda expression, because the version 3 monk of CELL-33 points to FOO.

It is useful to be able to "back-up" to the father of a cell, and to the father of that cell, and so on, if desired. Since which cell is the father of another cell changes as transformations apply, it is not sufficient to maintain a single static back-pointer for each cell. Instead, when a cell is created it is given a back-pointer list, initially NIL. However, the back-pointer list of each *subcell* is made to point to the newly created cell. A back-pointer consists of a cell and a version number. To obtain the back-pointer of a cell we use the function CELL!BP, which takes a cell and a version number and returns the back-pointer of the cell for that point in time. Suppose, for example, that CELL-45 has the back-pointer list:

```
( (23 . CELL-88) (14 . CELL-23) (11 . CELL-23) (10 . CELL-4) )
```

Then (CELL!BP CELL-45 15) will return the back-pointer (14 . CELL-23). The function BP!CELL will return the cell of a back-pointer. Then (BP!CELL (CELL!BP CELL-45 10)) will return CELL-4. If the back-pointer of CELL-45 is requested for some version less than 10, CELL!BP will return NIL, indicating that CELL-45 had no father at that time.

Recording the back-pointers of a cell will allow us to easily implement the function CELL!FATHERS. This function takes a cell and a version number and returns a list of the cell's cu:rent ancestors; that is, a list of all the cells currently on the direct path from the given cell to the root cell of the parse tree.

At creation, each cell is given a "range", which allows us to determine the times at which it or any of its subcells underwent transformation. A range consists of a minumum and maximum version, both of which are 0 at creation. If a transformation applies to CELL-45 at time 5, the range's maximum is changed to 5, and every range of the transformed cell's current ancestors is updated as well. Hence, if in the end CELL-45 has a range with minimum 5 and maximum 49, we know that every transformation which occurred from time 5 to time 49 applied to CELL-45 or one of its subcells.

Though the initial monk of a cell always has version 0, 0 is not included in the range. This is because the LISP program tree which is input to the transformation system is converted to the cell internal form representation all at once, immediately giving every cell a monk of version 0. But transformation only begins at one subtree of the parse tree, and brothers of that subtree are not transformed until later. To maintain the continuity of the range indicated, we define the minimum of a cell's range to be only the time of the first actual transformation to reach that cell or one of its subcells.

Two slots are left open in a cell at its creation for the eventual storage of its environment and index of bindings. These will be installed immediately before transformation of the cell is attempted. The function SEXPR!RECORD-ENVDEX is called within each transformation set, and will cause the two structures to be stored in the input cell. Although the transformers do not access this common contextual information via the cell they transform, they *could* if they knew how. However, it seems cleaner to keep them uninformed about the temporary duplication of information. Later, when the transformation process ends and a user wishes to obtain common contextual information about an expression, it will be provided him from the copy retained in that expression's cell.

4.3.2. The Transformation of Cells

How is it possible to neatly package the history of each subtree of a program into its own cell, given that any number of unspecified transformations will be engaged in collapsing code or moving pieces of the program from place to place? In an attempt to avoid having more than one cell to represent a particular node of the parse tree, yet at the same time maintain the integrity of a node's history, I have developed special monks which keep cells clean and tidy. These monks are created as needed by RESPONSE!HACK; the transformers need not concern themselves with such low level details. Indeed, the transformers know nothing at all of the internal representation of the argument program.

When a transformer is called via XFORM, it is passed the cell to be transformed, and, if the transformer performs a global transformation, the environment and index of bindings as well. The transformer's response consists of a success/fail flag, the resultant expression, and a justification. If the transformation fails, it returns the same cell as was input. Otherwise the returned object must be either:

(1) the same cell given to the transformer (the "input" cell),

(2) a cell which is a "subcell" of the input cell,

(3) a newly made cell, perhaps copied from elsewhere in the tree,

(4) a list of cells, or

(5) nothing.

I will treat each of these cases in turn, after which we will discuss the completion of histories.

4.3.2.1. *Transformers Which Call Slaves*

If the cell returned from a transformer is the same as the input cell, the transformer must have called slave transformers to perform a local transformation. In this case, a new monk of the same type as before will be created and installed in the input cell. Though the subtree pointed to by the new monk is the same one the last monk pointed to, the new monk carries with it the new version number, transformation name, and justification, signifying that a transformer did apply to the cell.

Any transformation performed by slave transformers will have been recorded in the cells which were input to those slave transformers, thereby updating only the histories of the relavent subcells, and leaving other subcells of the parent transformation cell both unchanged and

```
(DEFUN XFORM-SLAVE (XFORMER CELL ARGS PTCPOINTER)
       (LET ((RESPONSE (APPLY XFORMER (CONS CELL ARGS))))
            (COND ((WAS-CHANGED? RESPONSE)
                   (CELL!STORE-MONK
                      CELL
                      (RESPONSE!SLAVE-HACK RESPONSE CELL XFORMER PTCPOINTER))
                   (CLOAKRACK!HANG-SLAVE CELL XFORMTIME)
                   (XFORMER!RECORD XFORMER CELL XFORMTIME))
                  (T (CELL!STORE-HERMIT
                        CELL
                        (RESPONSE!SLAVE-HACKF RESPONSE XFORMER PTCPONTER))
                     (RAGRACK!HANG-SLAVE CELL XFORMTIME)
                     (XFORMER!RECORDF XFORMER CELL XFORMTIME)))))

(DEFUN SLAVE-RESPOND (APPLIED? OBJECT JUST)
       (COND ((NOT APPLIED?) (LIST **SLAVE-RESPONSE** *FAILED* JUST))
             (T (LIST **SLAVE-RESPONSE** OBJECT JUST))))
```

TABLE 4-10. Slave transformations allow the histories of subcells to be updated.

uncopied. The pointers in the parse tree which connect the subcells to the parent cell remain intact. The use of slave transformers in the transformational component permits the accountable component to maintain the locality of transformation.

Slave transformers are expected to return success/fail flags, resultant cells, and justifications, just as their masters do. XFORM-SLAVE installs these in the history of the slave transformer's input cell, along with a pointer to the parent transformation cell. Every monk has such a "PTC" pointer; A non-nil PTC pointer indicates that the transformer which caused the monk's creation was a slave transformer. In particular, it was a slave to the transformer which created the monk of the same version contained in the parent cell pointed to. Thus, for example, if CELL-14 contains a version 5 monk with a PTC pointer to CELL-59, we know that CELL-59 contains a version 5 monk as well, and the transformer indicated in that monk was the master of the transformer indicated in CELL-14's version 5 monk. The definition of XFORM-SLAVE is given in Table 4-10.

4.3.2.2. *Transformers Which Simplify*

A transformer which returns a subcell of the input cell was able to simplify the expression by "bypassing" some part of its structure, as in the rule (APPEND <x> NIL) => <x>. (I use the function CELL!FATHERS to determine if the cell returned by a transformation is a subcell of the input cell. If the input cell is a member of the fathers of the returned cell, then the returned cell is a subcell.)

How can this returned subcell be incorporated into the input cell? If we install the most recent monk of the returned subcell as the new monk of the input cell, we confuse the histories of the two pieces of code. Someone interested only in following the progress of the expression <x> would suddenly be left hanging at the point when the transformation occurred, unless some complicated method of backpointers and justifications was concocted. My decision was to instead create a special type of monk called a *POINTER*. The cell which represents <x> is left unchanged, just as the expression itself was untouched by the transformer. The new monk is installed in the input cell along with the normally recorded information. The subtree of this *POINTER* monk, however, is the <x> cell. The type of the resultant input cell is defined to be the type of the object of the *POINTER* monk, and someone interested in following the history of the input cell, upon confronting the *POINTER*, continues his study by picking up the history of the <x> cell at the time the transformation occurred. The history of the <x> cell, however, remains totally contained in that cell (unless it, too, is simplified to point to a subcell), and is oblivious to the effects of transformations performed on cells above it in the parse tree.

4.3.2.3. *Transformers Which Replace*

A transformer which returns a new cell, perhaps copied from an expression elsewhere in the parse tree or newly created, intends for the subtree of that new cell to completely replace the subtree of the input cell. This might occur in situations when an expression is being replaced by its truth value, when a procedure definition is substituted for a call, or when actual arguments are being substituted for formal arguments, for example. In all cases, the new cell should be "deeply" copied (that is, copied at all levels) from its source so that future transformations which apply to it or its subcells in the new context will not affect the original copy. Transformers which perform such replacements or substitutions are responsible for providing the new copy of the returned cell (note the calls to the function SEXPR!COPY in Table 4-1). Copied cells in some sense begin their existence at the time of their substitution, and thus are not responsible for the history of the original cell. Thus, the latest monk of the copied cell is installed as the new monk in the input cell, and the copied cell itself is thrown away. The justification for the transformation should provide the information necessary to determine the source of the new cell should the user desire it.

4.3.2.4. *Transformers Which Listify*

The object returned from a transformer may be a list of cells, indicating that one cell is to be replaced by many. Under what circumstances might this occur? A transformer which returns a list of cells is nearly always a slave transformer, but since slaves are treated almost exactly as regular transformers, it will not enter into the discussion here. Recall the example given in sections 3.4.2 and 3.4.4 where this occurred with conditional clauses:

```
(COND <clauses-1> (T (COND <clauses-2>)))
                     => (COND <clauses-1> <clauses-2>)
```

A transformer to do this would be placed in the *COND* transformation set, and after verifying that the input expression is of the correct form, calls a slave transformer on the final clause K of the input conditional expression. The slave transformer returns the list of clauses <clauses-2>. Clause K should have a special monk installed in its history to show that it now points to the list <clauses-2>. To allow a cell to point to a *list* of subtrees, I have created another special monk called a *POINTER-LIST*.

Another case in which a slave transformer is used to return a list of expressions is in implementation of the transformation rule:

```
(PLUS (PLUS <Y> <Z>) <X>) => (PLUS <X> <Y> <Z>)
```

A transformer to perform this transformation would appear in the *FCALL* transformation set and apply to the outer function call, but would call a slave transformer on the inner function call that returns a list of its arguments.

The use of *POINTER-LIST* monks complicates accessing and conversion functions which now must be able to suddenly handle a list of subtrees when they were only expecting a single tree. A special MAPCAR function was writter which checks for *POINTER-LIST* cells before passing them as arguments to the stated function. Because such system functions must be able to detect and deal with *POINTER-LIST* cells, the type of such cells is defined to be *POINTER-LIST*, rather than the type of the first subtree in the list of subtrees pointed to. A *user* of the accountable component, however, need not know of their existence, and will receive expected type information in answer to the same query.

4.3.2.5. *Transformers Which Delete*

A transformer which returns nothing must do so by returning a "nothing" flag, which occurs in this system as the special variable **NOTHING**. Such transformers currently *must* be slaves, but again, that will not enter into our discussion.

When a nothing flag is returned, a special *NOTHING* monk is created and installed in the input cell, complete with version number, transformer name, etc. just as though nothing unusual had happened. The subtree pointed to by a *NOTHING* monk is NIL, and the type of cells which contain *NOTHING* monks is defined to be *NOTHING*. No transformer can ever apply to such a cell again since the cell will never be detected; it no longer exists as far as the transformational component is concerned.

The use of *NOTHING* monks is currently restricted to the case in which the deleted expression occurred as an element of a list in the program. That is, as an element of an argument list to a function call, a dummy argument list, a list of conditional clauses, etc. This allows system functions which manipulate cells to correctly ignore the nothing cell, which is actually still very much a something. This restriction should not hamper the transformation implementor, since any transformation which deletes an object not in a list can be implemented as a transformer which simplifies (the expressions such transformers bypass are essentially deleted).

4.3.2.6. *Completing Histories*

We have discussed the maintenance of cell histories by installing monks with information regarding the success or failure of transformations to those cells. However, there is another possibility which I have not yet provided for. It may be the case that a CELL-17 is eliminated from the parse tree by a transformation. When the resulting parse tree is converted back to executable LISP code, CELL-17 will properly be omitted, since it is not pointed to by any of the current monks of cells above it. However, someone interested in following only the history of CELL-17 will not be aware of the point at which some cell higher in the parse tree ceases to reference CELL-17. Reaching the last monk in the cell, that person will assume (and rightly so) that CELL-17 occurs in the final parse tree as the subtree recorded by that monk.

For example, if the *AND* cell which represents the expression (AND T (CONS X Y)) is replaced by T (assuming that the expression is in predicate position), then the *ATOM* T, the *FCALL* expression (CONS X Y), the *FUNCNAME* CONS, and the *ATOM*s X and Y should all have their histories brought to an end. To do this, I have created a special monk of type *REPLACED*. The subtree of this monk is empty, and when installed in a cell simply signifies that some transformation occurred which caused this cell to be replaced. Whenever a transformation returns a new cell or a nothing flag, the histories of all the subcells of the input cell are completed by installing the *REPLACED* monk.

Now consider the transformation (APPEND NIL <x>) => <x>. This transformation bypasses cells to return a subcell of the input cell, as in section 4.3.2.2. In order to bring the histories of the *FUNCNAME* cell containing APPEND and the *ATOM* cell containing NIL to a close, I have created another special monk called a *BYPASS* monk. Again, the *BYPASS* monk has an empty subtree, and when installed in a cell signifies that this cell was bypassed by a simplifying transformer and therefore does not remain in the resultant program.

In general, the history of any cell which becomes invisible to future transformations should be "completed" with one of the special monks. The transformers themselves are not responsible for this; they need know nothing of the existence of cells at all, since the transformational component should be oblivious to the existence of the accountable component. Instead, all such housekeeping (cell-keeping!) is performed by the function CELL!TERMINATE, which is called by RESPONSE!HACK and RESPONSE!SLAVE-HACK. These latter two are able to determine which types of the possible special monks *POINTER*, *REPLACED*, *BYPASSED*, *POINTER-LIST*, or *NOTHING* to create, simply by determining in which of the five

```
(DEFUN RESPONSE!HACK (RESPONSE INPUTCELL XFORMER PTCPOINTER)
       (LET
        ((OBJECT (RESPONSE!OBJECT RESPONSE))
         (JUST (RESPONSE!JUST RESPONSE)))
        (COND
          ((EQ OBJECT INPUTCELL)   ; must have been a master transformer
           (COND ((RESPONSE!SLAVE? RESPONSE)
                  (ERROR '|Slaves can't return same cell -- RESPONSE!HACK|)))
           (MONK!CREATE
              (SEXPR!TYPE OBJECT)
              XFORMTIME
              (SEXPR!TREE OBJECT)
              XFORMER
              JUST
              **MASTER**))
          ((CELL!SUBTREE? INPUTCELL OBJECT)  ; sign of a **POINTER**
           (CELL!STORE-BP OBJECT (CELL!BP INPUTCELL))
           (CELL!TERMINATE INPUTCELL (LIST OBJECT) PTCPOINTER)
           (MONK!CREATE **POINTER** XFORMTIME OBJECT XFORMER JUST PTCPOINTER))
          ((CELL? OBJECT)  ; must be a new cell
           (CELL!TERMINATE INPUTCELL NIL PTCPOINTER)
           (MONK!CREATE (SEXPR!TYPE OBJECT)
                        XFORMTIME
                        (SEXPR!TREE OBJECT)
                        XFORMER
                        JUST
                        PTCPOINTER))
          ((CELLIST? OBJECT)  ; sign of a *POINTER-LIST*
           (MAPC (FUNCTION (LAMBDA (POINTEE)
                                    (CELL!STORE-BP POINTEE
                                                   (CELL!BP INPUTCELL))))
                 OBJECT)
           (CELL!TERMINATE INPUTCELL OBJECT PTCPOINTER)
           (MONK!CREATE **POINTER-LIST** XFORMTIME OBJECT XFORMER JUST PTCPOINTER))
          (T (ERROR '|Unknown response object -- RESPONSE!HACK| OBJECT)))))

(DEFUN RESPONSE!SLAVE-HACK (RESPONSE INPUTCELL XFORMER PTCPOINTER)
       (LET ((OBJECT (RESPONSE!OBJECT RESPONSE))
             (JUST (RESPONSE!JUST RESPONSE)))
            (COND ((EQ OBJECT **BYPASS**)
                   (MONK!CREATE **BYPASS** XFORMTIME NIL XFORMER JUST PTCPOINTER))
                  ((OR (EQ OBJECT **NOTHING**) (EQ OBJECT **REPLACED**))
                   (CELL!TERMINATE INPUTCELL NIL PTCPOINTER)
                   (MONK!CREATE OBJECT XFORMTIME NIL XFORMER JUST PTCPOINTER))
                  (T (RESPONSE!HACK RESPONSE INPUTCELL XFORMER PTCPOINTER)))))
```

TABLE 4-11. Functions which perform "cell-keeping".

categories the returned object falls.

The definitions of the functions RESPONSE!HACK and RESPONSE!SLAVE-HACK are given in Table 4-11. After determining the type of response object returned by the transformer (same cell, subcell, new cell, list of cells, etc.), up to three things happen:

```
(DEFUN CELL!TERMINATE (CELL *TERM-OBJLIST* *TERM-PTCPOINTER*)
       (DECLARE (SPECIAL *TERM-OBJLIST* *TERM-PTCPOINTER*))
       (COND
         ((NULL *TERM-OBJLIST*)
          (SEXPR!MAPCAR
           (FUNCTION
            (LAMBDA (SON)
                    (XFORM-SLAVE 'SEXPR!REPLACED
                                 SON
                                 (LIST)
                                 *TERM-PTCPOINTER*)))
           (CELL!CSONS CELL)))
         (T (SEXPR!MAPCAR
             (FUNCTION
              (LAMBDA (SON)
                      (COND ((MEMQ SON *TERM-OBJLIST*) NIL)
                            (T (XFORM-SLAVE 'SEXPR!BYPASS
                                            SON
                                            (LIST)
                                            *TERM-PTCPOINTER*)
                               (CELL!TERMINATE SON
                                               *TERM-OBJLIST*
                                               *TERM-PTCPOINTER*)))))
             (CELL!CSONS CELL)))))
```

TABLE 4-12. Cell histories are completed using CELL!TERMINATE.

- back-pointers are updated. The back-pointer list of the returned object is updated to include references to the input cell, since the object's subtree is about to be installed in the input cell via a new monk.
- the relavent old subtrees of the input cell are terminated. CELL!TERMINATE takes a cell whose current subcells are to be terminated, a list of subcells of the input cell which are not to be terminated (it checks each of the former against the latter before terminating), and the input cell which acts as the parent transformation cell for the termination process. The special termination monks *BYPASS* and *REPLACE* are installed via system defined slave transformers which use the PTC pointer information. The definition of CELL!TERMINATE is given in Table 4-12.
- the new monk is created and returned to XFORM (or XFORM-SLAVE), which will then install it in the input cell.

4.4. Retrieval of Information

At the start of the research for this thesis, I had grand ideas of a system in which the user had only to hint at what he didn't understand about the transformation process. The accountable component would immediately understand the source of his confusion and provide him with the information he needed. This, I thought, would be a reasonable goal; very AI-ish, and much easier than the even more idealistic goal in which the system could detect "unusual" transformation situations and explain them without any prompting by the user.

Such goals may yet be attained, however, for the present I have accepted the fact that providing simple functions for accessing the information stored in cells and monks will have to be sufficient. Given a cell and a transformation time, for example, the function DISP-JUST will display the justification for the transformation which occurred to that cell at that time. Or, given a transformation time, the function LOCATE-NEWCELL will return the cell which was transformed at that time. There are a many such functions for retrieving information; their usefulness, however, is limited to those people who know which ones to call to get the desired information. The user should be able to request information with the least amount of hassle. Obviously then, the retrieval functions should be packaged up and presented via a menu of some sort.

4.4.1. The Menus

Following the transformation of a list of evaluable s-expressions (the input accepted by the driver function TRANSFORM), the resulting list of cells (one for each s-expression) is bound to the atom XSEXPR. The transformed cells are cross-referenced by transformation time in CLOAKRACK, and cells to which transformers applied but failed are cross-referenced by transformation time in RAGRACK. In addition, a list of the cells and times for which each transformer succeeded or failed to apply has been recorded in the property list associated with the name of that transformer. To give the user access to the information stored in these structures, the query element presents the user with a number of choices via the Menu Menu.

The Menu Menu allows the user to select either the Code Menu, the Transformer Menu, or the Version Menu. Each of these present the user with his choice of code, transformer, or transformation time, respectively, to be the subject of his query. If he selects the Code Menu, he may then specify any one of the expressions from the list of those he gave to the function TRANSFORM. Thus, for example, if he transformed a list of five expressions, he may enter "3" to point to the third expression. The Transformer Menu gives the user access to statistics concerning the number of transformers given to the system, the number that were applied, and how many always failed or always applied. The user may ask to see a list of the names of the transformers that fall into any of these categories, or enter one transformer name and study its activities. The Version Menu will tell the user what the final transformation time was, then request a version number as input. The expression transformed at that time will be displayed, and the user may continue to ask questions about that expression, or return to the menu to inquire about a new transformation time. A diagram of the menus is given in Figure 4-2.

The functions used to support the menu system are simple and straightforward. I defined a

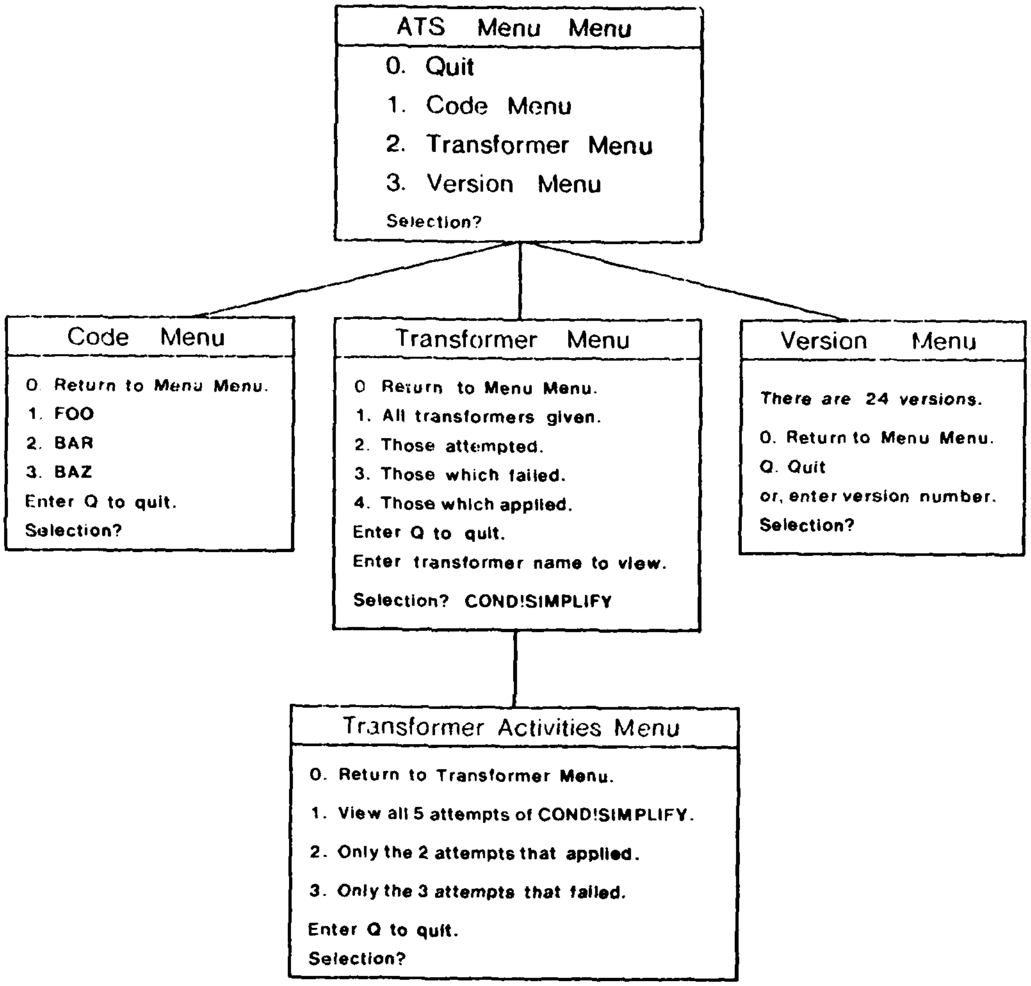


FIGURE 4-2. The menu system.

general menu display function, DISP-MENU, which takes a title, a list of selections, and a list of messages to be displayed after the selections. This function is used by each of the menu functions MENU-MENU, CODE-MENU, TRANSFORMER-MENU, and VERSION-MENU. If the user selects an expression from the Code Menu, that expression will be passed to the function WALK. This function will allow the user to obtain a variety of information stored in the cell which represents that expression. If the user selects a transformer from the Transformer Menu, he will be given a number of statistics on the activities of that transformer, and then will be allowed to "walk" any of the expressions transformed by that transformer. And finally, if the user selects a transformation time from the Version Menu, he will walk the cell which was transformed at that time.

4.4.2. Walking

The function `WALK` takes any cell and optionally, a version. If no version is given, he will begin walking at version 0. The expression will be displayed along with the version number, the expression's structur‿ type, and the name of the transformer which created it. The user may then enter any of the commands described in Chapter Two (and some others besides) to obtain additional information.

The simplest method of retrieving information from the cells is for the user simply to enter "+" at each prompt, which will display the expression as it occurred in the next version. This has the effect of simulating the transformation process as it applied to the expression. If, as the user walks through this process, step by step, he wishes to stop along the way and inquire in more detail about some transformation , he may easily do so. Entering "w" (for "why"), for example, will cause the justification for the current transformation to be displayed. Entering "f" (for "failed") will cause the names of the transformers which were attempted but failed at this point in the process to be displayed along with their justifications. Entering "?" will display the menu of commands available to the user.

If the user has a pretty good idea of what he wants to look at, he may go directly to piece of code he is interested in by using the "up", "down", "over", and "set version" commands described in Chapter Two. These allow him to pick up his walk anywhere in time (transformation time) or space (point in code). If at any time during his walk he wishes to jump down to the sub-expression which was just transformed, the user may enter the command "j" (for "jump"), rather than the correct sequence of downs and overs. After exploring around all he wants, he may enter "r" (for return) to return to the original expression. And, as a special added feature, the mechanism used to remember the expression jumped from (a stack!) was generalized and made available to the user via the command "m" (for memorize). This adds the current expression and version number to the stack (stored in `CELL-ENVIRONMENT`) without jumping anywhere. The user may walk anywhere he likes, and then enter "r" to return the last expression and version which were either jumped from or "memorized".

4.5. An Example

[illegible] function of the system is `TRANSFORM`. I call that here on the list:

```
[illegible]
```

```
            (COND ((NULL Y) Y)
                  (T (FOO (CAT Y)))))
    (DEFUN FOO (X)
            (CONS X X))
    (DEFUN BAR (FROB)
            (COND ((ATOM  FROB) (TEST FROB FROB))
                  ((OR FROB (CDR FROB)) (CAR FROB))
                  (T (CAR (FOO FROB))))))
```

which is bound to the atom DAFS.

```
    (transform dafs)

    Transformation completed.
```

When the transformation process is complete, the user may enter the menu system, invoking the query element.

```
    Enter Menu System? (enter y or n) y

         THE MENU MENU

    0 - Quit
    1 - Code Menu
    2 - Transformer Menu
    3 - Version Menu
    Selection? (end with CR): 1
```

Upon entering the code menu, the name of each function definition transformed (or an expression type if the the expression transformed is not a function definition) will be displayed so the user may select one of them to examine.

```
         CODE MENU

    0 - Return to Menu Menu
    1 - TEST
    2 - FOO
    3 - BAR
    Enter Q to quit.
    Selection? (end with CR): 3
```

The selected expression is first displayed as it was input to the transformation system.

```
    (DEFUN BAR (FROB)
            (COND ((ATOM FROB) (TEST FROB FROB))
                  ((OR FROB (CDR FROB)) (CAR FROB))
                  (T (CAR (FOO FROB)))))
```

```
Version: 0    Expression type: *DEFETTE*    Transformer: DEFETTE!CREATE
:?
```

The user may type "?" at any time to see a list of possible commands.

```
d - down     to first sub-expression of this cell
u - up       to parent expression
o - over     to brother expression
j - jump     jump to sub-expression just changed.
r - return   return to expression last jumped from or memorized
+ - add      increment version by one and print
- - subtract decrement version by one and print
s - set      set version to number prompted for
n - next     go to next version of this cell and print
p - previous go to previous version of this cell and print
l - last     go to last version of this cell and print
m - memorize remember this cell and this version
a - again    print current version of expression
h - how many display version numbers for this expression
t - type?    what is type of top level expression
v - version? what is version of top level exprssion?
w - why?     get justification of current transformation
x - xformer? what transformer produced this cell's current monk.
e - evaluate display the boolean value of this expression
f - fails    look at the transformers which failed
b - break    break to LISP
q - quit     quit to top level
? - huh?     prints this info
:+
```

The user may re-enact the transformation process by entering a "+" at each prompt.

```
(DEFUN BAR (FROB)
    (COND ((ATOM FROB)
           ((LAMBDA (X-2 Y-1)
                    (COND ((NULL Y-1) Y-1)
                          (T (FOO (CAT Y-1))))) FROB FROB))
          ((OR FROB (CDR FROB)) (CAR FROB))
          (T (CAR (FOO FROB)))))

Version: 3   Expression type: *DEFETTE*  Transformer: FUNCNAME!SUB-DEF
:+
(DEFUN BAR (FROB)
      (COND ((ATOM FROB)
             ((LAMBDA (X-2 Y-1)
                      (COND ((NULL Y-1) NIL)
                            (T (FOO (CAT Y-1))))) FROB FROB))
            ((OR FROB (CDR FROB)) (CAR FROB))
            (T (CAR (FOO FROB)))))

  Version: 4    Expression type: •ATOM•    Transformer: SEXPR!FORM-BVAL
  :+


(DEFUN BAR (FROB)
    (COND ((ATOM FROB)
           ((LAMBDA (X-2 Y-1)
                    (COND ((NULL Y-1) NIL)
                          (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT Y-1)))))
            FROB
            FROB))
          ((OR FROB (CDR FROB)) (CAR FROB))
          (T (CAR (FOO FROB)))))

  Version: 5   Expression type: •DEFETTE•  Transformer: FUNCNAME!SUB-DEF
  :+


(DEFUN BAR (FROB)
    (COND ((ATOM FROB)
           ((LAMBDA NIL
                    (COND ((NULL FROB) NIL)
                          (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT FROB))))))
          ((OR FROB (CDR FROB)) (CAR FROB))
          (T (CAR (FOO FROB)))))

  Version: 6    Expression type: •FCALL•    Transformer: DEFCALL!TRIM
  :+


(DEFUN BAR (FROB)
    (COND ((ATOM FROB)
           (COND ((NULL FROB) NIL)
                 (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT FROB)))))
          ((OR FROB (CDR FROB)) (CAR FROB))
          (T (CAR (FOO FROB)))))

  Version: 7    Expression type: •COND•    Transformer: DEFCALL!SIMPLIFY
  :+
(DEFUN BAR (FROB)
    (COND ((ATOM FROB)
          (COND ((NULL FROB) NIL)
         (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT FROB)))))
         ((OR T (CDR FROB)) (CAR FROB))
         (T (CAR (FOO FROB)))))

  Version: 8    Expression type: *ATOM*    Transformer: SEXPR!FORM-BVAL
  :+


(DEFUN BAR (FROB)
    (COND ((ATOM FROB)
          (COND ((NULL FROB) NIL)
               (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT FROB)))))
         ((OR T) (CAR FROB))
         (T (CAR (FOO FROB)))))

  Version: 9    Expression type: *OR*    Transformer: OR!TRIM
  :+


(DEFUN BAR (FROB)
    (COND ((ATOM FROB)
          (COND ((NULL FROB) NIL)
               (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT FROB)))))
         (T (CAR FROB))
         (T (CAR (FOO FROB)))))

  Version: 10    Expression type: *ATOM*    Transformer: OR!SIMPLIFY
  :+


(DEFUN BAR (FROB)
    (COND ((ATOM FROB)
          (COND ((NULL FROB) NIL)
               (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT FROB)))))
         (T (CAR FROB))))

  Version: 11    Expression type: *COND*    Transformer: COND!TRIM
  :+


No more transformations occurred. This is the final version.

  (DEFUN BAR (FROB)
    (COND ((ATOM FROB)
     (COND ((NULL FROB) NIL)
        (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT FROB)))))
    (T (CAR FROB))))

Version: 11    Expression type: •COND•    Transformer: COND!TRIM
:q
```

If the user requests a non-existent version of the expression, he is warned as shown above, and the expression is redisplayed. When he is through walking throught the transformation process, the user quits by entering "q" and is returned to the code menu.

```
   CODE MENU

0 - Return to Menu Menu
1 - TEST
2 - FOO
3 - BAR
Enter Q to quit.
Selection? (end with CR): 0


   THE MENU MENU

0 - Quit
1 - Code Menu
2 - Transformer Menu
3 - Version Menu
Selection? (end with CR): 3


   VERSION MENU

There are 11 versions.
0 - Return to Menu Menu
Q - quit
or, enter version number.
Selection? (end with CR): 7
```

The version menu allows the user to begin walking the code at a particular point in time and space: the expression transformed at time 7, in this case. After the code is displayed, the user may enter any of the usual walking commands. For example, he might enter "u" to see more of the context of the expression which was transformed.

```
(COND ((NULL FROB) NIL)
   (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT FROB))))
```

```
Version: 7    Expression type: *COND*    Transformer: DEFCALL!SIMPLIFY
:u



((ATOM FROB)
 (COND ((NULL FROB) NIL)
       (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT FROB)))))

Version: 7    Expression type: *COND*    Transformer: DEFCALL!SIMPLIFY
:u



(COND ((ATOM FROB)
       (COND ((NULL FROB) NIL)
             (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT FROB)))))
      ((OR FROB (CDR FROB)) (CAR FROB))
      (T (CAR (FOO FROB))))

Version: 7    Expression type: *COND*    Transformer: DEFCALL!SIMPLIFY
:—
```

To see what the code looked like immediatly before the transformation, the user enters "—".

```
(COND ((ATOM FROB)
       ((LAMBDA NIL
                (COND
                 ((NULL FROB) NIL)
                 (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT FROB)))))))
      ((OR FROB (CDR FROB)) (CAR FROB))
      (T (CAR (FOO FROB))))

Version: 6    Expression type: *FCALL*    Transformer: DEFCALL!TRIM
:q
```

The user may explore as much as he desires, and when he quits he will be returned to the menu he came from.

```
      VERSION MENU

There are 11 versions.
0 - Return to Menu Menu
Q - quit
or, enter version number.
Selection? (end with CR): 0
```

```
          THE MENU MENU

     0 - Quit
     1 - Code Menu
     2 - Transformer Menu
     3 - Version Menu
     Selection? (end with CR): 2


          TRANSFORMER MENU

     0 - Return to Menu Menu
     1 - Transformers given in transformation sets
     2 - Transformers attempted
     3 - Transformers which always failed
     4 - Transformers which always succeeded
     Enter transformer name to view.
     Selection? (end with CR): 2
```

The transformer menu gives the user access to a variety of statistics. Each of the categories listed above will display the number of transformers in that category and give their names.

```
     14 transformers attempted:

     (OR!SIMPLIFY OR!TRIM
                  SEXPR!REPLACED
                  SEXPR!BYPASS
                  ATOM!SUB-ACTARG
                  SEXPR!NOTHING
                  COND!SIMPLIFY
                  COND!TRIM
                  DEFCALL!SIMPLIFY
                  DEFCALL!TRIM
                  FCALL!SIMPLIFY
                  FCALL!SIMP-NOT
                  FUNCNAME!SUB-DEF
                  SEXPR!FORM-BVAL)

     Enter any character to continue: x
```

```
 TRANSFORMER MENU

0 - Return to Menu Menu
1 - Transformers given in transformation sets
2 - Transformers attempted
3 - Transformers which failed
4 - Transformers which succeeded
Enter Q to quit.
Enter transformer name to view.
Selection? (end with CR): 4


11 transformers succeeded:

   (OR!SIMPLIFY OR!TRIM
                SEXPR!REPLACED
                SEXPR!BYPASS
                ATOM!SUB-ACTARG
                SEXPR!NOTHING
                COND!TRIM
                DEFCALL!SIMPLIFY
                DEFCALL!TRIM
                FUNCNAME!SUB-DEF
                SEXPR!FORM-BVAL)


6 transformers always succeeded:

   (OR!SIMPLIFY OR!TRIM
                SEXPR!REPLACED
                SEXPR!BYPASS
                ATOM!SUB-ACTARG
                SEXPR!NOTHING)

   Enter any character to continue: x


 TRANSFORMER MENU

0 - Return to Menu Menu
1 - Transformers given in transformation sets
2 - Transformers attempted
3 - Transformers which failed
4 - Transformers which succeeded
Enter Q to quit.
Enter transformer name to view.
Selection? (end with CR): or!trim
```

If the user enters the name of a transformer, he is taken to the transformer activities menu and given some statistics on the performance of that transformer.

```
      TRANSFORMER ACTIVITIES MENU

0 - Return to Transformer Menu
1 - Observe all 1 attempts of transformer OR!TRIM
2 - Only its 1 successful applications
3 - Only its 0 unsuccessful applications
Enter Q to quit.
Selection? (end with CR): 2
```

The user may study both successful and unsuccessful applications of the transformer.

```
The transformer OR!TRIM applied at time 9.
Would you like to study the expression?  (enter y or n) y

(OR T)

Version: 11   Expression type: *OR*   Transformer: OR!TRIM
:—

(OR T (CDR FROB))

Version: 10   Expression type: *ATOM*   Transformer: SEXPR!FORM-BVAL
:w
```

While walking, entering the command "w" (for "why?") will produce a set of justifications for the transformation.

```
Expression was transformed because (not (atom x)) => (not (null x)) => x

(ATOM FROB)

Value is NIL because it is within false branch of the cond clause

((ATOM FROB)
 (COND ((NULL FROB) NIL)
       (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT FROB)))))

:u
```

```
((OR T (CDR FROB)) (CAR FROB))

  Version: 10    Expression type: *ATOM*    Transformer: SEXPR!FORM-BVAL

:u

  (COND ((ATOM FROB)
         (COND ((NULL FROB) NIL)
               (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT FROB)))))
        ((OR T (CDR FROB)) (CAR FROB))
        (T (CAR (FOO FROB))))

  Version: 10    Expression type: *ATOM*    Transformer: SEXPR!FORM-BVAL
  :+


(COND ((ATOM FROB)
        (COND ((NULL FROB) NIL)
              (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT FROB)))))
                                  ((OR T) (CAR FROB))
                                  (T (CAR (FOO FROB))))

  Version: 11    Expression type: *OR*    Transformer: OR!TRIM
  :q
```

After walking, the user is returned to the transformer activities menu. From there he may either inquire more about the same transformer, or return to the transformer menu.

```
        TRANSFORMER ACTIVITIES MENU

  0 - Return to Transformer Menu
  1 - Observe all 1 attempts of transformer OR!TRIM
  2 - Only its 1 successful applications
  3 - Only its 0 unsuccessful applications
  Enter Q to quit.
  Selection? (end with CR): 0


  TRANSFORMER MENU

  0 - Return to Menu Menu
  1 - Transformers given in transformation sets
  2 - Transformers attempted
  3 - Transformers which failed
  4 - Transformars which succeeded
```

```
Enter Q to quit.
Enter transformer name to view.
Selection? (end with C1): cond!simplify


 TRANSFORMER ACTIVITIES MENU

0 - Return to Transformer Menu
1 - Observe all 3 attempts of transformer COND!SIMPLIFY
2 - Only its 0 successful applications
3 - Only its 3 unsuccessful applications
Enter Q to quit.
Selection? (end with CR): 3

The transformer COND!SIMPLIFY failed 1 times during transformation time 3.
Would you like to see the expressions?  (enter y or n) y
```

If the user chooses to view an expression to which a transformer was applied but failed, after displaying the expression, the names of all transformers which were attempted (but failed) at that time are displayed along with their justifications.

```
Expression 1 of 1.  Continue?  (enter y or n) y

(COND ((NULL Y) NIL) (T ((LAMBDA (X-1) (CONS X-1 X-1)) (CAT Y))))

Version: 3    Expression type: *COND*    Transformer: COND!CREATE
Transformation time: 3  Expression type: *COND*

COND!TRIM failed to transform the expression because
         no predicates were known (maybe last).

COND!SIMPLIFY failed to transform the expression because
         patterns didn't match.

SEXPR!FORM-BVAL failed to transform the expression because
         the boolean value is unknown.
Done.
Enter any character to continue: x


  The transformer COND!SIMPLIFY failed 1 times during transformation time 6.
Would you like to see the expressions?  (enter y or n) n
The transformer COND!SIMPLIFY failed 1 times during transformation time 12.
Would you like to see the expressions?  (enter y or n) y
```

```
 Expression 1 of 1.  Continue?  (enter y or n) y


(COND ((ATOM FROB)
     (COND ((NULL FROB) NIL)
        (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT FROB)))))
                  (T (CAR FROB)))

 Version: 14    Expression type: *COND*    Transformer: COND!TRIM
 Transformation time: 12  Expression type: *COND*

 COND!SIMPLIFY failed to transform the expression because
          patterns didn't match.

 SEXPR!FORM-BVAL failed to transform the expression because
          the boolean value is unknown.
 Done.
 Enter any character to continue: x
```

If the user quits from any menu, he leaves the menu system and is returned to the driver function, which allows him to write out the transformed expressions.

```
       TRANSFORMER ACTIVITIES MENU

 0 - Return to Transformer Menu
 1 - Observe all 3 attempts of transformer COND!SIMPLIFY
 2 - Only its 0 successful applications
 3 - Only its 3 unsuccessful applications
 Enter Q to quit.
 Selection? (end with CR): q

 Write final output to a file?  (enter y or n) y
 Enter file name: bkerns:xformd>

 Written.
 T
```

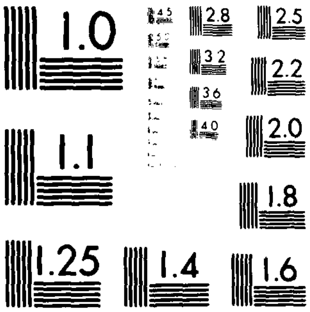
The file bkerns;xformd > now contains:

```
(DEFUN TEST (X Y)
       (COND ((NULL Y) NIL)
             (T ((LAMBDA (X-1) (CONS X-1 X-1)) (CAT Y)))))

(DEFUN FOO (X) (CONS X X))

(DEFUN BAR (FROB)
(COND ((ATOM FROB)
       (COND ((NULL FROB) NIL)
             (T ((LAMBDA (X-3) (CONS X-3 X-3)) (CAT FROB)))))
      (T (CAR FROB))))
```

AD-A110 115   MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTE--ETC F/G 9/2
AN ACCOUNTABLE SOURCE-TO-SOURCE TRANSFORMATION SYSTEM.(U)
JUN 81 B S STEELE   N00014-80-C-0505
UNCLASSIFIED   AI-TR-636   NL

2 OF 2
AD
A110115

END
DATE
FILMED
8-82
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

Chapter Five

Conclusions

IT WORKS. The code has run successfully with the applicable LISP programs given it, and the set of optimizing transformers I have written to demonstrate the system have even managed to significantly improve the execution time of those programs. That is not to say that the system was all I envisioned, or that it cannot be more . . .

5.1. Looking Back

It was my intention from the beginning that the transformational component be able to run independently of the accountable component (which in my mind meant independently of the cell representation), simply by rewriting the definitions of XFORM and RESPOND. Clearly, the functions which access pieces of the internal form must be aware of the representation of that form, but these functions can be written to accept any of a finite number of different representations at any time, so long as they are able to figure out what that representation is by means of checking type flags.

I still believe this is possible; however, it is clear that the system as currently implemented is *not* able to run with an internal form as simple as just the LISP parse tree. Certain contextual information is necessary for correct transformation. Without the predicate position information currently stored in the internal form of the parse tree, transformers dependent on that information cannot apply. Without back-pointer information, for example, it is impossible for a slave transformer to delete its input expression; it has no access to the father expression from which to unhook it. Although all such information might have been created and maintained in variables separate form the internal form and passed from control function to control function (as the environment and index of bindings were), some of it was more easily stored directly along with the LISP code to which it pertained. Thus, some sort of augmented parse tree is necessary. Furthermore, the current system is dependent on the fact that XFORM will install the returned expression in the internal form (whatever it is) by side-effecting that form, since many system functions do not "hold on" to their values. This means that the transformation system in its present implementation cannot transform itself, a fact I truly regret.

It was my further intention that the accountable system be implementable regardless of the nature of the implementation of the transformational component. While this is true to some degree, there are nevertheless dependencies I saw no simple way of avoiding. Some of them have been stated already in Chapter Three: the independence of transformations from the control structure, the restriction that transformations are called via XFORM and return via RESPOND, etc. In addition, in order for cell histories to be properly maintained, slave transformers must be used to keep transformations as localized as possible. Consider the result of consing up a new list of clauses to be returned by the transformation:

```
(COND <clauses-1> (T (COND <clauses-2>)))
                     => (COND <clauses-1> <clauses-2>)
```

First of all, since optimizing transformations are required to return equivalent expressions, the transformer would have to return a conditional expression. Thus, after constructing the new list of clauses (perhaps by appending <clauses-1> and <clauses-2>), it creates a new conditional expression via COND!CREATE, which takes a list of clauses and returns a conditional expression. This function will result in the creation of a new *COND* cell; when XFORM realizes that the returned expression is neither the same nor a subcell of the input cell, it will correctly assume that the returned cell is new. The new cell's subtree (the list of clauses) will be installed as the input cell's new monk, but only *after* the subcells of the input conditional have been terminated. Since these same subcells are contained in the newly consed list of clauses, the resulting *COND* cell consists of an empty clause list.

This example illustrates some of the dependencies of the accountable component on the transformation implementor. The implementor needs to be aware of the five different cases into which returned objects may fall, and write his transformers accordingly. Correctly maintaining the cell histories is tricky business, and I have not yet succeeded in making the mechanism robust enough to survive even the good intentions of uninformed transformation implementors.

5.2. Looking Ahead

I spent the bulk of my effort for this thesis in gathering information and developing mechanisms for recording it, rather than using that information. I believe that future work on this system should emphasize the analysis of the transformation process using the information gathered by the recording element. I dream of an accountable system, for example, that could reply to the question "Why didn't this simplifying transformer apply?" with an answer something like "Hmmm, if only I could have shown the predicate <predicate> to be false, I could have completely eliminated the computation <hairy expression>". Or to the question "Why *did* this transformer apply?", reply "if it hadn't been for this assertion you coded in by hand, none of this whole reduction would have happened." That is, the query element could not only *report* the facts, but perform some sort of analysis *using* those facts. It involves more carefully itemizing the prerequisites of a transformation and understanding exactly which of them were met (and why), and which were not. The "why" needs to be slightly more sophisticated than the justifications of the present system; if a prerequisite is met, the system should understand the source of the information which satisfied that prerequisite. Was it an assertion, deduced from context, or always true, and then, what is the significance of that source?

If a transformation did not apply, then how close did it come? To answer this question the system must know not only which prerequisites failed, but be able to suggest ways of satisfying them. Would an assertion solve the problem, or would such an assertion be a contradiction to current information. If a transformation failed because it tried to simplify an OR expression with one argument, for example, but was applied to an OR expression with two arguments, simply asserting that the expression had only one arguments would raise a contradiction.

And finally, though I would enjoy providing a super slick user interface for the accountable transformation system, such improvements are not as dependent on the implementation of this system so much as they are on the capabilities of the user's terminal and its host system. Just the same, I dream of display hacks which use multiple cursors and can remember where a particular s-expression is on the screen. Then instead of having to redisplay an expression to refer to it, the cursor simply jumps to the correct s-expression already displayed. When the user asks to see the next version of the displayed expression, the cursor jumps to the sub-expression about to be transformed, waits for a signal, then pops in the new sub-expression. A status line on the bottom of the screen keeps the user informed as to the current version, expression type, and transformer name. Then of course, menus pop up on the screen and a mouse is available to control the cursor. However, I will leave the implementation of such features to hackers equipped with the necessary terminals.

5.3. The Present

As is always the case, I have not accomplished all I had set out to do in this thesis. Though I believe the present implementation qualifies as an *accountable* system, it is not what I proposed that it be originally: a *responsible* system. To be responsible for its actions, it must have a clearer understanding of those actions and be able to analyze a situation to the extent that it can propose to the user the course of action necessary to correct any problems in the transformation process. An accountable system, however, is a step forward, and does provide a hitherto undeveloped service. It allows the user to observe the sequence of transformations applied and study their interactions with each other, it allows him to select any portion of code in any point in time, and it allows him to control the flow of information returned to him at his request.

References

[Allen and Cocke 1972]

Allen, F.E., and Cocke, J. "A catalogue of optimizing transformations." Design and Optimization of Compilers, R. Rustin, Ed. Englewood Cliffs, NJ, Prentice-Hall, 1972, pp 1-30

[Atkinson 1976]

Atkinson, R.R. *Optimization techniques for a structured programming language.* S.M. Thesis., MIT, Cambridge, MA. 1976.

[Bagwell 1970]

Bagwell, J.T. "Local Optimizations." SIGPLAN Proceedings, Vol 5, No 7, (July 1970), pp 52-66

[Boyle 1970]

Boyle, James M. *A Transformational Component for Programming Language Grammar.* Argonne National Laboratory Report ANL-7690, Argonne, Illinois, July 1970.

[Boyle 1976]

Boyle, James M. "Mathematical Software Transportability Systems – Have the Variations a Theme?" Proceedings of Workshop on Portability of Numerical Software, June 1976.

[Boyle and Matz 1977]

Boyle, James M. and Matz, Marilyn. "Automating Multiple Program Realizations." Proc. of the M.R.I. International Symposium XXIV: Computer Software Engineering. Polytechnic Press, Brooklyn, N.Y., 1977.

[Davis 1978]

Davis, Randall *Interactive Transfer of Expertise: Acquisition of New Inference Rules.* Computer Science Dept, Stanford Univ., Stanford, CA, December 1978.

[Gerhart 1975]

Gerhart, S. L. "Correctness-preserving program transformations." Conf. Rec. Second ACM Symp. on Princples of Programming Languages, January 1975, pp 54-66

[Geschke 1972]

Geschke, C.M. *Global Program Optimizations.* PhD. Th., Computer Science Department, Carnegie-Mellon University, 1972

[Kerns 1977]

Kerns, Barbara S. *An Experiment in Information Hiding.* Bachelor's Thesis Greenville College, Greenville, Illinois, (May 1977)

[Loveman 1977]

Loveman, David B. "Program Improvement by Source-to-Source Transformation." Journal of the ACM, Vol 24, No.1, January 1977, pp 121-145

[Loveman and Faneuf 1975]

Loveman, D. and Faneuf, R. "Program Optimization - theory and practice." Proc. of Conf. on Programming Languages and Compilers for Parallel and Vector Machines., SIGPLAN Notices (ACM) 10, 3 (March 1975), pp 97-102

[Nievergelt 1965]

Nievergelt, J. "On the automatic simplification of computer programs." Comm ACM, Vol 8, No 6 (June 1965), pp 366-370

[Pitman 1979]

Pitman, Kent M. "A Fortran->Lisp Translator." Proc. of Macsyma Users Conference, June 20-22, 1979, Washington, D.C., pp 200-214.

[Pitman 1980]

Pitman, Kent M. "Special Forms in LISP." to appear in proceedings of the Lisp Conference, August 24-27, 1980, Stanford, California.

[Schaefer 1973]

Schaefer, M. *A Mathematical Theory of Global Program Optimization.* Prentice-Hall, Englewood Cliffs, NJ, 1973.

[Scheifler 1977]

Scheifler, R. W. "An analysis of inline substitution for a structured programming language." Comm. ACM, Vol 20, No 9 (September 1977), pp 647-654

[Standish, et al 1976]

Standish, T., Harriman, D., Kibler, D., and Neighbors, J. M. *The Irvine Program Transformation Catalogue.* Computer Science Dept, U.C. Irvine, Irvine, CA (January 1976).

[Standish, et al 1976]

Standish, T., Harriman, D., Kibler, D., and Neighbors, J. M. "Improving and refining programs by program manipulation." Proc. 1976 ACM Annual Conf., Oct. 20-22,1976, pp 509-516

[Steele 1980]

Steele, Barbara K. "Strategies for Data Abstraction in LISP." to appear in Proc. of LISP Conference, August 24-27, 1980, Stanford, CA

[Schwartz 1974]

Schwartz, J.T. "Automatic and semiautomatic optimization of SETL." Proc. ACM Symp. Very High Level Languages, SIGPLAN Notices, Vol 9, No 4, April 1974

[Wegbreit 1976]

Wegbreit, B. "Goal-directed program transformation." IEEE Trans. on Software Engineering. SE-2, 2 (June 1976), pp 69-80

DATE
FILMED

8