Technical Report 474

RABBIT: A Compiler for SCHEME

Guy Lewis Steele

MIT Artificial Intelligence Laboratory

UNCLASSIFIED   sent 12/7/78   ADA 061996

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR474	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization)		5. TYPE OF REPORT & PERIOD COVERED Technical report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Guy Lewis Steele		8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0643
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 May 1978
		13. NUMBER OF PAGES 272
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)

compiler optimization	tail-recursion
code generation	lambda calculus
LISP	lexical scoping
macros	continuations

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

We have developed a compiler for the lexically-scoped dialect of LISP known as SCHEME. The compiler knows relatively little about specific data manipulation primitives such as arithmetic operators, but concentrates on general issues of environment and contral. Rather than having specialized knowledge about a large variety of control and environment constructs, the compiler handles only a small basis set which reflects the semantics of lambda-calculus. All of the traditional imperative constructs, such as sequencing, assignment, looping, GOTO, as well as many standard (cont'd)

DD FORM 1 JAN 73 1473   EDITION OF 1 NOV 65 IS OBSOLETE   UNCLASSIFIED
S/N 0102-014-6601
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

BLOCK 20 CONTINUED

LISP constructs such as AND, OR, and COND, are expressed as macros in terms of the applicative basis set. A small number of optimization techniques, coupled with the treatment of function calls as GOTO statements, serve to produce code as good as that produced by more traditional compilers. The macro approach enables speedy implementation of new constructs as desired without sacrificing efficiency in the generated code.

A fair amount of analysis is devoted to determining whether environments may be stack-allocated or must be heap-allocated. Heap-allocated environments are necessary in general because SCHEME (unlike Algol 60 and Algol 68, for example) allows procedures with free lexically scoped variable to be returned as the values of other procedures: the Algol stack-allocation environment strategy does not suffice. The methods used here indicate that a heap-allocating generalizaiton of the "display" technique leads to an efficient implementation of such "upward funargs". Moreover, compile-time optimization and analysis can eliminate many "funargs" entirely, and so far fewer environment structures need be allocated at run time than might be expected.

A subset of SCHEME (rather than triples, for example) serves as the representation intermedieate between the optimized SCHEME code and the final output code; code is expressed in this subset in the so'called constinuation-passing style. As a subset of SCHEME, it enjoys the same theoretical properties; one could even apply the same optimizer used on the input code to the intermediate code. However, the subset is so chosen that all temporary quantities are made manifest as variables, and no control stack is needed to evaluate it. As a result, this apparently applicative representation admits an imperative interpretation which permits easy transcription to fianl imperative machine code. These qualities suggest that an applicative language like SCHEME is a better candidate for an UNCOL than the more imperative candidates proposed to date.

This research was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-75-C-0643.

RABBIT:
A Compiler for SCHEME

(A Dialect of LISP)

A Study in

Compiler Optimization

Based on Viewing
LAMBDA as RENAME
and
PROCEDURE CALL as GOTO

using the techniques of

Macro Definition of Control and Environment Structures
Source-to-Source Transformation
Procedure Integration
and
Tail-Recursion

Guy Lewis Steele Jr.

Massachusetts Institute of Technology

May 1978

Revised version of a dissertation submitted (under the title "Compiler Optimization Based on Viewing LAMBDA as RENAME plus GOTO") to the Department of Electrical Engineering and Computer Science on May 12, 1977, in partial fulfillment of the requirements for the degree of Master of Science.

RABBIT: A Compiler for SCHEME (A Dialect of LISP)

A Study in Compiler Optimization
Based on Viewing LAMBDA as RENAME and PROCEDURE CALL as GOTO

using the techniques of
Macro Definition of Control and Environment Structures,
Source-to-Source Transformation, Procedure Integration, and Tail-Recursion

Guy Lewis Steele Jr.
Massachusetts Institute of Technology
May 1978

ABSTRACT

We have developed a compiler for the lexically-scoped dialect of LISP known as SCHEME. The compiler knows relatively little about specific data manipulation primitives such as arithmetic operators, but concentrates on general issues of environment and control. Rather than having specialized knowledge about a large variety of control and environment constructs, the compiler handles only a small basis set which reflects the semantics of lambda-calculus. All of the traditional imperative constructs, such as sequencing, assignment, looping, GOTO, as well as many standard LISP constructs such as AND, OR, and COND, are expressed as macros in terms of the applicative basis set. A small number of optimization techniques, coupled with the treatment of function calls as GOTO statements, serve to produce code as good as that produced by more traditional compilers. The macro approach enables speedy implementation of new constructs as desired without sacrificing efficiency in the generated code.

A fair amount of analysis is devoted to determining whether environments may be stack-allocated or must be heap-allocated. Heap-allocated environments are necessary in general because SCHEME (unlike Algol 60 and Algol 68, for example) allows procedures with free lexically scoped variables to be returned as the values of other procedures; the Algol stack-allocation environment strategy does not suffice. The methods used here indicate that a heap-allocating generalization of the "display" technique leads to an efficient implementation of such "upward funargs". Moreover, compile-time optimization and analysis can eliminate many "funargs" entirely, and so far fewer environment structures need be allocated at run time than might be expected.

A subset of SCHEME (rather than triples, for example) serves as the representation intermediate between the optimized SCHEME code and the final output code; code is expressed in this subset in the so-called continuation-passing style. As a subset of SCHEME, it enjoys the same theoretical properties; one could even apply the same optimizer used on the input code to the intermediate code. However, the subset is so chosen that all temporary quantities are made manifest as variables, and no control stack is needed to evaluate it. As a result, this apparently applicative representation admits an imperative interpretation which permits easy transcription to final imperative machine code. These qualities suggest that an applicative language like SCHEME is a better candidate for an UNCOL than the more imperative candidates proposed to date.

Thesis Supervisor: Gerald Jay Sussman
Title: Associate Professor of Electrical Engineering

Note

The first part of this report is a slightly revised version of a dissertation submitted in May 1977. Where it was of historical interest to reflect changes in the SCHEME language which ocurred in the following year and the effect they had on RABBIT, the text was left intact, with notes added of the form, "Since the dissertation was written, thus-and-so occurred." The second part, the Appendix, was not part of the dissertation, and is a complete listing of the source code for RABBIT, with extensive commentary.

It is intended that the first part should be self-contained, and provide a qualitative overview of the compilation methods used in RABBIT. The second part is provided for those readers who would like to examine the precise mechanisms used to carry out the general methods.

Thus there are five levels of thoroughness at which the reader may consume this document:
(1) The reader who wishes only to skim is advised to read sections 1, 5, 6, possibly 7, 8A, 8B, 8C, 10, 11, and 12. This will give a basic overview, including the use of macros and the optimizing techniques.
(2) The reader who also wants to know about the details of SCHEME, the run-time system, and a long example is advised to read the entire main text (about a third of the document).
(3) The reader who wants to understand the low-level organization of the algorithms, and read about the more tricky special cases, should read the main text and then the commentary on the code.
(4) The reader who additionally wants to understand the nit-picking details should read the code along with the commentary.
(5) The reader who wants a real feel for the techniques involved should read the entire document, invent three new SCHEME constructs and write macros for them, and then reimplement the compiler for another run-time environment. (He ought please also to send a copy of any documents on such a project to this author, who would be very interested!)

Acknowledgements

I would like to acknowledge the contributions to this work of the following people and other entities:

Gerald Jay Sussman, who is not only my thesis advisor but a colleague and a good friend; who is fun to hack programs with; who not only provided insights on the issues of programming, but also was willing to give me a kick in the right direction when necessary;

Jon Doyle, one of the first real "users" of SCHEME, who was always willing to discuss my problems, and who carefully proofread the thesis in one day when no one else would or could;

Richard Zippel, the other first real SCHEME user, who has discussed with me many possibilities for the practical use of SCHEME-like languages in such large systems as MACSYMA;

Carl Hewitt, whose actors metaphor inspired in part first SCHEME and then the investigations presented here;

Scott Fahlman, who has Great Ideas, and who paid some of his dues at the same place I did;

Jon L White, resident LISP compiler expert and agreeable office-mate, who likes both tea and ();

Dan Weinreb, Bernie Greenberg, Richard Stallman, Dave Moon, Howard Cannon, Alan Bawden, Henry Baker, and Richard Greenblatt for their companionship, advice, comments, enthusiasm, criticism, and/or constructive opposition;

the rest of the gang at the AI Lab and Project MAC (loosely known as the Lab for Computer Science), for their continued interest in my work and for the pleasant social atmosphere they provide;

Bill Wulf, Charles Geschke, Richard Johnsson, Charles Weinstock, and Steven Hobbs, whose work on BLISS-11 I found a great inspiration, for it told me that there was at least one beautiful compiler already;

Dan Friedman and Dave Wise, who also know that LISP is the One True Way;

Dick Gabriel, a most singular person (that's odd...), who knows that Lapin is best dealt with gingerly;

the National Science Foundation, which provided the fellowship under which this work was done;

Cindy Ellis and J.J. McCabe, who always treated me as just a regular guy;

Julie Genovese, my main (and only) groupie;

the congregation at the Brighton Evangelical Congregational Church, for their social and moral support;

Mittens Jr., our cat, who was willing to communicate when the rest of the world was asleep;

Chuck, the peculiar poodle, who carried on as best she could after Mittens Jr. had gone, and who still barks in the night;

my brother, David A. Steele, who has kept me up to date on cultural affairs, and who probably understands me better than anyone else;

and my parents, Guy L. Steele Sr. and Nalora Steele, who provided unbounded amounts of patience, encouragement, opportunity, and support.

Contents

1. Introduction

The work described here is a continuation (!) of that described in [SCHEME], [Imperative], and [Declarative]. Before enumerating the points of the thesis, we summarize here each of these documents.

A. Background

In [SCHEME] we (Gerald Jay Sussman and the author) described the implementation of a dialect of LISP named SCHEME with the properties of lexical scoping and tail-recursion; this implementation is embedded within MacLISP [Moon], a version of LISP which does not have these properties. The property of lexical scoping (that a variable can be referenced only from points textually within the expression which binds it) is a consequence of the fact that all functions are closed in the "binding environment". [Moses] That is, SCHEME is a "full-funarg" LISP dialect. {Note Full-Funarg Example} The property of tail-recursion implies that loops written in an apparently recursive form will actually be executed in an iterative fashion. Intuitively, function calls do not "push control stack"; instead, it is argument evaluation which pushes control stack. The two properties of lexical scoping and tail-recursion are not independent. In most LISP systems [LISP1.5M] [Moon] [Teitelman], which use dynamic scoping rather than lexical, tail-recursion is impossible because function calls must push control stack in order to be able to undo the dynamic bindings after the return of the function. On the other hand, it is possible to have a lexically scoped LISP which does not tail-recurse, but it is easily seen that such an implementation only wastes storage space needlessly compared to a tail-recursing implementation. [Steele] Together, these two properties cause

SCHEME to reflect lambda-calculus semantics much more closely than dynamically scoped LISP systems. SCHEME also permits the treatment of functions as full-fledged data objects; they may be passed as arguments, returned as values, made part of composite data structures, and notated as independent, unnamed ("anonymous") entities. (Contrast this with most ALGOL-like languages, in which a function can be written only by declaring it and giving it a name; imagine being able to use an integer value only by giving it a name in a declaration!) The property of lexical scoping allows this to be done in a consistent manner without the possibility of identifier conflicts (that is, SCHEME "solves the FUNARG problem" [Moses]). In [SCHEME] we also discussed the technique of "continuation-passing style", a way of writing programs in SCHEME such that no function ever returns a value.

In [Imperative] we explored ways of exploiting these properties to implement most traditional programming constructs, such as assignment, looping, and call-by-name, in terms of function application. Such applicative (lambda-calculus) models of programming language constructs are well-known to theoreticians (see [Stoy], for example), but have not been used in a practical programming system. All of these constructs are actually made available in SCHEME by macros which expand into these applicative definitions. This technique has permitted the speedy implementation of a rich user-level language in terms of a very small, easy-to-implement basis set of primitive constructs. In [Imperative] we continued the exploration of continuation-passing style, and noted that the escape operator CATCH is easily modelled by transforming a program into this style. We also pointed out that transforming a program into this style enforces a particular order of argument evaluation, and makes all intermediate computational quantities manifest as variables.

In [Declarative] we examined more closely the issue of tail-recursion, and demonstrated that the usual view of function calls as pushing a return address must lead to an either inefficient or inconsistent implementation, while the tail-recursive approach of SCHEME leads to a uniform discipline in which function calls are treated as GOTO statements which also pass arguments. We also noted that a consequence of lexical scoping is that the only code which can reference the value of a variable in a given environment is code which is closed in that environment or which receives the value as an argument; this in turn implies that a compiler can structure a run-time environment in any arbitrary fashion, because it will compile all the code which can reference that environment, and so can arrange for that code to reference it in the appropriate manner. Such references do not require any kind of search (as is commonly and incorrectly believed in the LISP community because of early experience with LISP interpreters which search a-lists) because the compiler can determine the precise location of each variable in an environment at compile time. It is not necessary to use a standard format, because neither interpreted code nor other compiled code can refer to that environment. (This is to be constrasted with "spaghetti stacks" [Bobrow and Wegbreit].) In [Declarative] we also carried on the analysis of continuation-passing style, and noted that transforming a program into this style elucidates traditional compilation issues such as register allocation because user variables and intermediate quantities alike are made manifest as variables on an equal footing. Appendix A of [Declarative] contained an algorithm for converting any SCHEME program (not containing ASET) to continuation-passing style.

We have implemented two compilers for the language SCHEME. The purpose was to explore compilation techniques for a language modelled on lambda-calculus, using lambda-calculus-style models of imperative programming constructs. Both compilers use the strategy of converting the source program to continuation-passing style.

The first compiler (known as CHEAPY) was written as a throw-away implementation to test the concept of conversion to continuation-passing style. The first half of CHEAPY is essentially the algorithm which appears in Appendix A of [Declarative], and the second is a simple code generator with almost no optimization. In conjunction with the writing of CHEAPY, the SCHEME interpreter was modified to interface to compiled functions. (This interface is described later in this report.)

The second compiler, with which we are primarily concerned here, is known as RABBIT. It, like CHEAPY, is written almost entirely in SCHEME (with minor exceptions due only to problems in interfacing with certain MacLISP I/O facilities). Unlike CHEAPY, it is fairly clever. It is intended to demonstrate a number of optimization techniques relevant to lexical environments and tail-recursive control structures. (The code for RABBIT, with commentary, appears in the Appendix.)

B. The Thesis

(1) Function calls are not expensive when compiled correctly; they should be thought of as GOTO statements that happen to pass arguments.

(2) The combination of cheap function calls, lexical scoping, tail-recursion, and "anonymous" notation of functions (which are not independent properties of a language, but aspects of a single unified approach) permits the definition of a wide variety of "imperative" constructs in applicative terms. Because these properties result from adhering to the principles of the well-known lambda-calculus [Church], such definitions can be lifted intact from existing literature and used directly.

(3) A macro facility (the ability to specify syntactic transformations) makes it practical to use these as the only definitions of imperative constructs in a programming system. Such a facility makes it extremely easy to define new constructs.

(4) A few well-chosen optimization strategies enable the compilation of these applicative definitions into the imperative low-level code which one would expect from a traditional compiler.

(5) The macro facility and the optimization techniques used by the compiler can be conceptually unified. The same properties which make it easy to write the macros make it easy to define optimizations correctly. In the same way that many programming constructs are defined in terms of a small, well-chosen basis set, so a large number of traditional optimization techniques fall out as special cases of the few used in RABBIT. This is no accident. The separate treatment of a large and diverse set of constructs necessitates separate optimization techniques for each. As the basis set of constructs is reduced, so is the set of interesting transformations. If the basis set is properly chosen, their combined effect is "multiplicative" rather than "additive".

(6) The technique of compiling by converting to continuation-passing style elucidates some important compilation issues in a natural way. Intermediate quantities are made manifest; so is the precise order of evaluation. Moreover, this is all expressed in a language isomorphic to a subset of the source language SCHEME; as a result the continuation-passing style version of a program inherits many of the philosophical and practical advantages. For example, the same optimization techniques can be applied at this level as at the original source level. While the use of continuation-passing style may not make the decisions any easier, it provides an effective and

natural way to express the results of those decisions.

(7) Continuation-passing style, while apparently applicative in nature, admits a peculiarly imperative interpretation as a consequence of the facts that it requires no control stack to be evaluated and that no functions ever return values. As a result, it is easily converted to an imperative machine language.

(8) A SCHEME compiler should ideally be a designer of good data structures, since it may choose any representation whatsoever for environments. RABBIT has a rudimentary design knowledge, involving primarily the preferral of registers to heap-allocated storage. However, there is room for knowledge of "bit-diddling" representations.

(9) We suggest that those who have tried to design useful UNCOL's (UNiversal Computer-Oriented Languages) [Sammet] [Coleman] have perhaps been thinking too imperatively, and worrying more about data manipulation primitives than about environment and control issues. As a result, proposed UNCOLs have been little more than generalizations of contemporary machine languages. We suggest that SCHEME makes an ideal UNCOL at two levels. The first level is the fully applicative level, to which most source-language constructs are easily reduced; the second is the continuation-passing style level, which is easily reduced to machine language. We envision building a compiler in three stages: (a) reduction of a user language to basic SCHEME, whether by macros, a parser of algebraic syntax, or some other means; (b) optimization by means of SCHEME-level source-to-source transformations, and conversion to continuation-passing style; and (c) generation of code for a particular machine. RABBIT addresses itself to the second stage. Data manipulation primitives are completely ignored at this stage, and are just passed along from input to output. These primitives, whether integer arithmetic, string

concatenation and parsing, or list structure manipulators, are chosen as a function of a particular source language and a particular target machine. RABBIT deals only with fundamental environment and control issues common to most modes of algorithmic expression.

(10) While syntactic issues tend to be rather superficial, we point out that algebraic syntax tends to obscure the fundamental nature of function calling and tail-recursion by arbitrarily dividing functions into syntactic classes such as "operators" and "functions". ([Standish], for example, uses much space to exhibit each conceptually singular transformation in a multiplicity of syntactic manifestations.) The lack of an "anonymous" notation for functions in most algebraic languages, and the inability to treat functions as data objects, is a distinct disadvantage. The uniformity of LISP syntax makes these issues easier to deal with.

To the LISP community in particular we address these additional points:

(11) Lexical scoping need not be as expensive as is commonly thought. Experience with lexically-scoped interpreters is misleading; lexical scoping is not inherently slower than dynamic scoping. While some implementations may entail access through multiple levels of structure, this occurs only under circumstances (accessing of variables through multiple levels of closure) which could not even be expressed in a dynamically scoped language. Unlike deep-bound dynamic variables, compiled lexical access requires no search; unlike shallow-bound dynamic variables, lexical binding does not require that values be put in a canonical value cell. The compiler has complete discretion over the manipulation of environments and variable values. The "display" technique used in Algol implementations can be generalized to provide an efficient solution to the FUNARG problem.

(12) Lexical scoping does not necessarily make LISP programming unduly difficult.

The very existence of RABBIT, a working compiler some fifty pages in length written in SCHEME, first implemented in about a month, part-time, substantiates this claim (which is, however, admitted to be mostly a matter of taste and experience). {Note Refinement of RABBIT} SCHEME has also been used to implement several AI problem-solving languages, including AMORD [Doyle].

2. The Source Language - SCHEME

The basic language processed by RABBIT is a subset of the SCHEME language as described in [SCHEME] and [Revised Report], the primary restrictions being that the first argument to ASET must be quoted and that the multiprocessing primitives are not accommodated. This subset is summarized here.

SCHEME is essentially a lexically scoped ("full funarg") dialect of LISP. Interpreted programs are represented by S-expressions in the usual manner. Numbers represent themselves. Atomic symbols are used as identifiers (with the conventional exception of T and NIL, which are conceptually treated as constants). All other constructs are represented as lists.

In order to distinguish the various other constructs, SCHEME follows the usual convention that a list whose car is one of a set of distinguished atomic symbols is treated as directed by a rule associated with that symbol. All other lists (those with non-atomic cars, or with undistinguished atoms in their cars) are <u>combinations</u>, or function calls. All subforms of the list are uniformly evaluated in an unspecified order, and then the value of the first (the function) is applied to the values of all the others (the arguments). Notice that the function position is evaluated in the same way as the argument positions (unlike most other LISP systems). (In order to be able to refer to MacLISP functions, global identifiers evaluate to a special kind of functional object if they have definitions as MacLISP functions of the EXPR, SUBR, or LSUBR varieties. Thus "(PLUS 1 2)" evaluates to 3 because the values of the subforms are <functional object for PLUS>, 1, and 2; and applying the first to the other two causes invocation of the MacLISP primitive PLUS.)

The atomic symbols which distinguish special constructs are as follows:

LAMBDA   This denotes a function. A form (LAMBDA (var1 var2 ... varn) body) will evaluate to a function of n arguments. The _parameters_ vari are identifiers (atomic symbols) which may be used in the body to refer to the respective _arguments_ when the function is invoked. Note that a LAMBDA-expression is not a function, but _evaluates_ to one, a crucial distinction.

IF This denotes a conditional form. (IF a b c) evaluates the _predicate_ a, producing a value x; if x is non-NIL, then the _consequent_ b is evaluated, and otherwise the _alternative_ c. If c is omitted, NIL is assumed.

QUOTE As in all LISP systems, this provides a way to specify any S-expression as a constant. (QUOTE x) evaluates to the S-expression x. This may be abbreviated to 'x, thanks to the MacLISP read-macro-character feature.

LABELS This primitive permits the local definition of one or more mutually recursive functions. The format is:

```
(LABELS ((name1 (LAMBDA ...))
         (name2 (LAMBDA ...))
         ...
         (namen (LAMBDA ...)))
        body)
```

This evaluates the body in an environment in which the names refer to the respective functions, which are themselves closed in that same environment. Thus references to these names in the bodies of the LAMBDA-expressions will refer to the labelled functions. {Note Generalized LABELS}

ASET' This is the primitive side-effect on variables. (ASET' var body) evaluates the body, assigns the resulting value to the variable var, and returns that value. {Note Non-quoted ASET} For implementation-dependent reasons, it is forbidden by RABBIT to use ASET' on a global variable which is the name of a primitive MacLISP function, or on a variable bound by LABELS. (ASET' is actually used very seldom in practice anyway, and all these restrictions are "good programming practice". RABBIT could be altered to lift these restrictions, at some expense and labor.)

CATCH This provides an escape operator facility. [Landin] [Reynolds] (CATCH var body) evaluates the body, which may refer to the variable var, which will denote an "escape function" of one argument which, when called, will return from the CATCH-form with the given argument as the value of the CATCH-form. Note that it is entirely possible to return from the CATCH-form several times. This raises a difficulty with optimization which will be discussed later.

Macros Any atomic symbol which has been defined in one of various ways to be a macro distinguishes a special construct whose meaning is determined by a macro function. This function has the responsibility of rewriting the form and returning a new form to be evaluated in place of the old one. In this way complex syntactic constructs can be expressed in terms of simpler ones.

3. The Target Language

The "target language" is a highly restricted subset of MacLISP, rather than any particular machine language for an actual hardware machine such as the PDP-10. RABBIT produces MacLISP function definitions which are then compiled by the standard MacLISP compiler. In this way we do not need to deal with the uninteresting vagaries of a particular piece of hardware, nor with the peculiarities of the many and various data-manipulation primitives (CAR, RPLACA, +, etc.). We allow the MacLISP compiler to deal with them, and concentrate on the issues of environment and control which are unique to SCHEME. While for production use this is mildly inconvenient (since the code must be passed through two compilers before use), for research purposes it has saved the wasteful re-implementation of much knowledge already contained in the MacLISP compiler.

On the other hand, the use of MacLISP as a target language does not by any means trivialize the task of RABBIT. The MacLISP function-calling mechanism cannot be used as a target construct for the SCHEME function call, because MacLISP's function calls are not guaranteed to behave tail-recursively. Since tail-recursion is a most crucial characteristic distinguishing SCHEME from most LISP systems, we must implement SCHEME function calls by more primitive methods. Similarly, since SCHEME is a full-funarg dialect of LISP while MacLISP is not, we cannot in general use MacLISP's variable-binding mechanisms to implement those of SCHEME. On the other hand, it is a perfectly legitimate optimization to use MacLISP mechanisms in those limited situations where they are applicable.

Aside from ordinary MacLISP data-manipulation primitives, the only MacLISP constructs used in the target language are PROG, GO, RETURN, PROGN, COND, SETQ, and ((LAMBDA ...) ...). PROG is never nested; there is only a single, outer PROG. RETURN is used only in the form (RETURN NIL) to exit this outer

PROG; it is never used to return a value of any kind. LAMBDA-expressions are used only to bind temporary variables. In addition, CONS, CAR, CDR, RPLACA, and RPLACD are used in the creation and manipulation of environments.

We may draw a parallel between each of these constructs and an equivalent machine-language (or rather, assembly language) construct:

PROG	A single program module.
GO	A branch instruction. PROG tags correspond to instruction labels.
RETURN	Exit from program module.
PROGN	Sequencing of several instructions.
COND	Conditional branches, used in a disciplined manner. One may think of

```
(COND (pred1 value1)
      (pred2 value2)
      ...
      (predn valuen))
```

as representing the sequence of code

```
        <code for pred1>
        JUMP-IF-NIL reg1,TAG1
        <code for value1>
        JUMP ENDTAG
TAG1:   <code for pred2>
        JUMP-IF-NIL reg1,TAG2
        <code for value2>
        JUMP ENDTAG
TAG2:   ...
        <code for predn>
        JUMP-IF-NIL reg1,TAGn
        <code for valuen>
        JUMP ENDTAG
TAGn:   LOAD-VALUE NIL
ENDTAG:
```

which admits of some optimizations, but we shall not worry about this. (The MacLISP compiler does, but we do not depend at all on this fact.)

SETQ Load register, or store into memory.

LAMBDA We use this primarily in the form:

```
((LAMBDA (q1 ... qn)
         (setq var1 q1)
         ...
         (setq varn qn))
 value1 ... valuen)
```

which we may think of as saving values on a temporary stack and then popping them into the variables:

```
<code for value1>              ;leaves result in reg1
PUSH reg1
...
<code for valuen>
PUSH reg1
POP varn
...
POP var1
```

This is in fact approximately how the MacLISP compiler will treat this construct. This is used to effect the simultaneous assignment of several values to several registers. It would be possible to do without the MacLISP LAMBDA in this case, by using extra intermediate variables, but it was decided that this task was less interesting than other issues within RABBIT, and that assignments of this kind would occur sufficiently often that it was desirable to get the MacLISP compiler to produce the best possible code in this case.

The form ((LAMBDA ...) ...) is also used in some situation where the user wrote such a form in the SCHEME code, and the arguments and LAMBDA-body are all "trivial", in a sense to be defined later.

CONS CONS is used, among other things, to "push" new values onto the current environment. While SCHEME variables can sometimes be represented as temporary MacLISP variables using LAMBDA, in general they must be kept in a "consed environment" in the heap; CAR and CDR are used to "index" the environment "stack" (which is not really a stack, but in general tree-like). (N.B. By using CONS for this purpose we can push the entire issue of environment retention off onto the LISP garbage collector. It would be possible to use array-like blocks for environments, and an Algol-like "display" pointer discipline for variable access. However, a retention strategy as opposed to a deletion strategy must be used in general, because SCHEME, unlike Algol 60 and 68, permits procedures to be the values of other procedures. Stack allocation does not suffice in general -- a heap must be used. Later we will see that RABBIT uses stack allocation of environments and a deletion strategy in simple cases, and reverts to heap allocation of environments and a retention strategy in more complicated situations.)

CAR, + Primitive MacLISP operators such as + and CAR are analogous to machine-language instructions such as ADD and LOAD-INDEXED. We leave to the MacLISP compiler the task of compiling large expressions involving these; but we are not avoiding the associated difficult issues such as register allocation, for we shall have to deal with them in compiling calls to SCHEME functions.

4. The Target Machine

Compiled code is interfaced to the SCHEME interpreter in two ways. The interpreter must be able to recognize functional objects which happen to be compiled and to invoke them with given arguments; and compiled code must be able to invoke any function, whether interpreted or compiled, with given arguments. (This latter interface is traditionally known as the "UUO Handler" as the result of the widespread use of the PDP-10 in implementing LISP systems. [DEC] [Moon] [Teitelman]) We define here an arbitrary standard form for functional objects, and a standard means for invoking them.

In the PDP-10 MacLISP implementation of SCHEME, a function is, in general, represented as a list whose car contains one of a set of distinguished atomic symbols. (Notice that LAMBDA is _not_ one of these; a LAMBDA-expression may _evaluate_ to a function, but is not itself a valid function.) This set of symbols includes EXPR, SUBR, and LSUBR, denoting primitive MacLISP functions of those respective types; BETA, denoting a SCHEME function whose code is interpretive; DELTA, denoting an escape function created by the interpreter for a CATCH form, or a continuation given by the interpreter to compiled code; CBETA, denoting a SCHEME function or continuation whose code is compiled; and EPSILON, denoting a continuation created when compiled code invokes interpreted code. Each of these function types requires a different invocation convention; the interpreter must distinguish these types and invoke them in the appropriate manner. For example, to invoke an EXPR the MacLISP FUNCALL construct must be used. A BETA must be invoked by creating an appropriate environment, using the given arguments, and then interpreting the code of the function.

We have arbitrarily defined the CBETA interface as follows: there are a number of "registers", in the form of global variables. Nine registers called

CONT, **ONE**, **TWO**, ..., **EIGHT** are used to pass arguments to compiled functions. **CONT** contains the continuation. The others contain the arguments prescribed by the user; if there are more than eight arguments, however, then they are passed as a list of all the arguments in register **ONE**, and the others are unused. (Any of a large variety of other conventions could have been chosen, such as the first seven arguments in seven registers and a list of all remaining arguments in **EIGHT**. We merely chose a convention which would be workable and convenient, reflect the typical finiteness of hardware register sets, and mirror familiar LISP conventions. The use of a list of arguments is analogous to the passing of an arbitrary number of arguments on a stack, sometimes known as the LSUBR convention. [Moon] [Declarative])

There is another register called **FUN**. A function is invoked by putting the functional object in **FUN**, its arguments in the registers already described, and the number of arguments in the register **NARGS**, and then exiting the current function. Control (at the MacLISP level) is then transferred to a routine (the "SCHEME UUO handler") which determines the type of the function in **FUN** and invokes it.

A continuation is invoked in exactly the same manner as any other kind of function, with two exceptions: a continuation does not itself require a continuation, so **CONT** need not be set up; and a continuation always takes a single argument, so **NARGS** need not be set to 1. {Note Multiple-Argument Continuations}

A CBETA form has additional fixed structure. Besides the atomic symbol CBETA in the car, there is always in the cadr the address of the code, and in the cddr the environment. The form of the environment is completely arbitrary as far as the SCHEME interpreter is concerned; indeed, the CHEAPY compiler and the RABBIT compiler use completely different formats for environments for compiled function. (Recall that this cannot matter since the only code which will ever be able to access that environment is the code belonging the the functional closure of which that environment is a part.) The "UUO handler" puts the cddr of **FUN** in the register **ENV**, and then transfers to the address in the cadr of **FUN**. When that code eventually exits, control returns to the "UUO handler", which expects the code to have set up **FUN** and any necessary arguments.

There is a set of "memory locations" -11-, -12-, ... which are used to hold intermediate quantities within a single user function. (Later we shall see that we think of these as being used to pass values between internally generated functions within a module. For this purpose we think of the "registers" and "memory locations" being arranged in a single sequence **CONT**, **ONE**, ..., **EIGHT**, -11-, -12-, ... There is in principle an unbounded number of these "memory locations", but RABBIT can determine (and in fact outputs as a declaration for the MacLISP compiler) the exact set of such locations used by any given function.) One may think of the "memory locations" as being local to each module, since they are never used to pass information between modules; in practice they are implemented as global MacLISP variables.

The registers **FUN**, **NARGS**, **ENV**, and the argument registers are the only global registers used by compiled SCHEME code (other than the "memory locations"). Except for global variables explicitly mentioned by the user program, all communication between compiled SCHEME functions is through these registers. It is useful to note that the continuation in **CONT** is generally analogous to the usual "control stack" which contains return addresses, and so we may think of **CONT** as our "stack pointer register".

5. Language Design Considerations

SCHEME is a lexically scoped ("full-funarg") dialect of LISP, and so is an applicative language which conforms to the spirit of the lambda-calculus. [Church] We divide the definition of the SCHEME language into two parts: the environment and control constructs, and the data manipulation primitives. Examples of the former are LAMBDA-expressions, combinations, and IF; examples of the latter are CONS, CAR, EQ, and PLUS. Note that we can conceive of a version of SCHEME which did not have CONS, for example, and more generally did not have S-expressions in its data domain. Such a version would still have the same environment and control constructs, and so would hold the same theoretical interest for our purposes here. (Such a version, however, would be less convenient for purposes of writing a meta-circular description of the language, however!)

By the "spirit of lambda-calculus" we mean the essential properties of the axioms obeyed by lambda-calculus expressions. Among these are the rules of alpha-conversion and beta-conversion. The first intuitively implies that we can uniformly rename a function parameter and all references to it without altering the meaning of the function. An important corollary to this is that we can in fact effectively locate all the references. The second implies that in a situation where a known function is being called with known argument expressions, we may substitute an argument expression for a parameter reference within the body of the function (provided no naming conflicts result, and that certain restrictions involving side effects are met). Both of these operations are of importance to an optimizing compiler. Another property which follows indirectly is that of tail-recursion. This property is exploited in expressing iteration in terms of applicative constructs, and is discussed in some detail in

[Declarative].

We realize that other systems of environment and control constructs also are reasonably concise, clear, and elegant, and can be axiomatized in useful ways, for example the guarded commands of Dijkstra. [Dijkstra] However, that of lambda-calculus is extremely well-understood, lends itself well to certain kinds of optimizations in a natural manner, and has behind it a body of literature which can be used directly by RABBIT to express non-primitive constructs.

The desire for uniform lexical scoping arises from other motives as well, some pragmatic, some philosophical. Many of these are described in [SCHEME], [Imperative], [Declarative], and [Revised Report]. It is often difficult to explain some of these to those who are used to dynamically scoped LISP systems. Any one advantage of lexical scoping may often be countered with "Yes, but you can do that in this other way in a dynamically scoped LISP." However, we are convinced that lexical scoping in its totality provides all of the advantages to be described in a natural, elegant, and integrated manner, largely as a consequence of its great simplicity.

There are those to whom lexical scoping is nothing new, for example the ALGOL community. For this audience, however, we should draw attention to another important feature of SCHEME, which is that functions are first-class data objects. They may be assigned or bound to variables, returned as values of other functions, placed in arrays, and in general treated as any other data object. Just as numbers have certain operations defined on them, such as addition, so functions have an important operation defined on them, namely invocation.

The ability to treat functions as objects is not at all the same as the ability to treat representations of functions as objects. It is the latter ability that is traditionally associated with LISP; functions can be represented as S-expressions. In a version of SCHEME which had no S-expression primitives, however, one could still deal with functions (i.e. closures) as such, for that ability is part of the fundamental environment and control facilities. Conversely, in a SCHEME which does have CONS, CAR, and CDR, there is no defined way to use CONS by itself to construct a function (although a primitive ENCLOSE is now provided which converts an S-expression representation of a function into a function), and the CAR or CDR of a function is in general undefined. The only defined operation on a function is invocation. {Note Operations on Functions}

We draw this sharp distinction between environment and control constructs on the one hand and data manipulation primitives on the other because only the former are treated in any depth by RABBIT, whereas much of the knowledge of a "real" compiler deals with the latter. A PL/I compiler must have much specific knowledge about numbers, arrays, strings, and so on. We have no new ideas to present here on such issues, and so have avoided this entire area. RABBIT itself knows almost nothing about data manipulation primitives beyond being able to recognize them and pass them along to the output code, which is a small subset of MacLISP. In this way RABBIT can concentrate on the interesting issues of environment and control, and exploit the expert knowledge of data manipulation primitives already built into the MacLISP compiler.

6. The Use of Macros

An important characteristic of the SCHEME language is that its set of primitive constructs is quite small. This set is not always convenient for expressing programs, however, and so a macro facility is provided for extending the expressive power of the language. A macro is best thought of as a syntax rewrite rule. As a simple example, suppose we have a primitive GCD which takes only two arguments, and we wish to be able to write an invocation of a GCD function with any number of arguments. We might then define (in a "production-rule" style) the conditional rule:

```
(XGCD)            =>  0
(XGCD x)          =>  x
(XGCD x . rest)   =>  (GCD x (XGCD . rest))
```

(Notice the use of LISP dots to refer to the rest of a list.) This is not considered to be a definition of a function XGCD, but a purely syntactic transformation. In principle all such transformations could be performed before executing the program. In fact, RABBIT does exactly this, although the SCHEME interpreter naturally does it incrementally, as each macro call is encountered.

Rather than use a separate production-rule/pattern-matching language, in practice SCHEME macros are defined as transformation functions from macro-call expressions to resulting S-expressions, just as they are in MacLISP. (Here, however, we shall continue to use production rules for purposes of exposition.) It is important to note that macros need not be written in the language for which they express rewrite rules; rather, they should be considered an adjunct to the interpreter, and written in the same language as the interpreter (or the compiler). To see this more clearly, consider a version of SCHEME which does not have S-expressions in its data domain. If programs in this language are represented as S-expressions, then the interpreter for that language cannot be written in that language, but in another meta-language which does deal with S-expressions. Macros, which transform one S-expression (representing a macro call) to another (the replacement form, or the interpretation of the call), clearly should be expressed in this meta-language also. The fact that in most LISP systems the language and the meta-language appear to coincide is a source of both power and confusion.

In the PDP-10 MacLISP implementation of SCHEME, four separate macro mechanisms are used in practice. One is the MacLISP read-macro mechanism [Moon], which performs transformations such as 'FOO => (QUOTE FOO) when an expression is read from a file. The other three are as described earlier, processed by the interpreter or compiler, and differ only in that one kind is recognized by the MacLISP interpreter as well while the other two are used only by SCHEME, and that of the latter two one kind is written in MacLISP and the other kind in SCHEME itself.

There is a growing library of SCHEME macros which express a variety of traditional programming constructs in terms of other, more primitive constructs, and eventually in terms of the small set of primitives. A number of these are catalogued in [Imperative] and [Revised Report]. Others were invented in the course of writing RABBIT. We shall give some examples here.

The BLOCK macro is similar to the MacLISP PROGN; it evaluates all its arguments and returns the value of the last one. One critical characteristic is that the last argument is evaluated "tail-recursively" (I use horror quotes because normally we speak of invocation, not evaluation, as being tail-recursive). An expansion rule is given for this in [Imperative] equivalent to:

```
(BLOCK x)          =>  x
(BLOCK x . rest)   =>  ((LAMBDA (DUMMY) (BLOCK . rest)) x)
```

This definition exploits the fact that SCHEME is evaluated in applicative order, and so will evaluate all arguments before applying a function to them. Thus, in the second subrule, x must be evaluated, and then the block of all the rest is. It is then clear from the first subrule that the last argument is evaluated "tail-recursively".

One problem with this definition is the occurrence of the variable DUMMY, which must be chosen so as not to conflict with any variable used by the user. This we refer to as the "GENSYM problem", in honor of the traditional LISP function which creates a "fresh" symbol. It would be nicer to write the macro in such a way that no conflict could arise no matter what names were used by the user. There is indeed a way, which ALGOL programmers will recognize as equivalent to the use of "thunks", or call-by-name parameters:

```
(BLOCK x)          =>  x
(BLOCK x . rest)   =>  ((LAMBDA (A B) (B))
                        x
                        (LAMBDA () (BLOCK . rest)))
```

Consider carefully the meaning of the right-hand side of the second subrule. First the expression x and the (LAMBDA () ...) must be evaluated (it doesn't matter in which order!); the result of the latter is a function (that is, a closure), which is later invoked in order to evaluate the rest of the arguments. There can be no naming conflicts here, because the scope of the variables A and B (which is <u>lexical</u>) does not contain any of the arguments to BLOCK written by the user. (We should note that we have been sloppy in speaking of the "arguments" to BLOCK, when BLOCK is properly speaking not a function at all, but merely a syntactic keyword used to recognize a situation where a syntactic rewriting rule is applicable. We would do better to speak of "argument expressions" or "macro arguments", but we shall continue to be sloppy where no confusion should arise.)

This is a technique which should be understood quite thoroughly, since it is the key to writing correct macro rules without any possibility of conflicts between names used by the user and those needed by the macro. As another example, let us consider the AND and OR constructs as used by most LISP systems. OR evaluates its arguments one by one, in order, returning the first non-NIL value obtained (without evaluating any of the following arguments), or NIL if all arguments produce NIL. AND is the dual to this; it returns NIL if any argument does, and otherwise the value of the last argument. A simple-minded approach to OR would be:

```
(OR)            =>  'NIL
(OR x . rest)   =>  (IF x x (OR . rest))
```

There is an objection to this, which is that the code for x is duplicated. Not only does this consume extra space, but it can execute erroneously if x has any side-effects. We must arrange to evaluate x only once, and then test its value:

```
(OR)            =>  'NIL
(OR x . rest)   =>  ((LAMBDA (V) (IF V V (OR . rest))) x)
```

This certainly evaluates x only once, but admits a possible naming conflict between the variable V and any variables used by rest. This is avoided by the same technique used for BLOCK:

```
(OR)            =>  'NIL
(OR x . rest)   =>  ((LAMBDA (V R) (IF V V (R)))
                     x
                     (LAMBDA () (OR . rest)))
```

Similarly, we can express AND as follows:

```
(AND)            =>  'T
(AND x)          =>  x
(AND x . rest)   =>  ((LAMBDA (V R) (IF V (R) 'NIL))
                      x
                      (LAMBDA () (AND . rest)))
```

(The macro rules are not precise duals because of the non-duality between NIL-ness and non-NIL-ness, and the requirement that a successful AND return the actual value of the last argument and not just T.) {Note Tail-Recursive OR}

As yet another example, consider a modification to BLOCK to allow a limited form of assignment statement: if (v := x) appears as a statement in a block, it "assigns" a value to the variable v whose scope is the remainder of the block. Let us assume that such a statement cannot occur as the last statement of a block (it would be useless to have one in that position, as the variable would have a null scope). We can write the rule:

```
(BLOCK x)                 =>  x
(BLOCK (v := x) . rest)   =>  ((LAMBDA (v) (BLOCK . rest)) x)
(BLOCK x . rest)          =>  ((LAMBDA (A B) (B))
                               x
                               (LAMBDA () (BLOCK . rest)))
```

The second subrule states that an "assignment" causes x to be evaluated and then bound to v, and that the variable v is visible to the rest of the block.

We may think of := as a "sub-macro keyword" which is used to mark an expression as suitable for transformation, but only in the context of a certain larger transformation. This idea is easily extended to allow other constructions, such as "simultaneous assignments" of the form

```
((var1 var2 ... varn) := value1 value2 ... valuen)
```

which first compute all the values and then assign to all the variables, and "exchange assignments" of the form (X :=: Y), as follows:

```
(BLOCK x)           =>  x
(BLOCK (v := x) . rest)
                    =>  ((LAMBDA (v) (BLOCK . rest)) x)
(BLOCK (vars := . values) . rest)
                    =>  ((LAMBDA vars (BLOCK . rest)) . values)
(BLOCK (x :=: y) . rest)
                    =>  ((LAMBDA (x y) (BLOCK . rest)) y x)
(BLOCK x . rest)    =>  ((LAMBDA (A B) (B))
                         x
                         (LAMBDA () (BLOCK . rest)))
```

Let us now consider a rule for the more complicated COND construct:

```
(COND)  =>  'NIL
(COND (x) . rest)  =>  (OR x (COND . rest))
(COND (x . r) . rest)  =>  (IF x (BLOCK . r) (COND . rest))
```

This defines the "extended" COND of modern LISP systems, which produces NIL if no clauses succeed, which returns the value of the predicate in the case of a singleton clause, and which allows more than one consequent in a clause. An important point here is that one can write these rules in terms of other macro constructs such as OR and BLOCK; moreover, any extensions to BLOCK, such as the limited assignment feature described above, are automatically inherited by COND. Thus with the above definition one could write

```
(COND ((NUMBERP X) (Y := (SQRT X)) (+ Y (SQRT Y)))
      (T (HACK X)))
```

where the scope of the variable Y is the remainder of the first COND clause.

SCHEME also provides macros for such constructs as DO and PROG, all of which expand into similar kinds of code using LAMBDA, IF, and LABELS (see below). In particular, PROG permits the use of GO and RETURN in the usual manner. In this manner all the traditional imperative constructs are expressed in an applicative manner. {Note ASET' Is Imperative}

None of this is particularly new; theoreticians have modelled imperative constructs in these terms for years. What is new, we think, is the serious proposal that a practical interpreter and compiler can be designed for a language in which such models serve as the <u>sole definitions</u> of these imperative constructs. {Note Dijkstra's Opinion} This approach has both advantages and disadvantages.

One advantage is that the base language is small. A simple-minded interpreter or compiler can be written in a few hours. (We have re-implemented the SCHEME interpreter from scratch a dozen times or more to test various representation strategies; this was practical only because of the small size of the language. Similarly, the CHEAPY compiler is fewer than ten pages of code, and could be rewritten in a day or less.) Once the basic interpreter is written, the macro definitions for all the complex constructs can be used without revision. Moreover, the same macro definitions can be used by both interpreter and compiler (or by several versions of interpreter and compiler!). Excepting the very few primitives such as LAMBDA and IF, it is not necessary to "implement a construct twice", once each in interpreter and compiler.

Another advantage is that new macros are very easy to write (using facilities provided in SCHEME). One can easily invent a new kind of DO loop, for example, and implement it in SCHEME for both interpreter and all compilers in less than five minutes. (In practice such new control constructs, such as the ITERATE loop described in [Revised Report], are indeed installed within five to ten minutes of conception, in a routine manner.)

A third advantage is that the attention of the compiler can be focused on the basic constructs. Rather than having specialized code for two dozen different constructs, the compiler can have much deeper knowledge about each of a few basic constructs. One might object that this "deeper knowledge" consists of recognizing the two dozen special cases represented by the separate constructs of the former case. This is true to some extent. It is also true, however, that in the latter case such deep knowledge will carry over to any new constructs which are invented and represented as macros.

Among the disadvantages of the macro approach are lack of speed and the discarding of information. Many people have objected that macros are of necessity slower than, say, the FSUBR implementation used by most LISP systems. This is true in many current interpretive implementations, but need not be true of compilers or more cleverly designed interpreters. Moreover, the FSUBR implementation is not general; it is very hard for a user to write a meaningful FSUBR and then describe to the compiler the best way to compile it. The macro approach handles this difficulty automatically. We do not object to the use of the FSUBR mechanism as a special-case "speed hack" to improve the performance of an interpreter, but we insist on recognizing the fact that it is not as generally useful as the macro approach.

Another objection relating to speed is that the macros produce convoluted code involving the temporary creation and subsequent invocation of many closures. We feel, first of all, that the macro writer should concern himself more with producing correct code than fast code. Furthermore, convolutedness can be eliminated by a few simple optimization techniques in the compiler, to be discussed below. Finally, function calls need not be as expensive as is popularly supposed. [Steele]

Information is discarded by macros in the situation, for example, where a DO macro expands into a large mess that is not obviously a simple loop; later compiler analysis must recover this information. This is indeed a problem. We feel that the compiler is probably better off having to recover the information anyway, since a deep analysis allows it to catch other loops which the user did not use DO to express for one reason or another. Another is the possibility that

DO could leave clues around in the form of declarations if desired.

Another difficulty with the discarding of information is the issuing of meaningful diagnostic messages. The user would prefer to see diagnostics mention the originally-written source constructs, rather than the constructs into which the macros expanded. (An example of this problem from another LISP compiler is that it may convert (MEMQ X '(A B C)) into (OR (EQ X 'A) (EQ X 'B) (EQ X 'C)); when by the same rule it converts (MEMQ X '(A)) (a piece of code generated by a macro) into (OR (EQ X 'A)), it later issues a warning that an OR had only one subform.) This problem can be partially circumvented if the responsibility for syntax-checking is placed on the macro definition at each level of expansion.

7. The Imperative Treatment of Applicative Constructs

Given the characteristics of lexical scoping and tail-recursive invocations, it is possible to assign a peculiarly imperative interpretation to the applicative constructs of SCHEME, which consists primarily of treating a function call as a GOTO. More generally, a function call is a GOTO that can pass one or more items to its target; the special case of passing no arguments is precisely a GOTO. It is never necessary for a function call to save a return address of any kind. It is true that return addresses are generated, but we adopt one of two other points of view, depending on context. One is that the return address, plus any other data needed to carry on the computation after the called function has returned (such as previously computed intermediate values and other return addresses) are considered to be packaged up into an additional argument (the continuation) which is passed to the target. This lends itself to a non-functional interpretation of LAMBDA, and a method of expressing programs called the continuation-passing style (similar to the message-passing actors paradigm [Hewitt]), to be discussed further below. The other view, more intuitive in terms of the traditional stack implementation, is that the return address should be pushed before evaluating arguments rather than before calling a function. This view leads to a more uniform function-calling discipline, and is discussed in [Declarative] and [Steele].

We are led by these points of view to consider a compilation strategy in which function calling is to be considered very cheap (unlike the situation with PL/I and ALGOL, where programmers avoid procedure calls like the plague -- see [Steele] for a discussion of this). In this light the code produced by the sample macros above does not seem inefficient, or even particularly convoluted. Consider the expansion of (OR a b c):

```
((LAMBDA (V R) (IF V V (R)))
 a
 (LAMBDA () ((LAMBDA (V R) (IF V V (R)))
             b
             (LAMBDA () ((LAMBDA (V R) (IF V V (R)))
                         c
                         (LAMBDA () 'NIL))))))
```

Then we might imagine the following (slightly contrived) compilation scenario. First, for expository purposes, we shall rename the variables in order to be able to distinguish them.

```
((LAMBDA (V1 R1) (IF V1 V1 (R1)))
 a
 (LAMBDA () ((LAMBDA (V2 R2) (IF V2 V2 (R2)))
             b
             (LAMBDA () ((LAMBDA (V3 R3) (IF V3 V3 (R3)))
                         c
                         (LAMBDA () 'NIL))))))
```

We shall assign a generated name to each LAMBDA-expression, which we shall notate by writing the name after the word LAMBDA. These names will be used as tags in the output code.

```
((LAMBDA name1 (V1 R1) (IF V1 V1 (R1)))
 a
 (LAMBDA name2 () ((LAMBDA name3 (V2 R2) (IF V2 V2 (R2)))
             b
             (LAMBDA name4 () ((LAMBDA name5 (V3 R3)
                                             (IF V3 V3 (R3)))
                         c
                         (LAMBDA name6 () 'NIL))))))
```

Next, a simple analysis shows that the variables R1, R2, and R3 always denote the LAMBDA-expressions named name2, name4, and name6, respectively. Now an optimizer might simply have substituted these values into the bodies of name1, name3, and name5 using the rule of beta-conversion, but we shall not apply that technique here. Instead we shall compile the six functions in a straightforward manner. We make use of the additional fact that all six functions are closed in identical environments (we count two environments as identical if they involve the same variable bindings, regardless of the number of "frames" involved; that is, the environment is the same inside and outside a (LAMBDA () ...)). Assume a simple target machine with argument registers called reg1, reg2, etc.

```
main:    <code for a>               ;result in reg1
         LOAD reg2,[name2]          ;[name2] is the closure for name2
         CALL-FUNCTION 2,[name1] ;call name1 with 2 arguments

name1:   JUMP-IF-NIL reg1,name1a
         RETURN                     ;return the value in reg1
name1a: CALL-FUNCTION 0,reg2       ;call function in reg2, 0 arguments

name2:   <code for b>               ;result in reg1
         LOAD reg2,[name4]          ;[name4] is the closure for name4
         CALL-FUNCTION 2,[name3] ;call name3 with 2 arguments

name3:   JUMP-IF-NIL reg1,name3a
         RETURN                     ;return the value in reg1
name3a: CALL-FUNCTION 0,reg2       ;call function in reg2, 0 arguments

name4:   <code for c>               ;result in reg1
         LOAD reg2,[name6]          ;[name6] is the closure for name6
         CALL-FUNCTION 2,[name5] ;call name5 with 2 arguments

name5:   JUMP-IF-NIL reg1,name5a
         RETURN                     ;return the value in reg1
name5a: CALL-FUNCTION 0,reg2       ;call function in reg2, 0 arguments

name6:   LOAD reg1,'NIL             ;constant NIL in reg1
         RETURN
```

Now we make use of our knowledge that certain variables always denote certain functions, and convert CALL-FUNCTION of a known function to a simple GOTO. (We have actually done things backwards here; in practice this knowledge is used before generating any code. We have fudged over this issue here, but will return to it later. Our purpose here is merely to demonstrate the treatment of function calls as GOTOs.)

```
main:    <code for a>               ;result in reg1
         LOAD reg2,[name2]          ;[name2] is the closure for name2
         GOTO name1
```

```
name1:  JUMP-IF-NIL reg1,name1a
        RETURN                     ;return the value in reg1
name1a: GOTO name2

name2:  <code for b>               ;result in reg1
        LOAD reg2,[name4]          ;[name4] is the closure for name4
        GOTO name3

name3:  JUMP-IF-NIL reg1,name3a
        RETURN                     ;return the value in reg1
name3a: GOTO name4

name4:  <code for c>               ;result in reg1
        LOAD reg2,[name6]          ;[name6] is the closure for name6
        GOTO name5

name5:  JUMP-IF-NIL reg1,name5a
        RETURN                     ;return the value in reg1
name5a: GOTO name6

name6:  LOAD reg1,'NIL             ;constant NIL in reg1
        RETURN
```

The construction [foo] indicates the creation of a closure for foo in the current environment. This will actually require additional instructions, but we shall ignore the mechanics of this for now since analysis will remove the need for the construction in this case. The fact that the only references to the variables R1, R2, and R3 are function calls can be detected and the unnecessary LOAD instructions eliminated. (Once again, this would actually be determined ahead of time, and no LOAD instructions would be generated in the first place. All of this is determined by a general pre-analysis, rather than a peephole post-pass.) Moreover, a GOTO to a tag which immediately follows the GOTO can be eliminated.

```
main:    <code for a>             ;result in reg1
name1:   JUMP-IF-NIL reg1,name1a
         RETURN                   ;return the value in reg1
name1a:
name2:   <code for b>             ;result in reg1
name3:   JUMP-IF-NIL reg1,name3a
         RETURN                   ;return the value in reg1
name3a:
name4:   <code for c>             ;result in reg1
name5:   JUMP-IF-NIL reg1,name5a
         RETURN                   ;return the value in reg1
name5a:
name6:   LOAD reg1,'NIL           ;constant NIL in reg1
         RETURN
```

This code is in fact about what one would expect out of an ordinary LISP compiler. (There is admittedly room for a little more improvement.) RABBIT indeed produces code of essentially this form, by the method of analysis outlined here.

Similar considerations hold for the BLOCK macro. Consider the expression (BLOCK a b c); conceptually this should perform a, b, and c sequentially. Let us examine the code produced:

```
((LAMBDA (A B) (B))
 a
 (LAMBDA () ((LAMBDA (A B) (B))
             b
             (LAMBDA () c))))
```

Renaming the variables and assigning names to LAMBDA-expressions:

```
((LAMBDA name1 (A1 B1) (B1))
 a
 (LAMBDA name2 () ((LAMBDA name3 (A2 B2) (B2))
                   b
                   (LAMBDA name4 () c))))
```

Producing code for the functions:

```
main:   <code for a>
        LOAD reg2,[name2]
        CALL-FUNCTION 2,[name1]

name1:  CALL-FUNCTION 0,reg2

name2:  <code for b>
        LOAD reg2,[name4]
        CALL-FUNCTION 2,[name3]

name3:  CALL-FUNCTION 0,reg2

name4:  <code for c>
        RETURN
```

Turning general function calls into direct GO's, on the basis of analysis of what variables must refer to constant functions:

```
main:   <code for a>
        LOAD reg2,[name2]
        GOTO name1

name1:  GOTO name2

name2:  <code for b>
        LOAD reg2,[name4]
        GOTO name3

name3:  GOTO name4

name4:  <code for c>
        RETURN
```

Eliminating useless GOTO and LOAD instructions:

```
main:   <code for a>
name1:
name2:  <code for b>
name3:
name4:  <code for c>
        RETURN
```

What more could one ask for?

Notice that this has fallen out of a general strategy involving only an approach to compiling function calls, and has involved no special knowledge of OR or BLOCK not encoded in the macro rules. The cases shown so far are actually special cases of a more general approach, special in that all the conceptual closures involved are closed in the same environment, and called from places that have not disturbed that environment, but only used "registers". In the more general case, the environments of caller and called function will be different. This divides into two subcases, corresponding to whether the closure was created by a simple LAMBDA or by a LABELS construction. The latter involves circular references, and so is somewhat more complicated; but it is easy to show that in the former case the environment of the caller must be that of the (known) called function, possibly with additional values added on. This is a consequence of lexical scoping. As a result, the function call can be compiled as a GOTO preceded by an environment adjustment which consists merely of lopping off some leading portion of the current one (intuitively, one simply "pops the unnecessary crud off the stack"). LABELS-closed functions also can be treated in this way, if one closes all the functions in the same way (which RABBIT presently does, but this is not always desirable). If one does, then it is easy to see the effect of expanding a PROG into a giant LABELS as outlined in [Imperative] and elsewhere: normally, a GOTO to a tag at the same level of PROG will involve no adjustment of environment, and so compile into a simple GOTO instruction, whereas a GOTO to a tag at an outer level of PROG probably will involve adjusting the environment from that of the inner PROG to that of the outer. All of this falls out of the proper imperative treatment of function calls.

8. Compilation Strategy

The overall approach RABBIT takes to the compilation of SCHEME code may be summarized as follows:

(1) Alpha-conversion (renaming of variables) and macro-expansion (expansion of syntactic rewrite rules).

(2) Preliminary analysis (variable references, "trivial" expressions, and side effects).

(3) Optimization (meta-evaluation).

(4) Conversion to continuation-passing style.

(5) Environment and closure analysis.

(6) Code generation.

During (1) a data structure is built which is structurally a copy of the user program but in which all variables have been renamed and in which at each "node" of the program tree are additional slots for extra information. These slots are filled in during (2). In (3) the topology of the structure may be modified to reflect transformations made to the program; routines from (2) may be called to update the information slots. In (4) a new data structure is contructed from the old one, radically different in structure, but nevertheless also tree-like in form. During (5) information is added to slots in the second structure. In (6) this information is used to produce the final code.

A. Alpha-conversion and macro-expansion

In this phase a copy of the user program is made. The user program is conceptually a tree structure; each node is one of several kinds of construct (constant, variable, LAMBDA-expression, IF-expression, combination, etc.). Some kinds of nodes have subnodes; for example, a LAMBDA-expression node has a subnode representing the body, and a combination node has a subnode for each argument. The copying is performed in the obvious way by a recursive tree-walk. In the process all bound variables are renamed. Each bound variable is assigned a new generated name at the point of binding, and each node for a reference to a bound variable contains this generated name, not the original name. From this point on all variables are dealt with in terms of their new names. (This is possible because, as a consequence of lexical scoping, we can identify all references to each bound variable.) These new names are represented as atomic symbols, and the property lists of these symbols will later be used to store information about the variables.

As each subform of the user program is examined, a check is made for a macro call, which is a list whose car is an atomic symbol with one of several macro-defining properties. When such a call is encountered, the macro call is expanded, and the tree-walk is resumed on the code returned by the expansion process.

B. Preliminary analysis

The preliminary analysis ("phase 1") is in three passes, each involving a tree-walk of the node structure, filling in information slots at each node. (Two passes would have sufficed, but for reasons of clarity and modularity there is one pass for each type of analysis.)

The first pass (ENV-ANALYZE) analyzes variable references. For each node we determine the set of all local (bound) variables referenced at or below that node. For example, for a variable-reference node this set is empty (for a global variable), or the singleton set of the variable itself (for a local variable); for a LAMBDA-expression, it is the set for its body minus the variables bound by that LAMBDA-expression; for an IF-expression, it is the union of the sets for the predicate, consequent, and alternative; and so on. We also compute for each node the set of bound variables which appear in an ASET' at or below the node. (This set will be a subset of the first set, but no non-trivial use of this property is used in this pass.) Finally, for each variable we store several properties on its property list, including a list of all nodes which reference the variable (for "reading") and a list of of all ASET' nodes which modify the variable. These lists are built incrementally, with an entry added as each reference is encountered during the tree walk. (This exemplifies the general strategy for passing data around; any information which cannot be passed conveniently up and down the tree, but which must jump laterally between branches, is accumulated on the property lists of the variables. It may appear to be "lucky" that all such information has to do with variables, but this is actually an extremely deep property of our notation. The entire point of using identifiers is to relate textually separated constructions. We depend on alpha-conversion to give all variables distinct names (by "names" we really mean "compile-time data structures") so that the information for variables which the user happened to give the same name will not be confused.)

The second pass (TRIV-ANALYZE) locates "trivial" portions of the program. (Cf. [Wand and Friedman].) Constants and variables are trivial; an IF-expressions is trivial iff the predicate, consequent, and alternative are all trivial; an ASET' is trivial iff its body is trivial; a combination is trivial iff the function is either a global variable which is the name of a MacLISP primitive, or a LAMBDA-expression whose body is trivial, and the arguments are all trivial. LAMBDA-expressions, LABELS-forms (which contain LAMBDA-expressions), and CATCH-forms are never trivial. The idea is that a trivial expression is one that MacLISP could evaluate itself, without benefit of SCHEME control structures. (No denigration of MacLISP's ability is intended by this terminology!) Note particularly the two special cases of combinations distinguished here (in which the function position contains either the name of a MacLISP primitive or a LAMBDA-expression); they are very important, and shall be referred to respectively as TRIVFN-combinations and LAMBDA-combinations.

The third pass (EFFS-ANALYZE) analyzes the possible side-effects caused by each node, and the side-effects which could affect it. It actually produces two sets of analyses, one liberal and one conservative. Where there is any uncertainty as to what side-effects may be involved, it assumes none in one case and all possible in the other. The liberal estimation is used only to issue error messages to the user about possible conflicts which might result as a consequence of the freedom to evaluate arguments to combinations in any order. The user is given the benefit of doubt, and warned only of a "provable" conflict. (Actually, the "proof" is a little sloppy, and can err in both directions, but in practice it has issued no false alarms and a number of helpful warnings.) The conservative estimation is used by the optimizer, which will move expressions only if it can prove that there will be no conflict.

Side effects are grouped into classes: ASET, RPLACA and RPLACD (which are considered distinct), FILE (input/output operations), and CONS. These are not intended to be exhaustive; there is also an internal notation for "any side-effect whatever". The use of classes enables the analysis to realize, for example, that RPLACA cannot affect the value of a variable *per se*. There is a moderately large body of data in RABBIT about the side-effects of MacLISP primitive functions. For example, CAR, CDR, CAAR, CADR, and so on are known not to have side-effects, and to be respectively affected only by RPLACA, RPLACD, RPLACA, RPLACA or RPLACD, and so on. Similarly, RABBIT knows that ASET' affects the values of variables, but cannot affect the outcome of a CAR operation. (It may affect the value of the expression (CAR X), but only because a variable reference is a subnode of the combination. The effects, or affectability, of a combination are the union of the effects, or affectibility, of all arguments plus those of the function.) The CONS side-effect is a special case. This side-effect cannot affect anything, and two instances of it may be performed in the "wrong" order, but performing a single instance twice will produce distinct (as determined by EQ) and therefore incorrect results. In particular, closures of LAMBDA-expressions involve the CONS side-effect. (The definition of SCHEME says nothing about whether EQ is a valid operation on closures, but in general it is not a good idea to produce unnecessary multiple copies.) On the other hand, LAMBDA-expressions occurring in function position of a LAMBDA-combination do not incur the CONS side-effect. The CONS side-effect is given special treatment in the optimizer. {Note Side-Effect Classifications}

C. Optimization

Once the preliminary analysis is done, the optimization phase performs certain transformations on the code. The result is an equivalent program which will (probably) compile into more efficient code. This new program is itself structurally a valid SCHEME program; that is, all transformations are contained within the language. The transformations are thus similar to those performed on macro calls, consisting of a syntactic rewriting of an expression, except that the situations where such transformations are applicable are more easily recognized in the case of macro calls. It should be clear that the optimizer and the macro-functions are conceptually at the same level in that they may be written in the same meta-language that operates on representations of SCHEME programs. {Note Non-deterministic Optimization}

The simplest transformation is that a combination whose function position contains a LAMBDA-expression and which has no arguments can be replaced by the body of the LAMBDA-expression:

```
((LAMBDA () body))  =>  body
```

Another is that, in the case of a LAMBDA-combination, if some parameter of the LAMBDA-expression is not referenced and the corresponding argument can be proved to have no side-effects (with an exception discussed below), then the parameter and argument can be eliminated:

```
((LAMBDA (x1 x2 x3) body) a1 a2 a3)
                    =>  ((LAMBDA (x1 x3) body) a1 a3)
if x2 is unreferenced in body and a2 has no side-effects
```

Repeated applications of this rule can lead to the preceding case.

A third rule is that, in a LAMBDA-combination, an argument can be substituted for one or more occurrences of a parameter in the body of the LAMBDA-expression. (This rule is related to the view of LAMBDA as a renaming operator discussed in [Declarative], and together with the two preceding rules make up the rule of beta-conversion.) Such a substitution is permissible only if (a) either the parameter is referred to only once or the argument has no side effects, and (b) the substitution will not alter the order in which expressions are evaluated in such a way as to allow possible side-effects to produce different results. Before performing the substitution it is necessary to show that side-effects will not interfere in this manner. This issue is discussed in [Allen and Cocke], [Geschke], and [Wulf], and characterized more accurately in [Standish]. There is also some difficulty if the parameter appears in an ASET'. Presently RABBIT does not attempt any form of substitution for such a parameter. (ASET' is so seldom used in SCHEME programs that this restriction makes very little difference.)

This third rule creates an exception to the second. If an argument with a side effect is referred to once, and is substituted for the reference, then the second rule must be invoked to eliminate the original occurrence of the argument, so that the side effect will not occur twice. This requires a little collusion between the two rules.

Even if such a substitution is permissible, it is not always desirable; time/space tradeoffs are involved. The current heuristic is that a substitution is desirable if (1) the parameter is referred to only once; or (2) the argument to be substituted in is a constant or variable; or (3) the argument is a LAMBDA-expression whose body is (3a) a constant, or (3b) a variable reference, or (3c) a combination which has no more arguments than the LAMBDA-expression requires and for which the arguments are all constants or variables. This heuristic was designed to be as conservative as possible while handling most cases which arise from typical macro-expansions.

The case where the expression substituted for a variable is a LAMBDA-expression constitutes an instance of procedure integration [Allen and Cocke]. The more general kind of procedure integration proposed in [Declarative], which would involve block compilation of several user functions, and possibly also user declarations or data type analysis, has not been implemented yet.

It would be possible to substitute a LAMBDA-expression for a variable reference in the case of a variable bound by a LABELS. This might be useful in the case of a LABELS produced by a simple-minded PROG macro, which produced a labelled function for each statement of the PROG; in such a case most labelled functions would be referred to only once. We have not implemented this yet in RABBIT. {Note Loop Unrolling}

Currently there is not any attempt to perform the inverse of beta-conversion. This process would be that of locating common subexpressions of some single large expression, making that large expression the body of a LAMBDA-expression of one parameter, replacing all occurrences of the common subexpression by a reference to the parameter, and replacing the large expression by a combination whose function position contained the LAMBDA-expression and whose argument was a copy of the common subexpression. More generally, several common subexpressions could be isolated at once and made into several parameters of the LAMBDA-expression. For example, consider:

```
(LAMBDA (A B C)
        (LIST (/ (+ (- B) (SQRT (- (^ B 2) (* 4 A C))))
                 (* 2 A))
              (/ (- (- B) (SQRT (- (^ B 2) (* 4 A C))))
                 (* 2 A))))
```

Within the large expression (LIST ...) we might detect the common subexpressions (- B), (SQRT ...), and (* 2 A). Thus we would invent three parameters Q1, Q2, Q3 and transform the expression into:

```
(LAMBDA (A B C)
        ((LAMBDA (Q1 Q2 Q3)
                 (LIST (/ (+ Q1 Q2) Q3)
                       (/ (- Q1 Q2) Q3)))
         (- B)
         (SQRT (- (^ B 2) (* 4 A C)))
         (* 2 A)))
```

(There would be no problem of conflicting names as there is for macro rules, because we are operating on code for which all variables have already been renamed; Q1, Q2, and Q3 can be chosen as the next variables in the renaming sequence.)

This approach doesn't always work if side-effects are present; the abstracted (!) common subexpression may be evaluated too soon, or the wrong number of times. This can be solved by wrapping (LAMBDA () ⊛) around the common subexpression, and replacing references by a combination instead of a simple variable reference. For example:

```
(IF (HAIRYP X)
    (BLOCK (PRINT '|Here is some hair:|)
           (PRINT X)
           (PRINT '|End of hair.|))
    (BLOCK (PRINT '|This one is bald:|)
           (PRINT X)
           (PRINT '|End of baldness.|)))
```

We could not transform it into this:

```
((LAMBDA (Q1)
         (IF (HAIRYP X)
             (BLOCK (PRINT '|Here is some hair:|)
                    Q1
                    (PRINT '|End of hair.|))
             (BLOCK (PRINT '|This one is bald:|)
                    Q1
                    (PRINT '|End of baldness.|))))
 (PRINT X))
```

because x would be printed before the appropriate leading message. Instead, we transform it into:

```
((LAMBDA (Q1)
         (IF (HAIRYP X)
             (BLOCK (PRINT '|Here is some hair:|)
                    (Q1)
                    (PRINT '|End of hair.|))
             (BLOCK (PRINT '|This one is bald:|)
                    (Q1)
                    (PRINT '|End of baldness.|))))
 (LAMBDA () (PRINT X)))
```

This is similar to the call-by-name trick used in writing macro rules.

A more general transformation would detect <u>nearly</u> common subexpressions as follows:

```
((LAMBDA (Q1)
         (IF (HAIRYP X)
             (Q1 '|Here is some hair:|
                 '|End of hair.|)
             (Q1 '|This one is bald:|
                 '|End of baldness.|)))
 (LAMBDA (Q2 Q3)
         (BLOCK (PRINT Q2) (PRINT X) (PRINT Q3))))
```

In this way we can express the notion of subroutinization. {Note Subroutinization}

We point out these possibilities despite the fact that they have not been implemented in RABBIT because the problem of isolating common subexpressions seems not to have been expressed in quite this way in the literature on compilation strategies. We might speculate that this is because most compilers which use complex optimization strategies have been for ALGOL-like languages which do not treat functions as full-fledged data objects, or even permit the writing of "anonymous" functions in functions calls as LISP does.

RABBIT does perform folding on constant expressions [Allen and Cocke]; that is, any combination whose function is a side-effect-less MacLISP primitive and whose arguments are all constants is replaced by the result of applying the primitive to the arguments. There is presently no attempt to do the same thing for side-effect-less SCHEME functions, although this is conceptually no problem.

Finally, there are two transformations on IF expressions. One is simply that an IF expression with a constant predicate is simplified to its consequent or alternative (resulting in elimination of dead code). The other was adapted from [Standish], which does not have this precise transformation listed, but gives a more general rule. In its original form this transformation is:

```
(IF (IF a b c) d e)  =>  (IF a (IF b d e) (IF c d e))
```

One problem with this is that the code for d and e is duplicated. This can be avoided by the use of LAMBDA-expressions:

```
((LAMBDA (Q1 Q2)
         (IF a
             (IF b (Q1) (Q2))
             (IF c (Q1) (Q2))))
 (LAMBDA () d)
 (LAMBDA () e))
```

As before, there is no problem of name conflicts with Q1 and Q2. While this code may appear unnecessarily complex, the calls to the functions Q1 and Q2 will typically, as shown above, be compiled as simple GOTO's. As an example, consider the expression:

```
(IF (AND PRED1 PRED2) (PRINT 'WIN) (ERROR 'LOSE))
```

Expansion of the AND macro will result in:

```
(IF ((LAMBDA (V R) (IF V (R) 'NIL))
     PRED1
     (LAMBDA () PRED2))
    (PRINT 'WIN)
    (ERROR 'LOSE))
```

(For expository clarity we will not bother to rename all the variables, inasmuch as they are already distinct.) Because V and R have only one reference apiece (and there are no possible interfering side-effects), the corresponding arguments can be substituted for them.

```
(IF ((LAMBDA (V R) (IF PRED1 ((LAMBDA () PRED2)) 'NIL))
     PRED1
     (LAMBDA () PRED2))
    (PRINT 'WIN)
    (ERROR 'LOSE))
```

Now, since V and R have no referents at all, they and the corresponding arguments can be eliminated, since the arguments have no side-effects.

```
(IF ((LAMBDA () (IF PRED1 ((LAMBDA () PRED2)) 'NIL)))
    (PRINT 'WIN)
    (ERROR 'LOSE))
```

Next, the combination ((LAMBDA () ...)) is eliminated in two places:

```
(IF (IF PRED1 PRED2 'NIL)
    (PRINT 'WIN)
    (ERROR 'LOSE))
```

Now, the transformation on the nested IF's:

```
((LAMBDA (Q1 Q2)
         (IF PRED1
             (IF PRED2 (Q1) (Q2))
             (IF 'NIL (Q1) (Q2))))
 (LAMBDA () (PRINT 'WIN))
 (LAMBDA () (ERROR 'LOSE)))
```

Now one IF has a constant predicate and can be simplified:

```
((LAMBDA (Q1 Q2)
        (IF PRED1
            (IF PRED2 (Q1) (Q2))
            (Q2)))
 (LAMBDA () (PRINT 'WIN))
 (LAMBDA () (ERROR 'LOSE)))
```

The variable Q1 has only one referent, and so we substitute in, eliminate the variable and argument, and collapse a ((LAMBDA () ..)):

```
((LAMBDA (Q2)
        (IF PRED1
            (IF PRED2 (PRINT 'WIN) (Q2))
            (Q2)))
 (LAMBDA () (ERROR 'LOSE)))
```

Recalling that (Q2) is, in effect, a GOTO branching to the common piece of code, and that by virtue of later analysis no actual closure will be created for either LAMBDA-expression, this result is quite reasonable. {Note Evaluation for Control}

D. Conversion to Continuation-Passing Style

This phase is the real meat of the compilation process. It is of interest primarily in that it transforms a program written in SCHEME into an equivalent program (the continuation-passing-style version, or CPS version), written in a language isomorphic to a subset of SCHEME with the property that interpreting it requires no control stack or other unbounded temporary storage and no decisions as to the order of evaluation of (non-trivial) subexpressions. The importance of these properties cannot be overemphasized. The fact that it is essentially a subset of SCHEME implies that its semantics are as clean, elegant, and well-understood as those of the original language. It is easy to build an interpreter for this subset, given the existence of a SCHEME interpreter, which can execute the transformed program directly at this level. This cannot be said for other traditional intermediate compilation forms; building an interpreter for triples [Gries], for example, would be a tremendous undertaking. The continuation-passing version expresses all temporary intermediate results explicitly as variables appearing in the program text, and all temporary control structure in the form of LAMBDA-expressions (that is, closures). It is explicit in directing the order of operations; there is no non-trivial freedom at any point in the evaluation process.

As a result, once the CPS version of a program has been generated, the remainder of the compilation process is fairly easy. There is a reasonably direct correspondence between constructs in the CPS language and "machine-language" operations (if one assumes CONS to be a "machine-language" primitive for augmenting environment structure, which we do). The later passes are complicated only by the desire to handle certain special cases in an optimal manner, most particularly the case of a function call whose function position contains a variable which can be determined to refer to a known LAMBDA-expression. This analysis must be done after the CPS conversion because it applies to continuations as well as LAMBDA-expressions written by the user or generated by macros.

The CPS language differs from SCHEME in only two respects. First, each primitive function is different, in that it returns no value; instead, it accepts an additional argument, the continuation, which must be a "function" of one argument, and by definition invokes the continuation tail-recursively, giving it as an argument the computed "value" of the primitive function. We extend this by convention to non-primitive functions, and so all functions are considered to take a continuation as one of its arguments (by convention the first -- this differs from the convention used in the examples in [SCHEME], [Imperative], and [Declarative]). Continuations, however, do not themselves take continuations as arguments.

Second, no combination may have a non-trivial argument. In strict continuation-passing style (as described in note {Evalorder} of [Imperative]), this implies that no combination can have another combination as an argument, or an IF-expression with a non-trivial consequent or alternative, etc. We relax this to allow as arguments any trivial form in the sense described above for the preliminary triviality analysis. We note that, in principle, trivial expressions require no unbounded space on the part of the SCHEME interpreter to evaluate, and that the compiler need not worry about control and environment issues for trivial expressions. (Trivial expressions do require unbounded space on the part of the MacLISP run-time system, because the point of the triviality analysis is that trivial expressions can be handled by MacLISP! The question of what should be considered trivial is actually a function of the characteristics of our target machine. We note that, at the least, variables, constants, and LAMBDA-expressions should be considered trivial. That the preliminary triviality analysis does not consider LAMBDA-expressions trivial is a trick so that all closures will be processed by the CPS-conversion process, and the fact that we call it a triviality analysis is a white lie. See, however, [Wand and Friedman].)

The effect of the restriction on combinations is startling. On the one hand, they do not so constrain the language as to be useless; on the other hand, they require a radically different approach to the expression of algorithms. It is easy to see that no control stack is necessary to evaluate such code, for, as mentioned in [SCHEME], control stack is used only to keep track of intermediate values and return addresses, and these arise only in the case of combinations with non-trivial arguments, and conditionals with non-trivial predicates.

An algorithm for converting SCHEME programs to continuation-passing style was given in Appendix A of [Declarative]. {Note Old CPS Algorithm} The one used in RABBIT is almost identical, except that for the convenience of the code-generation phase a distinction is made between ordinary LAMBDA-expressions and continuations, and between combinations used to invoke "functions" and those used to invoke continuations. These sets can in fact be consistently distinguished, and it affords a certain amount of error-checking; for example, a LAMBDA-expression should never appear in the "function" position of a continuation-invoking combination. Another fine point is that ASET' can never be applied to a variable bound by a continuation. Except for such differences arising from their uses, the two sets of constructs are treated more or less identically in later phases. An additional difference between the algorithm in [Declarative] and the one in RABBIT is that trivial subforms are treated as single nodes in the CPS version; these nodes have pointers to the non-CPS versions of the relevant code, which are largely ignored by later processing until the final code is to be generated.

It must be emphasized that there is not necessarily a unique CPS version for a given SCHEME program; there is as much freedom as there is in the original program to re-order the evaluation of subexpressions. In the course of the conversion process decisions must be made as to what order to use in evaluating arguments to combinations. The current decision procedure is fairly simple-minded, consisting mostly of not making copies of constants and the values of variables. The point here, as earlier, is not so much that RABBIT has a much better algorithm than other compilers as that it has a far cleaner way of expressing the result. (For a complex decision procedure for argument ordering, see [Wulf].) {Note Non-deterministic CPS Conversion}

E. Environment and closure analysis

This phase consists of four passes over the CPS version of the program. As with the earlier preliminary analysis, each pass determines one related set of information and attaches this information to nodes of the program tree and to property lists.

The first pass (CENV-ANALYZE) analyzes variable references for the CPS version in a manner similar to that of the first pass of the preliminary analysis. The results of this previous analysis are used here in the case of trivial expressions; with this exception the analysis is redone completely, because additional variables are introduced by the CPS conversion. (None of these new variables can appear in an ASET', however, and so the analysis of written variables need not be done over.) In addition, for each variable reference which does not occur in the function position of a combination, we mark that variable with a non-nil VARIABLE-REFP property, used later to determine whether closures need to be created for known functions.

The second pass (BIND-ANALYZE) determines for each LAMBDA-expression whether a closure will be needed for it at run time. There are three possibilities:

(1) If the function denoted by the LAMBDA-expression is bound to some variable, and that variable is referenced other than in function position, then the closure is being treated as data, and must be a full (standard CBETA format) closure. If the function itself occurs in non-function position other than in a LAMBDA-combination, it must be fully closed.

(2) If the closure is bound to some variable, and that variable is referenced

only in function position, but some of these references occur within other partially or fully closed functions, then this function must be partially closed. By this we mean that the environment for the closure must be "consed up", but no pointer to the code need be added on as for a full closure. This function will always be called from places that know the name of the function and so can just perform a GO to the code, but those such places which are within closures must have a complete copy of the necessary environment.

(3) In other cases (functions bound to variables referenced only in function position and never within a closed function, or functions occurring in function position of LAMBDA-combinations), the function need not be closed. This is because the environment can always be fully recovered from the environment at the point of call.

In order to determine this information, it is necessary to determine, for each node, the set of variables referred to from within closed functions at or below that node. Thus this process and the process of determining which functions to close are highly interdependent, and so must be accomplished in a single pass.

The second pass also generates a name for each LAMBDA-expression (to be used as tags in the output code, as discussed in the examples earlier), and for non-closed functions determines which variables will be assigned to "registers" or "memory locations". For these non-closed functions it may determine that certain variables need not be assigned locations at all (they are never referenced, or are bound to other non-closed functions -- the latter circumstance is important when a variable is known to denote a certain function, but the optimizer was too conservative to perform beta-substitution for fear of duplicating code and thus wasting space). Finally, for each variable which is (logically, at run time not necessarily actually) bound to a known function (and which never appears in an ASET'), a property KNOWN-FUNCTION is put on its property list whose value is the node of the CPS version of that function. This property is used later in generating code for combinations in whose function positions such variables appear.

The third pass (DEPTH-ANALYZE) examines each LAMBDA-expression and determines the precise registers or memory locations through which arguments are to be passed to each. Closed functions take their arguments in the standard registers described earlier; non-closed functions may take their arguments in any desired places. (Partially closed functions could also, but there is little advantage to this.) The allocation strategy in RABBIT for non-closed functions is presently merely stack-like; the deeper the nesting of a function, the higher in the ordering of "registers" and "memory locations" are the locations assigned. (See e.g. [Johnsson] for a detailed analysis of the register allocation problem.)

The fourth pass (CLOSE-ANALYZE) determines the precise format of the environment to be constructed for each closure. That is, while the third pass handles cases for which stack-allocation of environments will suffice, the fourth pass deals with heap-allocated environment structures. Recall that the format of an environment can be completely arbitrary, since the only code which can possibly refer to an environment is the function for a closure of which the environment was created. Therefore the compiler which compiles that function has a free hand in determining the structure of the environment. For the sake of simplicity, RABBIT chooses to generate code which represents environments simply as a list of variable values. Several environment lists may share a common tail. The environment for a closure need not contain any variables not needed by the closed function, but it may if this will allow the sharing of a single structure among several closures. (There is a problem with variables modified by ASET' which is discussed in the next paragraph.)

For each LAMBDA-expression which must be closed, three sets of variables are computed: (1) the variables which will already be in the "consed" environment structure at the time the closure is to be created; (2) additional variables which must be added ("consed on") to the existing structure to create the closure (because at that point they are spread out in "registers") {Note Heap-Allocated Contours}; (3) variables which must be added to the environment immediately after entering the function because they must eventually be added in for closures later and they are referred to in ASET' constructs. The third set arises from a requirement that ASET' constructs must have a consistent effect, and confusion can arise if a variable's value can be in more than one place. If the value were allowed to be both in a "register" and in an environment structure, or in several different environment structures, then altering the value in one place would not affect the other places. To assure consistency, this third set is computed, and such variables must at run time be placed in an environment structure to be shared by all others which refer to such variables.

For every LABELS statement a set of variables is computed which is the set of variables to be added to the existing environment on entry to the LABELS body, in order to share this new structure among all the closures to be created for the LABELS functions.

F. Code generation

Given the foregoing analysis, the generation of code is straightforward, and largely consists of using the information already gathered to detect special cases. The special cases of interest relate almost entirely to function calls and closures (indeed, there is little else in the language for RABBIT's purposes!).

RABBIT has provision for "block compiling" a number of functions into a single module. This permits an optimization in which one function can transfer control directly to another without going through the "UUO handler". Even if several user functions are not compiled into a single module, this is still of advantage, because a single user function can produce a large number of output functions, as a consequence of the code-generation techniques.

A module consists of a single MacLISP function whose body is a single PROG. This PROG has no local variables, but does have a number of tags, one for each function in the module. On entry to the module, the register **ENV** will contain the "environment" for the function to be executed. As noted above, the format of this is arbitrary. For functions compiled by RABBIT, this is a list whose car is a tag within the PROG and whose cdr is the "real environment". {Note Code Pointers} At the beginning of the PROG there is always the code

```
(GO (PROG2 NIL (CAR **ENV**)
           (SETQ **ENV** (CDR **ENV**))))
```

the effect of which is to put the "real environment" in **ENV** and then perform a computed GO to the appropriate tag. (This is the only circumstance in which the MacLISP PROG2 and computed GO constructs are used by RABBIT-compiled code. Either could be eliminated at the expense of more bookkeeping, the former by using a temporary intermediate variable, the latter by using a giant COND with non-computed GO statements (which is effectively how the MacLISP compiler compiles a computed GO anyway). As always, such trivial issues are left to the MacLISP compiler when they do not bear on the issues of interest in compiling SCHEME code.) For small functions, often the "main entry point" is the only closed function, and it would be possible to eliminate the computed GO, but RABBIT always outputs one, because is is cheap and provides a useful error check.

Once the computed GO has been performed, the code following the tag is responsible for performing its bit of computation and then exiting. It may exit by setting the **FUN** register to another function, setting up appropriate argument registers, and then doing (RETURN NIL) to exit the module and enter the UUO handler; or it may exit by directly transferring control to another function within the module by performing a GO to the appropriate tag, after setting up the arguments and **ENV**. In the latter case the arguments may actually be passed through "memory locations" rather than the standard "registers". (Conceptually, in this optimized case the environment needed for the function being called is being passed, not in **ENV**, but spread out in those registers and locations lower than those being used to pass the arguments.)

Starting with the CPS version of one or more user functions, the generation of the code for a module proceeds iteratively. Code for each function is generated in turn, producing one segment of code and a tag; this tag and code will become part of the body of the module. In processing a function, other functions may be encountered; in general, each such function is added to the list of outstanding functions for the module, and is replaced by code to generate a closure for that function. When all functions have been processed, the outer structure of the module is created.

Many situations are treated specially. For example,

```
((LAMBDA ...) ...)
```

does not cause the LAMBDA-expression to be added to the list of outstanding functions; rather, a MacLISP PROGN is constructed consisting of the argument set-up followed by the code for the body of the LAMBDA-expression. A more subtle case is

```
(FOO (LAMBDA ...) ...)
```

where FOO is the name of a MacLISP primitive and the LAMBDA-expression is the continuation. In this case a PROGN is constructed consisting of calling the MacLISP primitive on the other arguments, putting this value into the appropriate location, and then executing the body of the LAMBDA-expression. (It should be noted that all these special cases must be anticipated by the analysis preceding the code generation phase.)

In the case of ((LAMBDA ...) ...), we must also handle the argument set-up a little carefully, because parameters which are never referred to or which represent known non-closed functions need not actually be passed. However, the corresponding argument for the first case must nevertheless be evaluated because it may have a side-effect. A good example is the result of expanding BLOCK (neglecting the effects of optimization): there is a (continuation-passing style) combination of the form:

```
((LAMBDA (C A B) (B C)) cont x (LAMBDA (C) y))
```

The argument x need not be passed, but presumably has a side effect and so must be evaluated. The second LAMBDA-expression need not be closed, and so requires neither evaluation nor passing. The output code uses a PROGN to evaluate the arguments which are potentially for effect. In this way the end result of a

BLOCK construct actually turns out to be a MacLISP PROGN. (The routine LAMBDACATE in the Appendix is responsible for this analysis.) {Note Evaluation for Effect}

Another case of interest is a combination whose function position contains a variable with a KNOWN-FUNCTION property. The value of this property is the node for the CPS version of the function, which provides information about possible code generation strategies. We can decide which arguments needn't be passed as for the ((LAMBDA ...) ...) case, and can also arrange to call the function with a direct (MacLISP) GO to the appropriate tag within the module. The set-up of the environment depends on whether the function is non-closed or partially closed; in the latter case the partial closure is the environment, and in the former the environment can be recovered from the current one (and may even be the same).

A certain amount of "peephole optimization" [McKeeman] is also performed, primarily to make it easier for people to inspect the code produced, since the MacLISP compiler will handle them anyway. Examples of these are avoiding the generation of SETQ of a variable to the value of that same variable; reduction of car-cdr chains to single functions, such as (CAR (CDR (CDR x))) to (CADDR x); removal of nested PROGN's such as

```
(PROGN a (PROGN b c) d)  =>  (PROGN a b c d)
```

and the like; and simplification of nested COND's, such as

```
(COND (a b)                 (COND (a b)
      (T (COND (c d)   =>         (c d)
               ...)))             ...)
```

One of the effects of this last peephole optimization is that many times, when the user writes a COND in a piece of SCHEME code, that COND is expanded into IF constructs, and then re-contracted by the peephole optimization into an equivalent COND! (This fact is of no practical consequence, but looks cute.)

9. Example: Compilation of Iterative Factorial

Here we shall provide a complete example of the compilation of a simple function IFACT (iterative factorial), to show what quantities are computed in the course of analyzing the code. We shall need some notation for the data structures involved. Every node of the program is represented by a small data structure which has a type and several named components. (In the actual implementation, a node is represented as two such structures; one contains named components common to all program nodes, and the other contains components specific to a given node type. We shall gloss over this detail here.) For example, a LAMBDA-expression is represented by a structure of type LAMBDA with components named UVARS (user variable names), VARS (the alpha-converted names), BODY (the node representing the body), ENV (the environment of the node), and so on. We shall represent a data structure as the name of its type, with the components written below it and indented, with colons after each component name. For example:

```
LAMBDA
    UVARS:  (A B)
    VARS:   (VAR-43 VAR-44)
    BODY:   COMBINATION
                ARGS:  VARIABLE
                           VAR:   F
                       VARIABLE
                           VAR:   VAR-44
                       VARIABLE
                           VAR:   VAR-43
```

Notice that the value of a component may itself be a structure. These structures are always arranged in a tree, so no notation for cycles will be needed. In the case where a component contains a list of things, we will write the things as a LISP list unless the things are structures, in which case we will simply write them in a vertical stack, as shown in the example above. To conserve space, in any single diagram we will show only the named components of interest. Components may seem to appear and then disappear in the series of diagrams, but in practice they all exist simultaneously.

The source code for our example:

```
(DEFINE IFACT
        (LAMBDA (N)
                (LABELS ((F (LAMBDA (M A)
                                    (IF (= M 0) A
                                        (F (- M 1) (* M A))))))
                        (F N 1))))
```

The alpha-conversion process copies the program and produces a tree of structures. All the bound variables are renamed, and VARIABLE nodes refer to these new names. The GLOBALP component in a VARIABLE node is non-NIL iff the reference is to a global variable. The ENV component is simply an a-list relating the user names of variables to the new names; this a-list is computed during the conversion as the new names are created at LAMBDA, LABELS, and CATCH nodes.

```
LAMBDA
    ENV:    ()
    UVARS:  (N)
    VARS:   (VAR-1)
    BODY:
        LABELS
            ENV:        ((N VAR-1))
            UFNVARS:    (F)
            FNVARS:     (FNVAR-2)
            FNDEFS:
                LAMBDA
                    ENV:    ((F FNVAR-2) (N VAR-1))
                    UVARS:  (M A)
                    VARS:   (VAR-3 VAR-4)
                    BODY:   IF
                                ENV:    ((A VAR-4) (M VAR-3) (F FNVAR-2) (N VAR-1))
                                PRED:   COMBINATION
                                            ENV:    ***   (see below)
                                            ARGS:   VARIABLE
                                            ENV:    ***
                                            VAR:    =
                                            GLOBALP: T
                                        VARIABLE
                                            ENV:    ***
                                            VAR:    VAR-3
                                            GLOBALP: NIL
                                        CONSTANT
                                            ENV:    ***
                                            VALUE:  0
                    CON:    VARIABLE
                                ENV:    ***
                                VAR:    VAR-4
                                GLOBALP: NIL
                    ALT:    COMBINATION
                                ENV:    ***
                                ARGS:   VARIABLE
                                            ENV:    ***
                                            VAR:    FNVAR-2
                                            GLOBALP: NIL
                                        COMBINATION
                                            ENV:    ***
                                            ARGS:   VARIABLE
                                                        ENV:    ***
                                                        VAR:    -
                                                        GLOBALP: T
                                                    VARIABLE
                                                        ENV:    ***
                                                        VAR:    VAR-3
                                                        GLOBALP: NIL
                                                    CONSTANT
                                                        ENV:    ***
                                                        VALUE:  1
                                        COMBINATION
                                            ENV:    ***
                                            ARGS:   VARIABLE
                                                        ENV:    ***
                                                        VAR:    *
                                                        GLOBALP: T
                                                    VARIABLE
                                                        ENV:    ***
                                                        VAR:    VAR-3
                                                        GLOBALP: NIL
                                                    VARIABLE
                                                        ENV:    ***
                                                        VAR:    VAR-4
                                                        GLOBALP: NIL
BODY:   COMBINATION
            ENV:    ((F FNVAR-2) (N VAR-1))
            ARGS:   VARIABLE
                        ENV:    ((F FNVAR-2) (N VAR-1))
                        VAR:    FNVAR-2
                        GLOBALP: NIL
                    VARIABLE
                        ENV:     ((F FNVAR-2) (N VAR-1))
                        VAR:     VAR-1
                        GLOBALP: NIL
                    CONSTANT
                        ENV:     ((F FNVAR-2) (N VAR-1))
                        VALUE:   1
```

The reader is asked to imagine that the expression

```
((A VAR-4) (M VAR-3) (F FNVAR-2) (N VAR-1))
```

occurs where *** appears in the diagram. It should be clear how the ENV components are computed on the basis of variables bound at the LAMBDA and LABELS nodes. The ENV information propagates down the tree to VARIABLE nodes, where it is used to supply the correct new name for the one used by the original code.

The first step in the preliminary analysis is the determination of referenced variables:

```
LAMBDA
    REFS:   ()
    VARS:   (VAR-1)
    BODY:
        LABELS
            REFS:       (VAR-1)
            FNVARS:     (FNVAR-2)
            FNDEFS:
                LAMBDA
                    REFS:   (FNVAR-2)
                    VARS:   (VAR-3 VAR-4)
                    BODY:   IF
                                REFS:   (FNVAR-2 VAR-3 VAR-4)
                                PRED:   COMBINATION
                                            REFS:   (VAR-3)
                                            ARGS:   VARIABLE
                                                        REFS:   ()
                                                        VAR:    =
                                                        GLOBALP: T
                                                    VARIABLE
                                                        REFS:   (VAR-3)
                                                        VAR:    VAR-3
                                                        GLOBALP: NIL
                                                    CONSTANT
                                                        REFS:   ()
                                            VALUE:  0
                    CON:    VARIABLE
                                REFS:   (VAR-4)
                                VAR:    VAR-4
                                GLOBALP: NIL
                    ALT:    COMBINATION
                                REFS:   (FNVAR-2 VAR-3 VAR-4)
                                ARGS:   VARIABLE
                                            REFS:   (FNVAR-2)
                                            VAR:    FNVAR-2
                                            GLOBALP: NIL
                                        COMBINATION
                                            REFS:   (VAR-3)
                                            ARGS:   VARIABLE
                                                        REFS:   ()
                                                        VAR:    -
                                                        GLOBALP: T
                                                    VARIABLE
                                                        REFS:   (VAR-3)
                                                        VAR:    VAR-3
                                                        GLOBALP: NIL
                                                    CONSTANT
                                                        REFS:   ()
                                                        VALUE:  1
                                        COMBINATION
                                            REFS:   (VAR-3 VAR-4)
                                            ARGS:   VARIABLE
                                                        REFS:   ()
                                                        VAR:    *
                                                        GLOBALP: T
                                                    VARIABLE
                                                        REFS:   (VAR-3)
                                                        VAR:    VAR-3
                                                        GLOBALP: NIL
                                                    VARIABLE
                                                        REFS:   (VAR-4)
                                                        VAR:    VAR-4
                                                        GLOBALP: NIL
BODY:   COMBINATION
            REFS:   (FNVAR-2 VAR-1)
            ARGS:   VARIABLE
                        REFS:   (FNVAR-2)
                        VAR:    FNVAR-2
                        GLOBALP: NIL
                    VARIABLE
                        REFS:   (VAR-1)
                        VAR:    VAR-1
                        GLOBALP: NIL
                    CONSTANT
                        REFS:   ()
                        VALUE:  1
```

The REFS component is a list of all <u>local</u> variables referenced at or below the node. Notice that, in general, the REFS component of a node is the union (considering them as sets) of the REFS components of its subnodes. In this way the information sifts up from the VARIABLE nodes. At a LAMBDA, LABELS, or CATCH, the variables bound at that node are filtered out of the REFS sifting up. The REFS for the outer function must always be (), a useful error check. In this example, we see that VAR-1 (N) is not referenced by the function FNVAR-2 (F). This indicates that a closure for this function need not contain the value for VAR-1 in its environment. (We will not actually use the information for this purpose, since later analysis will determine that the function need not have a closure constructed for it.) Another component ASETVARS is computed for each node, which contains the set of variables appearing in an ASET' at or below the node. We have omitted this information from the diagram since the value is the empty set in all cases. Certain properties are placed on the property list of each variable as well, which are not shown here.

The next pass locates trivial subforms:

```
LAMBDA
    TRIVP:  NIL
    VARS:   (VAR-1)
    BODY:
        LABELS
            TRIVP:      NIL
            FNVARS:     (FNVAR-2)
            FNDEFS:
                LAMBDA
                    TRIVP:  NIL
                    VARS:   (VAR-3 VAR-4)
                    BODY:   IF
                                TRIVP:  NIL
                                PRED:   COMBINATION
                                            TRIVP:  T
                                            ARGS:   VARIABLE
                                                        TRIVP:  T
                                                        VAR:    =
                                                        GLOBALP: T
                                                    VARIABLE
```

```
                                                TRIVP:  T
                                                VAR:    VAR-3
                                                GLOBALP: NIL
                                            CONSTANT
                                                TRIVP:  T
                                                VALUE:  0
                        CON:    VARIABLE
                                    TRIVP:  T
                                    VAR:    VAR-4
                                    GLOBALP: NIL
                        ALT:    COMBINATION
                                    TRIVP:  NIL
                                    ARGS:   VARIABLE
                                                TRIVP:  T
                                                VAR:    FNVAR-2
                                                GLOBALP: NIL
                                            COMBINATION
                                                TRIVP:  T
                                                ARGS:   VARIABLE
                                                            TRIVP:  T
                                                            VAR:    -
                                                            GLOBALP: T
                                                        VARIABLE
                                                            TRIVP:  T
                                                            VAR:    VAR-3
                                                            GLOBALP: NIL
                                                        CONSTANT
                                                            TRIVP:  T
                                                            VALUE:  1
                                            COMBINATION
                                                TRIVP:  T
                                                ARGS:   VARIABLE
                                                            TRIVP:  T
                                                            VAR:    *
                                                            GLOBALP: T
                                                        VARIABLE
                                                            TRIVP:  T
                                                            VAR:    VAR-3
                                                            GLOBALP: NIL
                                                        VARIABLE
                                                            TRIVP:  T
                                                            VAR:    VAR-4
                                                            GLOBALP: NIL
BODY:   COMBINATION
            TRIVP:  NIL
            ARGS:   VARIABLE
                        TRIVP:  T
                        VAR:    FNVAR-2
                        GLOBALP: NIL
                    VARIABLE
                        TRIVP:  T
                        VAR:    VAR-1
                        GLOBALP: NIL
```

```
CONSTANT
    TRIVP:  T
    VALUE:  1
```

Constants and variables are always trivial, and trivial combinations (involving only MacLISP primitives) are located. As before, in this pass information sifts up from below. One possibility not yet explored in RABBIT is to isolate entire SCHEME functions (for example FNVAR-2), determine that it is, as a whole, trivial, compile it as a simple MacLISP SUBR, and reference it as a primitive. This would in turn render trivial the combination (F N 1) in the body of the LABELS, for example.

The analysis of side-effects merely determines that no side-effects are present, and is uninteresting for our example. The optimization pass finds no transformations worth making. We will skip over these steps to the conversion to continuation-passing style. As a simple S-expression, this may be rendered as:

```
(LAMBDA (CONT-5 VAR-1)
        (LABELS ((FNVAR-2
                   (LAMBDA (CONT-6 VAR-3 VAR-4)
                           (IF (= VAR-3 0)
                               (CONT-6 VAR-4)
                               (FNVAR-2 CONT-6
                                        (- VAR-3 1)
                                        (* VAR-3 VAR-4))))))
                (FNVAR-2 CONT-5 VAR-1 1)))
```

In rendering this as a tree of data structures, we use structures of type CLAMBDA instead of LAMBDA, etc., in order to prevent confusion. Trivial forms are represented by structures of type TRIVIAL with pointers to the data structures from before. We will not notate such data structures in the following diagrams, but will simply write an S-expression as a reminder of what the trivial form was. The types RETURN and CONTINUATION are like CCOMBINATION and CLAMBDA, but are distinguished as discussed above for convenience and for purposes of consistency checking.

```
CLAMBDA
    VARS:   (CONT-5 VAR-1)
    BODY:   CLABELS
                FNVARS: (FNVAR-2)
                FNDEFS: CLAMBDA
                            VARS:   (CONT-6 VAR-3 VAR-4)
                            BODY:   CIF
                                        PRED:   TRIVIAL
                                                    (= VAR-3 0)
                                        CON:    RETURN
                                                    CONT:   CVARIABLE
                                                                VAR:    CONT-6
                                                    VAL:    TRIVIAL
                                                                VAR-4
                                        ALT:    CCOMBINATION
                                                    ARGS:   TRIVIAL
                                                                FNVAR-2
                                                            CVARIABLE
                                                                VAR:    CONT-6
                                                            TRIVIAL
                                                                (- VAR-3 1)
                                                            TRIVIAL
                                                                (* VAR-3 VAR-4)
                BODY:   CCOMBINATION
                            ARGS:   TRIVIAL
                                        FNVAR-2
                                    CVARIABLE
                                        VAR:    CONT-5
                                    TRIVIAL
                                        VAR-1
                                    TRIVIAL
                                        1
```

The first post-conversion analysis pass computes ENV and REFS components as before, this time including the variables introduced to represent continuations. The ENV in this case is not an a-list, but simply a list of variables, since no renaming is taking place. The ENV information sifts down from above during the tree walk, and on the way back the REFS information sifts up. For a TRIVIAL node, the REFS information is taken from the pre-conversion node referenced by the TRIVIAL node; this REFS information is shown here as a reminder. As before, the REFS information for a node is always a subset of the

ENV information.

```
CLAMBDA
    ENV:    ()
    REFS:   ()
    VARS:   (CONT-5 VAR-1)
    BODY:   CLABELS
                ENV:    (CONT-5 VAR-1)
                REFS:   (CONT-5 VAR-1)
                FNVARS: (FNVAR-2)
                FNDEFS: CLAMBDA
                            ENV:    (FNVAR-2 CONT-5 VAR-1)
                            REFS:   (FNVAR-2)
                            VARS:   (CONT-6 VAR-3 VAR-4)
                            BODY:   CIF
                                        ENV:    ***
                                        REFS:   (FNVAR-2 CONT-6 VAR-3 VAR-4)
                                        PRED:   TRIVIAL
                                                    REFS:   (VAR-3)
                                                    (= VAR-3 0)
                                        CON:    RETURN
                                                    ENV:    ***
                                                    REFS:   (CONT-6 VAR-4)
                                                    CONT:   CVARIABLE
                                                                ENV:    ***
                                                                REFS:   (CONT-6)
                                                                VAR:    CONT-6
                                                    VAL:    TRIVIAL
                                                                REFS:   (VAR-4)
                                                                VAR-4
                                        ALT:    CCOMBINATION
                                                    ENV:    ***
                                                    REFS:   (FNVAR-2 CONT-6 VAR-3 VAR-4)
                                                    ARGS:   TRIVIAL
                                                                REFS:   (FNVAR-2)
                                                                FNVAR-2
                                                            CVARIABLE
                                                                ENV:    ***
                                                                REFS:   (CONT-6)
                                                                VAR:    CONT-6
                                                            TRIVIAL
                                                                REFS:   (VAR-3)
                                                                (- VAR-3 1)
                                                            TRIVIAL
                                                                REFS:   (VAR-3 VAR-4)
                                                                (* VAR-3 VAR-4)
                BODY:   CCOMBINATION
                            ENV:    (FNVAR-2 CONT-5 VAR-1)
                            REFS:   (FNVAR-2 CONT-5 VAR-1)
                            ARGS:   TRIVIAL
                                        REFS:   (FNVAR-2)
                                        FNVAR-2
```

```
CVARIABLE
    ENV:    (FNVAR-2 CONT-5 VAR-1)
    REFS:   (CONT-5)
    VAR:    CONT-5
TRIVIAL
    REFS:   (VAR-1)
    VAR-1
TRIVIAL
    REFS:   ()
    1
```

The reader is asked to imagine that where *** occurs the expression

```
(CONT-6 VAR-3 VAR-4 FNVAR-2 CONT-5 VAR-1)
```

had been written instead. An additional operation performed on this pass is to flag all variables referenced in other than function position. These include VAR-1, VAR-3, etc.; but FNVAR-2 is <u>not</u> among them. This will be of importance below.

The next pass determines all variables referenced by closures at or below each node, and also decides which functions will actually be closed. It is determined that FNVAR-2 need not be closed, because it is referred to only in function position (as determined by the previous pass), and is not referred to by any other closures. As a result, no closures are created at all in this function, and so all the computed sets of variables are empty. This pass also assigns the name F-7 to the outer function, for use later as a tag.

The third pass computes the "depth" of each function, which determines through what registers or other locations arguments will be passed for each function. In this case the outer CLAMBDA is assigned depth 0, and the one labelled FNVAR-2 is assigned depth 2, because it is not closed, and is contained in a depth 0 function of 2 arguments. In this way registers are allocated in a purely stack-like manner; all closed functions are of depth 0, and all unclosed ones are at a depth determined by that of the containing function and its number of arguments.

One way to think about this trick is as follows. A closure consists of a pointer to a piece of code and a set of values determined at the time of closure. When the closure is invoked, we execute the code, making available to it (a) the set of values (its environment), and (b) some additional arguments. Slicing these components a different way, we may think of calling the bare code, supplying all the values as arguments; we pass the arguments in some registers, and the environment values in some other registers. Put yet another way, if we can determine that every caller of the closed function can reconstruct the necessary environment at the time of the call (because it will have available the necessary values anyway), then we can avoid constructing the closure at the point where the function should be closed, and instead arrange for each caller to pass the environment through specified registers. As mentioned earlier, the compiler has a completely free hand in determining the format of an environment!

As it happens, the function labelled FNVAR-2 does not reference CONT-5 or VAR-1, and so this argument is of no importance here. It is determined that the following register assignments will apply:

CONT-5	**CONT**
VAR-1	**ONE**
FNVAR-2	<none>
CONT-6	**TWO**
VAR-3	**THREE**
VAR-4	**FOUR**

{Note Continuation Variable Hack} We will see below that some unnecessary shuffling of values results; a more complicated register assignment technique would be useful here. (One was outlined in [Declarative], but it has not been implemented. See also [Wulf] and [Johnsson].)

The fourth post-conversion analysis pass determines the format of environments for closed functions. Since there are none in this example, this analysis is of little interest here.

Finally, we are ready to generate code. Consider the S-expression form:

```
(LAMBDA (CONT-5 VAR-1)
        (LABELS ((FNVAR-2
                  (LAMBDA (CONT-6 VAR-3 VAR-4)
                          (IF (= VAR-3 0)
                              (CONT-6 VAR-4)
                              (FNVAR-2 CONT-6
                                       (- VAR-3 1)
                                       (* VAR-3 VAR-4))))))
                (FNVAR-2 CONT-5 VAR-1 1)))
```

The first function encountered is the outer one (named F-7). In analyzing its body we note the LABELS, and place all the labelled functions (that is, FNVAR-2) on the queue of functions yet to be processed. We then analyze the body of the LABELS. This is a combination, and so we analyze each argument, producing code for each. Each argument must be TRIVIAL, a (C)VARIABLE, or a (C)LAMBDA-expression. (We shall refer to this set of possibilities as "meta-trivial", which means what "trivial" did in [Imperative].) The variable FNVAR-2 refers to a known function which is not closed, and so we need not set up **FUN**. The others may be referred to as **CONT**, **ONE**, and the constant 1, respectively. These are to be passed to FNVAR-2 through the registers **TWO**, **THREE**, and **FOUR** (as determined by the register allocation pass). Thus the code for F-7 looks like this:

```
F-7     ((LAMBDA (Q-40 Q-41 Q-42)
                 (SETQ **FOUR** Q-42)
                 (SETQ **THREE** Q-41)
                 (SETQ **TWO** Q-40))
         **CONT** **ONE** '1)
        (GO FNVAR-2)
```

The first form sets up the arguments, using a standard "simultaneous assignment" construction. The second branches to the code for FNVAR-2. Because a known function is being called, it is not necessary to set up **NARGS**. Because FNVAR-2 requires no closure, it is not necessary to set up **ENV**.

The next function on the queue to process is FNVAR-2. Its body is an IF (actually a CIF); this is compiled into a COND containing the code for the predicate, consequent, and alternative:

```
(COND (<predicate> <consequent>)
      (T <alternative>))
```

The predicate is guaranteed to be meta-trivial. It is, in this example, a trivial combination; this is compiled by changing all the variable references appropriately, producing (= **THREE** '0).

The consequent involves calling an unknown continuation which is in **TWO**. The returned value is in **FOUR**. The code produced is:

```
(SETQ **FUN** **TWO**)
(SETQ **ONE** **FOUR**)
(RETURN NIL)
```

The (RETURN NIL) exits the module, passing control to the dispatcher in the SCHEME interpreter, which will arrange to invoke the continuation.

The code for the alternative is similar to that for the body of F-7, because we are calling the known function FNVAR-2. The generated code is:

```
((LAMBDA (Q-43 Q-44)
         (SETQ **FOUR** Q-44)
         (SETQ **THREE** Q-43))
 (- **THREE** '1)
 (* **THREE** **FOUR**))
(GO FNVAR-2)
```

The argument set-up ought to involve copying **TWO** into **TWO**, but a peephole optimization eliminates that SETQ.

Putting all this together, the code for FNVAR-2 is:

```
FNVAR-2 (COND ((= **THREE** '0)
               (SETQ **FUN** **TWO**)
               (SETQ **ONE** **FOUR**)
               (RETURN NIL))
              (T ((LAMBDA (Q-43 Q-44)
                          (SETQ **FOUR** Q-44)
                          (SETQ **THREE** Q-43))
                  (- **THREE** '1)
                  (* **THREE** **FOUR**))
                 (GO FNVAR-2)))
```

(We have glossed over the peephole optimizations which eliminate occurrences of PROGN in such places as COND clauses.)

There are no more functions to be processed, and so we now create the final module. The final output, with comments inserted by RABBIT for debugging purposes, and declarations supplied by RABBIT for the benefit of the MacLISP compiler, looks like this:

```
(PROGN 'COMPILE
       (COMMENT MODULE FOR FUNCTION IFACT)
       (DEFUN ?-37 ()
              (PROG ()
                    (DECLARE (SPECIAL ?-37))
                    (GO (PROG2 NIL (CAR **ENV**) (SETQ **ENV** (CDR **ENV**))))
               F-7  (COMMENT (DEPTH = 0) (FNP = NIL) (VARS = (CONT-5 N)))
                    ((LAMBDA (Q-40 Q-41 Q-42)
                             (SETQ **FOUR** Q-42)
                             (SETQ **THREE** Q-41)
                             (SETQ **TWO** Q-40))
                     **CONT** **ONE** '1)
                    (COMMENT (DEPTH = 2) (FNP = NOCLOSE) (VARS = (CONT-6 M A)))
                    (GO FNVAR-2)
               FNVAR-2
                    (COMMENT (DEPTH = 2) (FNP = NOCLOSE) (VARS = (CONT-6 M A)))
                    (COND ((= **THREE** '0)
                           (SETQ **FUN** **TWO**)
                           (SETQ **ONE** **FOUR**)
                           (RETURN NIL))
                          (T ((LAMBDA (Q-43 Q-44)
                                      (SETQ **FOUR** Q-44)
                                      (SETQ **THREE** Q-43))
                              (- **THREE** '1)
                              (* **THREE** **FOUR**))
```

```
                        (COMMENT (DEPTH = 2) (FNP = NOCLOSE) (VARS = (CONT-6 M A)))
                        (GO FNVAR-2)))))
    (SETQ ?-37 (GET '?-37 'SUBR))
    (SETQ IFACT (LIST 'CBETA ?-37 'F-7))
    (DEFPROP ?-37 IFACT USER-FUNCTION))
```

In the interpolated comments, FNP refers to whether the function being entered or being called is closed or not (the possibilities are NIL, NOCLOSE, and EZCLOSE). The VARS are the passed variables, expressed as the names from the original source code, except for those introduced by the CPS conversion. The form (SETQ IFACT ...) constructs the closure for the globally defined function IFACT. The DEFPROP form provides debugging information.

The points of interest in this example are the isolation of trivial subforms, and the analysis of the function FNVAR-2 which allows it to be called with GO. Examination of the output code will show that FNVAR-2 is coded as an iterative loop. While the register allocation leaves something to be desired, the inner loop does surprisingly little shuffling. (This should be compared with the code suggested in [Declarative] for this function.)

For those who prefer "real" machine language, we give a plausible transcription of the MacLISP code into our hypothetical machine language:

```
IFACT:  PUSH CONT                 ;CONT contains the return address
        PUSH ONE
        PUSH 1
        POP FOUR
        POP THREE
        POP TWO
        GOTO FNVAR2
```

```
FNVAR2: JUMP-IF-ZERO THREE,FNV2A
        MOVE ONE,FOUR
        RETURN (TWO)                ;return to address in TWO
FNV2A:  MOVE TEMP,THREE             ;TEMP is used to evaluate
        ADD TEMP,1                  ; trivial forms
        PUSH TEMP
        MOVE TEMP,THREE
        MUL TEMP,FOUR
        PUSH TEMP
        POP FOUR
        POP THREE
        GOTO FNVAR2
```

While this is not the world's most impressively tight code, it again shows the essential iterative structure of the inner loop. The primary problem is the absence of analysis of which registers are used when. Leaving aside the question of allocating registers, one could at least determine when assigning values to registers for argument set-up can occur sequentially rather than simultaneously.

There are a few other obvious optimizations which have not been performed, for example the elimination of (GO FNVAR-2) just before the tag FNVAR-2. While this would not have been difficult, we knew that the MacLISP compiler would take care of this for us; since it is not a very interesting issue, we let it slide.

10. Performance Measurements

RABBIT has provision for metering runtime usage, and for controlling whether certain options in the optimizer are used. The standard test case has been RABBIT compiling itself (!); by running both interpreted and compiled version of this task, some comparisons have been made. Two different compiled versions have also been tested, where the code was produced with or without using the optimizer.

The overall speed gain of unoptimized compiled code over interpreted code has been measured to be a factor of 25. The speed gain ratio excluding time for garbage collection was 17, and the garbage collection time ratio was 34. (The SCHEME interpreter does a lot of consing. The straight runtime ratio of 17 is roughly typical for standard LISP compilers on non-numeric code.)

The overall speed ratio of optimized compiled code to unoptimized compiled code has been measured to be 1.2. The speed ratio excluding garbage collection was 1.37, and the garbage collection time ratio was 1.07. We conclude that the amount of consing was reduced very little, despite optimizations which may eliminate closures, because the phase-2 analysis of closures eliminated most consing from that source anyway. Eliminations of register shuffling because of substitutions of one variable for another were probably more significant.

Combining these figures yields an overall speed ratio for optimized compiled code over interpreted code of about 30.

Turning now to the analysis of compilation time, as opposed to running time, we have found that using the optimizer approximately doubles the cost of compilation. It might be possible to reduce this with a more clever optimizer; presently RABBIT wastes much time re-doing certain analysis unnecessarily. The extra time needed by the optimizer excluding garbage collection is only half again the overall compilation time, but the garbage collection time triples, because the optimizer copies and re-copies parts of the program.

There is also one error check which is very expensive; it checks every argument of a combination against every other argument to check for possible side-effect conflicts (this is the "liberal" analysis in EFFS-ANALYZE, and the testing done by CHECK-COMBINATION-PEFFS). Use of this error check increased compilation time by thirty percent.

11. Comparison with Other Work

The only other work we know of similar to ours is that in [Wand and Friedman]. They use a technique from category theory known as factorization to isolate trivial expressions. As far as they go, their work is similar to ours; they have written a compiler for LISP code, producing output code which uses continuations. However, they indicate that they cannot interface compiled and interpreted code correctly. Moreover, while they use continuations, they do not make general use of closures, and in fact there is no clue that closures are permitted in their source language, or that functions are permissible as data objects. (In fact, there is evidence to the contrary in several examples they give involving an expression

```
(MAPCAR (QUOTE (LAMBDA ...)) ...)
```

These seem to indicate that they have not made the crucial distinction between treating a function as a data object and treating a representation of a function as data.) Wand and Friedman do realize the importance of tail-recursion, but fail to mention the necessity for lexical scoping (perhaps taking it for granted). We feel that the contributions of category theory may provide interesting new ways to analyze programs, but also feel that Wand and Friedman have not, in the work cited, explored it thoroughly, since they have not even explored the issue of closures as such.

Somewhat more distantly related is the work of Carter and others at the IBM T.J. Watson Research Lab. [Carter] This work is similar in spirit, in that it uses "macro definitions" of complex operators, which are integrated into the program being compiled, followed by source-to-source program transformations which optimize the resulting code. However, they have primarily worked with definitions of complex data manipulations, such as string concatenation, whereas this report has dealt exclusively with environment and control operations. (Also, as a matter of taste, we find SCHEME a simpler and more tractable language to deal with than the low-level dialect of PL/I used in [Carter], partly because of its closeness to lambda-calculus and partly because SCHEME inherits from LISP the natural ability to deal with representations of its own programs.)

12. Conclusions and Future Work

Lexical scoping, tail-recursion, the conceptual treatment of functions (as opposed to representations thereof) as data objects, and the ability to notate "anonymous" functions make SCHEME an excellent language in which to express program transformations and optimizations. Imperative constructs are easily modelled by applicative definitions. Anonymous functions make it easy to avoid needless duplication of code and conflict of variable names. A language with these properties is useful not only at the preliminary optimization level, but for expressing the results of decisions about order of evaluation and storage of temporary quantities. These properties make SCHEME as good a candidate as any for an UNCOL. The proper treatment of functions and function calls leads to generation of excellent imperative low-level code.

We have emphasized the ability to treat functions as data objects. We should point out that one might want to have a very simple run-time environment which did not support complex environment structures, or even stacks. Such an end environment does not preclude the use of the techniques described here. Many optimizations result in the elimination of LAMBDA-expressions; post CPS-conversion analysis eliminates the need to close many of the remaining LAMBDA-expressions. One could use the macros and internal representations of RABBIT to describe intermediate code transformations, and require that the final code not actually create any closures. As a concrete example, imagine writing an operating system in SCHEME, with machine words as the data domain (and functions excluded from the run-time data domain). We could still meaningfully write, for example:

```
(IF (OR (STOPPED (PROCESS I))
        (AWAITING-INPUT (PROCESS I)))
    (SCHEDULE-LOOP (+ I 1))
    (SCHEDULE-PROCESS I))
```

While the intermediate expansion of this code would conceptually involve the use of functions as data objects, optimizations would reduce the final code to a form which did not require closures at run time.

An experiment we would like to try would be to use CGOL [Pratt], a program which parses ALGOL-like syntax and produces LISP code, as a front end for RABBIT. The result would be a compiler for an ALGOL-like language which would produce code by the processes of parsing (by CGOL); macro-expansion, optimization, and output of MacLISP code (by RABBIT); and generation of PDP-10 machine language (by the MacLISP compiler).

Among the interesting issues we have not dealt with or have not yet implemented in RABBIT are: compilation of data manipulation primitives, interaction of such primitives, procedure integration of the most general form, and complex register allocation. A particularly interesting issue is that of data type analysis. Such analysis would solve certain problems which cannot easily be solved now by RABBIT. For example, consider the piece of code:

```
(IF (OR A B) X Y)
```

The macro-expansion and optimization phases will reduce this to:

```
(IF A (IF A X Y) (IF B X Y))
```

The difficulty is that RABBIT has no way of knowing that A is known to be non-null in the first inner IF by virtue of the testing of A in the outer IF. If it could realize this, then the code would reduce to the more reasonable:

```
        (IF A X (IF B X Y))
```

Compare this with the case of (IF (AND ...) ...) presented earlier.

One particularly nagging difficulty concerns an interaction between CATCH and optimization by substituting expressions for variables. The problem is that if an expression with a side-effect is substituted into a place which is evaluated after the return of a call to an unknown function (where it had been written at a place normally evaluated before the call), and if a CATCH is performed within that unknown function, and the escape function is subsequently called more than once, then the expression with a side-effect will be evaluated twice instead of once. There is no possible way to decide whether this can happen, other than to be fearful of all unknown function calls. In practice this defeats most optimization. We have ignored this difficulty in RABBIT. It probably indicates that escape functions are even more intractable than we had earlier believed. It would not be so bad if we could insist that an escape function be called no more than once (or rather, that a CATCH be returned from no more than once, implying that if the escape function is used it must be dynamically within the body of the CATCH). If this restriction is enforced, or if CATCH is forbidden, then in fact no continuation can be invoked more than once, which, with other suitable restrictions, accounts for the ability of most languages to use stacks instead of trees for their control stacks.

Notes

{Note ASET' Is Imperative}

It is true that ASET' is an actual imperative which produces a side effect, and is not expressed applicatively. ASET' is used only for two purposes in practice: to initialize global variables (often relating to MacLISP primitives), and to implement objects with state (cells, in the PLASMA sense [Smith and Hewitt] [Hewitt and Smith]). If we were to redesign SCHEME from scratch, I imagine that we would introduce cells as our primitive side-effect rather than ASET'. The decision to use ASET' was motivated primarily by the desire to interface easily to the MacLISP environment (and, as a corollary, to be able to implement SCHEME in three days instead of three years!).

We note that in approximately one hundred pages of SCHEME code written by three people, the non-quoted ASET has never been used, and ASET' has been used only a dozen times or so, always for one of the two purposes mentioned above. In most situations where one would like to write an assignment of some kind, macros which expand into applicative constructions suffice.

{Note Code Pointers}

Conceptually a closure is made up of a pointer to some code (a "script" [Smith and Hewitt]) and an environment. In a RABBIT-formatted CBETA, the pointer to the code is encoded into two levels: a pointer to a particular piece of MacLISP code, plus a tag within that PROG. This implementation was forced upon us by MacLISP. If we could easily create pointers into the middle of a PROG, we could avoid this two-level encoding.

On the other hand, this is not just an engineering kludge, but can be provided with a reasonable semantic explanation: rather than compiling a lot of little functions, we compile a single big function which is a giant CASE statement. Wherever we wish to make a closure of a little function, we actually close a different little function which calls the big function with an extra argument to dispatch on.

{Note Continuation Variable Hack}

Since the dissertation was written, a simple modification to the routine which converts to continuation-passing style has eliminated some of the register shuffling. The effect of the change was to perform substitutions of one continuation variable for another, in situations such as:

```
((CLAMBDA (CONT-3 ...) ...)
 CONT-2 ...)
```

where CONT-2 would be substituted for CONT-3 in the body of the CLAMBDA-expression. Once this is done, CONT-3 is unreferenced, and so is not really passed at all by virtue of the phase-2 analysis. The result is that continuations are not copied back and forth from register to register. In the iterative factorial example in the text, the actual register assignment would be:

CONT-5	**CONT**
VAR-1	**ONE**
FNVAR-2	<none>
VAR-3	**TWO**
VAR-4	**THREE**

This optimization is discussed more thoroughly in the Appendix near the routine CONVERT-COMBINATION.

{Note Dijkstra's Opinion}

In [Dijkstra] a remark is made to the effect that defining the while-do construct in terms of function calls seems unusually clumsy. In [Steele] we reply that this is due partly to Dijkstra's choice of ALGOL for expressing the definition. Here we would add that, while such a definition is completely workable and is useful for compilation purposes, we need never tell the user that we defined while-do in this manner! Only the writer of the macros needs to know the complexity involved; the user need not, and should not, care as long as the construction works when he uses it.

{Note Evaluation for Control}

It is usual in a compiler to distinguish at least three "evaluation contexts": value, control, and effect. (See [Wulf], for example.) Evaluation for control occurs in the predicate of an IF, where the point is not so much to produce a data object as simply to decide whether it is true or false. The results of AND, OR, and NOT operations in predicates are "encoded in the program counter". When compiling an AND, OR, or NOT, a flag is passed down indicating whether it is for value or for control; in the latter case, two tags are also passed down, indicating the branch targets for success or failure. (This is called "anchor pointing" in [Allen and Cocke].)

In RABBIT this notion falls out automatically without any special handling, thanks to the definition of AND and OR as macros expanding into IF statements. If we were also to define NOT as a macro

```
(NOT x)  =>  (IF x 'NIL 'T)
```

then nearly all such special "evaluation for control" cases would be handled by virtue of the nested-IF transformation in the optimizer.

One transformation which ought to be in the optimizer is

```
(IF ((LAMBDA (X Y ...) <body>) A B ...) <con> <alt>)
   =>  ((LAMBDA (X Y ...) (IF <body> <con> <alt>)) A B ...)
```

which could be important if the <body> is itself as IF. (This transformation would occur at a point (in the optimizer) where no conflicts between X, Y, ... and variables used in <con> and <alt> could occur.)

{Note Evaluation for Effect}

This is the point where the notion of evaluation for effect is handled (see {Note Evaluation for Control}). It is detected as the special case of evaluation for value where no one refers to the value! This may be construed as the distinction between "statement" and "expression" made in Algol-like languages.

{Note Full-Funarg Example}

As an example of the difference between lexical and dynamic scoping, consider the classic case of the "funarg problem". We have defined a function MAPCAR which, given a function and a list, produces a new list of the results of the function applied to each element of the given list:

```
(DEFINE MAPCAR
        (LAMBDA (FN L)
                (IF (NULL L) NIL
                    (CONS (FN (CAR L)) (MAPCAR FN (CDR L))))))
```

Now suppose in another program we have a list X and a number L, and want to add L to every element of X:

```
(MAPCAR (LAMBDA (Z) (+ Z L)) X)
```

This works correctly in a lexically scoped language such as SCHEME, because the L in the function (LAMBDA (Z) (+ Z L)) refers to the value of L at the point the LAMBDA-expression is evaluated. In a dynamically scoped language, such as standard LISP, the L refers to the most recent run-time binding of L, which is the binding in the definition of MAPCAR (which occurs between the time the LAMBDA-expression is passed to MAPCAR and the time the LAMBDA-expression is invoked).

{Note Generalized LABELS}

Since the dissertation was written, and indeed after [Revised Report] came out, the format of LABELS in SCHEME was generalized to permit labelled functions to be defined using any of the same three formats permitted by DEFINE in [Revised Report]. RABBIT has been updated to reflect this change, and the code for it appears in the Appendix.

{Note Heap-Allocated Contours}

RABBIT maintains heap-allocated environments as a simple chained list of variable values. However, all the variables which are added on at once as a single set may be regarded as a new "contour" in the Algol sense. Such contours could be heap-allocated arrays (vectors), and so an environment would be a chained list of such little arrays. The typical Algol implementation technique using a "display" (a margin array whose elements point at successive elements (contours) of the environment chain) is clearly applicable here. One advantage of the list-of-all-values representation actually used in RABBIT is that null contours automatically add no content to the environment structure, which makes it easier to recognize later, in the code generator, that no environment adjustments are necessary in changing between two environments which differ only by null contours (see the code for ADJUST-KNOWNFN-CENV in the Appendix).

{Note Loop Unrolling}

In the case of a LABELS used to implement a loop, the substitution of a labelled function for the variable which names it would constitute an instance of loop unrolling [Allen and Cocke], particularly if the substitution permitted subsequent optimizations such as eliminating dead code. Here, as elsewhere, a specific optimization technique falls out as a consequence of the more general technique of beta-conversion.

{Note Multiple-Argument Continuations}

One could easily define a SCHEME-like language in which continuations could take more than one argument (that is, functions could return several values); see the discussion in [Declarative]. We have elected not to provide for this in SCHEME and RABBIT.

{Note Non-deterministic CPS Conversion}

As with optimization, so the conversion to continuation-passing style involves decisions which ideally could be made non-deterministically. The decisions made at this level will affect later decisions involving register allocation, etc., which cannot easily be foreseen at this stage.

{Note Non-deterministic Optimization}

To simplify the implementation, RABBIT uses only a deterministic (and very conservative) optimizer. Ideally, an optimizer would be non-deterministic in structure; it could try an optimization, see how the result interacted with other optimizations, and back out if the end result is not as good as desired. We have experimented briefly with the use of the AMORD language [Doyle] to build a non-deterministic compiler, but have no significant results yet.

We can see more clearly the fundamental unity of macros and other optimizations in the light of this hypothetical non-deterministic implementation. Rather than trying to guess ahead of time whether a macro expansion or optimization is desirable, it goes ahead and tries, and then measures the utility of the result. The only difference between a macro and other optimizations is that a macro call is an all-or-nothing situation: if it cannot be expanded for some reason, it is of infinite disutility, while if it can its disutility is finite. This leads to the idea of non-deterministic macro expansions, which we have not pursued.

{Note Non-quoted ASET}

The SCHEME interpreter permits one to compute the name of the variable, but for technical and philosophical reasons RABBIT forbids this. We shall treat "ASET'" as a single syntactic object (think "ASETQ").

Hewitt (private communication) and others have objected that the ASET primitive is "dangerous" in that one cannot predict what variable may be clobbered, and in that it makes one dependent on the representation of variables (since one can "compute up" an arbitrary variable to be set). The first is a valid objection on the basis of programming style or programming philosophy.

(Indeed, on this basis alone it was later decided to remove ASET from the SCHEME language, leaving only ASET' in [Revised Report].) The second is only slightly true; the compiler can treat ASET with an non-quoted first argument as a sort of macro. Let V1, V2, ..., VN be the names of the bound variables accessible to the occurrence of ASET in question. These names are all distinct, for if two are the same, one variable "shadows" another, and so we may omit the one shadowed (and so inaccessible). Then we may write the transformation:

```
(ASET a b)  =>  ((LAMBDA (Q1 Q2)
                         (COND ((EQ Q1 'V1) (ASET' V1 Q2))
                               ((EQ Q1 'V2) (ASET' V2 Q2))
                               ...
                               ((EQ Q1 'VN) (ASET' VN Q2))
                               (T (GLOBAL-SET P Q1 Q2))))
                 a
                 b)
```

This transformation is to be made after the alpha-conversion process, which renames all variables; Q1 and Q2 are two more generated variables guaranteed not to conflict with V1, ..., VN. This expansion makes quite explicit the fact that we are comparing against a list of <u>symbols</u> to decide which <u>variable</u> to modify. The actual run-time representation of variables is not exploited, the one exception being the GLOBAL-SET operator, which raises questions about the meaning of the global environment and the user interface which we are not prepared to answer.

(See also {Note ASET' Is Imperative}.)

{Note Old CPS Algorithm}

We reproduce here Appendix A of [Declarative]:

Here we present a set of functions, written in SCHEME, which convert a SCHEME expression from functional style to pure continuation-passing style. {Note PLASMA CPS}

```
(ASET' GENTEMPNUM 0)

(DEFINE GENTEMP
        (LAMBDA (X)
                (IMPLODE (CONS X (EXPLODEN (ASET' GENTEMPNUM (+ GENTEMPNUM 1)))))))
```

GENTEMP creates a new unique symbol consisting of a given prefix and a unique number.

```
(DEFINE CPS (LAMBDA (SEXPR) (SPRINTER (CPC SEXPR NIL '#CONT#))))
```

CPS (Continuation-Passing Style) is the main function; its argument is the expression to be converted. It calls CPC (C-P Conversion) to do the real work, and then calls SPRINTER to pretty-print the result, for convenience. The symbol #CONT# is used to represent the implied continuation which is to receive the value of the expression.

```
(DEFINE CPC
        (LAMBDA (SEXPR ENV CONT)
                (COND ((ATOM SEXPR) (CPC-ATOM SEXPR ENV CONT))
                      ((EQ (CAR SEXPR) 'QUOTE)
                       (IF CONT "(,CONT ,SEXPR) SEXPR))
                      ((EQ (CAR SEXPR) 'LAMBDA)
                       (CPC-LAMBDA SEXPR ENV CONT))
                      ((EQ (CAR SEXPR) 'IF)
                       (CPC-IF SEXPR ENV CONT))
                      ((EQ (CAR SEXPR) 'CATCH)
                       (CPC-CATCH SEXPR ENV CONT))
                      ((EQ (CAR SEXPR) 'LABELS)
                       (CPC-LABELS SEXPR ENV CONT))
                      ((AND (ATOM (CAR SEXPR))
                            (GET (CAR SEXPR) 'AMACRO))
                       (CPC (FUNCALL (GET (CAR SEXPR) 'AMACRO) SEXPR) ENV CONT))
                      (T (CPC-FORM SEXPR ENV CONT)))))
```

CPC merely dispatches to one of a number of subsidiary routines based on the form of the expression SEXPR. ENV represents the environment in which SEXPR will be evaluated; it is a list of the variable names. When CPS initially calls CPC, ENV is NIL. CONT is the continuation which will receive the value of SEXPR. The double-quote (") is like a single-quote, except that within the quoted expression any subexpressions preceded by comma (,) are evaluated and substituted in (also, any subexpressions preceded by atsign (@) are substituted in a list segments). One special case handled directly by CPC is a quoted expression; CPC also expands any SCHEME macros encountered.

```
(DEFINE CPC-ATOM
        (LAMBDA (SEXPR ENV CONT)
                ((LAMBDA (AT) (IF CONT "(,CONT ,AT) AT))
                 (COND ((NUMBERP SEXPR) SEXPR)
                       ((MEMQ SEXPR ENV) SEXPR)
                       ((GET SEXPR 'CPS-NAME))
                       (T (IMPLODE (CONS '% (EXPLODEN SEXPR))))))))
```

For convenience, CPC-ATOM will change the name of a global atom. Numbers and atoms in the environment are not changed; otherwise, a specified name on the property list of the given atom is used (properties defined below convert "+"

into "++", etc.); otherwise, the name is prefixed with "%". Once the name has been converted, it is converted to a form which invokes the continuation on the atom. (If a null continuation is supplied, the atom itself is returned.)

```
(DEFINE CPC-LAMBDA
        (LAMBDA (SEXPR ENV CONT)
                ((LAMBDA (CN)
                         ((LAMBDA (LX) (IF CONT "(,CONT ,LX) LX))
                          "(LAMBDA (@(CADR SEXPR) ,CN)
                                   ,(CPC (CADDR SEXPR)
                                         (APPEND (CADR SEXPR) (CONS CN ENV))
                                         CN))))
                 (GENTEMP 'C))))
```

A LAMBDA expression must have an additional parameter, the continuation supplied to its body, added to its parameter list. CN holds the name of this generated parameter. A new LAMBDA expression is created, with CN added, and with its body converted in an environment containing the new variables. Then the same test for a null CONT is made as in CPC-ATOM.

```
(DEFINE CPC-IF
        (LAMBDA (SEXPR ENV CONT)
                ((LAMBDA (KN)
                         "((LAMBDA (,KN)
                                   ,(CPC (CADR SEXPR)
                                         ENV
                                         ((LAMBDA (PN)
                                                  "(LAMBDA (,PN)
                                                           (IF ,PN
                                                               ,(CPC (CADDR SEXPR)
                                                                     ENV
                                                                     KN)
                                                               ,(CPC (CADDDR SEXPR)
                                                                     ENV
                                                                     KN))))
                                          (GENTEMP 'P))))
                           ,CONT))
                 (GENTEMP 'K))))
```

First, the continuation for an IF must be given a name KN (rather, the name held in KN; but for convenience, we will continue to use this ambiguity, for the form of the name is indeed Kn for some number n), for it will be referred to in two places and we wish to avoid duplicating the code. Then, the predicate is converted to continuation-passing style, using a continuation which will receive the result and call it PN. This continuation will then use an IF to decide which converted consequent to invoke. Each consequent is converted using continuation KN.

```
(DEFINE CPC-CATCH
        (LAMBDA (SEXPR ENV CONT)
                ((LAMBDA (EN)
                         "((LAMBDA (,EN)
                                   ((LAMBDA (,(CADR SEXPR))
                                            ,(CPC (CADDR SEXPR)
                                                  (CONS (CADR SEXPR) ENV)
                                                  EN))
                                    (LAMBDA (V C) (,EN V))))
                           ,CONT))
                 (GENTEMP 'E))))
```

This routine handles CATCH as defined in [Sussman 75], and in converting it to continuation-passing style eliminates all occurrences of CATCH. The idea is to give the continuation a name EN, and to bind the CATCH variable to a continuation (LAMBDA (V C) ...) which ignores its continuation and instead exits the catch by calling EN with its argument V. The body of the CATCH is converted using continuation EN.

```
(DEFINE CPC-LABELS
        (LAMBDA (SEXPR ENV CONT)
                (DO ((X (CADR SEXPR) (CDR X))
                     (Y ENV (CONS (CAAR X) Y)))
                    ((NULL X)
                     (DO ((W (CADR SEXPR) (CDR W))
                          (Z NIL (CONS (LIST (CAAR W)
                                             (CPC (CADAR W) Y NIL))
                                       Z)))
                         ((NULL W)
                          "(LABELS ,(REVERSE Z)
                                   ,(CPC (CADDR SEXPR) Y CONT))))))))
```

Here we have used DO loops as defined in MacLISP (DO is implemented as a macro in SCHEME). There are two passes, one performed by each DO. The first pass merely collects in Y the names of all the labelled LAMBDA expressions. The second pass converts all the LAMBDA expressions using a null continuation and an environment augmented by all the collected names in Y, collecting them in Z. At the end, a new LABELS is constructed using the results in Z and a converted LABELS body.

```
(DEFINE CPC-FORM
        (LAMBDA (SEXPR ENV CONT)
                (LABELS ((LOOP1
                          (LAMBDA (X Y Z)
                                  (IF (NULL X)
                                      (DO ((F (REVERSE (CONS CONT Y))
                                              (IF (NULL (CAR Z)) F
                                                  (CPC (CAR Z)
                                                       ENV
                                                       `(LAMBDA (,(CAR Y)) ,F))))
                                           (Y Y (CDR Y))
                                           (Z Z (CDR Z)))
                                          ((NULL Z) F))
                                      (COND ((OR (NULL (CAR X))
                                                 (ATOM (CAR X)))
                                             (LOOP1 (CDR X)
                                                    (CONS (CPC (CAR X) ENV NIL) Y)
                                                    (CONS NIL Z)))
                                            ((EQ (CAAR X) 'QUOTE)
                                             (LOOP1 (CDR X)
                                                    (CONS (CAR X) Y)
                                                    (CONS NIL Z)))
                                            ((EQ (CAAR X) 'LAMBDA)
                                             (LOOP1 (CDR X)
                                                    (CONS (CPC (CAR X) ENV NIL) Y)
                                                    (CONS NIL Z)))
                                            (T (LOOP1 (CDR X)
                                                      (CONS (GENTEMP 'T) Y)
                                                      (CONS (CAR X) Z))))))))
                         (LOOP1 SEXPR NIL NIL))))
```

This, the most complicated routine, converts forms (function calls). This also operates in two passes. The first pass, using LOOP1, uses X to step down the expression, collecting data in Y and Z. At each step, if the next element of X can be evaluated trivially, then it is converted with a null continuation and added to Y, and NIL is added to Z. Otherwise, a temporary name TN for the result of the subexpression is created and put in Y, and the subexpression itself is put in Z. On the second pass (the DO loop), the final continuation-passing form is constructed in F from the inside out. At each step, if the element of Z is non-null, a new continuation must be created. (There is actually a bug in CPC-FORM, which has to do with variables affected by side-effects. This is easily fixed by changing LOOP1 so that it generates temporaries for variables even though variables evaluate trivially. This would only obscure the examples presented below, however, and so this was omitted.)

```
(LABELS ((BAR
          (LAMBDA (DUMMY X Y)
                  (IF (NULL X) '|CPS ready to go!|
                      (BAR (PUTPROP (CAR X) (CAR Y) 'CPS-NAME)
                           (CDR X)
                           (CDR Y))))))
        (BAR NIL
             '(+  -  *  //   ^  T  NIL)
             '(++ -- ** //// ^^ 'T 'NIL)))
```

This loop sets up some properties so that "+" will translate into "++" instead of "%+", etc.

Now let us examine some examples of the action of CPS. First, let us try our old friend FACT, the iterative factorial program.

```
(DEFINE FACT
        (LAMBDA (N)
                (LABELS ((FACT1 (LAMBDA (M A)
                                        (IF (= M 0) A
                                            (FACT1 (- M 1) (* M A))))))
                        (FACT1 N 1))))
```

Applying CPS to the LAMBDA expression for FACT yields:

```
(#CONT#
    (LAMBDA (N C7)
            (LABELS ((FACT1
                      (LAMBDA (M A C10)
                          ((LAMBDA (K11)
                                   (%= M 0
                                       (LAMBDA (P12)
                                           (IF P12 (K11 A)
                                               (-- M 1
                                                   (LAMBDA (T13)
                                                       (** M A
                                                           (LAMBDA (T14)
                                                               (FACT1 T13 T14 K11)))))))))
                              C10))))
                    (FACT1 N 1 C7))))
```

As an example of CATCH elimination, here is a routine which is a paraphrase of the SQRT routine from [Sussman 75]:

```
(DEFINE SQRT
        (LAMBDA (X EPS)
                ((LAMBDA (ANS LOOPTAG)
                         (CATCH RETURNTAG
                                (BLOCK (ASET' LOOPTAG (CATCH M M))
                                       (IF ---
                                           (RETURNTAG ANS)
                                           NIL)
                                       (ASET' ANS ===)
                                       (LOOPTAG LOOPTAG))))
                 1.0
                 NIL)))
```

Here we have used "---" and "===" as ellipses for complicated (and relatively uninteresting) arithmetic expressions. Applying CPS to the LAMBDA expression for SQRT yields:

```
(#CONT#
 (LAMBDA (X EPS C33)
     ((LAMBDA (ANS LOOPTAG C34)
          ((LAMBDA (E35)
               ((LAMBDA (RETURNTAG)
                    ((LAMBDA (E52)
                         ((LAMBDA (M) (E52 M))
                          (LAMBDA (V C) (E52 V))))
                     (LAMBDA (T51)
                             (%ASET' LOOPTAG T51
                                    (LAMBDA (T37)
                                        ((LAMBDA (A B C36) (B C36))
                                         T37
                                         (LAMBDA (C40)
                                            ((LAMBDA (K47)
                                                 ((LAMBDA (P50)
                                                      (IF P50
                                                          (RETURNTAG ANS K47)
                                                          (K47 'NIL)))
                                                  %---))
                                             (LAMBDA (T42)
                                                 ((LAMBDA (A B C41) (B C41))
                                                  T42
                                                  (LAMBDA (C43)
                                                      (%ASET' ANS %===
                                                            (LAMBDA (T45)
                                                                ((LAMBDA (A B C44)
                                                                         (B C44))
                                                                 T45
                                                                 (LAMBDA (C46)
                                                                     (LOOPTAG
                                                                      LOOPTAG
                                                                      C46))
                                                                 C43))))
                                                  C40))))
                                         E35))))))
                 (LAMBDA (V C) (E35 V))))
           C34))
      1.0
      'NIL
      C33)))
```

Note that the CATCHes have both been eliminated. It is left as an exercise for the reader to verify that the continuation-passing version correctly reflects the semantics of the original.

{Note Operations on Functions}

It would certainly be possible to define other operations on functions, such as determining the number of arguments required, or the types of the arguments and returned value, etc. (Indeed, after the dissertation was written, it was decided to include such an operator PROCP in [Revised Report].) The point is that functions need not conform to a specific representation such as S-expressions. At a low level, it may be useful to think of invocation as a generic operator which dispatches on the particular representation and invokes the function in an appropriate manner. Similarly, a debugging package might need to be able to distinguish the various representations. At the user level, however, it is perhaps best to hide this issue, and answer a type inquiry with merely "function".

{Note Refinement of RABBIT}

Since the original dissertation was written I have continued to refine and improve RABBIT. This effort has included a complete rewriting of the optimizer to make it more efficienct and at the same time more lucid. It also included accommodation of changes to SCHEME as documented in [Revised Report]. This work has spanned perhaps eight months' time, because the availability of computer time restricted me to testing RABBIT only once or twice a night. Thus, the actual time expended for the improvements was much less than ten hours a week.

{Note Side-Effect Classifications}

The division of side-effects into classes in RABBIT was not really necessary to the primary goals of RABBIT, but was undertaken as an interesting experiment for our own edification. One could easily imagine a more complex taxonomy. A case of particular interest not handled by RABBIT is dividing the ASET side-effect into ASET of each particular variable; thus an ASET on FOO would not affect a reference to the variable BAR. This could have been done in an ad hoc manner, but we are interested in a more general method dealing only with sets of effects and affectabilities.

{Note Subroutinization}

We have not said anything about how to locate candidate expressions for subroutinization. For examples of appropriate strategies, see [Geschke] and [Aho, Johnson, and Ullman]. Our point here is that SCHEME, thanks to the property of lexical scoping and the ability to write "anonymous" functions as LAMBDA-expressions, provides an ideal way to represent the result of such transformations.

{Note Tail-Recursive OR}

Since the dissertation was written, the SCHEME language was redefined in [Revised Report] to prescribe a "tail-recursive" interpretation for the last form in an AND or OR. This requirement necessitated a redefinition of OR which is in fact dual to the definition of AND.

References

[Aho, Johnson, and Ullman]
Aho, A.V., Johnson, S.C., and Ullman, J.D. "Code Generation for Expressions with Common Subexpressions." J. ACM 24, 1 (January 1977), 146-160.

[Allen and Cocke]
Allen, Frances E., and Cocke, John. "A Catalogue of Optimizing Transformations." In Rustin, Randall (ed.), Design and Optimization of Compilers. Proc. Courant Comp. Sci. Symp. 5. Prentice-Hall (Englewood Cliffs, N.J., 1972).

[Bobrow and Wegbreit]
Bobrow, Daniel G. and Wegbreit, Ben. "A Model and Stack Implementation of Multiple Environments." CACM 16, 10 (October 1973) pp. 591-603.

[Carter]
Carter, J. Lawrence. "A Case Study of a New Code Generation Technique for Compilers." Comm. ACM 20, 12 (December 1977), 914-920.

[Church]
Church, Alonzo. The Calculi of Lambda Conversion. Annals of Mathematics Studies Number 6. Princeton University Press (Princeton, 1941). Reprinted by Klaus Reprint Corp. (New York, 1965).

[Coleman]
Coleman, Samuel S. JANUS: A Universal Intermediate Language. Ph.D. thesis. University of Colorado (1974).

[DEC]
Digital Equipment Corporation. DecSystem 10 Assembly Language Handbook (third edition). (Maynard, Mass., 1973).

[Declarative]
Steele, Guy Lewis Jr. LAMBDA: The Ultimate Declarative. AI Memo 379. MIT AI Lab (Cambridge, November 1976).

[Dijkstra]
Dijkstra, Edsger W. A Discipline of Programming. Prentice-Hall (Englewood Cliffs, N.J., 1976).

[Doyle]
Doyle, Jon, de Kleer, Johan, Sussman, Gerald Jay, and Steele, Guy L. Jr. "AMORD: A Dependency-Based Problem-Solving Language." Submitted to the 1977 SIGART/SIGPLAN Artificial Intelligence and Programming Languages Conference.

[Geschke]
Geschke, Charles M. Global Program Optimizations. Ph.D. thesis. Carnegie-Mellon University (Pittsburgh, October 1972).

[Gries]
Gries, David. Compiler Construction for Digital Computers. John Wiley & Sons (New York, 1971), 252-257.

[Hewitt]
Hewitt, Carl. "Viewing Control Structures as Patterns of Passing Messages." AI Journal 8, 3 (June 1977), 323-364.

[Hewitt and Smith]
Hewitt, Carl, and Smith, Brian. "Towards a Programming Apprentice." IEEE Transactions on Software Engineering SE-1, 1 (March 1975), 26-45.

[Imperative]
Steele, Guy Lewis Jr., and Sussman, Gerald Jay. LAMBDA: The Ultimate Imperative. AI Memo 353. MIT AI Lab (Cambridge, March 1976).

[Johnsson]
Johnsson, Richard Karl. An Approach to Global Register Allocation. Ph.D. Thesis. Carnegie-Mellon University (Pittsburgh, December 1975).

[Landin]
Landin, Peter J. "A Correspondence between ALGOL 60 and Church's Lambda-Notation." CACM 8, 2-3 (February and March 1965).

[LISP1.5M]
McCarthy, John, et al. LISP 1.5 Programmer's Manual. The MIT Press (Cambridge, 1962).

[McKeeman]
McKeeman, W.M. "Peephole optimization." CACM 8, 7 (July 1965), 443-444.

[Moon]
Moon, David A. MACLISP Reference Manual, Revision 0. Project MAC, MIT (Cambridge, April 1974).

[Moses]
Moses, Joel. The Function of FUNCTION in LISP. AI Memo 199, MIT AI Lab (Cambridge, June 1970).

[Pratt]
Pratt, Vaughan R. CGOL: an Alternative External Representation for LISP Users. Working Paper 121. MIT AI Lab (Cambridge, March 1976).

[Revised Report]
Steele, Guy Lewis Jr., and Sussman, Gerald Jay. The Revised Report on SCHEME. MIT AI Memo 452 (Cambridge, January 1978).

[Reynolds]
Reynolds, John C. "Definitional Interpreters for Higher Order Programming Languages." ACM Conference Proceedings 1972.

[Sammet]
Sammet, Jean E. Programming Languages: History and Fundamentals. Prentice-Hall (Englewood Cliffs, N.J., 1969), 708-709.

[SCHEME]
Sussman, Gerald Jay, and Steele, Guy Lewis Jr. SCHEME: An Interpreter for Extended Lambda Calculus. AI Memo 349. MIT AI Lab (Cambridge, December 1975).

[Smith and Hewitt]
Smith, Brian C. and Hewitt, Carl. A PLASMA Primer (draft). MIT AI Lab (Cambridge, October 1975).

[Standish]
Standish, T.A., et al. The Irvine Program Transformation Catalogue. University of California (Irvine, January 1976).

[Steele]
Steele, Guy Lewis Jr. "Debunking the 'Expensive Procedure Call' Myth." Proc. ACM National Conference (Seattle, October 1977),153-162. Revised as MIT AI Memo 443 (Cambridge, October 1977).

[Stoy]
Stoy, Joseph E. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press (Cambridge, 1977).

[Teitelman]
Teitelman, Warren. InterLISP Reference Manual. Revised edition. Xerox Palo Alto Research Center (Palo Alto, 1975).

[Wand and Friedman]
Wand, Mitchell, and Friedman, Daniel P. Compiling Lambda Expressions Using Continuations. Technical Report 55. Indiana University (Bloomington, October 1976).

[Wulf]
Wulf, William A., et al. The Design of an Optimizing Compiler. American Elsevier (New York, 1975).

Appendix

We present here the complete working source code for RABBIT, written in SCHEME. (The listing of the code was produced by the "@" listing generator, written by Richard M. Stallman, Guy L. Steele Jr., and other contributors.)

The code is presented on successive odd-numbered pages. Commentary on the code is on the facing even-numbered page. An index appears at the end of the listing, indicating where each function is defined.

It should be emphasized that RABBIT was not written with efficiency as a particular goal. Rather, the uppermost goals were clarity, ease of debugging, and adaptability to changing algorithms during the development process Much information is generated, never used by the compilation process, and then thrown away, simply so that if some malfunction should occur it would be easier to conduct a post-mortem analysis. Information that is used for compilation is often retained longer than necessary. The overall approach is to create a big data structure and then, step by step, fill in slots, never throwing anything away, even though it may no longer be needed.

The algorithms could be increased in speed, particularly the optimizer, which often recomputes information needlessly. Determining whether or not the recomputation was necessary would have cluttered up the algorithms, however, making them harder to read and to modify, and so this was omitted. Similarly, certain improvements could dramatically decrease the space used. The larger functions in RABBIT can just barely be compiled with a memory size of 256K words on a PDP-10. However, it was deemed worthwhile to keep the extra information available for as long a time as possible.

The implementation of RABBIT has taken perhaps three man-months. This includes throwing away the original optimizer and rewriting it completely, and accomodating certain changes to the SCHEME language as they occurred. RABBIT was operational, without the optimizer, after about one man-month's work. The dissertation was written after the first version of the optimizer was demonstrated to work. The remaining time was spent analyzing the faults of the first optimizer, writing the second version, accomodating language changes, making performance measurements, and testing RABBIT on programs other than RABBIT itself.

The main modules of RABBIT are organized something like this:

```
COMFILE, TRANSDUCE, PROCESS-FORM   (Bookkeeping and file handling)
    COMPILE                            (Compile a function definition)
        ALPHATIZE                          (Convert input, rename variables)
            MACRO-EXPAND                       (Expand macro forms)
        META-EVALUATE                      (Source-to-source optimizations)
            PASS1-ANALYZE                      (Preliminary code analysis)
                ENV-ANALYZE                        (Environment analysis)
                TRIV-ANALYZE                       (Triviality analysis)
                EFFS-ANALYZE                       (Side effects analysis)
            META-IF-FUDGE                      (Transform nested IF expressions)
            META-COMBINATION-TRIVFN            (Constants folding)
            META-COMBINATION-LAMBDA            (Beta-conversion)
                SUBST-CANDIDATE                    (Substitution feasibility)
                META-SUBSTITUTE                    (Substitution, subsumption)
        CONVERT                            (Convert to continuation-passing style)
        CENV-ANALYZE                       (Environment analysis)
        BIND-ANALYZE                       (Bindings analysis)
        DEPTH-ANALYZE                      (Register allocation)
        CLOSE-ANALYZE                      (Environment structure design)
        COMPILATE-ONE-FUNCTION             (Generate code, producing one module)
            COMPILATE                          (Generate code for one subroutine)
                COMP-BODY                          (Compile procedure body)
                ANALYZE                            (Generate value-producing code)
                TRIV-ANALYZE                       (Generate "trivial" code)
```

```
AI:QUUX;RABBIT 568    GLS    12:14:50 Monday, May 15, 1978   FQ+1D.2H.39M.4S.
Created 23:29:15 Sunday, May 14, 1978   FQ+13H.53M.29S.

RRRRRRRR          AA        BBBBBBBB      BBBBBBBB        IIIIII      TTTTTTTTTT
RRRRRRRR          AA        BBBBBBBB      BBBBBBBB        IIIIII      TTTTTTTTTT
RRRRRRRR          AA        BBBBBBBB      BBBBBBBB        IIIIII      TTTTTTTTTT
RR      RR      AA  AA      BB      BB    BB      BB        II            TT
RR      RR      AA  AA      BB      BB    BB      BB        II            TT
RR      RR      AA  AA      BB      BB    BB      BB        II            TT
RR      RR    AA      AA    BB      BB    BB      BB        II            TT
RR      RR    AA      AA    BB      BB    BB      BB        II            TT
RR      RR    AA      AA    BB      BB    BB      BB        II            TT
RRRRRRRR      AA      AA    BBBBBBBB      BBBBBBBB          II            TT
RRRRRRRR      AA      AA    BBBBBBBB      BBBBBBBB          II            TT
RRRRRRRR      AA      AA    BBBBBBBB      BBBBBBBB          II            TT
RR  RR        AAAAAAAAAA    BB      BB    BB      BB        II            TT
RR  RR        AAAAAAAAAA    BB      BB    BB      BB        II            TT
RR  RR        AAAAAAAAAA    BB      BB    BB      BB        II            TT
RR    RR      AA      AA    BB      BB    BB      BB        II            TT
RR    RR      AA      AA    BB      BB    BB      BB        II            TT
RR    RR      AA      AA    BB      BB    BB      BB        II            TT
RR      RR    AA      AA    BBBBBBBB      BBBBBBBB        IIIIII          TT
RR      RR    AA      AA    BBBBBBBB      BBBBBBBB        IIIIII          TT
RR      RR    AA      AA    BBBBBBBB      BBBBBBBB        IIIIII          TT


AI:QUUX;RABBIT 568    GLS    12:14:50 Monday, May 15, 1978   FQ+1D.2H.39M.4S.
Created 23:29:15 Sunday, May 14, 1978   FQ+13H.53M.29S.

5555555555      666666        888888
5555555555      666666        888888
5555555555      666666        888888
55            66      66    88      88
55            66      66    88      88
55            66      66    88      88
55            66            88      88
55            66            88      88
55            66            88      88
55555555      66666666        888888
55555555      66666666        888888
55555555      66666666        888888
        55    66      66    88      88
        55    66      66    88      88
        55    66      66    88      88
55      55    66      66    88      88
55      55    66      66    88      88
55      55    66      66    88      88
  555555        666666        888888
  555555        666666        888888
  555555        666666        888888

Switch Settings: L[LISP] % A N 69V 110W X
Fonts: F[FONTS;22FG KST,,]
```

The DECLARE forms are for the benefit of the MacLISP compiler, which will process the result of compiling this file (i.e. RABBIT compiling itself). The first few forms are concerned with switch settings, allocation of memory within the MacLISP compiler, and loading of auxiliary functions which must be available at compile time.

The large block of SPECIAL declarations contains the name of every SCHEME function in the file. This is necessary because the run-time representation of a global variable is as a MacLISP SPECIAL variable. The compiled function objects will reside in MacLISP value cells, and SCHEME functions refer to each other through these cells.

The second set of SPECIAL declarations (variables whose names begin and end with a "*") specify variables used globally by RABBIT. These fall into three categories: variables containing properties of the SCHEME interpreter which are parameters for the compiler (e.g. **ARGUMENT-REGISTERS**); switches, primarily for debugging purposes, used to control certain compiler operations (e.g. *FUDGE*); and own variables for certain functions, used to generate objects or gather statistics (e.g. *GENTEMPNUM* and *DEPROGNIFY-COUNT*).

The PROCLAIM forms are to RABBIT as DECLARE forms are to the MacLISP compiler. These provide declarations to the incarnation of RABBIT which is compiling the file. The subforms of a PROCLAIM form are executed by RABBIT when it encounters the form in a file being compiled. (We will see later how this is done.)

```
;;; RABBIT COMPILER   -*-LISP-*-

(DECLARE (FASLOAD (QUUX) SCHMAC))
(DECLARE (MACROS T) (NEWIO T))
(DECLARE (ALLOC '(LIST (300000 450000 .2) FIXNUM 50000 SYMBOL 24000)))
(DECLARE (DEFUN DISPLACE (X Y) Y))

(DECLARE (SPECIAL EMPTY TRIVFN GENTEMP GENFLUSH GEN-GLOBAL-NAME PRINT-WARNING ADDPROP DELPROP SETPROP
                  ADJOIN UNION INTERSECT REMOVE SETDIFF PAIRLIS COMPILE PASS1-ANALYZE TEST-COMPILE
                  NODIFY ALPHATIZE ALPHA-ATOM ALPHA-LAMBDA ALPHA-IF ALPHA-ASET ALPHA-CATCH
                  ALPHA-LABELS ALPHA-LABELS-DEFN ALPHA-BLOCK MACRO-EXPAND ALPHA-COMBINATION
                  ENV-ANALYZE TRIV-ANALYZE TRIV-ANALYZE-FN-P EFFS-ANALYZE EFFS-UNION EFFS-ANALYZE-IF
                  EFFS-ANALYZE-COMBINATION CHECK-COMBINATION-PEFFS ERASE-NODES META-EVALUATE
                  META-IF-FUDGE META-COMBINATION-TRIVFN META-COMBINATION-LAMBDA SUBST-CANDIDATE
                  REANALYZE1 EFFS-INTERSECT EFFECTLESS EFFECTLESS-EXCEPT-CONS PASSABLE
                  META-SUBSTITUTE COPY-CODE COPY-NODES CNODIFY CONVERT MAKE-RETURN CONVERT-LAMBDA-FM
                  CONVERT-IF CONVERT-ASET CONVERT-CATCH CONVERT-LABELS CONVERT-COMBINATION
                  CENV-ANALYZE CENV-TRIV-ANALYZE CENV-CCOMBINATION-ANALYZE BIND-ANALYZE REFD-VARS
                  BIND-ANALYZE-CLAMBDA BIND-ANALYZE-CONTINUATION BIND-ANALYZE-CIF BIND-ANALYZE-CASET
                  BIND-ANALYZE-CLABELS BIND-ANALYZE-RETURN BIND-ANALYZE-CCOMBINATION
                  BIND-CCOMBINATION-ANALYZE DEPTH-ANALYZE FILTER-CLOSEREFS CLOSE-ANALYZE COMPILATE
                  DEPROGNIFY1 TEMPLOC ENVCARCDR REGSLIST SET-UP-ASETVARS COMP-BODY PRODUCE-IF
                  PRODUCE-ASET PRODUCE-LABELS PRODUCE-LAMBDA-COMBINATION PRODUCE-TRIVFN-COMBINATION
                  PRODUCE-TRIVFN-COMBINATION-CONTINUATION PRODUCE-TRIVFN-COMBINATION-CVARIABLE
                  PRODUCE-COMBINATION PRODUCE-COMBINATION-VARIABLE ADJUST-KNOWNFN-CENV
                  PRODUCE-CONTINUATION-RETURN PRODUCE-RETURN PRODUCE-RETURN-1 LAMBDACATE PSETQIFY
                  PSETQIFY-METHOD-2 PSETQIFY-METHOD-3 PSETQ-ARGS PSETQ-ARGS-ENV PSETQ-TEMPS
                  MAPANALYZE ANALYZE ANALYZE-CLAMBDA ANALYZE-CONTINUATION ANALYZE-CIF ANALYZE-CLABELS
                  ANALYZE-CCOMBINATION ANALYZE-RETURN LOOKUPICATE CONS-CLOSEREFS OUTPUT-ASET
                  CONDICATE DECARCDRATE TRIVIALIZE TRIV-LAMBDACATE COMPILATE-ONE-FUNCTION
                  COMPILATE-LOOP USED-TEMPLOCS REMARK-ON MAP-USER-NAMES COMFILE TRANSDUCE
                  PROCESS-FORM PROCESS-DEFINE-FORM PROCESS-DEFINITION CLEANUP SEXPRFY CSEXPRFY
                  CHECK-NUMBER-OF-ARGS DUMPIT STATS RESET-STATS INIT-RABBIT))

(DECLARE (SPECIAL *EMPTY* *GENTEMPNUM* *GENTEMPLIST* *GLOBAL-GEN-PREFIX* *ERROR-COUNT* *ERROR-LIST*
                  *TEST* *TESTING* *OPTIMIZE* *REANALYZE* *SUBSTITUTE* *FUDGE* *NEW-FUDGE*
                  *SINGLE-SUBST* *LAMBDA-SUBST* *FLUSH-ARGS* *STAT-VARS* *DEAD-COUNT* *FUDGE-COUNT*
                  *FOLD-COUNT* *FLUSH-COUNT* *CONVERT-COUNT* *SUBST-COUNT* *DEPROGNIFY-COUNT*
                  *LAMBDA-BODY-SUBST* *LAMBDA-BODY-SUBST-TRY-COUNT* *LAMBDA-BODY-SUBST-SUCCESS-COUNT*
                  *CHECK-PEFFS* **CONT+ARG-REGS** **ENV+CONT+ARG-REGS** **ARGUMENT-REGISTERS**
                  **NUMBER-OF-ARG-REGS** *BUFFER-RANDOM-FORMS* *DISPLACE-SW*))

(PROCLAIM (*EXPR PRINT-SHORT)
          (SET' *BUFFER-RANDOM-FORMS* NIL)
          (ALLOC '(LIST (240000 340000 1000) FIXNUM (30000 40000 1000)
                   SYMBOL (14000 24000 NIL) HUNK4 (20000 53000 NIL)
                   HUNK8 (20000 50000 NIL) HUNK16 (20000 60000 NIL))))

(SET' *STAT-VARS* '(*DEAD-COUNT* *FUDGE-COUNT* *FOLD-COUNT* *FLUSH-COUNT* *CONVERT-COUNT*
                    *SUBST-COUNT* *DEPROGNIFY-COUNT* *LAMBDA-BODY-SUBST-TRY-COUNT*
                    *LAMBDA-BODY-SUBST-SUCCESS-COUNT*))

(ALLOC '(LIST (240000 340000 1000) FIXNUM (30000 40000 1000)
         SYMBOL (14000 24000 NIL) HUNK4 (20000 50000 NIL)
         HUNK8 (20000 50000 NIL) HUNK16 (20000 70000 NIL)))

(APPLY 'GCTWA '(T))            ;GC USELESS ATOMS (CAN'T SAY (EVAL' (GCTWA T)) BECAUSE OF NCOMPLR)
(REPLACE)                      ;UNDO ANY DISPLACED MACROS
(SET' *DISPLACE-SW* NIL)       ;DON'T LET MACROS SELF-DISPLACE
(GRINDEF)                      ;LOAD THE GRINDER (PRETTY-PRINTER)

(DECLARE (/@DEFINE DEFINE |SCHEME FUNCTION|))        ;DECLARATIONS FOR LISTING PROGRAM
(DECLARE (/@DEFINE DEFMAC |MACLISP MACRO|))
(DECLARE (/@DEFINE SCHMAC |PDP-10 SCHEME MACRO|))
(DECLARE (/@DEFINE MACRO |SCHEME MACRO|))
```

The variable *EMPTY* is initialized to a unique object (a list cell whose car is *EMPTY* -- this is so that no other object can be EQ to it, but it can be easily recognized when printed) which is used to initialize components of structures. (We will see later how such structures are defined.) We do not use, say, NIL to represent an empty component because NIL might be a meaningful value for that component. The predicate EMPTY is true of the unique object.

TRIVFN is a predicate which is true of "trivial" functions. A function is trivial if it is a MacLISP primitive (an EXPR, SUBR, or LSUBR), or has been declared to be primitive via a *EXPR or *LEXPR proclamation.

(INCREMENT FOO) expands into the code (ASET' FOO (+ FOO 1)).

CATENATE is a utility macro which may be thought of as a function. Given any number of S-expressions it produces an atomic symbol whose print name is the concatenation of the print names of the S-expressions. Usually the S-expressions will be atomic symbols or numbers.

```
(CATENATE 'FOO '- 43)  =>  FOO-43
```

GENTEMP is used to generate a new unique symbol, given a specified prefix. The global variable *GENTEMPNUM* starts at zero and increases monotonicially. Each call to GENTEMP catenates the prefix, a hyphen, and a new value of *GENTEMPNUM*. Because the numeric suffixes of the generated symbols increase with time, one can determine in which order symbols were generated. We also will use different prefixes for different purposes, so that one can tell which part of the compiler generated a given symbol. This information can be invaluable for debugging purposes; from the names of the symbols appearing in a data structure, one can determine how that structure was created and in what order. (The generated symbols are themselves used primarily as simple markers, or as simple structures (property lists). The use of the print names amounts to tagging each marker or structure with a type and a creation timestamp. A LISP-like language encourages the inclusion of such information.)

```
(GENTEMP 'NODE)  =>  NODE-2534
```

A list of all generated symbols is maintained in *GENTEMPLIST*. GENFLUSH can be called to excise all generated symbols from the MacLISP obarray; this is periodically necessary when compiling a large file so that unneeded symbols may be garbage-collected. The symbols are initially interned on the obarray in the first place for ease of debugging (one can refer to them by name from a debugging breakpoint). GEN-GLOBAL-NAME is used to generate a symbol to be used as a run-time name by the compiled code. The prefix for such names is initially "?" for testing purposes, but is initialized by the file transducer as a function of the name of the file being compiled. This allows separately compiled files to be loaded together without fear of naming conflicts.

```
(COND ((NOT (BOUNDP '*EMPTY*))
       (SET' *EMPTY* (LIST '*EMPTY*))))

(DEFINE EMPTY
        (LAMBDA (X) (EQ X *EMPTY*)))


(DEFINE TRIVFN
        (LAMBDA (SYM)
                (GETL SYM '(EXPR SUBR LSUBR *EXPR *LEXPR))))


(DEFMAC INCREMENT (X) "(ASET' ,X (+ ,X 1)))

(DEFMAC CATENATE ARGS
       "(IMPLODE (APPEND @(MAPCAR '(LAMBDA (X)
                                    (COND ((OR (ATOM X) (NOT (EQ (CAR X) 'QUOTE)))
                                           "(EXPLODEN ,X))
                                          (T "(QUOTE ,(EXPLODEN (CADR X))))))
                                  ARGS))))


(COND ((NOT (BOUNDP '*GENTEMPNUM*))
       (SET' *GENTEMPNUM* 0)))

(COND ((NOT (BOUNDP '*GENTEMPLIST*))
       (SET' *GENTEMPLIST* NIL)))

(DEFINE GENTEMP
        (LAMBDA (X)
                (BLOCK (INCREMENT *GENTEMPNUM*)
                       (LET ((SYM (CATENATE X '|-| *GENTEMPNUM*)))
                            (ASET' *GENTEMPLIST* (CONS SYM *GENTEMPLIST*)) SYM))))

(DEFINE GENFLUSH
        (LAMBDA ()
                (BLOCK (AMAPC REMOB *GENTEMPLIST*)
                       (ASET' *GENTEMPLIST* NIL))))

(DEFINE GEN-GLOBAL-NAME
        (LAMBDA () (GENTEMP *GLOBAL-GEN-PREFIX*)))

(SET' *GLOBAL-GEN-PREFIX* '|?|)
```

WARN is a macro used to print a notice concerning an incorrect program being compiled. It generates a call to PRINT-WARNING, which maintains a count and a list of the error messages, and prints the message, along with any associated useful quantities.

```
    (WARN |FOO is greater than BAR| FOO BAR)
```

would print (assuming the values of FOO and BAR were 43 and 15)

```
    ;Warning: FOO is greater than BAR
    ; 43
    ; 15
```

WARN is used only to report errors in the program being compiled. The MacLISP ERROR function is used to signal internal inconsistencies in the compiler.

ASK is a macro which prints a message and then waits for a reply. Typically NIL means "no", and anything else means "yes".

SX and CSX are debugging aids which print intermediate data structures internal to the compiler in a readable form. They make use of SPRINTER (part of the MacLISP GRIND pretty-printing package) and of SEXPRFY and CSEXPRFY, which are defined below.

The EQCASE macro provides a simple dispatching control structure. The first form evaluates to an item, and the clause whose keyword matches the item is executed. If no clause matches, an error occurs. For example:

```
    (EQCASE TRAFFIC-LIGHT
            (RED (PRINT 'STOP))
            (GREEN (PRINT 'GO))
            (YELLOW (PRINT 'ACCELERATE) (CRASH)))
```

expands into the code:

```
    (COND ((EQ TRAFFIC-LIGHT 'RED) (PRINT 'STOP))
          ((EQ TRAFFIC-LIGHT 'GREEN) (PRINT 'GO))
          ((EQ TRAFFIC-LIGHT 'YELLOW) (PRINT 'ACCELERATE) (CRASH))
          (T (ERROR '|Losing EQCASE| TRAFFIC-LIGHT 'FAIL-ACT)))
```

```
(DEFMAC WARN (MSG . STUFF)
        "(PRINT-WARNING ',MSG (LIST @STUFF)))

(DEFINE PRINT-WARNING
        (LAMBDA (MSG STUFF)
                (BLOCK (INCREMENT *ERROR-COUNT*)
                       (ASET' *ERROR-LIST* (CONS (CONS MSG STUFF) *ERROR-LIST*))
                       (TYO 7 (SYMEVAL 'TYO))           ;BELL
                       (TERPRI (SYMEVAL 'TYO))
                       (PRINC '|;Warning: | (SYMEVAL 'TYO))
                       (TYO 7 (SYMEVAL 'TYO))           ;BELL
                       (PRINC MSG (SYMEVAL 'TYO))
                       (AMAPC PRINT-SHORT STUFF))))

(DEFUN PRINT-SHORT (X)
       ((LAMBDA (PRINLEVEL PRINLENGTH TERPRI)
                (TERPRI (SYMEVAL 'TYO))
                (PRINC '|; | (SYMEVAL 'TYO))
                (PRIN1 X (SYMEVAL 'TYO)))
        3 8 T))


(SCHMAC ASK (MSG)
        "(BLOCK (TERPRI) (PRINC ',MSG) (TYO 40) (READ)))


(DEFMAC SX (X) "(SPRINTER (SEXPRFY ,X NIL)))            ;DEBUGGING AID
(DEFMAC CSX (X) "(SPRINTER (CSEXPRFY ,X)))              ;DEBUGGING AID


(DEFMAC EQCASE (OBJ . CASES)
        "(COND @(MAPCAR '(LAMBDA (CASE)
                            (OR (ATOM (CAR CASE))
                                (ERROR '|Losing EQCASE clause|))
                            "((EQ ,OBJ ',(CAR CASE)) @(CDR CASE)))
                          CASES)
               (T (ERROR '|Losing EQCASE| ,OBJ 'FAIL-ACT))))
```

The next group of macros implement typed data structures with named components. ACCESSFN, CLOBBER, and HUNKFN allow definition of very general structure access functions. Their precise operation is not directly relevant to this exposition; suffice it to say that they are subsidiary to the DEFTYPE macro on the next page.

DEFTYPE defines structure "data types" with named components. These structures are implemented as MacLISP hunks. (A hunk is essentially a kind of list cell with more than two pointer components; it may be thought of as a short, fixed-length vector. Hunks are accessed with the function (CXR n hunk), which returns the nth component of the hunk. (RPLACX n hunk newval) analogously alters the nth component. CXR and RPLACX are thus similar to CAR/CDR and RPLACA/RPLACD.)

Slot 0 of each hunk is reserved for a "property list"; this feature is not used in RABBIT. Slot 1 always contains an atomic symbol which is the name of the type. Thus every structure explicitly bears its type. The form (HUNKFN TYPE 1) creates a function (actually a macro) called TYPE which when applied to a hunk will fetch slot 1. Slots 2 upward of a hunk are used to contain named components. A structure does <u>not</u> contain the component names. (However, the symbol which is the name of the type does have a list of the component names on its property list. This is useful for debugging purposes. There is, for example, a package which pretty-prints structured data types, showing the components explicitly as name-value pairs, which uses this information.)

```
(DECLARE (/@DEFINE ACCESSFN |ACCESS MACRO|))

(DEFMAC ACCESSFN (NAME UVARS FETCH . PUT)
        ((LAMBDA (VARS CNAME)
                 (DO ((A VARS (CDR A))
                      (B '*Z* "(CDR ,B))
                      (C NIL (CONS "(CAR ,B) C)))
                     ((NULL A)
                      "(PROGN 'COMPILE
                              (DEFMAC ,NAME *Z*
                                      ((LAMBDA ,(NREVERSE (CDR (REVERSE VARS)))
                                                ,FETCH)
                                       @(REVERSE (CDR C))))
                              (DEFMAC ,CNAME *Z*
                                      ((LAMBDA ,VARS
                                                ,(COND (PUT (CAR PUT))
                                                       (T ""(CLOBBER ,,FETCH
                                                                     ,THE-NEW-VALUE))))
                                       @(REVERSE C)))))))
         (COND (PUT UVARS)
               (T (APPEND UVARS '(THE-NEW-VALUE))))
         (CATENATE '|CLOBBER-| NAME)))

(DEFMAC CLOBBER (X Y)
        "(,(CATENATE '|CLOBBER-| (CAR X)) @(CDR X) ,Y))

(DECLARE (/@DEFINE HUNKFN |HUNK ACCESS MACRO|))

(DEFMAC HUNKFN (NAME SLOT)
        "(ACCESSFN ,NAME (THE-HUNK NEW-VALUE)
                   "(CXR ,,SLOT ,THE-HUNK)
                   "(RPLACX ,,SLOT ,THE-HUNK ,NEW-VALUE)))
```

Consider for example the form

```
(DEFTYPE LAMBDA (UVARS VARS BODY))
```

This defines a structured data type called LAMBDA with three named components UVARS, VARS, and BODY. It also defines a series of macros for manipulating this data type.

For access, the macros LAMBDA\UVARS, LAMBDA\VARS, and LAMBDA\BODY are defined. These each take a single argument, a data structure of type VARIABLE, and return the appropriate component. (The TYPE function can also be applied to the data object, and will return LAMBDA.)

For construction, a macro CONS-LAMBDA is defined. For example, the form:

```
(CONS-LAMBDA (UVARS = LIST1)
             (VARS = LIST2))
```

would construct a LAMBDA structure with the TYPE, UVARS, VARS, and BODY slots initialized respectively to LAMBDA, the value of LIST1, the value of LIST2, and the "empty object" (recall the EMPTY predicate above). Any component names (possibly none!) may be initialized in a CONS-xxx form, and any components not mentioned will be initialized to the empty object. (The "=" signs are purely syntactic sugar for mnemonic value. They can be omitted.)

For alteration of components, a macro ALTER-LAMBDA is defined. For example, the form

```
(ALTER-LAMBDA FOO
              (UVARS := LIST1)
              (BODY := (LIST A B)))
```

would alter the UVARS and BODY components of the value of FOO (which should be a LAMBDA structure - this is not checked) to be respectively the values of LIST1 and (LIST A B). Any non-zero number of components may be modified by a single ALTER-xxx form. (The ":=" signs are purely syntactic sugar also.)

A great advantage of using these structure definitions is that it is very easy to add or delete components during the development of the program. In particular, when a new component is added to a type, it is not necessary to find all instances of creations of objects of that type; they will simply automatically initialize the new slot to the empty object. Only parts of the program which are relevant to the use of the new component need be changed.

```
(DECLARE (/@DEFINE DEFTYPE |DATA TYPE|))

;;; SLOT 0 IS ALWAYS THE PROPERTY LIST, AND SLOT 1 THE HUNK TYPE.

(HUNKFN TYPE 1)

(DEFMAC DEFTYPE (NAME SLOTS SUPP)
         "(PROGN 'COMPILE
                (DEFMAC ,(CATENATE '|CONS-| NAME) KWDS
                        (PROGN (DO ((K KWDS (CDR K)))
                                   ((NULL K))
                                   (OR ,(COND ((CDR SLOTS) "(MEMQ (CAAR K) ',SLOTS))
                                              (T "(EQ (CAAR K) ',(CAR SLOTS))))
                                       (ERROR ',(CATENATE '|Invalid Keyword Argument to CONS-|
                                                          NAME)
                                              (CAR K)
                                              'FAIL-ACT)))
                               "(HUNK ',',NAME
                                         @(DO ((S ',SLOTS (CDR S))
                                               (X NIL
                                                  (CONS ((LAMBDA (KWD)
                                                                  (COND (KWD (CAR (LAST KWD)))
                                                                        (T '*EMPTY*)))
                                                          (ASSQ (CAR S) KWDS))
                                                         X)))
                                              ((NULL S) (NREVERSE X)))
                                         NIL)))
                (DEFMAC ,(CATENATE '|ALTER-| NAME) (OBJ . KWDS)
                        (PROGN (DO ((K KWDS (CDR K)))
                                   ((NULL K))
                                   (OR ,(COND ((CDR SLOTS) "(MEMQ (CAAR K) ',SLOTS))
                                              (T "(EQ (CAAR K) ',(CAR SLOTS))))
                                       (ERROR ',(CATENATE '|Invalid Keyword Argument to ALTER-|
                                                          NAME)
                                              (CAR K)
                                              'FAIL-ACT)))
                               (DO ((I (+ (LENGTH KWDS) 1) (- I 1))
                                    (VARS NIL (CONS (GENSYM) VARS)))
                                   ((= I 0)
                                   "((LAMBDA ,VARS
                                             ,(BLOCKIFY
                                               (MAPCAR '(LAMBDA (K V)
                                                                "(CLOBBER (,(CATENATE ',NAME
                                                                                       '|\|
                                                                                       (CAR K))
                                                                           (,(CAR VARS)))
                                                                          (,V)))
                                                       KWDS
                                                       (CDR VARS))))
                                     (LAMBDA () ,OBJ)
                                     @(MAPCAR '(LAMBDA (K) "(LAMBDA () ,(CAR (LAST K))))
                                              KWDS))))))
                @(DO ((S SLOTS (CDR S))
                      (N 2 (+ N 1))
                      (X NIL (CONS "(HUNKFN ,(CATENATE NAME '|\| (CAR S))
                                              ,N)
                                   X)))
                     ((NULL S) (NREVERSE X)))
                (DEFPROP ,NAME ,SLOTS COMPONENT-NAMES)
                (DEFPROP ,NAME ,SUPP SUPPRESSED-COMPONENT-NAMES)
                '(TYPE ,NAME DEFINED)))
```

On this page are two groups of utility functions. One group manipulates property lists, and the other manipulates sets of objects represented as lists.

For (ADDPROP SYM VAL PROP), the PROP property of the symbol SYM should be a list of things. The object VAL is added to this list if it is not already a member of the list.

DELPROP performs the inverse of ADDPROP; it removes an object from a list found as the property of a symbol.

(SETPROP SYM VAL PROP) puts the property-value pair PROP,VAL on the property list of SYM; but if SYM already has a PROP property, it is an error unless the new value is the same as (EQ to) the existing one. That is, a redundant SETPROP is permitted, but not a conflicting one.

(ADJOIN ITEM SET) produces a new set SET ∪ {ITEM}.
UNION produces the union of two sets.
INTERSECT produces the intersection of two sets.
(REMOVE ITEM SET) produces a new set SET - {ITEM}.
(SETDIFF SET1 SET2) produces the set SET1 - SET2.

All of the set operations are accomplished non-destructively; that is, the given arguments are not modified. Examples:

```
(ADJOIN 'A '(A B C)) => (A B C)
(ADJOIN 'A '(B C D)) => (A B C D)
(UNION '(A B C) '(B D F)) => (D F A B C)
(INTERSECT '(A B C) '(B D F)) => (B)
(REMOVE 'B '(A B C)) => (A C)
(SETDIFF '(A B C) '(B D F)) => (A C)
```

```
;;; ADD TO A PROPERTY WHICH IS A LIST OF THINGS

(DEFINE ADDPROP
        (LAMBDA (SYM VAL PROP)
                (LET ((L (GET SYM PROP)))
                     (IF (NOT (MEMQ VAL L))
                         (PUTPROP SYM (CONS VAL L) PROP)))))

;;; INVERSE OF ADDPROP

(DEFINE DELPROP
        (LAMBDA (SYM VAL PROP)
               (PUTPROP SYM (DELQ VAL (GET SYM PROP)) PROP)))

;;; LIKE PUTPROP, BUT INSIST ON NOT CHANGING A VALUE ALREADY THERE

(DEFINE SETPROP
        (LAMBDA (SYM VAL PROP)
                (LET ((L (GETL SYM (LIST PROP))))
                     (IF (AND L (NOT (EQ VAL (CADR L))))
                         (ERROR '|Attempt to redefine a unique property|
                                (LIST 'SETPROP SYM VAL PROP)
                                'FAIL-ACT)
                         (PUTPROP SYM VAL PROP)))))

;;; OPERATIONS ON SETS, REPRESENTED AS LISTS

(DEFINE ADJOIN
        (LAMBDA (X S)
                (IF (MEMQ X S) S (CONS X S))))

(DEFINE UNION
        (LAMBDA (X Y)
                (DO ((Z Y (CDR Z))
                     (V X (ADJOIN (CAR Z) V)))
                    ((NULL Z) V))))

(DEFINE INTERSECT
        (LAMBDA (X Y)
                (IF (NULL X)
                    NIL
                    (IF (MEMQ (CAR X) Y)
                        (CONS (CAR X) (INTERSECT (CDR X) Y))
                        (INTERSECT (CDR X) Y)))))

(DEFINE REMOVE
        (LAMBDA (X S)
                (IF (NULL S)
                    S
                    (IF (EQ X (CAR S))
                        (CDR S)
                        ((LAMBDA (Y)
                                 (IF (EQ Y (CDR S)) S
                                     (CONS (CAR S) Y)))
                         (REMOVE X (CDR S)))))))

(DEFINE SETDIFF
        (LAMBDA (X Y)
                (DO ((Z X (CDR Z))
                     (W NIL (IF (MEMQ (CAR Z) Y)
                                W
                                (CONS (CAR Z) W))))
                    ((NULL Z) W))))
```

The PAIRLIS function is similar to, but not identical to, the function of the same name in the LISP 1.5 Manual. The difference is that the pairs of the association list produced are 2-lists rather than single conses. This was done purely so that structures produced by PAIRLIS would be more readable when printed; the ease of debugging was considered worth the additional CONS and access time.

```
    (PAIRLIS '(A B C) '(X Y Z) '((F P) (G Q)))
         =>  ((C Z) (B Y) (A X) (F P) (G Q))
```

The COMPILE function is the main top-level function of the compiler. It is responsible for invoking each phase of the compiler in order. NAME is the name of a function (an atomic symbol), and LAMBDA-EXP the corresponding lambda-expression; these are easily extracted, for example, from a SCHEME DEFINE-form. SEE-CRUD is NIL for normal processing, or T for debugging purposes. OPTIMIZE is a switch controlling whether the optimization phase should be invoked; it can be T, NIL, or MAYBE (meaning to ask the (human) debugger).

The overall flow within COMPILE is as follows: check number of arguments; apply ALPHATIZE to the lambda-expression to produce the pass 1 data structure; optionally optimize this data structure; perform pass 1 analysis; convert the pass 1 data structure to a pass 2 (continuation-passing style) data structure; perform pass 2 analysis; generate code. The value of COMPILE is the MacLISP code produced by the code generator.

PASS1-ANALYZE is a separate function so that it can be used by the optimizer to re-analyze newly created subexpressions.

CL is a debugging utility. (CL FOO) causes the function FOO (which should be defined in the running SCHEME into which the compiler has been loaded) to be compiled. Various debugging facilities, such as SEE-CRUD, are enabled. This is done by using TEST-COMPILE.

```
(DEFINE PAIRLIS
        (LAMBDA (L1 L2 L)
                (DO ((V L1 (CDR V))
                     (U L2 (CDR U))
                     (E L (CONS (LIST (CAR V) (CAR U)) E)))
                    ((NULL V) E))))


(DEFINE COMPILE
        (LAMBDA (NAME LAMBDA-EXP SEE-CRUD OPTIMIZE)
                (BLOCK (CHECK-NUMBER-OF-ARGS NAME
                                             (LENGTH (CADR LAMBDA-EXP))
                                             T)
                       (LET ((ALPHA-VERSION (ALPHATIZE LAMBDA-EXP NIL)))
                            (IF (AND SEE-CRUD (ASK |See alpha-conversion?|))
                                (SX ALPHA-VERSION))
                            (LET ((OPT (IF (EQ OPTIMIZE 'MAYBE)
                                           (ASK |Optimize?|)
                                           OPTIMIZE)))
                                 (LET ((META-VERSION
                                        (IF OPT
                                            (META-EVALUATE ALPHA-VERSION)
                                            (PASS1-ANALYZE ALPHA-VERSION NIL NIL))))
                                      (OR (AND (NULL (NODE\REFS META-VERSION))
                                               (NULL (NODE\ASETS META-VERSION)))
                                          (ERROR '|ENV-ANALYZE lost - COMPILE|
                                                 NAME
                                                 'FAIL-ACT))
                                      (IF (AND SEE-CRUD OPT (ASK |See meta-evaluation?|))
                                          (SX META-VERSION))
                                      (LET ((CPS-VERSION (CONVERT META-VERSION NIL (NOT (NULL OPT)))))
                                           (IF (AND SEE-CRUD (ASK |See CPS-conversion?|))
                                               (CSX CPS-VERSION))
                                           (CENV-ANALYZE CPS-VERSION NIL NIL)
                                           (BIND-ANALYZE CPS-VERSION NIL NIL)
                                           (DEPTH-ANALYZE CPS-VERSION 0)
                                           (CLOSE-ANALYZE CPS-VERSION NIL)
                                           (COMPILATE-ONE-FUNCTION CPS-VERSION NAME))))))))

(DEFINE PASS1-ANALYZE
        (LAMBDA (NODE REDO OPT)
                (BLOCK (ENV-ANALYZE NODE REDO)
                       (TRIV-ANALYZE NODE REDO)
                       (IF OPT (EFFS-ANALYZE NODE REDO))
                       NODE)))


(SCHMAC CL (FNNAME) "(TEST-COMPILE ',FNNAME))

(DEFINE TEST-COMPILE
        (LAMBDA (FNNAME)
                (LET ((FN (GET FNNAME 'SCHEME!FUNCTION)))
                     (COND (FN (ASET' *TESTING* T)
                               (ASET' *TEST* NIL)        ;PURELY TO RELEASE FORMER GARBAGE
                               (ASET' *ERROR-COUNT* 0)
                               (ASET' *ERROR-LIST* NIL)
                               (ASET' *TEST* (COMPILE FNNAME FN T 'MAYBE))
                               (SPRINTER *TEST*)
                               "(,(IF (ZEROP *ERROR-COUNT*) 'NO *ERROR-COUNT*) ERRORS))
                           (T "(,FNNAME NOT DEFINED))))))
```

Here are the structured data types used for the pass 1 intermediate representation. Each piece of the program is represented as a NODE, which has various pieces of information associated with it. The FORM component is a structure of one of the types CONSTANT, VARIABLE, LAMBDA, IF, ASET, CATCH, LABELS, or COMBINATION. This structure holds information specific to a given type of program node, whereas the NODE structure itself holds information which is needed at every node of the program structure. (One may think of the FORM component as a PASCAL record variant.)

The ALPHATIZE routine and its friends take the S-expression definition of a function (a lambda-expression) and make a copy of it using NODE structures. This copy, like the S-expression, is a tree. Subsequent analysis routines will all recur on this tree, passing information up and down the tree, either distributing information from parent node to child nodes, or collating information from child nodes to pass back to parent nodes. Some information must move laterally within the tree, from branch to branch; this is accomplished exclusively by using the property lists of symbols, usually those generated for renamings of variables (since all lateral information is associated with variable references - which is no accident!).

The function NODIFY is used for constructing a node, with certain slots properly initialized. In particular, the METAP slot is initialized to NIL, indicating a node not yet processed by META-EVALUATE; this fact will be used later in the optimizer. A name is generated for the node, and the node is put on the property list of the name. This property is for debugging purposes only; given the name of a node one can get the node easily. The name itself may also be used for another purpose by CONVERT-COMBINATION, to represent the intermediate quantity which is the value of the form represented by the node.

```
;;; ALPHA-CONVERSION

;;; HERE WE RENAME ALL VARIABLES, AND CONVERT THE EXPRESSION TO AN EQUIVALENT TREE-LIKE FORM
;;; WITH EXTRA SLOTS TO BE FILLED IN LATER.  AFTER THIS POINT, THE NEW NAMES ARE USED FOR
;;; VARIABLES, AND THE USER NAMES ARE USED ONLY FOR ERROR MESSAGES AND THE LIKE.  THE TREE-LIKE
;;; FORM WILL BE USED AND AUGMENTED UNTIL IT IS CONVERTED TO CONTINUATION-PASSING STYLE.

;;; WE ALSO FIND ALL USER-NAMED LAMBDA-FORMS AND SET UP APPROPRIATE PROPERTIES.
;;; THE USER CAN NAME A LAMBDA-FORM BY WRITING (LAMBDA (X) BODY NAME).

(DEFTYPE NODE (NAME SEXPR ENV REFS ASETS TRIVP EFFS AFFD PEFFS PAFFD METAP SUBSTP FORM) (SEXPR))
         ;NAME:  A GENSYM WHICH NAMES THE NODE'S VALUE
         ;SEXPR: THE S-EXPRESSION WHICH WAS ALPHATIZED TO MAKE THIS NODE
         ;       (USED ONLY FOR WARNING MESSAGES AND DEBUGGING)
         ;ENV:   THE ENVIRONMENT OF THE NODE (USED ONLY FOR DEBUGGING)
         ;REFS:  ALL VARIABLES BOUND ABOVE AND REFERENCED BELOW OR BY THE NODE
         ;ASETS: ALL LOCAL VARIABLES SEEN IN AN ASET BELOW THIS NODE (A SUBSET OF REFS)
         ;TRIVP: NON-NIL IFF EVALUATION OF THIS NODE IS TRIVIAL
         ;EFFS:  SET OF SIDE EFFECTS POSSIBLY OCCURRING AT THIS NODE OR BELOW
         ;AFFD:  SET OF SIDE EFFECTS WHICH CAN POSSIBLY AFFECT THIS NODE OR BELOW
         ;PEFFS: ABSOLUTELY PROVABLE SET OF EFFS
         ;PAFFD: ABSOLUTELY PROVABLE SET OF AFFD
         ;METAP: NON-NIL IFF THIS NODE HAS BEEN EXAMINED BY THE META-EVALUATOR
         ;SUBSTP:FLAG INDICATING WHETHER META-SUBSTITUTE ACTUALLY MADE A SUBSTITUTION
         ;FORM:  ONE OF THE BELOW TYPES

(DEFTYPE CONSTANT (VALUE))
         ;VALUE: THE S-EXPRESSION VALUE OF THE CONSTANT
(DEFTYPE VARIABLE (VAR GLOBALP))
         ;VAR:   THE NEW UNIQUE NAME FOR THE VARIABLE, GENERATED BY ALPHATIZE.
         ;       THE USER NAME AND OTHER INFORMATION IS ON ITS PROPERTY LIST.
         ;GLOBALP:  NIL UNLESS THE VARIABLE IS GLOBAL (IN WHICH CASE VAR IS THE ACTUAL NAME)
(DEFTYPE LAMBDA (UVARS VARS BODY))
         ;UVARS: THE USER NAMES FOR THE BOUND VARIABLES (STRICTLY FOR DEBUGGING (SEE SEXPRFY))
         ;VARS:  A LIST OF THE GENERATED UNIQUE NAMES FOR THE BOUND VARIABLES
         ;BODY:  THE NODE FOR THE BODY OF THE LAMBDA-EXPRESSION
(DEFTYPE IF (PRED CON ALT))
         ;PRED:  THE NODE FOR THE PREDICATE
         ;CON:   THE NODE FOR THE CONSEQUENT
         ;ALT:   THE NODE FOR THE ALTERNATIVE
(DEFTYPE ASET (VAR BODY GLOBALP))
         ;VAR:   THE GENERATED UNIQUE NAME FOR THE ASET VARIABLE
         ;BODY:  THE NODE FOR THE BODY OF THE ASET
         ;GLOBALP:  NIL UNLESS THE VARIABLE IS GLOBAL (IN WHICH CASE VAR IS THE ACTUAL NAME)
(DEFTYPE CATCH (UVAR VAR BODY))
         ;UVAR:  THE USER NAME FOR THE BOUND VARIABLE (STRICTLY FOR DEBUGGING (SEE SEXPRFY))
         ;VAR:   THE GENERATED UNIQUE NAME FOR THE BOUND VARIABLE
         ;BODY:  THE NODE FOR THE BODY OF THE CATCH
(DEFTYPE LABELS (UFNVARS FNVARS FNDEFS BODY))
         ;UFNVARS:  THE USER NAMES FOR THE BOUND LABELS VARIABLES
         ;FNVARS:  A LIST OF THE GENERATED UNIQUE NAMES FOR THE LABELS VARIABLES
         ;FNDEFS:  A LIST OF THE NODES FOR THE LAMBDA-EXPRESSIONS
         ;BODY:  THE NODE FOR THE BOY OF THE LABELS
(DEFTYPE COMBINATION (ARGS WARNP))
         ;ARGS:  A LIST OF THE NODES FOR THE ARGUMENTS (THE FIRST IS THE FUNCTION)
         ;WARNP: NON-NIL IFF CHECK-COMBINATION-PEFFS HAS DETECTED A CONFLICT IN THIS COMBINATION

(DEFINE NODIFY
        (LAMBDA (FORM SEXPR ENV)
                (LET ((N (CONS-NODE (NAME = (GENTEMP 'NODE))
                                    (FORM = FORM)
                                    (SEXPR = SEXPR)
                                    (ENV = ENV)
                                    (METAP = NIL))))
                     (PUTPROP (NODE\NAME N) N 'NODE)
                     N)))
```

ALPHATIZE takes an S-expression to convert, and an environment. The latter is a list of 2-lists; each 2-list is of the form (user-name new-name). This is used for renaming each variable to a unique name. The unique names are generated within ALPHA-LAMBDA, ALPHA-LABELS, and ALPHA-CATCH, where the variable bindings are encountered. The new name pairings are tacked onto the front of the then-current environment, and the result used as the environment for converting the body.

ALPHATIZE merely does a dispatch on the type of form, to one of the sub-functions for the various types. It also detects forms which are really macro calls, and expands them by calling MACRO-EXPAND, which returns the form to be used in place of the macro call. (BLOCK is handled as a separate special case. In the interpreter, BLOCK is handled specially rather than going through the general MACRO mechanism. This is done purely for speed. Defining BLOCK as a macro in the compiler can confuse the interpreter in which the compiler runs, and so it was decided simply to handle BLOCK as a special case in the compiler also.) ALPHATIZE allows the S-expression to contain already converted code in the form of NODEs; this fact is exploited by the optimizer (see META-IF-FUDGE below), but has no use in the initial conversion.

ALPHA-ATOM creates a CONSTANT structure for numbers and the special symbols NIL and T. Otherwise a VARIABLE structure is created. If the symbol (it better be a symbol!) occurs in the environment, the new-name is used, and otherwise the symbol itself. The slot GLOBALP is set to T iff the symbol was not in the environment.

ALPHA-LAMBDA generates new names for all the bound variables. It then converts its body, after using PAIRLIS to add the user-name/new-name pairs to the environment. The result is used to make a LAMBDA structure. A copy is made of the list of variables in the UVARS slot; it must be copied because later META-COMBINATION-LAMBDA may splice out elements of that list. If so, it will also splice out corresponding members of VARS, but that list was freshly consed by ALPHA-LAMBDA.

```
;;; ON NODE NAMES THESE PROPERTIES ARE CREATED:
;;;     NODE            THE CORRESPONDING NODE

(DEFINE ALPHATIZE
        (LAMBDA (SEXPR ENV)
                (COND ((ATOM SEXPR)
                       (ALPHA-ATOM SEXPR ENV))
                      ((HUNKP SEXPR)
                       (IF (EQ (TYPE SEXPR) 'NODE)
                           SEXPR
                           (ERROR '|Peculiar hunk - ALPHATIZE| SEXPR 'FAIL-ACT)))
                      ((EQ (CAR SEXPR) 'QUOTE)
                       (NODIFY (CONS-CONSTANT (VALUE = (CADR SEXPR))) SEXPR ENV))
                      ((EQ (CAR SEXPR) 'LAMBDA)
                       (ALPHA-LAMBDA SEXPR ENV))
                      ((EQ (CAR SEXPR) 'IF)
                       (ALPHA-IF SEXPR ENV))
                      ((EQ (CAR SEXPR) 'ASET)
                       (ALPHA-ASET SEXPR ENV))
                      ((EQ (CAR SEXPR) 'CATCH)
                       (ALPHA-CATCH SEXPR ENV))
                      ((EQ (CAR SEXPR) 'LABELS)
                       (ALPHA-LABELS SEXPR ENV))
                      ((EQ (CAR SEXPR) 'BLOCK)
                       (ALPHA-BLOCK SEXPR ENV))
                      ((AND (ATOM (CAR SEXPR))
                            (EQ (GET (CAR SEXPR) 'AINT) 'AMACRO))
                       (ALPHATIZE (MACRO-EXPAND SEXPR) ENV))
                      (T (ALPHA-COMBINATION SEXPR ENV)))))

(DEFINE ALPHA-ATOM
        (LAMBDA (SEXPR ENV)
                (IF (OR (NUMBERP SEXPR) (NULL SEXPR) (EQ SEXPR 'T))
                    (NODIFY (CONS-CONSTANT (VALUE = SEXPR)) SEXPR ENV)
                    (LET ((SLOT (ASSQ SEXPR ENV)))
                         (NODIFY (CONS-VARIABLE (VAR = (IF SLOT (CADR SLOT) SEXPR))
                                                (GLOBALP = (NULL SLOT)))
                                 SEXPR
                                 ENV)))))

(DEFINE ALPHA-LAMBDA
        (LAMBDA (SEXPR ENV)
                (LET ((VARS (DO ((I (LENGTH (CADR SEXPR)) (- I 1))
                                 (V NIL (CONS (GENTEMP 'VAR) V)))
                                ((= I 0) (NREVERSE V)))))
                     (IF (CDDDR SEXPR)
                         (WARN |Malformed LAMBDA expression| SEXPR))
                     (NODIFY (CONS-LAMBDA (UVARS = (APPEND (CADR SEXPR) NIL))
                                          ;;SEE META-COMBINATION-LAMBDA
                                          (VARS = VARS)
                                          (BODY = (ALPHATIZE (CADDR SEXPR)
                                                             (PAIRLIS (CADR SEXPR)
                                                                      VARS
                                                                      ENV))))
                             SEXPR
                             ENV))))
```

ALPHA-IF simply converts the predicate, consequent, and alternative, and makes an IF structure.

ALPHA-ASET checks for a non-quoted first argument. (Presently RABBIT does not allow for computed ASET variables. Since RABBIT was written, such computed variables have in fact been banned from the SCHEME language [Revised Report].) For simplicity, it also does not allow altering a global variable which is the name of a MacLISP primitive. This restriction is related only to the kludginess of the PDP-10 MacLISP SCHEME implementation, and is not an essential problem with the language. The ERROR function was used here rather than WARN because the problems are hard to correct for and occur infrequently. Aside from these difficulties, ALPHA-ASET is much like ALPHA-ATOM on a variable; it looks in the environment, converts the body, and then constructs an ASET structure.

ALPHA-CATCH generates a new name "CATCHVAR-nn" for the bound variable, tacks it onto the environment, and converts the body; it then constructs a CATCH structure.

ALPHA-LABELS generates new names "FNVAR-n" for all the bound variables; it then constructs in LENV the new environment, using PAIRLIS. It then converts all the bound function definitions and the body, using this environment. In this way all the function names are apparent to all the functions. A LABELS structure is then created.

```
(DEFINE ALPHA-IF
        (LAMBDA (SEXPR ENV)
                (NODIFY (CONS-IF (PRED = (ALPHATIZE (CADR SEXPR) ENV))
                                 (CON = (ALPHATIZE (CADDR SEXPR) ENV))
                                 (ALT = (ALPHATIZE (CADDDR SEXPR) ENV)))
                        SEXPR
                        ENV)))

(DEFINE ALPHA-ASET
        (LAMBDA (SEXPR ENV)
                (LET ((VAR (COND ((OR (ATOM (CADR SEXPR))
                                      (NOT (EQ (CAADR SEXPR) 'QUOTE)))
                                  (ERROR '|Can't Compile Non-quoted ASET Variable|
                                         SEXPR
                                         'FAIL-ACT))
                                 (T (CADADR SEXPR)))))
                     (LET ((SLOT (ASSQ VAR ENV)))
                          (IF (AND (NULL SLOT) (TRIVFN VAR))
                              (ERROR '|Illegal to ASET a MacLISP primitive|
                                     SEXPR
                                     'FAIL-ACT))
                          (NODIFY (CONS-ASET (VAR = (IF SLOT (CADR SLOT) VAR))
                                             (GLOBALP = (NULL SLOT))
                                             (BODY = (ALPHATIZE (CADDR SEXPR) ENV)))
                                  SEXPR
                                  ENV)))))

(DEFINE ALPHA-CATCH
        (LAMBDA (SEXPR ENV)
                (LET ((VAR (GENTEMP 'CATCHVAR)))
                     (NODIFY (CONS-CATCH (VAR = VAR)
                                         (UVAR = (CADR SEXPR))
                                         (BODY = (ALPHATIZE (CADDR SEXPR)
                                                            (CONS (LIST (CADR SEXPR) VAR)
                                                                  ENV))))
                             SEXPR
                             ENV))))

(DEFINE ALPHA-LABELS
        (LAMBDA (SEXPR ENV)
                (LET ((UFNVARS (AMAPCAR (LAMBDA (X)
                                                (IF (ATOM (CAR X))
                                                    (CAR X)
                                                    (CAAR X)))
                                        (CADR SEXPR))))
                     (LET ((FNVARS (DO ((I (LENGTH UFNVARS) (- I 1))
                                        (V NIL (CONS (GENTEMP 'FNVAR) V)))
                                       ((= I 0) (NREVERSE V)))))
                          (LET ((LENV (PAIRLIS UFNVARS FNVARS ENV)))
                               (NODIFY (CONS-LABELS (UFNVARS = UFNVARS)
                                                    (FNVARS = FNVARS)
                                                    (FNDEFS = (AMAPCAR
                                                               (LAMBDA (X)
                                                                       (ALPHA-LABELS-DEFN X LENV))
                                                               (CADR SEXPR)))
                                                    (BODY = (ALPHATIZE (CADDR SEXPR) LENV)))
                                       SEXPR
                                       ENV))))))
```

ALPHA-LABELS-DEFN parses one LABELS definition clause. An extension to the SCHEME language (made just after the publication of [Revised Report]!) allows a LABELS definition to take on any of the same three forms permitted by DEFINE. Thus this LABELS form actually defines FOO, BAR, and BAZ to be equivalent functions:

```
(LABELS ((FOO (LAMBDA (X Y) (BLOCK (PRINT X) (+ X Y))))
         (BAR (X Y) (PRINT X) (+ X Y))
         ((BAZ X Y) (PRINT X) (+ X Y)))
        (LIST (FOO 1 2) (BAR 1 2) (BAZ 1 2)))
```

ALPHA-BLOCK implements the standard macro definition of BLOCK. (BLOCK x) is simply x, and (BLOCK x . y) expands into:

```
((LAMBDA (A B) (B)) x (LAMBDA () (BLOCK . y)))
```

MACRO-EXPAND takes a macro call and expands it into a new form to be used in place of the macro call. In the PDP-10 MacLISP SCHEME implementation there are three different kinds of macros. Types MACRO and AMACRO are defined by MacLISP code, and so their defining functions are invoked using the MacLISP primitive FUNCALL. Type SMACRO is defined by SCHEME code which is in the value cell of an atomic symbol; thus SYMEVAL is used to get the contents of the value cell, and this SCHEME function is then invoked.

ALPHA-COMBINATION converts all the subforms of a combination, making a list of them, and creates a COMBINATION structure. If the function position contains a variable, it performs a consistency check using CHECK-NUMBER-OF-ARGS to make sure the right number of arguments is present.

```
(DEFINE ALPHA-LABELS-DEFN
        (LAMBDA (LDEF LENV)
                (ALPHATIZE (IF (ATOM (CAR LDEF))
                               (IF (CDDR LDEF)
                                   "(LAMBDA ,(CADR LDEF) ,(BLOCKIFY (CDDR LDEF)))
                                   (CADR LDEF))
                               "(LAMBDA ,(CDAR LDEF) ,(BLOCKIFY (CDR LDEF))))
                           LENV)))

(DEFINE ALPHA-BLOCK
        (LAMBDA (SEXPR ENV)
                (COND ((NULL (CDR SEXPR))
                       (WARN |BLOCK with no forms|
                             "(ENV = ,(AMAPCAR CAR ENV)))
                       (ALPHATIZE NIL ENV))
                      (T (LABELS ((MUNG
                                   (LAMBDA (BODY)
                                           (IF (NULL (CDR BODY))
                                               (CAR BODY)
                                               "((LAMBDA (A B) (B))
                                                 ,(CAR BODY)
                                                 (LAMBDA () ,(MUNG (CDR BODY))))))))
                                 (ALPHATIZE (MUNG (CDR SEXPR)) ENV))))))

(DEFINE MACRO-EXPAND
        (LAMBDA (SEXPR)
                (LET ((M (GETL (CAR SEXPR) '(MACRO AMACRO SMACRO))))
                     (IF (NULL M)
                         (BLOCK (WARN |missing macro definition| SEXPR)
                                "(ERROR '|Undefined Macro Form| ',SEXPR 'FAIL-ACT))
                         (EQCASE (CAR M)
                                 (MACRO (FUNCALL (CADR M) SEXPR))
                                 (AMACRO (FUNCALL (CADR M) SEXPR))
                                 (SMACRO ((SYMEVAL (CADR M)) SEXPR)))))))

(DEFINE ALPHA-COMBINATION
        (LAMBDA (SEXPR ENV)
                (LET ((N (NODIFY (CONS-COMBINATION
                                  (WARNP = NIL)
                                  (ARGS = (AMAPCAR (LAMBDA (X) (ALPHATIZE X ENV))
                                                   SEXPR)))
                                 SEXPR
                                 ENV)))
                     (LET ((M (NODE\FORM (CAR (COMBINATION\ARGS (NODE\FORM N))))))
                          (IF (AND (EQ (TYPE M) 'VARIABLE)
                                   (VARIABLE\GLOBALP M))
                              (CHECK-NUMBER-OF-ARGS
                               (VARIABLE\VAR M)
                               (LENGTH (CDR (COMBINATION\ARGS (NODE\FORM N))))
                               NIL))
                          N))))
```

Once the S-expression function definition has been copied as a NODE tree, COMPILE calls PASS1-ANALYZE to fill in various pieces of information. (If optimization is to be performed, COMPILE instead calls META-EVALUATE. META-EVALUATE in turn calls PASS1-ANALYZE in a coroutining manner we will examine later.) PASS1-ANALYZE in turn calls ENV-ANALYZE, TRIV-ANALYZE, and EFFS-ANALYZE in order. Each of these has roughly the same structure. Each takes a node and a flag called REDOTHIS. Normally REDOTHIS is NIL and the information has not yet been installed in the node, and so the routine proceeds to analyze the node and install the appropriate information.

When invoked by the optimizer, however, there may be information in the node already, but that information may be incorrect or obsolete as a result of the optimizing transformations. If REDOTHIS is non-NIL, then the given node <u>must</u> be reanalyzed, even if the information is already present. If REDOTHIS is <u>in</u> fact the symbol ALL, then all descendants of the given node must be reanalyzed. Otherwise, only the given node requires re-analysis, plus any descendants which have not had the information installed at all. We will see later how these mechanisms are used in the optimizer.

The purpose of ENV-ANALYZE is to fill in for each node the slots REFS and ASETS. The first is a set (represented as a list) of the new-names of all variables bound above the node and referenced at or below the node, and the second (a subset of the first) is a set of such names which appear in an ASET at or below the node. These lists are computed recursively. A CONSTANT node has no such references; a VARIABLE node (with GLOBALP = NIL) refers to its own variable. An ASET node adds its variable to the ASET list for its body. Most other kinds of nodes merely merge together the lists for their immediate descendants. In order to satisfy the "bound above the node" requirement, those structures which bind variables (LAMBDA, CATCH, LABELS) filter out their own bound variables from the two sets.

As an example, consider this function:

```
(LAMBDA (X)
        ((LAMBDA (Y)
                 ((LAMBDA (W)
                          (ASET' Z (* X Y)))
                  (ASET' Y (- Y 1))))
         (- X 3)))
```

The node for (- X 3) would have a REFS list (X) and an ASET list (). The node for the ASET on Z would have REFS=(X Y) (or perhaps (Y X)) and ASETS=(); Z does not appear in the ASETS list because it is not bound above. The node for the combination ((LAMBDA (W) ...) ...) would have REFS=(X Y) and ASETS=(Y). The node for the lambda-expression (LAMBDA (Y) ...) would have REFS=(X) and ASETS=(), because Y is filtered out.

```
;;; ENVIRONMENT ANALYSIS.

;;; FOR NODES ENCOUNTERED WE FILL IN:
;;;     REFS
;;;     ASETS
;;; ON VARIABLE NAMES THESE PROPERTIES ARE CREATED:
;;;     BINDING         THE NODE WHERE THE VARIABLE IS BOUND
;;;     USER-NAME       THE USER'S NAME FOR THE VARIABLE (WHERE BOUND)
;;;     READ-REFS       VARIABLE NODES WHICH READ THE VARIABLE
;;;     WRITE-REFS      ASET NODES WHICH SET THE VARIABLE

;;; NORMALLY, ON RECURRING TO A LOWER NODE WE STOP IF THE INFORMATION
;;; IS ALREADY THERE.  MAKING THE PARAMETER "REDOTHIS" BE "ALL" FORCES
;;; RE-COMPUTATION TO ALL LEVELS; MAKING IT "ONCE" FORCES
;;; RECOMPUTATION OF THIS NODE BUT NOT OF SUBNODES.

(DEFINE ENV-ANALYZE
        (LAMBDA (NODE REDOTHIS)
                (IF (OR REDOTHIS (EMPTY (NODE\REFS NODE)))
                    (LET ((FM (NODE\FORM NODE))
                          (REDO (IF (EQ REDOTHIS 'ALL) 'ALL NIL)))
                         (EQCASE (TYPE FM)
                                (CONSTANT
                                 (ALTER-NODE NODE
                                            (REFS := NIL)
                                            (ASETS := NIL)))
                                (VARIABLE
                                 (ADDPROP (VARIABLE\VAR FM) NODE 'READ-REFS)
                                 (IF (VARIABLE\GLOBALP FM)
                                     (SETPROP (VARIABLE\VAR FM) (VARIABLE\VAR FM) 'USER-NAME))
                                 (ALTER-NODE NODE
                                            (REFS := (AND (NOT (VARIABLE\GLOBALP FM))
                                                          (LIST (VARIABLE\VAR FM))))
                                            (ASETS := NIL)))
                                (LAMBDA
                                 (DO ((V (LAMBDA\VARS FM) (CDR V))
                                      (UV (LAMBDA\UVARS FM) (CDR UV)))
                                     ((NULL V))
                                     (SETPROP (CAR V) (CAR UV) 'USER-NAME)
                                     (SETPROP (CAR V) NODE 'BINDING))
                                 (LET ((B (LAMBDA\BODY FM)))
                                      (ENV-ANALYZE B REDO)
                                      (ALTER-NODE NODE
                                                (REFS := (SETDIFF (NODE\REFS B)
                                                                  (LAMBDA\VARS FM)))
                                                (ASETS := (SETDIFF (NODE\ASETS B)
                                                                   (LAMBDA\VARS FM))))))
                                (IF
                                 (LET ((PRED (IF\PRED FM))
                                       (CON (IF\CON FM))
                                       (ALT (IF\ALT FM)))
                                      (ENV-ANALYZE PRED REDO)
                                      (ENV-ANALYZE CON REDO)
                                      (ENV-ANALYZE ALT REDO)
                                      (ALTER-NODE NODE
                                                (REFS := (UNION (NODE\REFS PRED)
                                                                (UNION (NODE\REFS CON)
                                                                       (NODE\REFS ALT))))
                                                (ASETS := (UNION (NODE\ASETS PRED)
                                                                 (UNION (NODE\ASETS CON)
                                                                        (NODE\ASETS ALT)))))))
                                (ASET
                                 (LET ((B (ASET\BODY FM))
                                       (V (ASET\VAR FM)))
                                      (ENV-ANALYZE B REDO)
                                      (ADDPROP V NODE 'WRITE-REFS)
                                      (IF (ASET\GLOBALP FM)
                                          (ALTER-NODE NODE
```

It should be easy to see the the topmost node of the node-tree must have REFS=() and ASETS=(), because no variables are bound above it. This fact is used in COMPILE for a consistency check. (After writing this last sentence, I noticed that in fact this consistency check was not being performed, and that it was a good idea. On being installed, this check immediately caught a subtle bug in the optimizer. Consistency checks pay off!)

Another purpose accomplished by ENV-ANALYZE is the installation of several useful properties on the new-names of bound variables. Two properties, READ-REFS and WRITE-REFS, accumulate for each variable the set of VARIABLE nodes which refer to it and the set of ASET nodes that refer to it. These lists are very important to the optimizer. A non-empty WRITE-REFS set also calls for special action by the code generator.

When a LAMBDA node is encountered, that node is put onto each new-name under the BINDING property, and the user-name is put under the USER-NAME property; these are used only for debugging.

```
                      (REFS := (NODE\REFS B))
                      (ASETS := (NODE\ASETS B)))
          (ALTER-NODE NODE
                      (REFS := (ADJOIN V (NODE\REFS B)))
                      (ASETS := (ADJOIN V (NODE\ASETS B)))))))
(CATCH
 (LET ((B (CATCH\BODY FM))
       (V (CATCH\VAR FM)))
      (SETPROP V (CATCH\UVAR FM) 'USER-NAME)
      (SETPROP V NODE 'BINDING)
      (ENV-ANALYZE B REDO)
      (ALTER-NODE NODE
                  (REFS := (REMOVE V (NODE\REFS B)))
                  (ASETS := (REMOVE V (NODE\ASETS B))))))
(LABELS
 (DO ((V (LABELS\FNVARS FM) (CDR V))
      (UV (LABELS\UFNVARS FM) (CDR UV))
      (D (LABELS\FNDEFS FM) (CDR D))
      (R NIL (UNION R (NODE\REFS (CAR D))))
      (A NIL (UNION A (NODE\ASETS (CAR D)))))
     ((NULL V)
      (LET ((B (LABELS\BODY FM)))
           (ENV-ANALYZE B REDO)
           (ALTER-NODE NODE
                       (REFS := (SETDIFF
                                 (UNION R (NODE\REFS B))
                                 (LABELS\FNVARS FM)))
                       (ASETS := (SETDIFF
                                  (UNION A (NODE\ASETS B))
                                  (LABELS\FNVARS FM))))))
     (SETPROP (CAR V) (CAR UV) 'USER-NAME)
     (SETPROP (CAR V) NODE 'BINDING)
     (ENV-ANALYZE (CAR D) REDO)))
(COMBINATION
 (LET ((ARGS (COMBINATION\ARGS FM)))
      (AMAPC (LAMBDA (X) (ENV-ANALYZE X REDO)) ARGS)
      (DO ((A ARGS (CDR A))
           (R NIL (UNION R (NODE\REFS (CAR A))))
           (S NIL (UNION S (NODE\ASETS (CAR A)))))
          ((NULL A)
           (ALTER-NODE NODE
                       (REFS := R)
                       (ASETS := S)))))))))))
```

TRIV-ANALYZE fills in the TRIVP slot for each node. This is a flag which, if non-NIL, indicates that the code represented by that node and its descendants is "trivial", i.e. it can be executed as simple host machine (MacLISP) code because no SCHEME closures are involved. Constants and variables are trivial, as are combinations with trivial arguments and a provably trivial function. While lambda-expressions are in general non-trivial (because a closure must be constructed), a special case is made for ((LAMBDA ...) ...), i.e. a combination whose function is a lambda-expression. This is possible because the code generator will not really generate a closure for the lambda-expression. This is the first example of a trichotomy we will encounter repeatedly. Combinations are divided into three kinds: those with a lambda-expression in the function position, those with a trivial MacLISP primitive (satisfying the predicate TRIVFN) in the function position, and all others.

All other expressions are, in general, trivial iff all their subparts are trivial. Note that a LABELS is trivial iff its body is trivial; the non-triviality of the bound functions does not affect this.

The triviality flag is used by phase 2 to control conversion to continuation-passing style. This in turn affects the code generator, which compiles trivial forms straightforwardly into MacLISP code, rather than using the more complex techniques required by non-trivial SCHEME code. It would be possible to avoid triviality analysis entirely; the net result would only be less optimal final code.

```
;;; TRIVIALITY ANALYSIS

;;; FOR NODES ENCOUNTERED WE FILL IN:
;;;     TRIVP

;;; A COMBINATION IS TRIVIAL IFF ALL ARGUMENTS ARE TRIVIAL, AND
;;; THE FUNCTION CAN BE PROVED TO BE TRIVIAL.  WE ASSUME CLOSURES
;;; TO BE NON-TRIVIAL IN THIS CONTEXT, SO THAT THE CONVERT FUNCTION
;;; WILL BE FORCED TO EXAMINE THEM.

(DEFINE TRIV-ANALYZE
        (LAMBDA (NODE REDOTHIS)
                (IF (OR REDOTHIS (EMPTY (NODE\TRIVP NODE)))
                    (LET ((FM (NODE\FORM NODE))
                          (REDO (IF (EQ REDOTHIS 'ALL) 'ALL NIL)))
                         (EQCASE (TYPE FM)
                                 (CONSTANT
                                  (ALTER-NODE NODE (TRIVP := T)))
                                 (VARIABLE
                                  (ALTER-NODE NODE (TRIVP := T)))
                                 (LAMBDA
                                  (TRIV-ANALYZE (LAMBDA\BODY FM) REDO)
                                  (ALTER-NODE NODE (TRIVP := NIL)))
                                 (IF
                                  (TRIV-ANALYZE (IF\PRED FM) REDO)
                                  (TRIV-ANALYZE (IF\CON FM) REDO)
                                  (TRIV-ANALYZE (IF\ALT FM) REDO)
                                  (ALTER-NODE NODE
                                             (TRIVP := (AND (NODE\TRIVP (IF\PRED FM))
                                                            (NODE\TRIVP (IF\CON FM))
                                                            (NODE\TRIVP (IF\ALT FM))))))
                                 (ASET
                                  (TRIV-ANALYZE (ASET\BODY FM) REDO)
                                  (ALTER-NODE NODE (TRIVP := (NODE\TRIVP (ASET\BODY FM)))))
                                 (CATCH
                                  (TRIV-ANALYZE (CATCH\BODY FM) REDO)
                                  (ALTER-NODE NODE (TRIVP := NIL)))
                                 (LABELS
                                  (AMAPC (LAMBDA (F) (TRIV-ANALYZE F REDO))
                                         (LABELS\FNDEFS FM))
                                  (TRIV-ANALYZE (LABELS\BODY FM) REDO)
                                  (ALTER-NODE NODE (TRIVP := NIL)))
                                 (COMBINATION
                                  (LET ((ARGS (COMBINATION\ARGS FM)))
                                       (TRIV-ANALYZE (CAR ARGS) REDO)
                                       (DO ((A (CDR ARGS) (CDR A))
                                            (SW T (AND SW (NODE\TRIVP (CAR A)))))
                                           ((NULL A)
                                            (ALTER-NODE NODE
                                                       (TRIVP := (AND SW
                                                                      (TRIV-ANALYZE-FN-P
                                                                       (CAR ARGS))))))
                                           (TRIV-ANALYZE (CAR A) REDO)))))))))

(DEFINE TRIV-ANALYZE-FN-P
        (LAMBDA (FN)
                (OR (AND (EQ (TYPE (NODE\FORM FN)) 'VARIABLE)
                         (TRIVFN (VARIABLE\VAR (NODE\FORM FN))))
                    (AND (EQ (TYPE (NODE\FORM FN)) 'LAMBDA)
                         (NODE\TRIVP (LAMBDA\BODY (NODE\FORM FN)))))))
```

EFFS-ANALYZE analyzes the code for side-effects. In each node the four slots EFFS, AFFD, PEFFS, and PAFFD are filled in. Each is a set of side effects, which may be the symbol NONE, meaning no side effects; ANY, meaning all possible side effects; or a list of specific side effect names. Each such name specifies a category of possible side effects. Typical names are ASET, RPLACD, and FILE (which means input/output transactions).

The four slots EFFS, AFFD, PEFFS, and PAFFD refer to the node they are in and all nodes beneath it. Thus each is computed by taking the union of the corresponding sets of all immediate descendants, then adjoining any effects due to the current node.

EFFS is the set of side effects which may possibly be caused at or below the current node; PEFFS is the set of side effects which can be proved to occur at or below the node. These may differ because of ignorance on RABBIT's part. For example, the node for a combination (RPLACA A B) will have the side-effect name RPLACA adjoined to both EFFS and PEFFS, because the RABBIT knows that RPLACA causes an RPLACA side effect (how this is known will be discussed later). On the other hand, for a combination (FOO A B), where FOO is some user function, RABBIT can only conjecture that FOO can cause any conceivable side effect, but cannot prove it. Thus EFFS will be forced to be ANY, while PEFFS will not.

AFFD is the set of side effects which can possibly affect the evaluation of the current node or its descendants. For example, an RPLACA side effect can affect the evaluation of (CAR X), but on the other hand an RPLACD side effect cannot. PAFFD is the corresponding set of side effects for which it can be proved. (This set is "proved" in a less rigorous sense than for PEFFS. The name RPLACA would be put in the PAFFD set for (CAR X), even though the user might know that while there are calls to RPLACA in his program, none of them ever modify X. PEFFS and PAFFD are only used by CHECK-COMBINATION-PEFFS to warn the user of potential conflicts anyway, and serve no other purpose. EFFS and AFFD, on the other hand, are used by the optimizer to prevent improper code motion. Thus EFFS and AFFD must be pessimistic, and err only on the safe side; while PEFFS and PAFFD are optimistic, so that the user will not be pestered with too many warning messages.)

The CONS side effect is treated specially. A node which causes the CONS side effect must not be duplicated, because each instance will create a new object; but whereas two RPLACA side effects may not be executed out of order, two CONS side effects may be.

The computation of AFFD and PAFFD for variables depends on whether the variable is global or not. If it is, SETQ and RPLACD can affect it (RPLACD can occur because of the peculiarities of the PDP-10 MacLISP implementation); otherwise, ASET can affect it if indeed any ASET refers to it (in which case ENV-ANALYZE will have left a WRITE-REFS property); otherwise, nothing can affect it. Similar remarks hold for the computation of EFFS and PEFFS for an ASET node. The name SETQ applies to modifications of global variables, while ASET applies to local variables.

```
;;; SIDE-EFFECTS ANALYSIS
;;; FOR NODES ENCOUNTERED WE FILL IN:  EFFS, AFFD, PEFFS, PAFFD
;;; A SET OF SIDE EFFECTS MAY BE EITHER 'NONE OR 'ANY, OR A SET.

(DEFINE EFFS-ANALYZE
        (LAMBDA (NODE REDOTHIS)
                (IF (OR REDOTHIS (EMPTY (NODE\EFFS NODE)))
                    (LET ((FM (NODE\FORM NODE))
                           (REDO (IF (EQ REDOTHIS 'ALL) 'ALL NIL)))
                          (EQCASE (TYPE FM)
                                  (CONSTANT
                                   (ALTER-NODE NODE
                                              (EFFS := 'NONE)
                                              (AFFD := 'NONE)
                                              (PEFFS := 'NONE)
                                              (PAFFD := 'NONE)))
                                  (VARIABLE
                                   (LET ((A (COND ((VARIABLE\GLOBALP FM) '(SETQ))
                                                  ((GET (VARIABLE\VAR FM) 'WRITE-REFS) '(ASET))
                                                  (T 'NONE))))
                                        (ALTER-NODE NODE
                                                   (EFFS := 'NONE)
                                                   (AFFD := A)
                                                   (PEFFS := 'NONE)
                                                   (PAFFD := A))))
                                  (LAMBDA
                                   (EFFS-ANALYZE (LAMBDA\BODY FM) REDO)
                                   (ALTER-NODE NODE
                                              (EFFS := '(CONS))
                                              (AFFD := NIL)
                                              (PEFFS := '(CONS))
                                              (PAFFD := NIL)))
                                  (IF (EFFS-ANALYZE-IF NODE FM REDO))
                                  (ASET
                                   (EFFS-ANALYZE (ASET\BODY FM) REDO)
                                   (LET ((ASETEFFS (IF (ASET\GLOBALP FM)
                                                       '(SETQ)
                                                       '(ASET))))
                                        (ALTER-NODE NODE
                                                   (EFFS := (EFFS-UNION ASETEFFS
                                                                        (NODE\EFFS (ASET\BODY FM))))
                                                   (AFFD := (NODE\AFFD (ASET\BODY FM)))
                                                   (PEFFS := (EFFS-UNION ASETEFFS
                                                                         (NODE\PEFFS (ASET\BODY FM))))
                                                   (PAFFD := (NODE\PAFFD (ASET\BODY FM))))))
                                  (CATCH
                                   (EFFS-ANALYZE (CATCH\BODY FM) REDO)
                                   (ALTER-NODE NODE
                                              (EFFS := (NODE\EFFS (CATCH\BODY FM)))
                                              (AFFD := (NODE\AFFD (CATCH\BODY FM)))
                                              (PEFFS := (NODE\PEFFS (CATCH\BODY FM)))
                                              (PAFFD := (NODE\PAFFD (CATCH\BODY FM)))))
                                  (LABELS
                                   (AMAPC (LAMBDA (F) (EFFS-ANALYZE F REDO))
                                          (LABELS\FNDEFS FM))
                                   (EFFS-ANALYZE (LABELS\BODY FM) REDO)
                                   (ALTER-NODE NODE
                                              (EFFS := (EFFS-UNION '(CONS)
                                                                   (NODE\EFFS (LABELS\BODY FM))))
                                              (AFFD := (NODE\AFFD (LABELS\BODY FM)))
                                              (PEFFS := (EFFS-UNION '(CONS)
                                                                    (NODE\PEFFS (LABELS\BODY FM))))
                                              (PAFFD := (NODE\PAFFD (LABELS\BODY FM)))))
                                  (COMBINATION
                                   (EFFS-ANALYZE-COMBINATION NODE FM REDO)))))))
```

(While it may be held that allowing ASET' on variables is unclean, and that the use of cells as in PLASMA is semantically neater, it is true that because of the lexical scoping rules it can always be determined whether a given variable is ever used in an ASET'. In this way one can say that variables are divided by the compiler into two classes: those which are implicitly cells, and those which are not.)

A closure (LAMBDA-expression) causes a CONS side-effect. This is not so much because SCHEME programs depend on being able to do EQ on closures (it is unclear whether this is a reasonable thing to specify in the semantics of SCHEME), as because there is no point in creating two closures when one will suffice. Adjoining CONS to EFFS will prevent the creation of such duplicate code by the optimizer. The same idea holds for LABELS (which has LAMBDA-expressions within it).

Notice that a LAMBDA node does not add to its four sets the information from its body's sets. This is because evaluation of a LAMBDA-expression does not immediately evaluate the body. Only later, when the resulting closure is invoked, is the body executed.

EFFS-UNION gives the union of two sets of side effects. It knows about the special symbols NONE and ANY.

EFFS-ANALYZE-IF computes the side-effect sets for IF nodes. It has been made a separate function only because its code is so bulky; it must perform a three-way union for each of four sets.

EFFS-ANALYZE-COMBINATION computes the side-effect sets for COMBINATION nodes. First the function is analyzed, then the arguments. The unions of the four sets over all the arguments are accumulated in EF, AF, PEF, and PAF. CHECK-COMBINATION-PEFFS is called to warn the user of any possible violations of the rule that SCHEME is privileged to choose the order in which to evaluate the subforms of a combination. Finally, there are three cases depending on the form of the function position.

If it is a variable, then the property list of the variable name is searched for information about that function. (The generated names for local variables will never have any such information; thus information will be found only for global variables. This information is used to augment the sets. (A clever technique not used in RABBIT would be to arrange for situations like ((LAMBDA (F) <body1>) (LAMBDA (...) <body2>), where F denotes a "known function" (see the description of BIND-ANALYZE below), to put on the property list of F the side-effect information for <body2>, to aid optimization in <body1>.)

If the function position is a LAMBDA-expression, then the four sets of the body of the LAMBDA-expression are unioned into the four sets for the COMBINATION node. This is because in this case we know that the body LAMBDA-expression will be executed in the course of executing the COMBINATION node.

In any other case, an unknown function is computed, and so it must be assumed that any side-effect is possible for EFFS and AFFD.

```
(DEFINE EFFS-UNION
        (LAMBDA (A B)
                (COND ((EQ A 'NONE) B)
                      ((EQ B 'NONE) A)
                      ((EQ A 'ANY) 'ANY)
                      ((EQ B 'ANY) 'ANY)
                      (T (UNION A B)))))

(DEFINE EFFS-ANALYZE-IF
        (LAMBDA (NODE FM REDO)
                (BLOCK (EFFS-ANALYZE (IF\PRED FM) REDO)
                       (EFFS-ANALYZE (IF\CON FM) REDO)
                       (EFFS-ANALYZE (IF\ALT FM) REDO)
                       (ALTER-NODE NODE
                                   (EFFS := (EFFS-UNION (NODE\EFFS (IF\PRED FM))
                                                        (EFFS-UNION (NODE\EFFS (IF\CON FM))
                                                                    (NODE\EFFS (IF\ALT FM)))))
                                   (AFFD := (EFFS-UNION (NODE\AFFD (IF\PRED FM))
                                                        (EFFS-UNION (NODE\AFFD (IF\CON FM))
                                                                    (NODE\AFFD (IF\ALT FM)))))
                                   (PEFFS := (EFFS-UNION (NODE\PEFFS (IF\PRED FM))
                                                         (EFFS-UNION (NODE\PEFFS (IF\CON FM))
                                                                     (NODE\PEFFS (IF\ALT FM)))))
                                   (PAFFD := (EFFS-UNION (NODE\PAFFD (IF\PRED FM))
                                                         (EFFS-UNION (NODE\PAFFD (IF\CON FM))
                                                                     (NODE\PAFFD (IF\ALT FM)))))))))

(SET' *CHECK-PEFFS* NIL)

(DEFINE EFFS-ANALYZE-COMBINATION
        (LAMBDA (NODE FM REDO)
                (LET ((ARGS (COMBINATION\ARGS FM)))
                     (EFFS-ANALYZE (CAR ARGS) REDO)
                     (DO ((A (CDR ARGS) (CDR A))
                          (EF 'NONE (EFFS-UNION EF (NODE\EFFS (CAR A))))
                          (AF 'NONE (EFFS-UNION AF (NODE\AFFD (CAR A))))
                          (PEF 'NONE (EFFS-UNION PEF (NODE\PEFFS (CAR A))))
                          (PAF 'NONE (EFFS-UNION PAF (NODE\PAFFD (CAR A)))))
                         ((NULL A)
                          (IF *CHECK-PEFFS* (CHECK-COMBINATION-PEFFS FM))
                          (COND ((EQ (TYPE (NODE\FORM (CAR ARGS))) 'VARIABLE)
                                 (LET ((V (VARIABLE\VAR (NODE\FORM (CAR ARGS)))))
                                      (LET ((VE (GET V 'FN-SIDE-EFFECTS))
                                            (VA (GET V 'FN-SIDE-AFFECTED)))
                                           (ALTER-NODE NODE
                                                       (EFFS := (IF VE (EFFS-UNION EF VE) 'ANY))
                                                       (AFFD := (IF VA (EFFS-UNION AF VA) 'ANY))
                                                       (PEFFS := (EFFS-UNION PEF VE))
                                                       (PAFFD := (EFFS-UNION PAF VA))))))
                                ((EQ (TYPE (NODE\FORM (CAR ARGS))) 'LAMBDA)
                                 (LET ((B (LAMBDA\BODY (NODE\FORM (CAR ARGS)))))
                                      (ALTER-NODE NODE
                                                  (EFFS := (EFFS-UNION EF (NODE\EFFS B)))
                                                  (AFFD := (EFFS-UNION AF (NODE\AFFD B)))
                                                  (PEFFS := (EFFS-UNION PEF (NODE\PEFFS B)))
                                                  (PAFFD := (EFFS-UNION PAF (NODE\PAFFD B))))))
                                (T (ALTER-NODE NODE
                                               (EFFS := 'ANY)
                                               (AFFD := 'ANY)
                                               (PEFFS := (EFFS-UNION PEF
                                                                     (NODE\PEFFS (CAR ARGS))))
                                               (PAFFD := (EFFS-UNION PAF
                                                                     (NODE\PAFFD (CAR ARGS))))))))
                         (EFFS-ANALYZE (CAR A) REDO)))))
```

CHECK-COMBINATION-PEFFS checks all the argument forms of a combination (including the function position) to see if they are all independent of each other with respect to side effects. If not, a warning is issued. This is because the semantics of SCHEME specify that the arguments may be evaluated in any order, and the user may not depend on a particular ordering.

The test is made by comparing all pairs of arguments within the combination. If the side-effects of one can "provably" affect the evaluation of the other, or if they both cause a side effect of the same category (other than CONS, which is special), then the results may depend on which order they are evaluated in. The test is not completely rigorous, and may err in either direction, but "probably" a reasonably written SCHEME program will satisfy the test.

This check is controlled by the switch *CHECK-PEFFS* in EFFS-ANALYZE-COMBINATION. This switch is provided because empirical tests show that performing the test slows down compilation by twenty to thirty percent. The test has proved valuable in trapping programming errors, and so is normally on, but it can be turned off for speed in compiling programs in which one has confidence.

EFFDEF is a macro which expands into a number of DEFPROP forms. This is used to define side-effect information about primitive functions. For example:

```
(EFFDEF CADR NONE (RPLACA RPLACD))
```

states that CADR causes no side-effects, and is "provably" affected by the RPLACA and RPLACD categories of side-effects. Similarly:

```
(EFFDEF MEMQ NONE (RPLACA RPLACD) T)
```

specifies the same information for MEMQ, but the "T" means that a call to MEMQ with constant arguments may be "folded" (evaluated, and the result compiled instead), despite the fact that some side effects can affect it. This represents a judgement that it is extremely unlikely that someone will write a program which modifies a constant argument to be given to MEMQ.

```
(DEFINE CHECK-COMBINATION-PEFFS
        (LAMBDA (FM)
                (IF (NOT (COMBINATION\WARNP FM))
                    (DO ((A (COMBINATION\ARGS FM) (CDR A)))
                        ((NULL A))
                        (DO ((B (CDR A) (CDR B)))
                            ((NULL B))
                            (IF (NOT (EFFECTLESS (EFFS-INTERSECT (NODE\PEFFS (CAR A))
                                                                 (NODE\PAFFD (CAR B)))))
                                (BLOCK (WARN |co-argument may affect later one|
                                             (NODE\SEXPR (CAR A))
                                             "(EFFECTS = ,(NODE\PEFFS (CAR A)))
                                             (NODE\SEXPR (CAR B))
                                             "(AFFECTED BY ,(NODE\PAFFD (CAR B))))
                                       (ALTER-COMBINATION FM (WARNP := T))))
                            (IF (NOT (EFFECTLESS (EFFS-INTERSECT (NODE\PEFFS (CAR B))
                                                                 (NODE\PAFFD (CAR A)))))
                                (BLOCK (WARN |co-argument may affect earlier one|
                                             (NODE\SEXPR (CAR B))
                                             "(EFFECTS = ,(NODE\PEFFS (CAR B)))
                                             (NODE\SEXPR (CAR A))
                                             "(AFFECTED BY ,(NODE\PAFFD (CAR A))))
                                       (ALTER-COMBINATION FM (WARNP := T))))
                            (IF (NOT (EFFECTLESS-EXCEPT-CONS (EFFS-INTERSECT (NODE\PEFFS (CAR A))
                                                                             (NODE\PEFFS (CAR B)))))
                                (BLOCK (WARN |co-arguments may have interfering effects|
                                             (NODE\SEXPR (CAR A))
                                             "(EFFECTS = ,(NODE\PEFFS (CAR A)))
                                             (NODE\SEXPR (CAR B))
                                             "(EFFECTS = ,(NODE\PEFFS (CAR B))))
                                       (ALTER-COMBINATION FM (WARNP := T)))))))))

(DEFMAC EFFDEF (FN EFFS AFFD . FOLD)
        "(PROGN (DEFPROP ,FN ,EFFS FN-SIDE-EFFECTS)
                (DEFPROP ,FN ,AFFD FN-SIDE-AFFECTED)
                ,(AND FOLD "(DEFPROP ,FN T OKAY-TO-FOLD))))

(DECLARE (/@DEFINE EFFDEF |SIDE EFFECTS|))
```

This page contains declarations of side-effect information for many standard primitive functions. The EFFDEF macro used to make the declarations is described on the previous page.

```
(PROGN 'COMPILE
       (EFFDEF + NONE NONE)
       (EFFDEF - NONE NONE)
       (EFFDEF * NONE NONE)
       (EFFDEF // NONE NONE)
       (EFFDEF = NONE NONE)
       (EFFDEF < NONE NONE)
       (EFFDEF > NONE NONE)
       (EFFDEF CAR NONE (RPLACA))
       (EFFDEF CDR NONE (RPLACD))
       (EFFDEF CAAR NONE (RPLACA))
       (EFFDEF CADR NONE (RPLACA RPLACD))
       (EFFDEF CDAR NONE (RPLACA RPLACD))
       (EFFDEF CDDR NONE (RPLACD))
       (EFFDEF CAAAR NONE (RPLACA))
       (EFFDEF CAADR NONE (RPLACA RPLACD))
       (EFFDEF CADAR NONE (RPLACA RPLACD))
       (EFFDEF CADDR NONE (RPLACA RPLACD))
       (EFFDEF CDAAR NONE (RPLACA RPLACD))
       (EFFDEF CDADR NONE (RPLACA RPLACD))
       (EFFDEF CDDAR NONE (RPLACA RPLACD))
       (EFFDEF CDDDR NONE (RPLACD))
       (EFFDEF CAAAAR NONE (RPLACA))
       (EFFDEF CAAADR NONE (RPLACA RPLACD))
       (EFFDEF CAADAR NONE (RPLACA RPLACD))
       (EFFDEF CAADDR NONE (RPLACA RPLACD))
       (EFFDEF CADAAR NONE (RPLACA RPLACD))
       (EFFDEF CADADR NONE (RPLACA RPLACD))
       (EFFDEF CADDAR NONE (RPLACA RPLACD))
       (EFFDEF CADDDR NONE (RPLACA RPLACD))
       (EFFDEF CDAAAR NONE (RPLACA RPLACD))
       (EFFDEF CDAADR NONE (RPLACA RPLACD))
       (EFFDEF CDADAR NONE (RPLACA RPLACD))
       (EFFDEF CDADDR NONE (RPLACA RPLACD))
       (EFFDEF CDDAAR NONE (RPLACA RPLACD))
       (EFFDEF CDDADR NONE (RPLACA RPLACD))
       (EFFDEF CDDDAR NONE (RPLACA RPLACD))
       (EFFDEF CDDDDR NONE (RPLACD))
       (EFFDEF CXR NONE (RPLACA RPLACD))
       (EFFDEF RPLACA (RPLACA) NONE)
       (EFFDEF RPLACD (RPLACA) NONE)
       (EFFDEF RPLACX (RPLACA RPLACD) NONE)
       (EFFDEF EQ NONE NONE)
       (EFFDEF ATOM NONE NONE)
       (EFFDEF NUMBERP NONE NONE)
       (EFFDEF TYPEP NONE NONE)
       (EFFDEF SYMBOLP NONE NONE)
       (EFFDEF HUNKP NONE NONE)
       (EFFDEF FIXP NONE NONE)
       (EFFDEF FLOATP NONE NONE)
       (EFFDEF BIGP NONE NONE)
       (EFFDEF NOT NONE NONE)
       (EFFDEF NULL NONE NONE)
       (EFFDEF CONS (CONS) NONE)
       (EFFDEF LIST (CONS) NONE)
       (EFFDEF APPEND (CONS) (RPLACD))
       (EFFDEF MEMQ NONE (RPLACA RPLACD) T)
       (EFFDEF ASSQ NONE (RPLACA RPLACD) T)
       (EFFDEF PRINT (FILE) (FILE RPLACA RPLACD))
       (EFFDEF PRIN1 (FILE) (FILE RPLACA RPLACD))
       (EFFDEF PRINC (FILE) (FILE RPLACA RPLACD))
       (EFFDEF TERPRI (FILE) (FILE))
       (EFFDEF TYO (FILE) (FILE))
       (EFFDEF READ ANY (FILE))
       (EFFDEF TYI ANY (FILE))
       'SIDE-EFFECTS-PROPERTIES)
```

ERASE-NODE and ERASE-ALL-NODES are convenient mnemonic macros used to invoke ERASE-NODES.

ERASE-NODES is used by the optimizer to destroy nodes which have been removed from the program tree because of some optimization. If ALLP is NIL (ERASE-NODE), then only the given node is erased; if it is T (ERASE-ALL-NODES), then the given node and all descendants, direct and indirect, are erased.

Erasing a node may involve removing certain properties from property lists. This is necessary to maintain the consistency of the properties. For example, if a VARIABLE node is erased, then that node must be removed from the READ-REFS property of the variable name. The optimizer depends on this so that, for example, it can determine whether all references to a variable have been erased.

It should be noted in passing that in principle all occurrences of ASET on a given variable could be erased, thereby reducing its WRITE-REFS property to NIL. Because the EFFS-ANALYZE computation on VARIABLE nodes used the WRITE-REFS property, a VARIABLE node might have ASET in its AFFD set after the optimizer had removed all the ASET nodes. Because of the tree-walking discipline of the optimizer, the VARIABLE nodes will not be reanalyzed immediately. This cannot hurt, however; it may just cause the optimizer later to be more cautious than necessary when examining a VARIABLE node. (If this doesn't make sense, come back after reading the description of the optimizer.)

The flag *TESTING* is used to determine whether or not to remove the node from the NODE property on the node's name. When debugging, it is very useful to keep this information around to trace the optimizer's actions; but when compiling a large function for "production" purposes, the discarded nodes may bloat memory, and so they must be removed from the NODE property in order that they may be garbage-collected by LISP.

```
;;; THIS ROUTINE IS USED TO UNDO ANY PASS 1 ANALYSIS ON A NODE.

(DEFMAC ERASE-NODE (NODE) "(ERASE-NODES ,NODE NIL))
(DEFMAC ERASE-ALL-NODES (NODE) "(ERASE-NODES ,NODE T))

(DEFINE ERASE-NODES
        (LAMBDA (NODE ALLP)
                (LET ((FM (NODE\FORM NODE)))
                     (OR (EQ (TYPE NODE) 'NODE)
                         (ERROR '|Cannot erase a non-node| NODE 'FAIL-ACT))
                     (EQCASE (TYPE FM)
                             (CONSTANT)
                             (VARIABLE
                              (DELPROP (VARIABLE\VAR FM) NODE 'READ-REFS))
                             (LAMBDA
                              (IF ALLP (ERASE-ALL-NODES (LAMBDA\BODY FM)))
                              (IF (NOT *TESTING*)
                                  (AMAPC (LAMBDA (V) (REMPROP V 'BINDING)) (LAMBDA\VARS FM))))
                             (IF (COND (ALLP (ERASE-ALL-NODES (IF\PRED FM))
                                             (ERASE-ALL-NODES (IF\CON FM))
                                             (ERASE-ALL-NODES (IF\ALT FM)))))
                             (ASET
                              (IF ALLP (ERASE-ALL-NODES (ASET\BODY FM)))
                              (DELPROP (ASET\VAR FM) NODE 'WRITE-REFS))
                             (CATCH
                              (IF ALLP (ERASE-ALL-NODES (CATCH\BODY FM)))
                              (IF (NOT *TESTING*)
                                  (REMPROP (CATCH\VAR FM) 'BINDING)))
                             (LABELS
                              (COND (ALLP (AMAPC (LAMBDA (D) (ERASE-ALL-NODES D))
                                                 (LABELS\FNDEFS FM))
                                          (ERASE-ALL-NODES (LABELS\BODY FM))))
                              (IF (NOT *TESTING*)
                                  (AMAPC (LAMBDA (V) (REMPROP V 'BINDING)) (LABELS\FNVARS FM))))
                             (COMBINATION
                              (IF ALLP (AMAPC (LAMBDA (A) (ERASE-ALL-NODES A))
                                              (COMBINATION\ARGS FM)))))
                     (IF (NOT *TESTING*)
                         (REMPROP (NODE\NAME NODE) 'NODE)))))
```

META-EVALUATE is the top-level function of the optimizer. It accepts a node, and returns a node (not necessarily the same one) for an equivalent program.

The METAP flags in the nodes are used to control re-analysis. META-EVALUATE checks this flag first thing, and returns the given node immediately if its METAP flag is non-NIL, meaning the node has already been properly optimized. Otherwise it examines the node more carefully.

Some rules about the organization of the optimizer:
[1] A node returned by a call to META-EVALUATE will always have its METAP flag set.
[2] The descendants of a node must be meta-evaluated before any information in them is used.
[3] If a node has its METAP flag set, so do all of its descendants. Moreover, REANALYZE1 has been applied to the node, so all of the information filled in by pass-1 analysis (ENV-ANALYZE, TRIV-ANALYZE, and EFFS-ANALYZE) is up-to-date.

When COMPILE calls META-EVALUATE, all the METAP flags are NIL, and no pass-1 analysis has been performed. META-EVALUATE, roughly speaking, calls itself recursively, and meta-evaluates the node tree from the bottom up. After meta-evaluating all the descendants of a node, it applies REANALYZE1 to perform pass-1 analysis on that node, sets the METAP flag, and returns the node. Exceptions can be made to this discipline if a non-trivial optimization occurs.

If the (meta-evaluated) predicate part of an IF node is itself an IF node (and the debugging switch *FUDGE* is set), then META-IF-FUDGE is called. If it is a constant, then the value of the constant is used to select either the consequent CON or the alternative ALT. The other one is then erased, and the IF node is itself erased. The selected component node is then returned (it has already been meta-evaluated). The statistics counter *DEAD-COUNT* counts occurrences of this "dead code elimination" optimization.

The other two interesting cases are COMBINATION nodes whose function position contains either a trivial function or a LAMBDA node. META-COMBINATION-TRIVFN and META-COMBINATION-LAMBDA handle these respective cases.

```
;;; THE VALUE OF META-EVALUATE IS THE (POSSIBLY NEW) NODE RESULTING FROM THE GIVEN ONE.

(SET' *FUDGE* T)                                      ;SWITCH TO CONTROL META-IF-FUDGE
(SET' *DEAD-COUNT* 0)                                 ;COUNT OF DEAD-CODE ELIMINATIONS

(DEFINE META-EVALUATE
        (LAMBDA (NODE)
                (IF (NODE\METAP NODE)
                    NODE
                    (LET ((FM (NODE\FORM NODE)))
                         (EQCASE (TYPE FM)
                                 (CONSTANT
                                  (REANALYZE1 NODE)
                                  (ALTER-NODE NODE (METAP := T)))
                                 (VARIABLE
                                  (REANALYZE1 NODE)
                                  (ALTER-NODE NODE (METAP := T)))
                                 (LAMBDA
                                  (ALTER-LAMBDA FM (BODY := (META-EVALUATE (LAMBDA\BODY FM))))
                                  (REANALYZE1 NODE)
                                  (ALTER-NODE NODE (METAP := T)))
                                 (IF
                                  (ALTER-IF FM
                                            (PRED := (META-EVALUATE (IF\PRED FM)))
                                            (CON := (META-EVALUATE (IF\CON FM)))
                                            (ALT := (META-EVALUATE (IF\ALT FM))))
                                  (IF (AND *FUDGE* (EQ (TYPE (NODE\FORM (IF\PRED FM))) 'IF))
                                      (META-IF-FUDGE NODE)
                                      (IF (EQ (TYPE (NODE\FORM (IF\PRED FM))) 'CONSTANT)
                                          (LET ((CON (IF\CON FM))
                                                (ALT (IF\ALT FM))
                                                (VAL (CONSTANT\VALUE (NODE\FORM (IF\PRED FM)))))
                                               (ERASE-NODE NODE)
                                               (ERASE-ALL-NODES (IF\PRED FM))
                                               (INCREMENT *DEAD-COUNT*)
                                               (IF VAL
                                                   (BLOCK (ERASE-ALL-NODES ALT) CON)
                                                   (BLOCK (ERASE-ALL-NODES CON) ALT)))
                                          (BLOCK (REANALYZE1 NODE)
                                                 (ALTER-NODE NODE (METAP := T))))))
                                 (ASET
                                  (ALTER-ASET FM (BODY := (META-EVALUATE (ASET\BODY FM))))
                                  (REANALYZE1 NODE)
                                  (ALTER-NODE NODE (METAP := T)))
                                 (CATCH
                                  (ALTER-CATCH FM (BODY := (META-EVALUATE (CATCH\BODY FM))))
                                  (REANALYZE1 NODE)
                                  (ALTER-NODE NODE (METAP := T)))
                                 (LABELS
                                  (DO ((D (LABELS\FNDEFS FM) (CDR D)))
                                      ((NULL D))
                                      (RPLACA D (META-EVALUATE (CAR D))))
                                  (ALTER-LABELS FM (BODY := (META-EVALUATE (LABELS\BODY FM))))
                                  (REANALYZE1 NODE)
                                  (ALTER-NODE NODE (METAP := T)))
                                 (COMBINATION
                                  (LET ((FN (NODE\FORM (CAR (COMBINATION\ARGS FM)))))
                                       (COND ((AND (EQ (TYPE FN) 'VARIABLE)
                                                   (TRIVFN (VARIABLE\VAR FN)))
                                              (META-COMBINATION-TRIVFN NODE))
                                             ((EQ (TYPE FN) 'LAMBDA)
                                              (META-COMBINATION-LAMBDA NODE))
                                             (T (DO ((A (COMBINATION\ARGS FM) (CDR A)))
                                                    ((NULL A))
                                                    (RPLACA A (META-EVALUATE (CAR A))))
                                                (REANALYZE1 NODE)
                                                (ALTER-NODE NODE (METAP := T)))))))))))
```

For an IF nested within another IF, the transformation shown in the comment is performed. This involves constructing an S-expression of the appropriate form and then calling ALPHATIZE to convert it into a node-tree. (The node-tree could be constructed directly, but it is easier to call ALPHATIZE. This is the reason why ALPHATIZE merely returns a NODE if it encounters one in the S-expression; META-IF-FUDGE inserts various nodes in the S-expression it constructs.) The original two IF nodes are erased, a statistics counter *FUDGE-COUNT* is incremented, and the new expression is meta-evaluated and returned in place of the nested IF nodes.

(The statistics counter shows that this optimization is performed with modest frequency, arising from cases such as (IF (AND ...)...).)

META-COMBINATION-TRIVFN performs the standard recursive meta-evaluation of all the arguments, and then checks to see whether the combination can be "folded". This is possible all the arguments are constants, and if the function has no side effects and cannot be affected by side-effects, or has an OKAY-TO-FOLD property. If this is the case, the function is applied to the arguments, the combination node and its descendants are erased, the statistics counter *FOLD-COUNT* is bumped, and a new CONSTANT node containing the result is created and meta-evaluated. This might typically occur for (NOT NIL) => T, or (+ 3 4) => 7, or (MEMQ 'BAR '(FOO BAR BAZ)) => '(BAR BAZ). If this optimization is not permissible, then the usual reanalysis and setting of the METAP flag is performed.

(The statistics counter shows that even in a very large program such as RABBIT this optimization is performed fewer than a dozen times. This may be due to my programming style, or because there are very few macros in the code for RABBIT which might expand into foldable code.)

```
;;; TRANSFORM (IF (IF A B C) D E) INTO:
;;;    ((LAMBDA (D1 E1)
;;;             (IF A (IF B (D1) (E1)) (IF C (D1) (E1))))
;;;     (LAMBDA () D)
;;;     (LAMBDA () E))

(SET' *FUDGE-COUNT* 0)                                  ;COUNT OF IF-FUDGES

(DEFINE META-IF-FUDGE
        (LAMBDA (NODE)
                (LET ((FM (NODE\FORM NODE)))
                     (LET ((PFM (NODE\FORM (IF\PRED FM))))
                          (LET ((N (ALPHATIZE (LET ((CONVAR (GENTEMP 'META-CON))
                                                    (ALTVAR (GENTEMP 'META-ALT)))
                                                   "((LAMBDA (,CONVAR ,ALTVAR)
                                                              (IF ,(IF\PRED PFM)
                                                                  (IF ,(IF\CON PFM)
                                                                      (,CONVAR)
                                                                      (,ALTVAR))
                                                                  (IF ,(IF\ALT PFM)
                                                                      (,CONVAR)
                                                                      (,ALTVAR))))
                                                     (LAMBDA () ,(IF\CON FM))
                                                     (LAMBDA () ,(IF\ALT FM))))
                                              (NODE\ENV NODE))))       ;DOESN'T MATTER
                               (ERASE-NODE NODE)
                               (ERASE-NODE (IF\PRED FM))
                               (INCREMENT *FUDGE-COUNT*)
                               (META-EVALUATE N))))))

;;; REDUCE A COMBINATION WITH A SIDE-EFFECT-LESS TRIVIAL
;;; FUNCTION AND CONSTANT ARGUMENTS TO A CONSTANT.

(SET' *FOLD-COUNT* 0)                                   ;COUNT OF CONSTANT FOLDINGS

(DEFINE META-COMBINATION-TRIVFN
        (LAMBDA (NODE)
                (LET ((FM (NODE\FORM NODE)))
                     (LET ((ARGS (COMBINATION\ARGS FM)))
                          (RPLACA ARGS (META-EVALUATE (CAR ARGS)))
                          (DO ((A (CDR ARGS) (CDR A))
                               (CONSTP (LET ((FNNAME (VARIABLE\VAR (NODE\FORM (CAR ARGS)))))
                                            (OR (AND (EQ (GET FNNAME
                                                              'FN-SIDE-EFFECTS)
                                                         'NONE)
                                                     (EQ (GET FNNAME
                                                              'FN-SIDE-AFFECTED)
                                                         'NONE))
                                                (GET FNNAME 'OKAY-TO-FOLD)))
                                       (AND CONSTP (EQ (TYPE (NODE\FORM (CAR A))) 'CONSTANT))))
                              ((NULL A)
                               (COND (CONSTP
                                      (LET ((VAL (APPLY (VARIABLE\VAR (NODE\FORM (CAR ARGS)))
                                                        (AMAPCAR (LAMBDA (X)
                                                                         (CONSTANT\VALUE
                                                                          (NODE\FORM X)))
                                                                 (CDR ARGS)))))
                                           (ERASE-ALL-NODES NODE)
                                           (INCREMENT *FOLD-COUNT*)
                                           (META-EVALUATE (ALPHATIZE "(QUOTE ,VAL) NIL))))
                                     (T (REANALYZE1 NODE)
                                        (ALTER-NODE NODE (METAP := T)))))
                              (RPLACA A (META-EVALUATE (CAR A))))))))
```

META-COMBINATION-LAMBDA performs several interesting optimizations on combinations of the form ((LAMBDA ...) ...). It is controlled by several debugging switches, and keeps several statistics counters, which we will not describe further.

First all the arguments, but not the LAMBDA-expression, are meta-evaluated by the first DO loop. Next, the body of the LAMBDA node is meta-evaluated and kept in the variable B in the second DO loop. This loop iterates over the LAMBDA variables and the corresponding arguments. For each variable-argument pair, SUBST-CANDIDATE determines whether the argument can "probably" be legally substituted for occurrences of the variable in the body. If so, META-SUBSTITUTE is called to attempt such substitution. When the loop finishes, B has the body with all possible substitutions performed. This body is then re-meta-evaluated. (The reason for this is explained later in the discussion of META-SUBSTITUTE.)

Next an attempt is made to eliminate LAMBDA variables. A variable and its corresponding argument may be eliminated if the variable has no remaining references, and the argument either has no side effects or has been successfully substituted. (If an argument has side effects, then SUBST-CANDIDATE will give permission to attempt substitution only if no more than one reference to the corresponding variable exists. If the substitution fails, then the argument may not be eliminated, because its side effects must not be lost. It the substitution succeeds, then the argument must be eliminated, because the side effects must not be duplicated.) A consistency check ensures that in fact the variable is unreferenced within the body as determined by its REFS and ASETS slots; then the argument and variable are deleted, and the nodes of the argument are erased.

When all possible variable-argument pairs have been eliminated, then there are two cases. If the LAMBDA has no variables left, then the combination containing it can be replaced by the body of the LAMBDA node. In this case the LAMBDA and COMBINATION nodes are erased. Otherwise the LAMBDA and COMBINATION nodes are reanalyzed and their METAP flags are set.

(The statistics counters show that when RABBIT compiles itself these three optimizations are performed hundreds of times. This occurs because many standard macros make use of closures to ensure that variables local to the code for the macro do not conflict with user variables. These closures often can be substituted into the code by the compiler and eliminated.)

```
(SET' *FLUSH-ARGS* T)                                ;SWITCH TO CONTROL VARIABLE ELIMINATION
(SET' *FLUSH-COUNT* 0)                               ;COUNT OF VARIABLES ELIMINATED
(SET' *CONVERT-COUNT* 0)                             ;COUNT OF FULL BETA-CONVERSIONS

(DEFINE
 META-COMBINATION-LAMBDA
 (LAMBDA (NODE)
         (LET ((FM (NODE\FORM NODE)))
              (LET ((ARGS (COMBINATION\ARGS FM)))
                   (DO ((A (CDR ARGS) (CDR A)))
                       ((NULL A))
                       (RPLACA A (META-EVALUATE (CAR A)))
                       (ALTER-NODE (CAR A) (SUBSTP := NIL)))
                   (LET ((FN (NODE\FORM (CAR ARGS))))
                        (DO ((V (LAMBDA\VARS FN) (CDR V))
                             (A (CDR ARGS) (CDR A))
                             (B (META-EVALUATE (LAMBDA\BODY FN))
                                (IF (SUBST-CANDIDATE (CAR A) (CAR V) B)
                                    (META-SUBSTITUTE (CAR A) (CAR V) B)
                                    B)))
                            ((NULL V)
                             (ALTER-LAMBDA FN (BODY := (META-EVALUATE B)))
                             (DO ((V (LAMBDA\VARS FN) (CDR V))
                                  (A (CDR ARGS) (CDR A)))
                                 ((NULL A))
                                 (IF (AND *FLUSH-ARGS*
                                          (NULL (GET (CAR V) 'READ-REFS))
                                          (NULL (GET (CAR V) 'WRITE-REFS))
                                          (OR (EFFECTLESS-EXCEPT-CONS (NODE\EFFS (CAR A)))
                                              (NODE\SUBSTP (CAR A))))
                                     (BLOCK (IF (OR (MEMQ V (NODE\REFS (LAMBDA\BODY FN)))
                                                    (MEMQ V (NODE\ASETS (LAMBDA\BODY FN))))
                                                (ERROR '|Reanalysis lost - META-COMBINATION-LAMBDA|
                                                       NODE
                                                       'FAIL-ACT))
                                            (DELQ (CAR A) ARGS)
                                            (ERASE-ALL-NODES (CAR A))
                                            (INCREMENT *FLUSH-COUNT*)
                                            (ALTER-LAMBDA FN
                                                          (VARS := (DELQ (CAR V) (LAMBDA\VARS FN)))
                                                          (UVARS := (DELQ (GET (CAR V) 'USER-NAME)
                                                                          (LAMBDA\UVARS FN)))))))
                             (COND ((NULL (LAMBDA\VARS FN))
                                    (OR (NULL (CDR ARGS))
                                        (ERROR '|Too many args in META-COMBINATION-LAMBDA|
                                               NODE
                                               'FAIL-ACT))
                                    (LET ((BOD (LAMBDA\BODY FN)))
                                         (ERASE-NODE (CAR ARGS))
                                         (ERASE-NODE NODE)
                                         (INCREMENT *CONVERT-COUNT*)
                                         BOD))
                                   (T (REANALYZE1 (CAR ARGS))
                                      (ALTER-NODE (CAR ARGS) (METAP := T))
                                      (REANALYZE1 NODE)
                                      (ALTER-NODE NODE (METAP := T))))))))))
```

(SUBST-CANDIDATE ARG VAR BOD) is a predicate which is true iff it is apparently legal to attempt to substitute the argument ARG for the variable VAR in the body BOD. This predicate is very conservative, because there is no provision for backing out of a bad choice. The decision is made on this basis:
[1] There must be no ASET references to the variable. (This is overly restrictive, but is complicated to check for correctly, and makes little difference in practice.)
[2] One of three conditions must hold:
 [2a] There is at most one reference to the variable. (Code with possible side effects must not be duplicated. Exceptions occur, for example, if there are two references, one in each branch of an IF, so that only one can be executed. This is hard to detect, and relaxing this restriction is probably not worthwhile.)
 [2b] The argument is a constant or variable. (This is always safe because the cost of a constant or variable is no worse than the cost of referencing the variable it replaces.)
 [2c] The argument is a LAMBDA-expression, and either:
  [2c1] There is no more than one reference. (This is tested again because of the presence of debugging switches in SUBST-CANDIDATE which can control various tests independently to help localize bugs.)
  [2c2] The body of the LAMBDA-expression is a combination, all of whose descendants are constants or variables, and the number of arguments of the combination (not counting the function) does not exceed the number of arguments taken by the LAMBDA-expression. (The idea here is that substitution of the LAMBDA-expression into function position of some combination will later allow reduction to a combination which is no worse than the original one. This test is a poor heuristic if references to the variable VAR occur in other than function position within BOD, because then several closures will be made instead of one, but is very good for code typically produced by the expansion of macros. In retrospect, perhaps ENV-ANALYZE should maintain a third property besides READ-REFS and WRITE-REFS called, say, NON-FN-REFS. This would be the subset of READ-REFS which occur in other than function position of a combination. SUBST-CANDIDATE could then use this information. Alternatively, META-SUBSTITUTE could, as it walked the node-tree of the body, keep track of whether a variable was encountered in function position, and refuse to substitute a LAMBDA-expression for a variable not in such a position which had more than one reference. This might in turn prevent other optimizations, however.)

```
(SET' *SUBSTITUTE* T)           ;SWITCH TO CONTROL SUBSTITUTION
(SET' *SINGLE-SUBST* T)         ;SWITCH TO CONTROL SUBSTITUTION OF EXPRESSIONS WITH SIDE EFFECTS
(SET' *LAMBDA-SUBST* T)         ;SWITCH TO CONTROL SUBSTITUTION OF LAMBDA-EXPRESSIONS

(DEFINE SUBST-CANDIDATE
        (LAMBDA (ARG VAR BOD)
                (AND *SUBSTITUTE*
                     (NOT (GET VAR 'WRITE-REFS))        ;BE PARANOID FOR NOW
                     (OR (AND *SINGLE-SUBST*
                              (NULL (CDR (GET VAR 'READ-REFS))))
                         (MEMQ (TYPE (NODE\FORM ARG)) '(CONSTANT VARIABLE))
                         (AND *LAMBDA-SUBST*
                              (EQ (TYPE (NODE\FORM ARG)) 'LAMBDA)
                              (OR (NULL (CDR (GET VAR 'READ-REFS)))
                                  (LET ((B (NODE\FORM (LAMBDA\BODY (NODE\FORM ARG)))))
                                       (OR (MEMQ (TYPE B) '(CONSTANT VARIABLE))
                                           (AND (EQ (TYPE B) 'COMBINATION)
                                                (NOT (> (LENGTH (CDR (COMBINATION\ARGS B)))
                                                        (LENGTH (LAMBDA\VARS (NODE\FORM ARG)))))
                                                (DO ((A (COMBINATION\ARGS B) (CDR A))
                                                     (P T (AND P (MEMQ (TYPE (NODE\FORM (CAR A)))
                                                                       '(CONSTANT VARIABLE)))))
                                                    ((NULL A) P)))))))))))
```

REANALYZE1 calls PASS1-ANALYZE on the given node. The argument T means that optimization is in effect, and so EFFS-ANALYZE must be invoked after ENV-ANALYZE and TRIV-ANALYZE (EFFS-ANALYZE information is used only by the optimizer). The argument *REANALYZE* specifies whether reanalysis should be forced to all descendant nodes, or whether reanalysis of the current node will suffice. This variable normally contains the symbol ONCE, meaning reanalyze only the current node. META-EVALUATE normally ensures, before analyzing a node, that all descendant nodes are analyzed. Thus the initial pass-1 analysis occurs incrementally, interleaved with the meta-evaluation process.

The switch *REANALYZE* may be set to the symbol ALL to force all descendants of a node to be reanalyzed before analyzing the node itself. This ability is provided to test for certain bugs in the optimizer. If the incremental analysis should fail for some reason, then the descendant nodes may not contain correct information (for example, their information slots may be empty!). The ALL setting ensures that a consistent analysis is obtained. If the optimizer's behavior differs depending on whether *REANALYZE* contains ONCE or ALL, then a problem with the incremental analysis is implicated. This switch has been very useful for isolating such bugs.

The next group of functions are utilities for META-SUBSTITUTE which deal with sets of side-effects.

EFFS-INTERSECT takes the intersection of two sets of side-effects. It is just like INTERSECT, except that it also knows about the two special sets ANY and NONE.

EFFECTLESS is a predicate which is true of an empty set of side-effects.

EFFECTLESS-EXCEPT-CONS is a predicate true of a set of side-effects which is empty except possibly for the CONS side-effect.

PASSABLE takes a node and two sets of side-effects, which should be the EFFS and AFFD sets from some other node. PASSABLE is a predicate which is true if the given node, which originally preceded the second in the standard evaluation order, can legitimately be postponed until after the second is evaluated. That is, it is true iff the first node can "pass" the second during the substitution process.

```
(DEFINE REANALYZE1
        (LAMBDA (NODE)
                (PASS1-ANALYZE NODE *REANALYZE* T)))

(SET' *REANALYZE* 'ONCE)



;;; HERE WE DETERMINE, FOR EACH VARIABLE NODE WHOSE VAR IS THE ONE
;;; GIVEN, WHETHER IT IS POSSIBLE TO SUBSTITUTE IN FOR IT; THIS IS
;;; DETERMINED ON THE BASIS OF SIDE EFFECTS.  THIS IS DONE BY
;;; WALKING THE PROGRAM, STOPPING WHEN A SIDE-EFFECT BLOCKS IT.
;;; A SUBSTITUTION IS MADE IFF IS VARIABLE NODE IS REACHED IN THE WALK.

;;; THERE IS A BUG IN THIS THEORY TO THE EFFECT THAT A CATCH
;;; WHICH RETURNS MULTIPLY CAN CAUSE AN EXPRESSION EXTERNAL
;;; TO THE CATCH TO BE EVALUATED TWICE.  THIS IS A DYNAMIC PROBLEM
;;; WHICH CANNOT BE RESOLVED AT COMPILE TIME, AND SO WE SHALL
;;; IGNORE IT FOR NOW.

;;; WE ALSO RESET THE METAP FLAG ON ALL NODES WHICH HAVE A
;;; SUBSTITUTION AT OR BELOW THEM, SO THAT THE META-EVALUATOR WILL
;;; RE-PENETRATE TO SUBSTITUTION POINTS, WHICH MAY ADMIT FURTHER
;;; OPTIMIZATIONS.


(DEFINE EFFS-INTERSECT
        (LAMBDA (A B)
                (COND ((EQ A 'ANY) B)
                      ((EQ B 'ANY) A)
                      ((EQ A 'NONE) A)
                      ((EQ B 'NONE) B)
                      (T (INTERSECT A B)))))

(DEFINE EFFECTLESS
        (LAMBDA (X) (OR (NULL X) (EQ X 'NONE))))

(DEFINE EFFECTLESS-EXCEPT-CONS
        (LAMBDA (X) (OR (EFFECTLESS X) (EQUAL X '(CONS)))))

(DEFINE PASSABLE
        (LAMBDA (NODE EFFS AFFD)
                (BLOCK (IF (EMPTY (NODE\EFFS NODE))
                           (ERROR '|Pass 1 Analysis Missing - PASSABLE|
                                  NODE
                                  'FAIL-ACT))
                       (AND (EFFECTLESS (EFFS-INTERSECT EFFS (NODE\AFFD NODE)))
                            (EFFECTLESS (EFFS-INTERSECT AFFD (NODE\EFFS NODE)))
                            (EFFECTLESS-EXCEPT-CONS (EFFS-INTERSECT EFFS (NODE\EFFS NODE)))))))
```

META-SUBSTITUTE takes a node-tree ARG, a variable name VAR, and another node-tree BOD, and wherever possible substitutes copies of ARG for occurrences of VAR within BOD. The complexity of this process is due almost entirely to the necessity of determining the extent of "wherever possible".

META-SUBSTITUTE merely spreads out the EFFS and AFFD slots of ARG to make them easy to refer to, makes an error check, and then passes the buck to the internal LABELS routine SUBSTITUTE, which does the real work.

SUBSTITUTE recurs over the structure of the node-tree. At each node it first checks to see whether VAR is in the REFS set of that node. This is purely an efficiency hack: if VAR is not in the set, then it cannot occur anywhere below that node in the tree, and so SUBSTITUTE can save itself the work of a complete recursive search of that portion of the node-tree.

SUBSTITUTE plays another efficiency trick in cahoots with META-EVALUATE to save work. Whenever SUBSTITUTE actually replaces an occurrence of VAR with a copy of ARG, the copy of ARG will have its METAP flag turned off (set to NIL). Now SUBSTITUTE propagates the METAP flag back up the node-tree; when all sub-nodes of a node have had SUBSTITUTE applied to them, then if the METAP flag of the current node is still set, it is set to the AND of the flags of the subnodes. Thus any node below which a substitution has occurred will have its METAP flag reset. More to the point, any node which after the substitution still has its METAP flag set has had no substitutions occur below it. META-EVALUATE can then be applied to BOD after all substitutions have been tried (this occurs in META-COMBINATION-LAMBDA), and META-EVALUATE will only have to re-examine those parts of BOD which have changed. In particular, if no substitutions were successful, META-EVALUATE will not have to re-examine BOD at all.

If the variable is referenced at or below the node, it breaks down into cases according to the type of the node.

For a CONSTANT, no action is necessary.

For a VARIABLE, no action is taken unless the variable matches VAR, in which case the node is erased and a copy of ARG is made and returned in its place. The SUBSTP slot of the original ARG is set as a flag to META-COMBINATION-LAMBDA (q.v.), to let it know that at least one substitution succeeded.

For a LAMBDA, substitution can occur in the body only if ARG has no side-effects except possibly CONS. This is because evaluation of the LAMBDA-expression (to produce a closure) will not necessarily cause evaluation of the side-effect in ARG at the correct time. The special case of a LAMBDA occurring as the function in a COMBINATION is handled separately below.

For an IF, substitution is attempted in the predicate. It is attempted in the other two sub-trees only if ARG can pass the predicate.

For an ASET' or a CATCH, substitution is attempted in the body. The same is true of LABELS, but substitution is also attempted in the labelled function definitions.

```
(SET' *SUBST-COUNT* 0)                             ;COUNT OF SUBSTITUTIONS
(SET' *LAMBDA-BODY-SUBST* T)                       ;SWITCH TO CONTROL SUBSTITUTION IN LAMBDA BODIES
(SET' *LAMBDA-BODY-SUBST-TRY-COUNT* 0)             ;COUNT THEREOF - TRIES
(SET' *LAMBDA-BODY-SUBST-SUCCESS-COUNT* 0)         ;COUNT THEREOF - SUCCESSES


(DEFINE
 META-SUBSTITUTE
 (LAMBDA
  (ARG VAR BOD)
  (LET ((EFFS (NODE\EFFS ARG))
        (AFFD (NODE\AFFD ARG)))
       (IF (EMPTY EFFS)
           (ERROR '|Pass 1 Analysis Screwed Up - META-SUBSTITUTE| ARG 'FAIL-ACT))
       (LABELS
        ((SUBSTITUTE
          (LAMBDA (NODE)
                  (IF (OR (EMPTY (NODE\REFS NODE))
                          (NOT (MEMQ VAR (NODE\REFS NODE))))    ;EFFICIENCY HACK
                      NODE
                      (LET ((FM (NODE\FORM NODE)))
                           (EQCASE (TYPE FM)
                                   (CONSTANT NODE)
                                   (VARIABLE
                                    (IF (EQ (VARIABLE\VAR FM) VAR)
                                        (BLOCK (ERASE-ALL-NODES NODE)
                                               (INCREMENT *SUBST-COUNT*)
                                               (ALTER-NODE ARG (SUBSTP := T))
                                               (COPY-CODE ARG))
                                        NODE))
                                   (LAMBDA
                                    (IF (AND (EFFECTLESS-EXCEPT-CONS EFFS) (EFFECTLESS AFFD))
                                         (ALTER-LAMBDA FM (BODY := (SUBSTITUTE (LAMBDA\BODY FM)))))
                                    (IF (NODE\METAP NODE)
                                        (ALTER-NODE NODE (METAP := (NODE\METAP (LAMBDA\BODY FM)))))
                                    NODE)
                                   (IF
                                    (ALTER-IF FM (PRED := (SUBSTITUTE (IF\PRED FM))))
                                    (IF (PASSABLE (IF\PRED FM) EFFS AFFD)
                                        (ALTER-IF FM
                                                  (CON := (SUBSTITUTE (IF\CON FM)))
                                                  (ALT := (SUBSTITUTE (IF\ALT FM)))))
                                    (IF (NODE\METAP NODE)
                                        (ALTER-NODE NODE
                                                    (METAP := (AND (NODE\METAP (IF\PRED FM))
                                                                   (NODE\METAP (IF\CON FM))
                                                                   (NODE\METAP (IF\ALT FM))))))
                                    NODE)
                                   (ASET
                                    (ALTER-ASET FM (BODY := (SUBSTITUTE (ASET\BODY FM))))
                                    (IF (NODE\METAP NODE)
                                        (ALTER-NODE NODE (METAP := (NODE\METAP (ASET\BODY FM)))))
                                    NODE)
                                   (CATCH
                                    (ALTER-CATCH FM (BODY := (SUBSTITUTE (CATCH\BODY FM))))
                                    (IF (NODE\METAP NODE)
                                        (ALTER-NODE NODE (METAP := (NODE\METAP (CATCH\BODY FM)))))
                                    NODE)
                                   (LABELS
                                    (ALTER-LABELS FM (BODY := (SUBSTITUTE (LABELS\BODY FM))))
                                    (DO ((D (LABELS\FNDEFS FM) (CDR D))
                                         (MP (NODE\METAP (LABELS\BODY FM))
                                             (AND MP (NODE\METAP (CAR D)))))
                                        ((NULL D)
                                         (IF (NODE\METAP NODE)
                                             (ALTER-NODE NODE (METAP := MP))))
                                        (RPLACA D (SUBSTITUTE (CAR D))))
                                    NODE)
```

The most complicated case is the COMBINATION. First it is determined (in the variable X) whether ARG can correctly pass *all* of the arguments of the combination. (It is not possible to substitute into *any* argument unless *all* can be passed, because at this time it has not been decided in what order to evaluate them. This decision is the free choice of CONVERT-COMBINATION below.) If it can, then substitution is attempted in all of the arguments except the function itself. Then two kinds of function are distinguished. If it is not a LAMBDA, a straightforward recursive call to SUBSTITUTE is used. If it is, then substitution is attempted in the *body* of the LAMBDA (*not* in the LAMBDA itself; substitution in a LAMBDA requires that ARG be EFFECTLESS-EXCEPT-CONS, but in this special case we know that the LAMBDA-expression will be invoked immediately, and so it is all right if ARG has side-effects).

```
                            (COMBINATION
                             (LET ((ARGS (COMBINATION\ARGS FM)))
                                  (DO ((A ARGS (CDR A))
                                       (X T (AND X (PASSABLE (CAR A) EFFS AFFD))))
                                      ((NULL A)
                                       (IF X (DO ((A (CDR ARGS) (CDR A)))
                                                 ((NULL A))
                                                 (RPLACA A (SUBSTITUTE (CAR A)))))
                                       (IF (AND *LAMBDA-BODY-SUBST*
                                                (EQ (TYPE (NODE\FORM (CAR ARGS))) 'LAMBDA))
                                           (LET ((FN (NODE\FORM (CAR ARGS))))
                                                (INCREMENT *LAMBDA-BODY-SUBST-TRY-COUNT*)
                                                (COND (X
                                                       (INCREMENT
                                                        *LAMBDA-BODY-SUBST-SUCCESS-COUNT*)
                                                       (ALTER-LAMBDA
                                                        FN
                                                        (BODY := (SUBSTITUTE
                                                                  (LAMBDA\BODY FN))))))
                                                (IF (NODE\METAP (CAR ARGS))
                                                    (ALTER-NODE
                                                     (CAR ARGS)
                                                     (METAP := (NODE\METAP
                                                                (LAMBDA\BODY FN))))))
                                           (IF X (RPLACA ARGS (SUBSTITUTE (CAR ARGS)))))))
                                  (DO ((A ARGS (CDR A))
                                       (MP T (AND MP (NODE\METAP (CAR A)))))
                                      ((NULL A)
                                       (IF (NODE\METAP NODE)
                                           (ALTER-NODE NODE (METAP := MP))))))
                             NODE)))))))
(SUBSTITUTE BOD)))))
```

COPY-CODE is used by META-SUBSTITUTE to make copies of node-trees representing code. It invokes COPY-NODES with appropriate additional arguments.

COPY-NODES does the real work. The argument ENV is analogous to the argument ENV taken by ALPHATIZE. However, variables are not looked-up in ENV by COPY-NODES; ENV is maintained only to install in the new nodes for debugging purposes. The argument RNL is a "rename list" for variables. When a node is copied which binds variables, new variables are created for the copy. RNL provides a mapping from generated names in the original code to generated names in the copy (as opposed to ENV, which maps user names to generated names in the copy). Thus, when a LAMBDA node is copied, new names are generated, and PAIRLIS is used to pair new names with the LAMBDA\VARS of the old node, adding the new pairs to RNL.

A neat trick to aid debugging is that the new names are generated by using the old names as the arguments to GENTEMP. In this way the name of a generated variable contains a history of how it was created. For example, VAR-34-73-156 was created by copying the LAMBDA node which bound VAR-34-73, which in turn was copied from the node which bound VAR-34. Copies of CATCH and LABELS variables are generated in the same way.

The large EQCASE handles the different types of nodes. The result is then given to NODIFY, the same routine which creates nodes for ALPHATIZE. Recall that NODIFY initializes the METAP slot to NIL; the next meta-evaluation which comes along will cause pass-1 analysis to be performed on the new copies.

Note particularly that the UVARS list of a LAMBDA node is copied, for the same reason that ALPHA-LAMBDA makes a copy: META-COMBINATION-LAMBDA may alter it destructively.

```
(DEFINE COPY-CODE
        (LAMBDA (NODE)
                (REANALYZE1 (COPY-NODES NODE (NODE\ENV NODE) NIL))))

(DEFINE
 COPY-NODES
 (LAMBDA (NODE ENV RNL)
         (NODIFY
          (LET ((FM (NODE\FORM NODE)))
               (EQCASE (TYPE FM)
                       (CONSTANT
                        (CONS-CONSTANT (VALUE = (CONSTANT\VALUE FM))))
                       (VARIABLE
                        (CONS-VARIABLE (VAR = (LET ((SLOT (ASSQ (VARIABLE\VAR FM) RNL)))
                                                   (IF SLOT (CADR SLOT) (VARIABLE\VAR FM))))
                                       (GLOBALP = (VARIABLE\GLOBALP FM))))
                       (LAMBDA
                        (LET ((VARS (AMAPCAR GENTEMP (LAMBDA\VARS FM))))
                             (CONS-LAMBDA (UVARS = (APPEND (LAMBDA\UVARS FM) NIL))
                                          (VARS = VARS)
                                          (BODY = (COPY-NODES
                                                   (LAMBDA\BODY FM)
                                                   (PAIRLIS (LAMBDA\UVARS FM) VARS ENV)
                                                   (PAIRLIS (LAMBDA\VARS FM) VARS RNL))))))
                       (IF (CONS-IF (PRED = (COPY-NODES (IF\PRED FM) ENV RNL))
                                    (CON = (COPY-NODES (IF\CON FM) ENV RNL))
                                    (ALT = (COPY-NODES (IF\ALT FM) ENV RNL))))
                       (ASET
                        (CONS-ASET (VAR = (LET ((SLOT (ASSQ (ASET\VAR FM) RNL)))
                                               (IF SLOT (CADR SLOT) (ASET\VAR FM))))
                                   (GLOBALP = (ASET\GLOBALP FM))
                                   (BODY = (COPY-NODES (ASET\BODY FM) ENV RNL))))
                       (CATCH
                        (LET ((VAR (GENTEMP (CATCH\VAR FM)))
                              (UVAR (CATCH\UVAR FM)))
                             (CONS-CATCH (UVAR = (CATCH\UVAR FM))
                                         (VAR = VAR)
                                         (BODY = (COPY-NODES
                                                  (CATCH\BODY FM)
                                                  (CONS (LIST UVAR VAR) ENV)
                                                  (CONS (LIST (CATCH\VAR FM) VAR) RNL))))))
                       (LABELS
                        (LET ((FNVARS (AMAPCAR GENTEMP (LABELS\FNVARS FM))))
                             (LET ((LENV (PAIRLIS (LABELS\UFNVARS FM) FNVARS ENV))
                                   (LRNL (PAIRLIS (LABELS\FNVARS FM) FNVARS RNL)))
                                  (CONS-LABELS (UFNVARS = (LABELS\UFNVARS FM))
                                               (FNVARS = FNVARS)
                                               (FNDEFS = (AMAPCAR
                                                          (LAMBDA (N) (COPY-NODES N LENV LRNL))
                                                          (LABELS\FNDEFS FM)))
                                               (BODY = (COPY-NODES (LABELS\BODY FM)
                                                                   LENV
                                                                   LRNL))))))
                       (COMBINATION
                        (CONS-COMBINATION (ARGS = (AMAPCAR (LAMBDA (N) (COPY-NODES N ENV RNL))
                                                           (COMBINATION\ARGS FM)))
                                          (WARNP = (COMBINATION\WARNP FM))))))
          (NODE\SEXPR NODE)
          ENV)))
```

The next several functions process the node-tree produced, analyzed, and optimized by pass 1, converting it to another representation. This new representation is a tree structure very similar to the node-tree, but has different components for the pass-2 analysis. We will call this the "cnode-tree". The "c" stands for "Continuation-passing style": for the conversion process transforms the node-tree into a form which uses continuation-passing to represent the control and data flow within the program.

We define a new collection of data types used to construct cnode-trees. The CNODE data type is analogous to the NODE data type; one component CFORM contains a variant structure which is specific to the programming construct represented by the CNODE.

The types CVARIABLE, CLAMBDA, CIF, CASET, CLABELS, and CCOMBINATION correspond roughly to their non-C counterparts in pass 1.

Type TRIVIAL is used to represent pieces of code which were designated trivial in pass 1 (TRIVP slot = T) by TRIV-ANALYZE; the NODE component is simply the pass-1 node-tree for the trivial code. This is the only case in which part of the pass-1 node-tree survives the conversion process to be used in pass 2.

A CONTINUATION is just like a CLAMBDA except that it has only one bound variable VAR. This variable can never appear in a CASET, and so the CONTINUATION type has no ASETVARS slot; all other slots are similar to those in a CLAMBDA structure.

A RETURN structure is just like a CCOMBINATION, except that whereas a CCOMBINATION may invoke a CLAMBDA which may take any number of arguments, a RETURN may invoke only a CONTINUATION on a single value. Thus, in place of the ARGS slot of a CCOMBINATION, which is a list of cnodes, a RETURN has two slots CONT and VAL, each of which is a cnode.

(In retrospect, this was somewhat of a design error. The motivation was that the world of closures could be dichotomized into LAMBDA-closures and continuation-closures, as a result of the fundamental semantics of the language: one world is used to pass values "down" into functions, and the other to pass values "up" from functions. Combinations can similarly be dichotimized, and I thought it would be useful to reflect this distinction in the data types to enforce and error-check this dichotomy. However, as it turned out, there is a great deal of code in pass 2 which had to be written twice, once for each "world", because the data types involved were different. It would be better to have a single structure for both CLAMBDA and CONTINUATION, with an additional slot flagging which kind it was. Then most code in pass 2 could operate on this structure without regard for which "world" it belonged to, and code which cared could check the flag.)

```
;;; CONVERSION TO CONTINUATION-PASSING STYLE

;;; THIS INVOLVES MAKING A COMPLETE COPY OF THE PROGRAM IN TERMS
;;; OF THE FOLLOWING NEW DATA STRUCTURES:

(DEFTYPE CNODE (ENV REFS CLOVARS CFORM))
        ;ENV     ENVIRONMENT (A LIST OF VARIABLES, NOT A MAPPING; DEBUGGING ONLY)
        ;REFS    VARIABLES BOUND ABOVE AND REFERENCED BELOW THIS CNODE
        ;CLOVARS  VARIABLES REFERRED TO AT OR BELOW THIS CNODE BY CLOSURES
        ;        (SHOULD BE A SUBSET OF REFS)
        ;CFORM   ONE OF THE BELOW TYPES
(DEFTYPE TRIVIAL (NODE))
        ;NODE    A PASS-1 NODE TREE
(DEFTYPE CVARIABLE (VAR))
        ;VAR     GENERATED VARIABLE NAME
(DEFTYPE CLAMBDA (VARS BODY FNP TVARS NAME DEP MAXDEP CONSENV CLOSEREFS ASETVARS))
        ;FNP     NON-NIL => NEEDN'T MAKE A FULL CLOSURE OF THIS
        ;        CLAMBDA.  MAY BE 'NOCLOSE OR 'EZCLOSE (THE FORMER
        ;        MEANING NO CLOSURE IS NECESSARY AT ALL, THE LATTER
        ;        THAT THE CLOSURE IS MERELY THE ENVIRONMENT).
        ;TVARS   THE VARIABLES WHICH ARE PASSED THROUGH TEMP LOCATIONS
        ;        ON ENTRY.  NON-NIL ONLY IF FNP='NOCLOSE; THEN IS
        ;        NORMALLY THE LAMBDA VARS, BUT MAY BE DECREASED
        ;        TO ACCOUNT FOR ARGS WHICH ARE THEMSELVES KNOWN NOCLOSE'S,
        ;        OR WHOSE CORRESPONDING PARAMETERS ARE NEVER REFERENCED.
        ;        THE TEMP VARS INVOLVED START IN NUMBER AT DEP.
        ;NAME    THE PROG TAG USED TO LABEL THE FINAL OUTPUT CODE FOR THE CLAMBDA
        ;DEP     DEPTH OF TEMPORARY REGISTER USAGE WHEN THE CLAMBDA IS INVOKED
        ;MAXDEP MAXIMUM DEPTH OF REGISTER USAGE WITHIN CLAMBDA BODY
        ;CONSENV  THE "CONSED ENVIRONMENT" WHEN THE CLAMBDA IS EVALUATED
        ;CLOSEREFS  VARIABLES REFERENCED BY THE CLAMBDA WHICH ARE NOT IN
        ;        THE CONSED ENVIRONMENT AT EVALUATION TIME, AND SO MUST BE
        ;        ADDED TO CONSENV AT THAT POINT TO MAKE THE CLOSURE
        ;ASETVARS  THE ELEMENTS OF VARS WHICH ARE EVER SEEN IN A CASET
(DEFTYPE CONTINUATION (VAR BODY FNP TVARS NAME DEP MAXDEP CONSENV CLOSEREFS))
        ;COMPONENTS ARE AS FOR CLAMBDA
(DEFTYPE CIF (PRED CON ALT))
(DEFTYPE CASET (CONT VAR BODY))
(DEFTYPE CLABELS (FNVARS FNDEFS FNENV EASY CONSENV BODY))
        ;FNENV  A LIST OF VARIABLES TO CONS ONTO THE ENVIRONMENT BEFORE
        ;       CREATING THE CLOSURES AND EXECUTING THE BODY
        ;EASY   NON-NIL IFF NO LABELED FUNCTION IS REFERRED TO
        ;       AS A VARIABLE.  CAN BE 'NOCLOSE OR 'EZCLOSE
        ;       (REFLECTING THE STATUS OF ALL THE LABELLED FUNCTIONS)
        ;CONSENV  AS FOR CLAMBDA
(DEFTYPE CCOMBINATION (ARGS))
        ;ARGS   LIST OF CNODES REPRESENTING ARGUMENTS
(DEFTYPE RETURN (CONT VAL))
        ;CONT   CNODE FOR CONTINUATION
        ;VAL    CNODE FOR VALUE
```

CNODIFY is for cnode-trees what NODIFY was for node-trees. It takes a CFORM and wraps a CNODE structure around it.

CONVERT is the main function of the conversion process; it is invoked by COMPILE on the result (META-VERSION) of pass 1. CONVERT dispatches on the type of node to be converted, often calling some specialist which may call it back recursively to convert subnodes. CONT may be a cnode, or NIL. If it is a cnode, then that cnode is the code for a continuation which is to receive as value that produced by the code to be converted. That is, when CONVERT finishes producing code for the given node (the first argument to CONVERT), then in effect a RETURN is created which causes the value of the generated code to be returned to the code represented by CONT (the second argument to CONVERT). Sometimes this RETURN cnode is created explicitly (as for CONSTANT and VARIABLE nodes), and sometimes only implicitly, by passing CONT down to a specialist converter.

MP is T if optimization was performed by pass 1, and NIL otherwise. This argument is for debugging purposes only: CONVERT compares this to the METAP slot of the pass-1 nodes in order to detect any failures of the incremental optimization and analysis process. CONVERT also makes some other consistency checks.

```
(DEFINE CNODIFY
        (LAMBDA (CFORM)
                (CONS-CNODE (CFORM = CFORM))))

(DEFINE CONVERT
        (LAMBDA (NODE CONT MP)
                (LET ((FM (NODE\FORM NODE)))
                     (IF (EMPTY (NODE\TRIVP NODE))
                         (ERROR '|Pass 1 analysis missing| NODE 'FAIL-ACT))
                     (OR (EQ (NODE\METAP NODE) MP)
                         (ERROR '|Meta-evaluation Screwed Up METAP| NODE 'FAIL-ACT))
                     (EQCASE (TYPE FM)
                            (CONSTANT
                             (OR (NODE\TRIVP NODE)
                                 (ERROR '|Non-trivial Constant| NODE 'FAIL-ACT))
                             (MAKE-RETURN (CONS-TRIVIAL (NODE = NODE)) CONT))
                            (VARIABLE
                             (OR (NODE\TRIVP NODE)
                                 (ERROR '|Non-trivial Variable| 'FAIL-ACT))
                             (MAKE-RETURN (CONS-TRIVIAL (NODE = NODE)) CONT))
                            (LAMBDA (MAKE-RETURN (CONVERT-LAMBDA-FM NODE NIL MP) CONT))
                            (IF (OR CONT (ERROR '|Null Continuation to IF| NODE 'FAIL-ACT))
                                (CONVERT-IF NODE FM CONT MP))
                            (ASET (OR CONT (ERROR '|Null Continuation to ASET| NODE 'FAIL-ACT))
                                  (CONVERT-ASET NODE FM CONT MP))
                            (CATCH (OR CONT (ERROR '|Null Continuation to CATCH| NODE 'FAIL-ACT))
                                   (CONVERT-CATCH NODE FM CONT MP))
                            (LABELS (OR CONT (ERROR '|Null Continuation to LABELS| NODE 'FAIL-ACT))
                                    (CONVERT-LABELS NODE FM CONT MP))
                            (COMBINATION (OR CONT (ERROR '|Null Continuation to Combination|
                                                         NODE
                                                         'FAIL-ACT))
                                         (CONVERT-COMBINATION NODE FM CONT MP))))))
```

MAKE-RETURN takes a CFORM (one of the types TRIVIAL, CVARIABLE, ...) and a continuation, and constructs an appropriate returning of the value of the CFORM to the continuation. First the CFORM is given to CNODIFY. If the continuation is in fact NIL (meaning none), this new cnode is returned; otherwise a RETURN cnode is constructed.

CONVERT-LAMBDA-FM takes a LAMBDA node and converts it into a CLAMBDA cnode. The two are isomorphic, except that an extra variable is introduced as an extra first parameter to the CLAMBDA. Conceptually, this variable will be bound to a continuation when the CLAMBDA is invoked at run time; this continuation is the one intended to receive the value of the body of the CLAMBDA. This is accomplished by creating a new variable name CONT-nnn, which is added into the lambda variables. A new CVARIABLE node is made from it, and given to CONVERT as the continuation when the body of the LAMBDA node is to be recursively converted.

The CNAME argument is used for a special optimization trick by CONVERT-COMBINATION, described below.

CONVERT-IF distinguishes several cases, to simplify the converted code. First, if the entire IF node is trivial, then a simple CTRIVIAL node may be created for it. Otherwise, the general strategy is to generate code which will bind the given continuation to a variable and evaluate the predicate. This predicate receives a continuation which will examine the resulting value (with a CIF), and then perform either the consequent or alternative, which are converted using the bound variable as the continuation. The reason that the original continuation is bound to a variable is because it would be duplicated by using it for two separate calls to CONVERT, thereby causing duplicate code to be generated for it. A schematic picture of the general strategy is:

```
NODE = (IF a b c)   and   CONT = k   becomes

((CLAMBDA (q)
          (RETURN (CONTINUATION (p)
                                (CIF p
                                     (RETURN q b)
                                     (RETURN q c)))
                  a))
 k)
```

Now there are two special cases which allow simplification. First, if the given continuation is already a cvariable, there is no point in creating a new one to bind it to. This eliminates the outer CCOMBINATION and CLAMBDA. Second, if the predicate a is trivial (but the whole IF is not, because the consequent b or the alternative c is non-trivial), then the CONTINUATION which binds p is unnecessary.

```
(DEFINE MAKE-RETURN
        (LAMBDA (CFORM CONT)
                (LET ((CN (CNODIFY CFORM)))
                     (IF CONT
                         (CNODIFY (CONS-RETURN (CONT = CONT) (VAL = CN)))
                         CN))))

(DEFINE CONVERT-LAMBDA-FM
        (LAMBDA (NODE CNAME MP)
                (LET ((CV (GENTEMP 'CONT))
                      (FM (NODE\FORM NODE)))
                     (CONS-CLAMBDA (VARS = (CONS CV (LAMBDA\VARS FM)))
                                   (BODY = (CONVERT (LAMBDA\BODY FM)
                                                    (CNODIFY
                                                     (CONS-CVARIABLE (VAR = (OR CNAME CV))))
                                                    MP))))))

;;; ISSUES FOR CONVERTING IF:
;;; (1) IF WHOLE IF IS TRIVIAL, MAY JUST CREATE A CTRIVIAL.
;;; (2) IF CONTINUATION IS NON-CVARIABLE, MUST BIND A VARIABLE TO IT.
;;; (3) IF PREDICATE IS TRIVIAL, MAY JUST STICK IT IN SIMPLE CIF.

(DEFINE CONVERT-IF
        (LAMBDA (NODE FM CONT MP)
                (IF (NODE\TRIVP NODE)
                    (MAKE-RETURN (CONS-TRIVIAL (NODE = NODE)) CONT)
                    (LET ((CVAR (IF (EQ (TYPE (CNODE\CFORM CONT)) 'CVARIABLE)
                                    NIL
                                    (GENTEMP 'CONT)))
                          (PVAR (IF (NODE\TRIVP (IF\PRED FM))
                                    NIL
                                    (NODE\NAME (IF\PRED FM)))))
                         (LET ((ICONT (IF CVAR
                                          (CNODIFY (CONS-CVARIABLE (VAR = CVAR)))
                                          CONT))
                               (IPRED (IF PVAR
                                          (CNODIFY (CONS-CVARIABLE (VAR = PVAR)))
                                          (CNODIFY (CONS-TRIVIAL (NODE = (IF\PRED FM)))))))
                              (LET ((CIF (CNODIFY
                                          (CONS-CIF
                                           (PRED = IPRED)
                                           (CON = (CONVERT (IF\CON FM) ICONT MP))
                                           (ALT = (CONVERT (IF\ALT FM)
                                                           (CNODIFY
                                                            (CONS-CVARIABLE
                                                             (VAR = (CVARIABLE\VAR
                                                                     (CNODE\CFORM ICONT)))))
                                                           MP))))))
                                   (LET ((FOO (IF PVAR
                                                  (CONVERT (IF\PRED FM)
                                                           (CNODIFY (CONS-CONTINUATION (VAR = PVAR)
                                                                                       (BODY = CIF)))
                                                           MP)
                                                  CIF)))
                                        (IF CVAR
                                            (CNODIFY
                                             (CONS-CCOMBINATION
                                              (ARGS = (LIST (CNODIFY
                                                             (CONS-CLAMBDA
                                                              (VARS = (LIST CVAR))
                                                              (BODY = FOO)))
                                                            CONT))))
                                            FOO))))))))
```

This is all done as follows. First CVAR and PVAR are bound to generated names if necessary, CVAR for binding the continuation and PVAR for binding the predicate value. Then ICONT and IPRED (the "I" is a mnemonic for "internal") are bound to the cnodes to be used for the two conversions of consequent and alternative, and for the predicate of the CIF, respectively. CIF is then bound to the cnode for the CIF code, including the conversions of consequent and alternative. Finally, using FOO as an intermediary, CONVERT-IF first conditionally arranges for conversion of a non-trivial predicate, and then conditionally arranges for the binding of a non-cvariable continuation. The result of all this is returned as the final conversion of the original IF node.

CONVERT-ASET is fairly straightforward, except that, as for IF nodes, a special case is made of trivial nodes, as determined by the TRIVP slot.

The CATCH construct may be viewed as the user's one interface between the "LAMBDA world" and the "continuation world". CONVERT-CATCH arranges its conversion in such a way as to eliminate CATCH entirely. Because CONTINUATION cnodes provide an explicit representation for the continuations involved, there is no need at this level to have an explicit CCATCH sort of cnode. The general idea is:

```
NODE = (CATCH a b)    and    CONT = k    becomes

((CLAMBDA (q)
          ((CLAMBDA (a) (RETURN q b))
           (CLAMBDA (*IGNORE* V) (RETURN q V))))
 k)
```

In the case where the given continuation k is already a cvariable, then it need not be bound to a new one q. Note that the (renamed) user catch variable a is bound to a CLAMBDA which ignores its own continuation, and returns the argument V to the continuation of the CATCH. Thus the user variable a is bound not to an actual CONTINUATION, but to a little CLAMBDA which interfaces properly between the CLAMBDA world and the CONTINUATION world. The uses of CVAR and ICONT are analogous to their uses in CONVERT-IF.

```
(DEFINE CONVERT-ASET
        (LAMBDA (NODE FM CONT MP)
                (IF (NODE\TRIVP NODE)
                    (MAKE-RETURN (CONS-TRIVIAL (NODE = NODE)) CONT)
                    (CONVERT (ASET\BODY FM)
                             (LET ((NM (NODE\NAME (ASET\BODY FM))))
                                  (CNODIFY
                                   (CONS-CONTINUATION
                                    (VAR = NM)
                                    (BODY = (CNODIFY
                                            (CONS-CASET
                                             (CONT = CONT)
                                             (VAR = (ASET\VAR FM))
                                             (BODY = (CNODIFY (CONS-CVARIABLE
                                                               (VAR = NM)))))))))
                             MP))))

;;; ISSUES FOR CONVERTING CATCH:
;;; (1) MUST BIND THE CATCH VARIABLE TO A FUNNY FUNCTION WHICH IGNORES ITS CONTINUATION:
;;; (2) IF CONTINUATION IS NON-CVARIABLE, MUST BIND A VARIABLE TO IT.

(DEFINE
 CONVERT-CATCH
 (LAMBDA (NODE FM CONT MP)
         (LET ((CVAR (IF (EQ (TYPE (CNODE\CFORM CONT)) 'CVARIABLE)
                         NIL
                         (GENTEMP 'CONT))))
              (LET ((ICONT (IF CVAR
                               (CNODIFY (CONS-CVARIABLE (VAR = CVAR)))
                               CONT)))
                   (LET ((CP (CNODIFY
                              (CONS-CCOMBINATION
                               (ARGS = (LIST (CNODIFY
                                              (CONS-CLAMBDA
                                               (VARS = (LIST (CATCH\VAR FM)))
                                               (BODY = (CONVERT (CATCH\BODY FM) ICONT MP))))
                                             (CNODIFY
                                              (CONS-CLAMBDA
                                               (VARS = '(*IGNORE* V))
                                               (BODY = (MAKE-RETURN
                                                         (CONS-CVARIABLE (VAR = 'V))
                                                         (CNODIFY
                                                          (CONS-CVARIABLE
                                                           (VAR = (CVARIABLE\VAR
                                                                   (CNODE\CFORM ICONT))))))))))))))
                        (IF CVAR (CNODIFY
                                  (CONS-CCOMBINATION
                                   (ARGS = (LIST (CNODIFY
                                                  (CONS-CLAMBDA (VARS = (LIST CVAR))
                                                                (BODY = CP)))
                                                 CONT))))
                            CP))))))
```

CONVERT-LABELS simply converts all the labelled function definitions using NIL as the continuation for each. This reflects the fact that no code directly receives the results of closing the definitions; rather, they simply become part of the environment. The body of the LABELS is converted using the given continuation.

To make things much simpler for the pass-2 analysis and the code generator, it is forbidden to use ASET' on a LABELS-bound variable. This is an arbitrary restriction imposed by RABBIT (out of laziness on my part and a desire to concentrate on more important issues), and not one inherent in the SCHEME language. This restriction is unnoticeable in practice, since one seldom uses ASET' at all, let alone on a LABELS variable.

The conversion of COMBINATION nodes is the most complex of all cases. First, a trivial combination becomes simply a TRIVIAL cnode. Otherwise, the overall idea is that each argument is converted, and the continuation given to the conversion is the conversion of all the following arguments. The conversion of the last argument uses a continuation which performs the invocation of function on arguments, using all the bound variables of the generated continuations. The end result is a piece of code which evaluates one argument, binds a variable to the result, evaluates the next, etc., and finally uses the results to perform a function call.

To simplify the generated code, the arguments are divided into two classes. One class consists of trivial arguments and LAMBDA-expressions (this class is precisely the class of "trivially evaluable" expressions defined in [Imperative]), and the other class consists of the remaining arguments. The successive conversion using successive continuations as in the general theory is only performed on the latter class of arguments. The trivially evaluable expressions are included along with the bound variables for non-trivial argument values in the final function call. For example, one might have something like:

```
NODE = (FOO (CONS A B) (BAR A) B (BAZ B))   and   CONT = k   becomes

(RETURN (CONTINUATION (x)
                      (RETURN (CONTINUATION (y)
                                            (FOO k (CONS A B) x B y))
                              (BAZ B)))
        (BAR A))
```

where FOO, (CONS A B), and B are trivial, but (BAR A) and (BAZ B) are not.

```
;;; ISSUES FOR CONVERTING LABELS:
;;; (1) MUST CONVERT ALL THE NAMED LAMBDA-EXPRESSIONS, USING A NULL CONTINUATION.
;;; (2) TO MAKE THINGS EASIER LATER, WE FORBID ASET ON A LABELS VARIABLE.

(DEFINE CONVERT-LABELS
        (LAMBDA (NODE FM CONT MP)
                (DO ((F (LABELS\FNDEFS FM) (CDR F))
                     (V (LABELS\FNVARS FM) (CDR V))
                     (CF NIL (CONS (CONVERT (CAR F) NIL MP) CF)))
                    ((NULL F)
                     (CNODIFY (CONS-CLABELS (FNVARS = (LABELS\FNVARS FM))
                                            (FNDEFS = (NREVERSE CF))
                                            (BODY = (CONVERT (LABELS\BODY FM) CONT MP)))))
                    (AND (GET (CAR V) 'WRITE-REFS)
                         (ERROR '|Are you crazy, using ASET on a LABELS variable?|
                                (CAR V)
                                'FAIL-ACT)))))
```

The separation into two classes is accomplished by the outer DO loop. DELAY-FLAGS is a list of flags describing whether the code can be "delayed" (not converted using strung-out continuations) because it is trivially evaluable. The inner DO loop of the three (which loops on variables A, D, and Z, not A, D, and F!) then constructs the final function call, using the "delayed" arguments and generated continuation variables. The names used for the variables are the names of the corresponding nodes, which were generated by NODIFY. Finally, the middle DO loop (which executes last because the "inner" DO loop occurs in the initialization, not the body, of the "middle" one) generates the strung-out continuations, converting the non-delayable arguments in reverse order, so as to generate the converted result from the inside out.

The net effect is that non-trivial arguments are evaluated from left-to-right, and trivial ones are also (as it happens, because of MacLISP semantics), but the two classes are intermixed. This is where RABBIT takes advantage of the SCHEME semantics which decree that arguments to a combination may be evaluated in any order. It is also why CHECK-COMBINATION-PEFFS tries to detect infractions of this rule.

A special trick is that if the given continuation is a variable, and the combination is of the form ((LAMBDA ...) ...), then it is arranged to use the given continuation as the continuation for converting the body of the LAMBDA, rather than the extra variable which is introduced for a continuation in the LAMBDA variables list (see CONVERT-LAMBDA-FM). This effectively constitutes the optimization of substituting one continuation variable for another, much as META-COMBINATION-LAMBDA may substitute one variable for another. (This turns out to be the only optimization of importance to be done on pass-2 cnode code; rather than building a full-blown optimizer for pass-2 cnode-trees, or arranging to make the optimizer usable on both kinds of data structures, it was easier to tweak the conversion of combinations.) The substitution is effected by passing a non-NIL CNAME argument to CONVERT-LAMBDA-FORM, as computed by the form (AND (NULL (CDR A)) ...).

```
;;; ISSUES FOR CONVERTING COMBINATIONS:
;;; (1) TRIVIAL ARGUMENT EVALUATIONS ARE DELAYED AND ARE NOT BOUND TO THE VARIABLE OF
;;;     A CONTINUATION.  WE ASSUME THEREBY THAT THE COMPILER IS PERMITTED TO EVALUATE
;;;     OPERANDS IN ANY ORDER.
;;; (2) ALL NON-DELAYABLE COMPUTATIONS ARE ASSIGNED NAMES AND STRUNG OUT WITH CONTINUATIONS.
;;; (3) IF CONT IS A CVARIABLE AND THE COMBINATION IS ((LAMBDA ...) ...) THEN WHEN CONVERTING
;;;     THE LAMBDA-EXPRESSION WE ARRANGE FOR ITS BODY TO REFER TO THE CVARIABLE CONT RATHER
;;;     THAN TO ITS OWN CONTINUATION.  THIS CROCK EFFECTIVELY PERFORMS THE OPTIMIZATION OF
;;;     SUBSTITUTING ONE VARIABLE FOR ANOTHER, ONLY ON CONTINUATION VARIABLES (WHICH COULDN'T
;;;     BE CAUGHT BY META-EVALUATE).

(DEFINE
 CONVERT-COMBINATION
 (LAMBDA (NODE FM CONT MP)
         (IF (NODE\TRIVP NODE)
             (MAKE-RETURN (CONS-TRIVIAL (NODE = NODE)) CONT)
             (DO ((A (COMBINATION\ARGS FM) (CDR A))
                  (DELAY-FLAGS NIL
                               (CONS (OR (NODE\TRIVP (CAR A))
                                         (EQ (TYPE (NODE\FORM (CAR A))) 'LAMBDA))
                                     DELAY-FLAGS)))
                 ((NULL A)
                  (DO ((A (REVERSE (COMBINATION\ARGS FM)) (CDR A))
                       (D DELAY-FLAGS (CDR D))
                       (F (CNODIFY
                           (CONS-CCOMBINATION
                            (ARGS = (DO ((A (REVERSE (COMBINATION\ARGS FM)) (CDR A))
                                         (D DELAY-FLAGS (CDR D))
                                         (Z NIL (CONS (IF (CAR D)
                                                          (IF (EQ (TYPE (NODE\FORM (CAR A)))
                                                                  'LAMBDA)
                                                              (CNODIFY
                                                               (CONVERT-LAMBDA-FM
                                                                (CAR A)
                                                                (AND (NULL (CDR A))
                                                                     (EQ (TYPE
                                                                          (CNODE\CFORM CONT))
                                                                         'CVARIABLE)
                                                                     (CVARIABLE\VAR
                                                                      (CNODE\CFORM CONT)))
                                                                MP))
                                                              (CNODIFY
                                                               (CONS-TRIVIAL
                                                                (NODE = (CAR A)))))
                                                          (CNODIFY
                                                           (CONS-CVARIABLE
                                                            (VAR = (NODE\NAME (CAR A))))))
                                                      Z)))
                                        ((NULL A) (CONS (CAR Z) (CONS CONT (CDR Z))))))))
                          (IF (CAR D) F
                              (CONVERT (CAR A)
                                       (CNODIFY (CONS-CONTINUATION
                                                 (VAR = (NODE\NAME (CAR A)))
                                                 (BODY = F)))
                                       MP))))
                      ((NULL A) F)))))))
```

Once the pass-2 cnode-tree is constructed, a pass-2 analysis is performed in a manner very similar to the pass-1 analysis. As before, successive routines are called which recursively process the code tree and pass information up and down, filling in various slots and putting properties on the property lists of variable names.

The first routine, CENV-ANALYZE, is similar to ENV-ANALYZE, but differs in some important respects. Two slots are filled in for each cnode. The slot ENV is computed from the top down, while REFS is computed from the bottom up.

ENV is the environment, a list of bound variables visible to the cnode. The ENV slot in the node-tree was a mapping (an alist), but this ENV is only a list. The argument ENV is used in the analysis of CVARIABLE and CASET nodes. The cnode slot ENV is included only for debugging purposes, and is never used by RABBIT itself.

REFS is analogous to the REFS slot of a node-tree: it is the set of variables bound above and referenced below the cnode. It differs from the pass-1 analysis in that variables introduced to name continuations and variables bound by continuations are also accounted for. In the case of a TRIVIAL cnode, however, the REFS are precisely those of the contained node.

The argument FNP to CENV-ANALYZE in non-NIL iff the given cnode occurs in "functional position" of a CCOMBINATION or RETURN cnode. This is used when a variable is encountered; on the property list a VARIABLE-REFP property is placed iff FNP is NIL, indicating that the variable was referenced in "variable (non-function) position". This information will be used by the next phase, BIND-ANALYZE.

```
;;; ENVIRONMENT ANALYSIS FOR CPS VERSION

;;; WE WISH TO DETERMINE THE ENVIRONMENT AT EACH CNODE,
;;; AND DETERMINE WHAT VARIABLES ARE BOUND ABOVE AND
;;; REFERRED TO BELOW EACH CNODE.

;;; FOR EACH CNODE WE FILL IN THESE SLOTS:
;;;     ENV     THE ENVIRONMENT SEEN AT THAT CNODE (A LIST OF VARS)
;;;     REFS    VARIABLES BOUND ABOVE AND REFERRED TO BELOW THAT CNODE
;;; FOR EACH VARIABLE REFERRED TO IN NON-FUNCTION POSITION
;;; BY A CVARIABLE OR CTRIVIAL CNODE WE GIVE A NON-NIL VALUE TO THE PROPERTY:
;;;     VARIABLE-REFP

;;; FNP IS NON-NIL IFF CNODE OCCURS IN FUNCTIONAL POSITION

(DEFINE
 CENV-ANALYZE
 (LAMBDA (CNODE ENV FNP)
         (LET ((CFM (CNODE\CFORM CNODE)))
              (ALTER-CNODE CNODE (ENV := ENV))
              (EQCASE (TYPE CFM)
                      (TRIVIAL
                       (CENV-TRIV-ANALYZE (TRIVIAL\NODE CFM) FNP)
                       (ALTER-CNODE CNODE
                                    (REFS := (NODE\REFS (TRIVIAL\NODE CFM)))))
                      (CVARIABLE
                       (LET ((V (CVARIABLE\VAR CFM)))
                            (ADDPROP V CNODE 'READ-REFS)
                            (OR FNP (PUTPROP V T 'VARIABLE-REFP))
                            (ALTER-CNODE CNODE
                                         (REFS := (AND (MEMQ V ENV)
                                                       (LIST (CVARIABLE\VAR CFM)))))))
                      (CLAMBDA
                       (LET ((B (CLAMBDA\BODY CFM)))
                            (CENV-ANALYZE B (APPEND (CLAMBDA\VARS CFM) ENV) NIL)
                            (LET ((REFS (SETDIFF (CNODE\REFS B) (CLAMBDA\VARS CFM))))
                                 (ALTER-CNODE CNODE (REFS := REFS)))))
                      (CONTINUATION
                       (LET ((B (CONTINUATION\BODY CFM)))
                            (CENV-ANALYZE B (CONS (CONTINUATION\VAR CFM) ENV) NIL)
                            (LET ((REFS (REMOVE (CONTINUATION\VAR CFM) (CNODE\REFS B))))
                                 (ALTER-CNODE CNODE (REFS := REFS)))))
                      (CIF
                       (LET ((PRED (CIF\PRED CFM))
                             (CON (CIF\CON CFM))
                             (ALT (CIF\ALT CFM)))
                            (CENV-ANALYZE PRED ENV NIL)
                            (CENV-ANALYZE CON ENV NIL)
                            (CENV-ANALYZE ALT ENV NIL)
                            (ALTER-CNODE CNODE
                                         (REFS := (UNION (CNODE\REFS PRED)
                                                         (UNION (CNODE\REFS CON)
                                                                (CNODE\REFS ALT)))))))
                      (CASET
                       (LET ((V (CASET\VAR CFM))
                             (CN (CASET\CONT CFM))
                             (B (CASET\BODY CFM)))
                            (PUTPROP (CASET\VAR CFM) T 'VARIABLE-REFP)
                            (CENV-ANALYZE CN ENV T)
                            (CENV-ANALYZE B ENV NIL)
                            (ALTER-CNODE CNODE
                                         (REFS := (LET ((R (UNION (CNODE\REFS CN)
                                                                  (CNODE\REFS B))))
                                                       (IF (MEMQ V ENV) (ADJOIN V R) R))))))
                      (CLABELS
                       (LET ((LENV (APPEND (CLABELS\FNVARS CFM) ENV)))
                            (DO ((F (CLABELS\FNDEFS CFM) (CDR F))
                                 (R NIL (UNION R (CNODE\REFS (CAR F)))))
```

This page intentionally left blank

except for

an annoying and self-referential little sentence.

```
                    ((NULL F)
                     (LET ((B (CLABELS\BODY CFM)))
                          (CENV-ANALYZE B LENV NIL)
                          (ALTER-CNODE CNODE
                                       (REFS := (SETDIFF (UNION R (CNODE\REFS B))
                                                         (CLABELS\FNVARS CFM))))))
                    (CENV-ANALYZE (CAR F) LENV NIL))))
          (CCOMBINATION
           (LET ((ARGS (CCOMBINATION\ARGS CFM)))
                (CENV-ANALYZE (CAR ARGS) ENV T)
                (COND ((AND (EQ (TYPE (CNODE\CFORM (CAR ARGS))) 'TRIVIAL)
                            (EQ (TYPE (NODE\FORM (TRIVIAL\NODE
                                                  (CNODE\CFORM (CAR ARGS)))))
                                'VARIABLE)
                            (TRIVFN (VARIABLE\VAR
                                     (NODE\FORM
                                      (TRIVIAL\NODE
                                       (CNODE\CFORM
                                        (CAR ARGS)))))))
                       (CENV-ANALYZE (CADR ARGS) ENV T)
                       (CENV-CCOMBINATION-ANALYZE CNODE
                                                  ENV
                                                  (CDDR ARGS)
                                                  (UNION (CNODE\REFS (CAR ARGS))
                                                         (CNODE\REFS (CADR ARGS)))))
                      (T (CENV-CCOMBINATION-ANALYZE CNODE
                                                    ENV
                                                    (CDR ARGS)
                                                    (CNODE\REFS (CAR ARGS)))))))
          (RETURN
           (LET ((C (RETURN\CONT CFM))
                 (V (RETURN\VAL CFM)))
                (CENV-ANALYZE C ENV T)
                (CENV-ANALYZE V ENV NIL)
                (ALTER-CNODE CNODE
                             (REFS := (UNION (CNODE\REFS C) (CNODE\REFS V))))))))))
```

The only purpose of CENV-TRIV-ANALYZE is to go through the code for a TRIVIAL cnode, looking for variables occurring in other than function position, in order to put appropriate VARIABLE-REFP properties. Notice that the types LAMBDA and LABELS do not occur in the EQCASE expression, as nodes of those types can never occur in trivial expressions.

CENV-CCOMBINATION-ANALYZE is a simple routine which analyzes CCOMBINATION cnodes; it is a separate routine only because it is used in more than one place in CENV-ANALYZE. It could have been made a local subroutine by using a LABELS in CENV-ANALYZE, but I elected not to do so for purely typographical reasons.

```
;;; THIS FUNCTION MUST GO THROUGH AND LOCATE VARIABLES APPEARING IN NON-FUNCTION POSITION.

(DEFINE CENV-TRIV-ANALYZE
        (LAMBDA (NODE FNP)
                (LET ((FM (NODE\FORM NODE)))
                     (EQCASE (TYPE FM)
                             (CONSTANT NIL)
                             (VARIABLE
                              (OR FNP (PUTPROP (VARIABLE\VAR FM) T 'VARIABLE-REFP)))
                             (LAMBDA
                              (OR FNP
                                  (ERROR '|Trivial closure - CENV-TRIV-ANALYZE| NODE 'FAIL-ACT))
                              (CENV-TRIV-ANALYZE (LAMBDA\BODY FM) NIL))
                             (IF
                              (CENV-TRIV-ANALYZE (IF\PRED FM) NIL)
                              (CENV-TRIV-ANALYZE (IF\CON FM) NIL)
                              (CENV-TRIV-ANALYZE (IF\ALT FM) NIL))
                             (ASET
                              (PUTPROP (ASET\VAR FM) T 'VARIABLE-REFP)
                              (CENV-TRIV-ANALYZE (ASET\BODY FM) NIL))
                             (COMBINATION
                              (DO ((A (COMBINATION\ARGS FM) (CDR A))
                                   (F T NIL))
                                  ((NULL A))
                                  (CENV-TRIV-ANALYZE (CAR A) F)))))))

(DEFINE CENV-CCOMBINATION-ANALYZE
        (LAMBDA (CNODE ENV ARGS FREFS)
                (DO ((A ARGS (CDR A))
                     (R FREFS (UNION R (CNODE\REFS (CAR A)))))
                    ((NULL A)
                     (ALTER-CNODE CNODE (REFS := R)))
                    (CENV-ANALYZE (CAR A) ENV NIL))))
```

The binding analysis is the most complicated phase of pass 2. It determines for each function whether or not a closure structure will be needed for it at run time (and if so, whether the closure structure must contain a pointer to the code); it determines for each variable whether or not it can be referred to by a run-time closure structure; and it determines for each function how arguments will be passed to it (because for internal functions not apparent to the "outside world", any arbitrary argument-passing convention may be adopted by the compiler to optimize register usage; in particular, arguments which are never referred to need never even be actually passed). If flow analysis determines that a given variable always denotes (a closure of) a given functional (CLAMBDA) expression, then a KNOWN-FUNCTION property is created to connect the variable directly to the function for the benefit of the code generator.

BIND-ANALYZE is just a simple dispatch to one of many specialists, one for each type of CNODE. TRIVIAL and CVARIABLE cnodes are handled directly because they are simple.

The argument FNP is NIL, EZCLOSE, or NOCLOSE, depending respectively on whether a full closure structure, a closure structure without a code pointer, or no closure structure will be needed if in fact CNODE turns out to be of type CLAMBDA (or CONTINUATION). Normally it is NIL, unless determined otherwise by a parent CLABELS or CCOMBINATION cnode.

The argument NAME is meaningful only if the CNODE argument is of type CLAMBDA or CONTINUATION. If non-NIL, it is a suggested name to use for the cnode. This name will later be used by the code generator as a tag. The only reason for using the suggestion rather than a generated name (and in fact one will be generated if the suggested name is NIL) is to make it easier to trace things while debugging.

REFD-VARS is a utility routine. Given a set of variables, it returns the subset of them that are actually referenced (as determined by the READ-REFS and WRITE-REFS properties which were set up by ENV-ANALYZE and CENV-ANALYZE).

```
;;; BINDING ANALYSIS.

;;; FOR EACH CNODE WE FILL IN:
;;;     CLOVARS         THE SET OF VARIABLES REFERRED TO BY CLOSURES
;;;                     AT OR BELOW THIS NODE (SHOULD ALWAYS BE A
;;;                     SUBSET OF REFS)
;;; FOR EACH CLAMBDA AND CONTINUATION WE FILL IN:
;;;     FNP     NON-NIL IFF REFERENCED ONLY AS A FUNCTION.
;;;             WILL BE 'EZCLOSE IF REFERRED TO BY A CLOSURE,
;;;             AND OTHERWISE 'NOCLOSE.
;;;     TVARS   VARIABLES PASSED THROUGH TEMP LOCATIONS WHEN CALLING
;;;             THIS FUNCTION
;;;     NAME    THE NAME OF THE FUNCTION (USED FOR THE PROG TAG)
;;; FOR EACH CLABELS WE FILL IN:
;;;     EASY    REFLECTS FNP STATUS OF ALL THE LABELLED FUNCTIONS
;;; FOR EACH VARIABLE WHICH ALWAYS DENOTES A CERTAIN FUNCTION WE
;;; PUT THE PROPERTIES:
;;;     KNOWN-FUNCTION          IFF THE VARIABLE IS NEVER ASET
;;; THE VALUE OF THE KNOWN-FUNCTION PROPERTY IS THE CNODE FOR
;;; THE FUNCTION DEFINITION.
;;; FOR EACH LABELS VARIABLE IN A LABELS OF THE 'EZCLOSE VARIETY
;;; WE PUT THE PROPERTY:
;;;     LABELS-FUNCTION
;;; TO INDICATE THAT ITS "EASY" CLOSURE MUST BE CDR'D TO GET THE
;;; CORRECT ENVIRONMENT (SEE PRODUCE-LABELS).

;;; NAME, IF NON-NIL, IS A SUGGESTED NAME FOR THE FUNCTION

(DEFINE BIND-ANALYZE
        (LAMBDA (CNODE FNP NAME)
                (LET ((CFM (CNODE\CFORM CNODE)))
                     (EQCASE (TYPE CFM)
                             (TRIVIAL
                              (ALTER-CNODE CNODE (CLOVARS := NIL)))
                             (CVARIABLE
                              (ALTER-CNODE CNODE (CLOVARS := NIL)))
                             (CLAMBDA
                              (BIND-ANALYZE-CLAMBDA CNODE FNP NAME CFM))
                             (CONTINUATION
                              (BIND-ANALYZE-CONTINUATION CNODE FNP NAME CFM))
                             (CIF
                              (BIND-ANALYZE-CIF CNODE CFM))
                             (CASET
                              (BIND-ANALYZE-CASET CNODE CFM))
                             (CLABELS
                              (BIND-ANALYZE-CLABELS CNODE CFM))
                             (CCOMBINATION
                              (BIND-ANALYZE-CCOMBINATION CNODE CFM))
                             (RETURN
                              (BIND-ANALYZE-RETURN CNODE CFM))))))

(DEFINE REFD-VARS
        (LAMBDA (VARS)
                (DO ((V VARS (CDR V))
                     (W NIL (IF (OR (GET (CAR V) 'READ-REFS)
                                    (GET (CAR V) 'WRITE-REFS))
                                (CONS (CAR V) W)
                                W)))
                    ((NULL V) (NREVERSE W)))))
```

For a CLAMBDA cnode, BIND-ANALYZE-CLAMBDA first analyzes the body. The CLOVARS component of the cnode is then calculated. If the CLAMBDA will have a run-time closure structure created for it, then any variable it references is obviously referred to by a closure. Otherwise, only the CLOVARS of its body are included in the set.

The TVARS component is the set of parameters for which arguments will be passed in a non-standard manner. Non-standard argument-passing is used only for NOCLOSE-type functions (though in principle it could also be used for EZCLOSE-type functions also). In this case, only referenced variables (as determined by REFD-VARS) are actually passed. The code generator uses TVARS for two purposes: when compiling the CLAMBDA itself, TVARS is used to determine which arguments are in which registers; and when compiling calls to the function, TVARS determines which registers to load (see LAMBDACATE).

The FNP slot is just filled in using the FNP parameter. If a name was not suggested for the NAME slot, an arbitrary name is generated.

BIND-ANALYZE-CONTINUATION is entirely analogous to BIND-ANALYZE-CLAMBDA.

BIND-ANALYZE-CIF straightforwardly analyzes recursively its sub-cnodes, and then passes the union of their CLOVARS up as its own CLOVARS.

BIND-ANALYZE-CASET tries to be a little bit clever about the obscure case produced by code such as:

```
    (ASET' FOO (LAMBDA ...))
```

where the continuation is a CONTINUATION cnode (rather than a CVARIABLE). It is then known that the variable bound by the CONTINUATION (<u>not</u> the variable set by the CASET!!) will have as its value the (closure of the) CLAMBDA-expression. This allows for the creation of a KNOWN-FUNCTION property, etc. This analysis is very similar to that performed by BIND-ANALYZE-RETURN (see below). Aside from this, the analysis of a CASET is simple; the CLOVARS component is merely the union of the CLOVAR slots of the sub-cnodes.

```
(DEFINE BIND-ANALYZE-CLAMBDA
        (LAMBDA (CNODE FNP NAME CFM)
                (BLOCK (BIND-ANALYZE (CLAMBDA\BODY CFM) NIL NIL)
                       (ALTER-CNODE CNODE
                                    (CLOVARS := (IF (EQ FNP 'NOCLOSE)
                                                    (CNODE\CLOVARS (CLAMBDA\BODY CFM))
                                                    (CNODE\REFS CNODE))))
                       (ALTER-CLAMBDA CFM
                                      (FNP := FNP)
                                      (TVARS := (IF (EQ FNP 'NOCLOSE)
                                                    (REFD-VARS (CLAMBDA\VARS CFM))
                                                    NIL))
                                      (NAME := (OR NAME (GENTEMP 'F)))))))

(DEFINE BIND-ANALYZE-CONTINUATION
        (LAMBDA (CNODE FNP NAME CFM)
                (BLOCK (BIND-ANALYZE (CONTINUATION\BODY CFM) NIL NIL)
                       (ALTER-CNODE CNODE
                                    (CLOVARS := (IF (EQ FNP 'NOCLOSE)
                                                    (CNODE\CLOVARS (CONTINUATION\BODY CFM))
                                                    (CNODE\REFS CNODE))))
                       (ALTER-CONTINUATION CFM
                                           (FNP := FNP)
                                           (TVARS := (IF (EQ FNP 'NOCLOSE)
                                                         (REFD-VARS (LIST (CONTINUATION\VAR CFM)))
                                                         NIL))
                                           (NAME := (OR NAME (GENTEMP 'C)))))))

(DEFINE BIND-ANALYZE-CIF
        (LAMBDA (CNODE CFM)
                (BLOCK (BIND-ANALYZE (CIF\PRED CFM) NIL NIL)
                       (BIND-ANALYZE (CIF\CON CFM) NIL NIL)
                       (BIND-ANALYZE (CIF\ALT CFM) NIL NIL)
                       (ALTER-CNODE CNODE
                                    (CLOVARS := (UNION (CNODE\CLOVARS (CIF\PRED CFM))
                                                       (UNION (CNODE\CLOVARS (CIF\CON CFM))
                                                              (CNODE\CLOVARS (CIF\ALT CFM)))))))))

(DEFINE BIND-ANALYZE-CASET
        (LAMBDA (CNODE CFM)
                (LET ((CN (CASET\CONT CFM))
                      (VAL (CASET\BODY CFM)))
                     (BIND-ANALYZE CN 'NOCLOSE NIL)
                     (COND ((AND (EQ (TYPE (CNODE\CFORM CN)) 'CONTINUATION)
                                 (EQ (TYPE (CNODE\CFORM VAL)) 'CLAMBDA))
                            (LET ((VAR (CONTINUATION\VAR (CNODE\CFORM CN))))
                                 (PUTPROP VAR VAL 'KNOWN-FUNCTION)
                                 (BIND-ANALYZE VAL
                                               (AND (NOT (GET VAR 'VARIABLE-REFP))
                                                    (IF (MEMQ VAR
                                                              (CNODE\CLOVARS
                                                               (CONTINUATION\BODY
                                                                (CNODE\CFORM CN))))
                                                        'EZCLOSE
                                                        (BLOCK (ALTER-CONTINUATION (CNODE\CFORM CN)
                                                                                   (TVARS := NIL))
                                                               'NOCLOSE)))
                                               NIL)))
                           (T (BIND-ANALYZE VAL NIL NIL)))
                     (ALTER-CNODE CNODE
                                  (CLOVARS := (UNION (CNODE\CLOVARS CN)
                                                     (CNODE\CLOVARS VAL)))))))
```

The binding analysis of a CLABELS is very tricky because of the possibility of mutually referent functions. For example, suppose a single CLABELS binds two CLAMBDA expressions with names FOO and BAR. Suppose that the body of FOO refers to BAR, and that of BAR to FOO. Should FOO and BAR be of FNP-type NIL, EZCLOSE, or NOCLOSE? If either is of type EZCLOSE, then the other must be also; but the decision cannot be made sequentially. It is even more complicated if one must be of type NIL.

An approximate solution is used here, to prevent having to solve complicated simultaneous constraints. It is arbitrarily decreed that all functions of a single CLABELS shall all have the same FNP type. If any one must be of type NIL, then they all are. Otherwise, it is tentatively assumed that they all may be of type NOCLOSE. If this assumption is disproved, then the analysis is retroactively patched up.

The outer DO loop of BIND-ANALYZE-CLABELS creates KNOWN-FUNCTION properties, and determines (in the variable EZ) whether any of the labelled functions needs a full closure structure. (This can be done before analyzing the functions, because it is determined entirely by the VARIABLE-REFP properties created in the previous phase.) The inner DO loop then analyzes the functions. When this is done, if EZ is NOCLOSE, and it turns out that it should have been EZCLOSE after all, then the third DO loop forcibly patches the CLAMBDA cnodes for the labelled functions, and the AMAPC form creates LABELS-FUNCTION properties as a flag for the code generator.

BIND-ANALYZE-RETURN simply analyzes the continuation and return value recursively, and then merges to two CLOVARS sets to produce its own CLOVARS set. A special case is when the two sub-cnodes are respectively a CONTINUATION and a CLAMBDA; then special work is done because it is known that the variable bound by the CONTINUATION will always denote the (closure of the) CLAMBDA-expression. A nasty trick is that if it turns out that the CLAMBDA can be of type NOCLOSE, then the TVARS slot of the CONTINUATION is forcibly set to NIL (i.e. the empty set). This is because no argument will really be passed. (This fact is also known by the LAMBDACATE routine in the code generator.)

```
(DEFINE BIND-ANALYZE-CLABELS
        (LAMBDA (CNODE CFM)
                (BLOCK (BIND-ANALYZE (CLABELS\BODY CFM) NIL NIL)
                       (DO ((V (CLABELS\FNVARS CFM) (CDR V))
                            (D (CLABELS\FNDEFS CFM) (CDR D))
                            (EZ 'NOCLOSE (AND (NULL (GET (CAR V) 'VARIABLE-REFP)) EZ)))
                           ((NULL V)
                            (ALTER-CLABELS CFM (EASY := EZ))
                            (DO ((V (CLABELS\FNVARS CFM) (CDR V))
                                 (D (CLABELS\FNDEFS CFM) (CDR D))
                                 (CV (CNODE\CLOVARS (CLABELS\BODY CFM))
                                      (UNION CV (CNODE\CLOVARS (CAR D)))))
                                ((NULL D)
                                 (ALTER-CNODE CNODE (CLOVARS := CV))
                                 (COND ((AND EZ (INTERSECT CV (LABELS\FNVARS CFM)))
                                        (DO ((D (CLABELS\FNDEFS CFM) (CDR D))
                                             (CV (CNODE\CLOVARS (CLABELS\BODY CFM))
                                                 (UNION CV (CNODE\CLOVARS (CAR D)))))
                                            ((NULL D)
                                             (ALTER-CNODE CNODE (CLOVARS := CV)))
                                            (ALTER-CLAMBDA (CNODE\CFORM (CAR D))
                                                           (FNP := 'EZCLOSE)
                                                           (TVARS := NIL))
                                            (ALTER-CNODE (CAR D)
                                                         (CLOVARS := (CNODE\REFS (CAR D)))))
                                        (AMAPC (LAMBDA (V) (PUTPROP V T 'LABELS-FUNCTION))
                                               (CLABELS\FNVARS CFM))
                                        (ALTER-CLABELS CFM (EASY := 'EZCLOSE)))))
                                 (BIND-ANALYZE (CAR D) EZ (CAR V))))
                           (PUTPROP (CAR V) (CAR D) 'KNOWN-FUNCTION)))))

(DEFINE BIND-ANALYZE-RETURN
        (LAMBDA (CNODE CFM)
                (LET ((CN (RETURN\CONT CFM))
                      (VAL (RETURN\VAL CFM)))
                     (BIND-ANALYZE CN 'NOCLOSE NIL)
                     (COND ((AND (EQ (TYPE (CNODE\CFORM CN)) 'CONTINUATION)
                                  (EQ (TYPE (CNODE\CFORM VAL)) 'CLAMBDA))
                             (LET ((VAR (CONTINUATION\VAR (CNODE\CFORM CN))))
                                  (PUTPROP VAR VAL 'KNOWN-FUNCTION)
                                  (BIND-ANALYZE VAL
                                               (AND (NOT (GET VAR 'VARIABLE-REFP))
                                                    (IF (MEMQ VAR
                                                              (CNODE\CLOVARS
                                                               (CONTINUATION\BODY
                                                                (CNODE\CFORM CN))))
                                                        'EZCLOSE
                                                        (BLOCK (ALTER-CONTINUATION (CNODE\CFORM CN)
                                                                                   (TVARS := NIL))
                                                               'NOCLOSE)))
                                               NIL)))
                            (T (BIND-ANALYZE VAL NIL NIL)))
                     (ALTER-CNODE CNODE
                                  (CLOVARS := (UNION (CNODE\CLOVARS CN)
                                                     (CNODE\CLOVARS VAL)))))))
```

BIND-ANALYZE-CCOMBINATION first analyzes the function position of the combination. It then distinguishes three cases: a trivial function, a CLAMBDA-expression function, and all others.

In the case of a trivial function, the continuation (which is the second item in ARGS) can be analyzed with FNP = NOCLOSE, because the compilation will essentially turn into "calculate all other arguments, apply the trivial function, and then give the result to the continuation". A CCOMBINATION which looks like:

```
(a-trivial-function (CONTINUATION (var) ...) arg1 ... argn)
```

is compiled almost as if it were:

```
((CONTINUATION (var) ...) (a-trivial-function arg1 ... argn))
```

and of course the continuation can be treated as of type NOCLOSE.

In the case of a CLAMBDA-expression, the arguments are all analyzed, and then the AMAPC expression goes back over the TVARS list of the CLAMBDA, and removes from the TVARS set each variable corresponding to an argument which the analysis has proved to be a NOCLOSE-type KNOWN-FUNCTION. This is because no actual argument will be passed at run time for such a function, and so there is no need to allocate a register through which to pass that argument.

In the third case, the arguments are analyzed straightforwardly by BIND-CCOMBINATION-ANALYZE.

BIND-CCOMBINATION-ANALYZE does the dirty work of analyzing arguments of a CCOMBINATION and updating the CLOVARS slot of the CCOMBINATION cnode. If VARS is non-NIL, then it is the variables of the CLAMBDA-expression which was in the function position of the CCOMBINATION. As the arguments are analyzed, KNOWN-FUNCTION properties are put on the variables as appropriate, and the correct value of FNP is determined for the recursive call to BIND-ANALYZE. If VARS is NIL, then this code depends on the fact that (CDR NIL)=NIL in MacLISP.

```
(DEFINE BIND-ANALYZE-CCOMBINATION
        (LAMBDA (CNODE CFM)
                (LET ((ARGS (CCOMBINATION\ARGS CFM)))
                     (BIND-ANALYZE (CAR ARGS) 'NOCLOSE NIL)
                     (LET ((FN (CNODE\CFORM (CAR ARGS))))
                          (COND ((AND (EQ (TYPE FN) 'TRIVIAL)
                                      (EQ (TYPE (NODE\FORM (TRIVIAL\NODE FN)))
                                          'VARIABLE)
                                      (TRIVFN (VARIABLE\VAR (NODE\FORM (TRIVIAL\NODE FN)))))
                                 (BIND-ANALYZE (CADR ARGS) 'NOCLOSE NIL)
                                 (BIND-CCOMBINATION-ANALYZE CNODE
                                                            (CDDR ARGS)
                                                            NIL
                                                            (CNODE\CLOVARS (CADR ARGS))))
                                ((EQ (TYPE FN) 'CLAMBDA)
                                 (BIND-CCOMBINATION-ANALYZE CNODE
                                                            (CDR ARGS)
                                                            (CLAMBDA\VARS FN)
                                                            (CNODE\CLOVARS (CAR ARGS)))
                                 (AMAPC (LAMBDA (V)
                                               (IF (LET ((KFN (GET V 'KNOWN-FUNCTION)))
                                                        (AND KFN
                                                             (EQ (EQCASE (TYPE (CNODE\CFORM KFN))
                                                                         (CLAMBDA
                                                                          (CLAMBDA\FNP
                                                                           (CNODE\CFORM KFN)))
                                                                         (CONTINUATION
                                                                          (CONTINUATION\FNP
                                                                           (CNODE\CFORM KFN))))
                                                                 'NOCLOSE)))
                                                   (ALTER-CLAMBDA
                                                    FN
                                                    (TVARS := (DELQ V (CLAMBDA\TVARS FN))))))
                                        (CLAMBDA\TVARS FN)))
                                (T (BIND-CCOMBINATION-ANALYZE CNODE
                                                              (CDR ARGS)
                                                              NIL
                                                              (CNODE\CLOVARS (CAR ARGS)))))))))

;;; VARS MAY BE NIL - WE DEPEND ON (CDR NIL)=NIL.

(DEFINE BIND-CCOMBINATION-ANALYZE
        (LAMBDA (CNODE ARGS VARS FCV)
                (DO ((A ARGS (CDR A))
                     (V VARS (CDR V))
                     (CV FCV (UNION CV (CNODE\CLOVARS (CAR A)))))
                    ((NULL A)
                     (ALTER-CNODE CNODE (CLOVARS := CV)))
                    (COND ((AND VARS
                                (MEMQ (TYPE (CNODE\CFORM (CAR A))) '(CLAMBDA CONTINUATION))
                                (NOT (GET (CAR V) 'WRITE-REFS)))
                           (PUTPROP (CAR V) (CAR A) 'KNOWN-FUNCTION)
                           (BIND-ANALYZE (CAR A)
                                         (AND (NOT (GET (CAR V) 'VARIABLE-REFP))
                                              (IF (MEMQ (CAR V) FCV)
                                                  'EZCLOSE
                                                  'NOCLOSE))
                                         NIL))
                          (T (BIND-ANALYZE (CAR A) NIL NIL))))))
```

DEPTH-ANALYZE allocates registers through which to pass arguments to NOCLOSE functions, i.e. for arguments corresponding to elements of TVARS sets. An unclever stack discipline is used for allocating registers. Each function is assigned a "depth", which is zero for a function whose FNP is NIL or EZCLOSE (such functions take their arguments in the standard registers **ONE** through **EIGHT**, assuming that **NUMBER-OF-ARG-REGS** is 8, as it is in the current SCHEME implementation). For a NOCLOSE function the depth is essentially the depth of the function in whose body the NOCLOSE function appears, plus the number of TVARS belonging to that other function (if it is of type NOCLOSE) or the number of standard argument registers used by it (if it is NIL or EZCLOSE). For example, consider this code:

```
(CLAMBDA (C X Y)
         ((CLAMBDA (K F Z)
                   ((CLAMBDA (Q W V)
                             ...)
                    CONT-57 '3 '4))
          (CONTINUATION (V) ...)
          (CLAMBDA (H) ...)
          'FOO))
```

Suppose that the outer CLAMBDA is of type EZCLOSE for some reason. Its depth is 0. The two CLAMBDA-expressions and CONTINUATION immediately within it have depth 3 (assuming the CONTINUATION and second CLAMBDA are of type NOCLOSE -- the first CLAMBDA definitely is). The innermost CLAMBDA is then of depth 4 (for Z, which will be in TVARS -- K and F will not be because they are names for NOCLOSE functions, assuming K and F have no WRITE-REFS properties).

To each function is also attached a MAXDEP value, which is in effect the number of registers used by that function, including all NOCLOSE functions within it. This is used in only one place in the code generator, to generate a SPECIAL declaration for the benefit of the MacLISP compiler, which compiles the output of RABBIT. For most constructs this is simply the numerical maximum over the depths of all sub-cnodes. Toward this end the maximum depth of the cnode is returned as the value of DEPTH-ANALYZE.

```
;;; DEPTH ANALYSIS FOR CPS VERSION.

;;; FOR EACH CLAMBDA AND CONTINUATION WE FILL IN:
;;;     DEP             DEPTH OF TEMP VAR USAGE AT THIS POINT
;;;     MAXDEP          MAX DEPTH BELOW THIS POINT

;;; VALUE OF DEPTH-ANALYZE IS THE MAX DEPTH

(DEFINE DEPTH-ANALYZE
        (LAMBDA (CNODE DEP)
                (LET ((CFM (CNODE\CFORM CNODE)))
                     (EQCASE (TYPE CFM)
                             (TRIVIAL DEP)
                             (CVARIABLE DEP)
                             (CLAMBDA
                              (LET ((MD (DEPTH-ANALYZE (CLAMBDA\BODY CFM)
                                                       (IF (EQ (CLAMBDA\FNP CFM) 'NOCLOSE)
                                                           (+ DEP (LENGTH (CLAMBDA\TVARS CFM)))
                                                           (MIN (LENGTH (CLAMBDA\VARS CFM))
                                                                (+ 1 **NUMBER-OF-ARG-REGS**))))))
                                   (ALTER-CLAMBDA
                                    CFM
                                    (DEP := (IF (EQ (CLAMBDA\FNP CFM) 'NOCLOSE) DEP 0))
                                    (MAXDEP := MD))
                                   MD))
                             (CONTINUATION
                              (LET ((MD (DEPTH-ANALYZE
                                         (CONTINUATION\BODY CFM)
                                         (IF (EQ (CONTINUATION\FNP CFM) 'NOCLOSE)
                                             (+ DEP (LENGTH (CONTINUATION\TVARS CFM)))
                                             2))))
                                   (ALTER-CONTINUATION
                                    CFM
                                    (DEP := (IF (EQ (CONTINUATION\FNP CFM) 'NOCLOSE) DEP 0))
                                    (MAXDEP := MD))
                                   MD))
                             (CIF
                              (MAX (DEPTH-ANALYZE (CIF\PRED CFM) DEP)
                                   (DEPTH-ANALYZE (CIF\CON CFM) DEP)
                                   (DEPTH-ANALYZE (CIF\ALT CFM) DEP)))
                             (CASET
                              (MAX (DEPTH-ANALYZE (CASET\CONT CFM) DEP)
                                   (DEPTH-ANALYZE (CASET\BODY CFM) DEP)))
                             (CLABELS
                              (LET ((DP (IF (EQ (CLABELS\EASY CFM) 'NOCLOSE)
                                            DEP
                                            (+ DEP (LENGTH (CLABELS\FNVARS CFM))))))
                                   (DO ((D (CLABELS\FNDEFS CFM) (CDR D))
                                        (MD (DEPTH-ANALYZE (CLABELS\BODY CFM) DP)
                                            (MAX MD (DEPTH-ANALYZE (CAR D) DP))))
                                       ((NULL D) MD))))
                             (CCOMBINATION
                              (DO ((A (CCOMBINATION\ARGS CFM) (CDR A))
                                   (MD 0 (MAX MD (DEPTH-ANALYZE (CAR A) DEP))))
                                  ((NULL A) MD)))
                             (RETURN
                              (MAX (DEPTH-ANALYZE (RETURN\CONT CFM) DEP)
                                   (DEPTH-ANALYZE (RETURN\VAL CFM) DEP)))))))
```

Just as DEPTH-ANALYZE assigns locations in registers ("stack locations") for variables, so CLOSE-ANALYZE assigns locations in consed ("heap-allocated") environment structures for variables. The general idea is that if the value of a an accessible variable is not in a register, then it is in the structure which is in the register **ENV**. This structure can in principle be any structure whatsoever, according to the whim of the compiler. RABBIT's whim is to be very unclever; the structure of **ENV** is always a simple list of variable values. Thus a variable in the **ENV** structure is always accessed by a series of CDR operations and then one CAR operation.

(More clever would be to maintain the environment as a chained list of vectors, each vector representing a non-null contour. Then a variable could be accessed by a series of "CDR" operations equal to the number of contours (rather than the number of variables) between the binding and the reference, followed by a single indexing operation into the contour-vector. The number of "CDR" operations could be reduced by having a kind of "cache" for the results of such contour operations; such a cache would in fact be equivalent to the "display" used in many Algol implementations. If such a display were maintained, a variable could be accessed simply by a two-level indexing operation.)

Within the compiler an environment structure is also represented as a simple list, with the name of a variable occupying the position which its value will occupy in the run-time environment.

For every CLAMBDA, CONTINUATION, and CLABELS, a slot called CONSENV is filled in, which is a list representing what the environment structure will look like when the closure(s) for that construct are to be constructed, if any. This is done by walking over the cnode-tree and doing to the environment representation precisely what will be done to the real environment at run time.

There is a problem with the possibility that a variable may initially be in a register (because it was passed as an argument, for example), but must be transferred to a consed environment structure because the variable is referred to by the code of a closure to be constructed. There are two cases: either the variable has no WRITE-REFS property, or it does.

If it does not, then there is no problem with the value of the variable being in two or more places, so it is simply copied and consed into the environment as necessary. The CLOSEREFS slot of a function is a list of such variables which must be added to the consed environment before constructing the closure.

If the variable does have WRITE-REFS, then the value of the variable must have a single "home", to prevent inconsistencies when it is altered. (This is far easier than arranging for every ASET' operation to update all extant copies of a variable's value.) It is arranged that such variables, if they are referred to be closures (are in the CLOVARS set of the CLAMBDA which binds them) will exist <u>only</u> in the consed environment. Thus for each CLAMBDA the ASETVARS set is that subset of the lambda variables which have WRITE-REFS and are in the CLOVARS set. Before the body of the CLAMBDA is executed, a piece of code inserted by the code generator will transfer the variables from their registers immediately into the consed environment, and the values in the registers are thereafter never referred to.

```
;;; CLOSURE ANALYSIS FOR CPS VERSION

;;; FOR EACH CLAMBDA, CONTINUATION, AND CLABELS WE FILL IN:
;;;     CONSENV         THE CONSED ENVIRONMENT OF THE CLAMBDA,
;;;                     CONTINUATION, OR CLABELS (BEFORE ANY
;;;                     CLOSEREFS HAVE BEEN CONSED ON)
;;; FOR EACH CLAMBDA AND CONTINUATION WE FILL IN:
;;;     CLOSEREFS       A LIST OF VARIABLES REFERENCED BY THE CLAMBDA
;;;                     OR CONTINUATION WHICH ARE NOT IN THE CONSED
;;;                     ENVIRONMENT AT THE POINT OF THE CLAMBDA OR
;;;                     CONTINUATION AND SO MUST BE CONSED ONTO THE
;;;                     ENVIRONMENT AT CLOSURE TIME; HOWEVER, THESE
;;;                     NEED NOT BE CONSED ON IF THE CLAMBDA OR
;;;                     CONTINUATION IS IN FUNCTION POSITION OF
;;;                     A FATHER WHICH IS A CCOMBINATION OR RETURN
;;; FOR THE CLAMBDA'S IN THE FNDEFS OF A CLABELS, THESE MAY BE
;;; SLIGHTLY ARTIFICIAL FOR THE SAKE OF OPTIMIZATION (SEE BELOW).
;;; FOR EACH CLAMBDA WE FILL IN:
;;;     ASETVARS        A LIST OF THE VARIABLES BOUND IN THE CLAMBDA
;;;                     WHICH ARE EVER ASET AND SO MUST BE CONSED
;;;                     ONTO THE ENVIRONMENT IMMEDIATELY IF ANY
;;;                     CLOSURES OCCUR IN THE BODY
;;; FOR EACH CLABELS WE FILL IN:
;;;     FNENV           VARIABLES TO BE CONSED ONTO THE CURRENT CONSENV
;;;                     BEFORE CLOSING THE LABELS FUNCTIONS

;;; CENV IS THE CONSED ENVIRONMENT (A LIST OF VARIABLES)

(DEFINE FILTER-CLOSEREFS
        (LAMBDA (REFS CENV)
                (DO ((X REFS (CDR X))
                     (Y NIL
                        (IF (OR (MEMQ (CAR X) CENV)
                                (LET ((KFN (GET (CAR X) 'KNOWN-FUNCTION)))
                                     (AND KFN
                                          (EQ (EQCASE (TYPE (CNODE\CFORM KFN))
                                                      (CLAMBDA
                                                       (CLAMBDA\FNP (CNODE\CFORM KFN)))
                                                      (CONTINUATION
                                                       (CONTINUATION\FNP (CNODE\CFORM KFN))))
                                              'NOCLOSE))))
                            Y
                            (CONS (CAR X) Y))))
                    ((NULL X) (NREVERSE Y)))))
```

For each CLABELS a set called FNENV is computed. This is strictly an efficiency hack, which attempts to arrange it such that the several closures constructed for a CLABELS share environment structure. The union over all the variables needed is computed, and these variables are, at run time, all consed onto the environment before any of the closures is constructed. The hope is that the intersection of these sets is large, so that the total environment consing is less than if a separate environment were consed for each labelled closure.

FILTER-CLOSEREFS is a utility routine which, given a set of variables and an environment representation, returns that subset of the variables which are not already in the environment and so do not denote known NOCLOSE functions. (Those variables which are already in the consed environment or which do denote NOCLOSE functions of course need not be added to that consed environment.)

The argument CENV to CLOSE-ANALYZE is the representation of the consed environment (in **ENV**) which will be present when the code for CNODE is executed. The only processing of interest occurs for CLAMBDA, CONTINUATION, and CLABELS cnodes.

The CLOSEREFS of a CLAMBDA are those which are referred to by the CLAMBDA and which are not already in CENV, provided the CLAMBDA is not of type NOCLOSE. The ASETVARS are precisely those VARS which have WRITE-REFS and are in CLOVARS.

The processing for a CONTINUATION is similar. As a consistency check, we make sure the bound variable has no WRITE-REFS (it should be impossible for an ASET' to refer to the bound variable of a CONTINUATION).

For a CLABELS, the FNENV set is first calculated and added to CENV. This new CENV is then used to process the definitions and body of the CLABELS.

```
(DEFINE CLOSE-ANALYZE
        (LAMBDA (CNODE CENV)
                (LET ((CFM (CNODE\CFORM CNODE)))
                     (EQCASE (TYPE CFM)
                             (TRIVIAL NIL)
                             (CVARIABLE NIL)
                             (CLAMBDA
                              (LET ((CR (AND (NOT (EQ (CLAMBDA\FNP CFM) 'NOCLOSE))
                                             (FILTER-CLOSEREFS (CNODE\REFS CNODE) CENV)))
                                    (AV (DO ((V (CLAMBDA\VARS (CNODE\CFORM CNODE)) (CDR V))
                                             (A NIL (IF (AND (GET (CAR V) 'WRITE-REFS)
                                                             (MEMQ (CAR V)
                                                                   (CNODE\CLOVARS
                                                                    (CLAMBDA\BODY CFM))))
                                                        (CONS (CAR V) A)
                                                        A)))
                                            ((NULL V) A))))
                                   (ALTER-CLAMBDA CFM
                                                  (CONSENV := CENV)
                                                  (CLOSEREFS := CR)
                                                  (ASETVARS := AV))
                                   (CLOSE-ANALYZE (CLAMBDA\BODY CFM)
                                                  (APPEND AV CR CENV))))
                             (CONTINUATION
                              (AND (GET (CONTINUATION\VAR CFM) 'WRITE-REFS)
                                   (ERROR '|How could an ASET refer to a continuation variable?|
                                          CNODE
                                          'FAIL-ACT))
                              (LET ((CR (AND (NOT (EQ (CONTINUATION\FNP CFM) 'NOCLOSE))
                                             (FILTER-CLOSEREFS (CNODE\REFS CNODE) CENV))))
                                   (ALTER-CONTINUATION CFM
                                                       (CONSENV := CENV)
                                                       (CLOSEREFS := CR))
                                   (CLOSE-ANALYZE (CONTINUATION\BODY CFM)
                                                  (APPEND CR CENV))))
                             (CIF
                              (CLOSE-ANALYZE (CIF\PRED CFM) CENV)
                              (CLOSE-ANALYZE (CIF\CON CFM) CENV)
                              (CLOSE-ANALYZE (CIF\ALT CFM) CENV))
                             (CASET
                              (CLOSE-ANALYZE (CASET\CONT CFM) CENV)
                              (CLOSE-ANALYZE (CASET\BODY CFM) CENV))
                             (CLABELS
                              ((LAMBDA (CENV)
                                       (BLOCK (AMAPC (LAMBDA (D) (CLOSE-ANALYZE D CENV))
                                                     (CLABELS\FNDEFS CFM))
                                              (CLOSE-ANALYZE (CLABELS\BODY CFM) CENV)))
                                (COND ((CLABELS\EASY CFM)
                                       (DO ((D (CLABELS\FNDEFS CFM) (CDR D))
                                            (R NIL (UNION R (CNODE\REFS (CAR D)))))
                                           ((NULL D)
                                            (LET ((E (FILTER-CLOSEREFS R CENV)))
                                                 (ALTER-CLABELS CFM
                                                                (FNENV := E)
                                                                (CONSENV := CENV))
                                                 (APPEND E CENV)))))
                                      (T (ALTER-CLABELS CFM
                                                        (FNENV := NIL)
                                                        (CONSENV := CENV))
                                         CENV))))
                             (CCOMBINATION
                              (AMAPC (LAMBDA (A) (CLOSE-ANALYZE A CENV))
                                     (CCOMBINATION\ARGS CFM)))
                             (RETURN
                              (CLOSE-ANALYZE (RETURN\CONT CFM) CENV)
                              (CLOSE-ANALYZE (RETURN\VAL CFM) CENV))))))
```

We now come to the code generator, which is altogether about one-fourth of all the code making up RABBIT. Part of this is because much code which is conceptually singular is duplicated in several places (partly as a result of the design error in which CCOMBINATION and RETURN nodes, or CLAMBDA and CONTINUATION nodes, are treated distinctly; and also because a powerful text editor made it very easy to make copies of the code for various purposes!). The rest is just because code generation is fairly tricky and requires checking for special cases. A certain amount of peephole optimization is performed; this is not so much to improve the efficiency of the output code, as to make the output code easier to read for a human debugging RABBIT. A large fraction of the output code (perhaps ten to twenty percent) is merely comments of various kinds intended to help the debugger of RABBIT figure out what happened.

One problem in the code generator is that most functions need to be able to return two things: the code generated for a given cnode-tree, and a list of functions encountered in the cnode-tree, for which code is to generated separately later. We solve this problem by a stylistic trick, namely the explicit use of continuation-passing style. Many functions in the code generator take an argument named "C". This argument is itself a function of two arguments: the generated code and the deferred-function list. The function which is given C is expected to compute its two results and then invoke C, giving it the two results as arguments. (In practice a function which gets an argument C also gets an argument FNS, which is a deferred-functions list; the function is expected to add its deferred functions onto this list FNS, and give the augmented FNS list to C along with the generated code.)

Other arguments which are frequently passed within the code generator are CENV (a representation of the consed environment); BLOCKFNS, a list describing external functions compiled together in this "block" or "module" (this is used to compile a direct GOTO rather than a more expensive call to an external function, the theory being that several functions might be compiled together in a single module as with the InterLISP "block compiler"; this theory is not presently implemented, however, and so BLOCKFNS always has just one entry); PROGNAME, a symbol which at run time will have as its value the MacLISP SUBR pointer for the current module (this SUBR pointer is consed into closures of compiled functions, and so any piece of code which constructs a closure will need to refer to the value of this symbol); and RNL, the "rename list", an alist pairing internal variable names to pieces of code for accessing them (when code to reference a variable is to be generated, the piece of code in RNL is used if the variable is found in RNL, and otherwise a reference to the variable name itself (which is therefore global) is output).

COMPILATE is the topmost routine of the code generator. FN is the cnode-tree for a function to be compiled. The topmost cnode should of course be of type CLAMBDA or CONTINUATION. For a CLAMBDA, the call to REGSLIST sets up the initial RNL (rename list) for references to the arguments. Also, when COMP-BODY has returned the code (the innermost LAMBDA-expression in COMPILATE is the argument C given to COMP-BODY), SET-UP-ASETVARS is called to take care of copying the variables in the ASETVARS set into the consed environment. The code for a CONTINUATION is similar, except that a CONTINUATION has no ASETVARS and only one bound variable.

```
;;; CODE GENERATION ROUTINES

;;; PROGNAME:   NAME OF A VARIABLE WHICH AT RUN TIME WILL HAVE
;;;             AS VALUE THE SUBR POINTER FOR THE PROG
;;; FN:         THE FUNCTION TO COMPILE (A CLAMBDA OR CONTINUATION CNODE)
;;; EXTERNALP:  NON-NIL IF THE FUNCTION IS EXTERNAL
;;; RNL:        INITIAL RENAME LIST (NON-NIL ONLY FOR NOCLOSE FNS).
;;;             ENTRIES ARE: (VAR . CODE)
;;; BLOCKFNS:   AN ALIST OF FUNCTIONS IN THIS BLOCK.
;;;             ENTRIES ARE: (USERNAME CNODE)
;;; FNS:        A LIST OF TUPLES FOR FUNCTIONS YET TO BE COMPILED;
;;;             EACH TUPLE IS (PROGNAME FN RNL)
;;; C:          A CONTINUATION, TAKING:
;;;             CODE:   THE PIECE OF MACLISP CODE FOR THE FUNCTION
;;;             FNS:    AN AUGMENTED FNS LIST

(DEFINE COMPILATE
        (LAMBDA (PROGNAME FN RNL BLOCKFNS FNS C)
                (LET ((CFM (CNODE\CFORM FN)))
                     (EQCASE (TYPE CFM)
                             (CLAMBDA
                              (LET ((CENV (APPEND (CLAMBDA\ASETVARS CFM)
                                                  (CLAMBDA\CLOSEREFS CFM)
                                                  (CLAMBDA\CONSENV CFM))))
                                   (COMP-BODY (CLAMBDA\BODY CFM)
                                              (REGSLIST CFM T (ENVCARCDR CENV RNL))
                                              PROGNAME
                                              BLOCKFNS
                                              CENV
                                              FNS
                                              (LAMBDA (CODE FNS)
                                                      (C (SET-UP-ASETVARS CODE
                                                                          (CLAMBDA\ASETVARS CFM)
                                                                          (REGSLIST CFM NIL NIL))
                                                         FNS)))))
                             (CONTINUATION
                              (LET ((CENV (APPEND (CONTINUATION\CLOSEREFS CFM)
                                                  (CONTINUATION\CONSENV CFM))))
                                   (COMP-BODY (CONTINUATION\BODY CFM)
                                              (IF (EQ (CONTINUATION\FNP CFM) 'NOCLOSE)
                                                  (IF (NULL (CONTINUATION\TVARS CFM))
                                                      (ENVCARCDR CENV RNL)
                                                      (CONS (CONS (CONTINUATION\VAR CFM)
                                                                  (TEMPLOC (CONTINUATION\DEP CFM)))
                                                            (ENVCARCDR CENV RNL)))
                                                  (CONS (CONS (CONTINUATION\VAR CFM)
                                                              (CAR **ARGUMENT-REGISTERS**))
                                                        (ENVCARCDR CENV RNL)))
                                              PROGNAME
                                              BLOCKFNS
                                              CENV
                                              FNS
                                              C)))))))
```

ARGUMENT-REGISTERS is a list of the standard "registers" through which arguments are passed. In the standard SCHEME implementation this list is:

```
(**ONE** **TWO** **THREE** **FOUR**
     **FIVE** **SIX** **SEVEN** **EIGHT**)
```

DEPROGNIFY1 is a peephole optimizer. It takes a MacLISP form and returns a list of MacLISP forms. The idea is that if the given form is (PROGN ...), the keyword PROGN is stripped off; also, any irrelevant computations (references to variables or constants other than in the final position) are removed. (ATOMFLUSHP, when NIL, suppresses the removal of symbols, which in some cases may be MacLISP PROG tags). The purpose of this is to avoid multiple nesting of PROGN forms:

```
(PROGN (PROGN a b) (PROGN (PROGN c (PROGN d e) f) g))
```

Any code generation routine which constructs a PROGN with a component Q generated by another routine generally says:

```
"(PROGN (SETQ FOO 3) @(DEPROGNIFY Q) (GO ,THE-TAG))
```

The "@" means that the list of forms returned by the call to DEPROGNIFY (which is actually a macro which expands into a call to DEPROGNIFY1) is to be substituted into the list (PROGN ...) being constructed by the '"' operator. Thus rather than the nested PROGN code shown above, the code generator would instead produce:

```
(PROGN a b c d e f g)
```

which is much easier to read when debugging the output of RABBIT.

TEMPLOC is a little utility which given the number (in the DEP ordering used by DEPTH-ANALYZE) of a register returns the name of that register. **CONT+ARG-REGS** is the same as **ARGUMENT-REGISTERS** except that the name **CONT** is tacked onto the front. **CONT** is considered to be register 0. If N is greater than the number of the highest standard argument register, then a new register name of the form "-N-" is invented. Thus the additional temporary registers are called -11-, -12-, -13-, etc.

```
;;; DEPROGNIFY IS USED ONLY TO MAKE THE OUTPUT PRETTY BY ELIMINATING
;;; UNNECESSARY OCCURRENCES OF "PROGN".

(DEFMAC DEPROGNIFY (FORM) "(DEPROGNIFY1 ,FORM NIL))

(SET' *DEPROGNIFY-COUNT* 0)

(DEFINE DEPROGNIFY1
        (LAMBDA (FORM ATOMFLUSHP)
                (IF (OR (ATOM FORM) (NOT (EQ (CAR FORM) 'PROGN)))
                    (LIST FORM)
                    (DO ((X (CDR FORM) (CDR X))
                         (Z NIL (COND ((NULL (CDR X)) (CONS (CAR X) Z))
                                      ((NULL (CAR X))
                                       (INCREMENT *DEPROGNIFY-COUNT*)
                                       Z)
                                      ((ATOM (CAR X))
                                       (COND (ATOMFLUSHP
                                              (INCREMENT *DEPROGNIFY-COUNT*)
                                              Z)
                                             (T (CONS (CAR X) Z))))
                                      ((EQ (CAAR X) 'QUOTE)
                                       (INCREMENT *DEPROGNIFY-COUNT*)
                                       Z)
                                      (T (CONS (CAR X) Z)))))
                        ((NULL X) (NREVERSE Z))))))

(DEFINE TEMPLOC
        (LAMBDA (N)
                (LABELS ((LOOP
                          (LAMBDA (REGS J)
                                  (IF (NULL REGS)
                                      (IMPLODE (APPEND '(-) (EXPLODEN N) '(-)))
                                      (IF (= J 0)
                                          (CAR REGS)
                                          (LOOP (CDR REGS) (- J 1)))))))
                        (LOOP **CONT+ARG-REGS** N))))
```

ENVCARCDR takes a set of variables VARS representing the consed environment, and an old rename list RNL, and adds to RNL new entries for the variables, supplying pieces of code to access the environment structure. For example, suppose RNL were NIL, and VARS were (A B C). Then ENVCARCDR would produce the list:

```
((C . (CAR (CDR (CDR **ENV**))))
 (B . (CAR (CDR **ENV**)))
 (A . (CAR **ENV**)))
```

where each variable has been paired with a little piece of code which can be used to access it at run time. This example is not quite correct, however, because the peephole optimizer DECARCDRATE is called on the little pieces of code; DECARCDRATE collapses CAR-CDR chains to make them easier to read, and so the true result of ENVCARCDR would be:

```
((C . (CADDR **ENV**))
 (B . (CADR **ENV**))
 (A . (CAR **ENV**)))
```

```

(DEFINE ENVCARCDR
        (LAMBDA (VARS RNL)
                (DO ((X '**ENV** "(CDR ,X))
                     (V VARS (CDR V))
                     (R RNL (CONS (CONS (CAR V) (DECARCDRATE "(CAR ,X))) R)))
                    ((NULL V) R))))
```

REGSLIST takes a CLAMBDA cnode, a switch AVP, and a rename list RNL. It tacks onto RNL new entries which describe how to access the arguments of the CLAMBDA. This is complicated because there are three cases. (1) A NOCLOSE function takes its arguments in non-standard registers. (2) Other functions of not more than **NUMBER-OF-ARGUMENT-REGISTERS** (the length of the **ARGUMENT-REGISTERS** list) arguments takes their arguments in the standard registers. (3) All other functions takes a list of arguments in the first argument register (**ONE**), except for the continuation in **CONT**. The switch AVP tells whether or not the elements of ASETVARS should be included (non-nil means do not include).

As an example, suppose the CLAMBDA is a NOCLOSE with DEP = 12 and TVARS = (A B C D), and suppose that AVP = T and RNL = NIL. Then the result would be:

```
((D . -15-) (C . -14-) (B . -13-) (A . -12-))
```

As another example, suppose the CLAMBDA is of type EZCLOSE with VARS = (K X Y Z) and ASETVARS = (Y), and suppose that AVP = NIL and RNL = ((A . -12-)). Then the result would be:

```
((Z . **THREE**) (X . **ONE**) (K . **CONT**) (A . -12-))
```

SET-UP-ASETVARS takes a piece of code (the code for a CLAMBDA body), an ASETVARS set AV, and a rename list. If there are no ASETVARS, then just the code is returned, but otherwise a PROGN-form is returned, which ahead of the code has a SETQ which adds the ASETVARS to the environment. (LOOKUPICATE takes a variable and a RNL and returns a piece of code for referring to that variable.) For example, suppose we had:

```
CODE = (GO FOO)
AV = (A C)
RNL = ((C . -14-) (B . -13-) (A . -12-))
```

Then SET-UP-ASETVARS would return the code:

```
(PROGN (SETQ **ENV** (CONS -12- (CONS -14- **ENV**))) (GO FOO))
```

```
;;; AVP NON-NIL MEANS THAT ASETVARS ARE TO BE EXCLUDED FROM THE CONSED LIST.

(DEFINE REGSLIST
        (LAMBDA (CLAM AVP RNL)
                (LET ((AV (AND AVP (CLAMBDA\ASETVARS CLAM))))
                     (IF (EQ (CLAMBDA\FNP CLAM) 'NOCLOSE)
                         (DO ((J (CLAMBDA\DEP CLAM) (+ J 1))
                              (TV (CLAMBDA\TVARS CLAM) (CDR TV))
                              (R RNL
                                 (IF (MEMQ (CAR TV) AV)
                                     R
                                     (CONS (CONS (CAR TV) (TEMPLOC J)) R))))
                             ((NULL TV) R))
                         (LET ((VARS (CLAMBDA\VARS CLAM)))
                              (IF (> (LENGTH (CDR VARS)) **NUMBER-OF-ARG-REGS**)
                                  (DO ((X (CAR **ARGUMENT-REGISTERS**) "(CDR ,X))
                                       (V (CDR VARS) (CDR V))
                                       (R (CONS (CONS (CAR VARS) '**CONT**) RNL)
                                          (IF (MEMQ (CAR V) AV)
                                              R
                                              (CONS (CONS (CAR V) (DECARCDRATE "(CAR ,X))) R))))
                                      ((NULL V) R))
                                  (DO ((V VARS (CDR V))
                                       (X **CONT+ARG-REGS** (CDR X))
                                       (R RNL
                                          (IF (MEMQ (CAR V) AV)
                                              R
                                              (CONS (CONS (CAR V) (CAR X)) R))))
                                      ((NULL V) R))))))))

(DEFINE SET-UP-ASETVARS
        (LAMBDA (CODE AV RNL)
                (IF (NULL AV)
                    CODE
                    "(PROGN (SETQ **ENV**
                                  ,(DO ((A (REVERSE AV) (CDR A))
                                        (E '**ENV** "(CONS ,(LOOKUPICATE (CAR A) RNL) ,E)))
                                       ((NULL A) E)))
                            @(DEPROGNIFY CODE)))))
```

In the continuation-passing style, functions do not return values; instead, they apply a continuation to the value. Thus, the body of a CLAMBDA-expression is a form which is not expected to produce a value. On the other hand, such a form will have subforms which do produce values, for example references to variables.

Thus the forms to be dealt with in the code generator can be divided into those which produce values and those which do not. Initially the latter will always be attacked, as the body of a "function"; later the former will be seen. COMP-BODY takes a valueless form and compiles it. The routine ANALYZE, which we will see later, handles valued forms.

COMP-BODY instantiates a by now familiar theme: it simply dispatches on the type of BODY to some specialist routine. In the case of a CLABELS, it first compiles the body of the CLABELS (which itself is valueless if the CLABELS is valueless, and so a recursive call to COMP-BODY is used), and then goes to PRODUCE-LABELS. For a CCOMBINATION or RETURN, it does a three-way (for RETURN, two-way) sub-dispatch on whether the function is a TRIVFN, a CLAMBDA (or CONTINUATION), or something else.

The PRODUCE series of routines produce code for valueless forms. PRODUCE-IF calls ANALYZE on the predicate (which will produce a value), and COMP-BODY on the consequent and alternative (which produce no value because the entire CIF does not). The three pieces of resulting code are respectively called PRED, CON, and ALT. These are then given to CONDICATE, which generates a MacLISP COND form to be output.

```
;;; RNL IS THE "RENAME LIST": AN ALIST DESCRIBING HOW TO REFER TO THE VARIABLES IN THE
;;; ENVIRONMENT.  CENV IS THE CONSED ENVIRONMENT SEEN BY THE BODY.

(DEFINE
 COMP-BODY
 (LAMBDA (BODY RNL PROGNAME BLOCKFNS CENV FNS C)
         (LET ((CFM (CNODE\CFORM BODY)))
              (EQCASE (TYPE CFM)
                      (CIF
                       (PRODUCE-IF BODY RNL PROGNAME BLOCKFNS CENV FNS C))
                      (CASET
                       (PRODUCE-ASET BODY RNL PROGNAME BLOCKFNS CENV FNS C))
                      (CLABELS
                       (OR (EQUAL CENV (CLABELS\CONSENV CFM))
                           (ERROR '|Environment disagreement| BODY 'FAIL-ACT))
                       (LET ((LCENV (APPEND (CLABELS\FNENV CFM) CENV)))
                            (COMP-BODY
                             (CLABELS\BODY CFM)
                             (ENVCARCDR LCENV RNL)
                             PROGNAME
                             BLOCKFNS
                             LCENV
                             FNS
                             (LAMBDA (LBOD FNS)
                                     (PRODUCE-LABELS BODY LBOD RNL PROGNAME BLOCKFNS FNS C)))))
                      (CCOMBINATION
                       (LET ((FN (CNODE\CFORM (CAR (CCOMBINATION\ARGS CFM)))))
                            (COND ((EQ (TYPE FN) 'CLAMBDA)
                                   (PRODUCE-LAMBDA-COMBINATION BODY RNL PROGNAME BLOCKFNS CENV FNS C))
                                  ((AND (EQ (TYPE FN) 'TRIVIAL)
                                        (EQ (TYPE (NODE\FORM (TRIVIAL\NODE FN))) 'VARIABLE)
                                        (TRIVFN (VARIABLE\VAR (NODE\FORM (TRIVIAL\NODE FN)))))
                                   (PRODUCE-TRIVFN-COMBINATION BODY RNL PROGNAME BLOCKFNS CENV FNS C))
                                  (T (PRODUCE-COMBINATION BODY RNL PROGNAME BLOCKFNS CENV FNS C)))))
                      (RETURN
                       (LET ((FN (CNODE\CFORM (RETURN\CONT CFM))))
                            (IF (EQ (TYPE FN) 'CONTINUATION)
                                (PRODUCE-CONTINUATION-RETURN BODY RNL PROGNAME BLOCKFNS CENV FNS C)
                                (PRODUCE-RETURN BODY RNL PROGNAME BLOCKFNS CENV FNS C))))))))

(DEFINE PRODUCE-IF
        (LAMBDA (CNODE RNL PROGNAME BLOCKFNS CENV FNS C)
                (LET ((CFM (CNODE\CFORM CNODE)))
                     (ANALYZE (CIF\PRED CFM)
                              RNL
                              PROGNAME
                              BLOCKFNS
                              FNS
                              (LAMBDA (PRED FNS)
                                      (COMP-BODY (CIF\CON CFM)
                                                 RNL
                                                 PROGNAME
                                                 BLOCKFNS
                                                 CENV
                                                 FNS
                                                 (LAMBDA (CON FNS)
                                                         (COMP-BODY (CIF\ALT CFM)
                                                                    RNL
                                                                    PROGNAME
                                                                    BLOCKFNS
                                                                    CENV
                                                                    FNS
                                                                    (LAMBDA (ALT FNS)
                                                                            (C (CONDICATE PRED
                                                                                          CON
                                                                                          ALT)
                                                                               FNS))))))))))
```

PRODUCE-ASET first calls ANALYZE on the body, which must produce a value (to be assigned to the CASET variable). There are then two cases, depending on whether the CASET\CONT is a CONTINUATION or not.

If it is, then the body of the continuation is compiled (using COMP-BODY), and then LAMBDACATE is called to generate the invocation of the continuation. The routine OUTPUT-ASET generates the actual MacLISP SETQ (or other construct) for the CASET variable, using the environment location provided by LOOKUPICATE. All in all this case is very much like a RETURN with an explicit CONTINUATION, except that just before the continuation is invoked a SETQ is stuck in.

If the CASET\CONT is not a CONTINUATION, then ANALYZE is called on the CASET\CONT, and then a piece of code is output which sets **FUN** to the continuation, **ONE** (which is in the car of **ARGUMENT-REGISTERS**) to the value of the body (after also setting the CASET variable, using OUTPUT-ASET), and does (RETURN NIL), which is the SCHEME run-time protocol for invoking a continuation.

```
(DEFINE
 PRODUCE-ASET
 (LAMBDA (CNODE RNL PROGNAME BLOCKFNS CENV FNS C)
         (LET ((CFM (CNODE\CFORM CNODE)))
              (ANALYZE (CASET\BODY CFM)
                       RNL
                       PROGNAME
                       BLOCKFNS
                       FNS
                       (LAMBDA (BODY FNS)
                               (LET ((CONTCFM (CNODE\CFORM (CASET\CONT CFM))))
                                    (IF (EQ (TYPE CONTCFM) 'CONTINUATION)
                                        (COMP-BODY (CONTINUATION\BODY CONTCFM)
                                                   (IF (CONTINUATION\TVARS CONTCFM)
                                                       (CONS (CONS (CAR (CONTINUATION\TVARS CONTCFM))
                                                                   (TEMPLOC (CONTINUATION\DEP
                                                                             CONTCFM)))
                                                             (ENVCARCDR CENV RNL))
                                                       (ENVCARCDR CENV RNL))
                                                   PROGNAME
                                                   BLOCKFNS
                                                   CENV
                                                   FNS
                                                   (LAMBDA (CODE FNS)
                                                           (C (LAMBDACATE
                                                               (LIST (CONTINUATION\VAR CONTCFM))
                                                               (CONTINUATION\TVARS CONTCFM)
                                                               (CONTINUATION\DEP CONTCFM)
                                                               (LIST (OUTPUT-ASET
                                                                      (LOOKUPICATE (CASET\VAR CFM)
                                                                                   RNL)
                                                                      BODY))
                                                               (REMARK-ON (CASET\CONT CFM))
                                                               '**ENV**
                                                               CODE)
                                                              FNS)))
                                        (ANALYZE
                                         (CASET\CONT CFM)
                                         RNL
                                         PROGNAME
                                         BLOCKFNS
                                         FNS
                                         (LAMBDA (CONT FNS)
                                                 (C `(PROGN (SETQ **FUN** ,CONT)
                                                            (SETQ ,(CAR **ARGUMENT-REGISTERS**)
                                                                  ,(OUTPUT-ASET
                                                                     (LOOKUPICATE (CASET\VAR CFM)
                                                                                  RNL)
                                                                     BODY))
                                                            (RETURN NIL))
                                                    FNS))))))))))
```

PRODUCE-LABELS takes an already-compiled body LBOD. FNENV-FIX is a (possibly empty) list of pieces of code which will fix up the consed environment by adding the variables common to all the closures to be made up (this set was computed by CLOSE-ANALYZE and put in the FNENV slot of the CLABELS). The code for this addition is built from the list of variables by CONS-CLOSEREFS.

There are then three cases, depending on the type of closures to be constructed (NOCLOSE, EZCLOSE, or NIL). Suppose that the CLABELS is:

```
    (CLABELS ((FOO (LAMBDA ...))
              (BAR (LAMBDA ...)))
             <body>)
```

Let us see roughly what code is produced for each case.

For a NIL type (full closures), the idea is merely to create all the closures in standard form (but with a null environment), add them all to the consed environment, and then go back and clobber the environment portion of the closures with the new resulting environment, plus any other variables needed. Now a standard closure looks like (CBETA <value of progname> <tag> . <environment>). (At run time the value of the progname will be a MacLISP SUBR pointer for the module; the tag identifies the particular routine in the module.) In the DO loop, FNS accumulates the function definitions (to be compiled separately later), RP accumulates RPLACD forms for clobbering the closures, and CB accumulates constructors of CBETA lists. For our example, the generated code looks like:

```
    ((LAMBDA (FOO BAR)
             (SETQ **ENV** (CONS ... (CONS X43 **ENV***)...))
             (RPLACD (CDDR BAR) (CONS ... (CONS X72 **ENV**)...))
             (RPLACD (CDDR FOO) (CONS ... (CONS X69 **ENV**)...))
             <body>)
     (LIST 'CBETA ?-453 'FOO-TAG)
     (LIST 'CBETA ?-453 'BAR-TAG))
```

where ?-453 is the PROGNAME for the module containing the CLABELS, and FOO-TAG and BAR-TAG are the tags (whose names will actually look like FNVAR-91) for FOO and BAR. (Now in fact CLOSE-ANALYZE creates a null FNENV for type NIL CLABELS, and so the first SETQ would in fact not appear. However, the decision as to the form of the FNENV is only a heuristic, and so PRODUCE-LABELS is written so as to be prepared for any possible choice of FNENV and CLOSEREFS of individual labelled functions. In this way the heuristic in CLOSE-ANALYZE can be freely adjusted without having to change PRODUCE-LABELS.)

For the EZCLOSE case the "closures" need only contain environments, not also code pointers. A trick is needed here, however, to build the circular environment. When adding the labelled functions to the environment, we must somehow cons in an object; but we want this object to possibly be the environment itself! What we do instead is to make up a list of the tag, and later RPLACD this list cell with the environment. The tag is never used, but is useful for debugging. This method also makes the code very similar to the NIL case, the only difference being that the atom CBETA and the value of the PROGNAME are not consed onto each closure.

```
(DEFINE
 PRODUCE-LABELS
 (LAMBDA (CNODE LBOD RNL PROGNAME BLOCKFNS FNS C)
          (LET ((CFM (CNODE\CFORM CNODE)))
               (LET ((VARS (CLABELS\FNVARS CFM))
                     (DEFS (CLABELS\FNDEFS CFM))
                     (FNENV (CLABELS\FNENV CFM)))
                    (LET ((FNENV-FIX (IF FNENV "((SETQ **ENV** ,(CONS-CLOSEREFS FNENV RNL))))))
                         (EQCASE (CLABELS\EASY CFM)
                                 (NIL
                                  (DO ((V VARS (CDR V))
                                       (D DEFS (CDR D))
                                       (FNS FNS (CONS (LIST PROGNAME (CAR D) NIL) FNS))
                                       (RP NIL (CONS "(RPLACD (CDDR ,(CAR V))
                                                                      ,(CONS-CLOSEREFS
                                                                        (CLAMBDA\CLOSEREFS
                                                                         (CNODE\CFORM (CAR D)))
                                                                        RNL))
                                                     RP))
                                       (CB NIL (CONS "(LIST 'CBETA ,PROGNAME ',(CAR V)) CB)))
                                      ((NULL V)
                                       (C "((LAMBDA ,VARS
                                                     @FNENV-FIX
                                                     @RP
                                                     @(DEPROGNIFY LBOD))
                                            @(NREVERSE CB))
                                          FNS))))
                                 (EZCLOSE
                                  (DO ((V VARS (CDR V))
                                       (D DEFS (CDR D))
                                       (FNS FNS (CONS (LIST PROGNAME (CAR D) NIL) FNS))
                                       (RP NIL (CONS "(RPLACD ,(CAR V)
                                                                        ,(CONS-CLOSEREFS
                                                                          (CLAMBDA\CLOSEREFS
                                                                           (CNODE\CFORM (CAR D)))
                                                                          RNL))
                                                     RP))
                                       (CB NIL (CONS "(LIST ',(CAR V)) CB)))
                                      ((NULL V)
                                       (C "((LAMBDA ,VARS
                                                     @FNENV-FIX
                                                     @RP
                                                     @(DEPROGNIFY LBOD))
                                            @(NREVERSE CB))
                                          FNS))))
                                 (NOCLOSE
                                  (C "(PROGN @FNENV-FIX @(DEPROGNIFY LBOD))
                                      (DO ((V VARS (CDR V))
                                            (D DEFS (CDR D))
                                            (FNS FNS (CONS (LIST PROGNAME (CAR D) RNL) FNS)))
                                           ((NULL V) FNS))))))))))
```

One problem is that these "closures" are not of the same form as ordinary EZCLOSE closures, which do not have the tag. This is the purpose of the LABELS-FUNCTION properties which BIND-ANALYZE created; when a call to an EZCLOSE function is generated, the presence of a LABELS-FUNCTION property indicates that the "closure" itself is not the environment, but rather its cdr is. (It would be possible to do without the cell containing the tag, by instead making up the environment with values of NIL, then constructing the "closures" as simple environments, and then going back and clobbering the environment structure with the closure objects, rather than clobbering the closure objects themselves. The decision not to do this was rather arbitrary.) The generated code for the EZCLOSE case thus looks like:

```
((LAMBDA (FOO BAR)
         (SETQ **ENV** (CONS ... (CONS X43 **ENV***)...))
         (RPLACD (CDDR BAR) (CONS ... (CONS X72 **ENV**)...))
         (RPLACD (CDDR FOO) (CONS ... (CONS X69 **ENV**)...))
         <body>)
 (LIST 'FOO-TAG)
 (LIST 'BAR-TAG))
```

In the NOCLOSE case, no closures are made at run time for the labelled functions, and so the code consists merely of the FNENV-FIX (which, again, using the current heuristic in CLOSE-ANALYZE will always be null in the NOCLOSE case) and the code for the body:

```
(PROGN (SETQ **ENV** (CONS ... (CONS X43 **ENV**)...)) <body>)
```

In any case, of course, the labelled functions are added to the FNS list which is handed back to C for later compilation.

PRODUCE-LAMBDA-COMBINATION generates code for the case of ((CLAMBDA ...) arg1 ... argn). First a number of consistency checks are performed, to make sure the pass-2 analysis is not completely awry. Then code is generated for the body of the CLAMBDA, using COMP-BODY. Then all the arguments, which are of course expected to produce values, are given to MAPANALYZE, which will call ANALYZE on each in turn and return a list of the pieces of generated code (here called ARGS in the continuation handed to MAPANALYZE). Finally, LAMBDACATE is called to generate the code for entering the body after setting up the arguments in an appropriate manner. Notice the use of SET-UP-ASETVARS to generate any necessary additional code for adding ASETVARS to the consed environment on entering the body. (A more complicated compiler would in this situation add the argument values to the consed environment directly, rather than first putting them in registers (which is done by LAMBDACATE) and then moving the registers into the consed environment (which is done by SET-UP-ASETVARS). To do this, however, would involve destroying the modular distinction between LAMBDACATE and SET-UP-ASETVARS. The extra complications were deemed not worthwhile because in practice the ASETVARS set is almost always empty anyway.)

```
(DEFINE
 PRODUCE-LAMBDA-COMBINATION
 (LAMBDA (CNODE RNL PROGNAME BLOCKFNS CENV FNS C)
         (LET ((CFM (CNODE\CFORM CNODE)))
              (LET ((FN (CNODE\CFORM (CAR (CCOMBINATION\ARGS CFM)))))
                   (AND (CLAMBDA\CLOSEREFS FN)
                        (ERROR '|Functional LAMBDA has CLOSEREFS| CNODE 'FAIL-ACT))
                   (OR (EQUAL CENV (CLAMBDA\CONSENV FN))
                       (ERROR '|Environment disagreement| CNODE 'FAIL-ACT))
                   (OR (EQ (CLAMBDA\FNP FN) 'NOCLOSE)
                       (ERROR '|Non-NOCLOSE LAMBDA in function position| CNODE 'FAIL-ACT))
                   (COMP-BODY
                    (CLAMBDA\BODY FN)
                    (ENVCARCDR (CLAMBDA\ASETVARS FN)
                               (REGSLIST FN T (ENVCARCDR CENV RNL)))
                    PROGNAME
                    BLOCKFNS
                    (APPEND (CLAMBDA\ASETVARS FN) CENV)
                    FNS
                    (LAMBDA (BODY FNS)
                            (MAPANALYZE (CDR (CCOMBINATION\ARGS CFM))
                                        RNL
                                        PROGNAME
                                        BLOCKFNS
                                        FNS
                                        (LAMBDA (ARGS FNS)
                                                (C (LAMBDACATE (CLAMBDA\VARS FN)
                                                              (CLAMBDA\TVARS FN)
                                                              (CLAMBDA\DEP FN)
                                                              ARGS
                                                              (REMARK-ON
                                                               (CAR (CCOMBINATION\ARGS CFM)))
                                                              '**ENV**
                                                              (SET-UP-ASETVARS
                                                               BODY
                                                               (CLAMBDA\ASETVARS FN)
                                                               (REGSLIST FN NIL NIL)))
                                                   FNS)))))))))
```

PRODUCE-TRIVFN-COMBINATION handles a case like (CONS continuation arg1 arg2), i.e. a CCOMBINATION whose function position contains a TRIVFN. First all the arguments (<u>excluding</u> the continuation!) are given to MAPANALYZE; then a dispatch is made on whether the continuation is a CONTINUATION or a CVARIABLE, and one of two specialists is called.

PRODUCE-TRIVFN-COMBINATION-CONTINUATION handles a case like (CONS (CONTINUATION (Z) <body>) arg1 arg2). The idea here is to compile it approximately as if it were

```
    ((CONTINUATION (Z) <body>) (CONS arg1 arg2))
```

That is, the arguments are evaluated, the trivial function is given them to produce a value, and that value is then given to the continuation. Accordingly, the body of the CONTINUATION is compiled using COMP-BODY, and then LAMBDACATE takes care of setting up the argument (the fourth argument to LAMBDACATE is a list of the MacLISP code for invoking the trivial function) and invoking the body of the (necessarily NOCLOSE) CONTINUATION.

```
(DEFINE PRODUCE-TRIVFN-COMBINATION
        (LAMBDA (CNODE RNL PROGNAME BLOCKFNS CENV FNS C)
                (LET ((CFM (CNODE\CFORM CNODE)))
                     (LET ((FN (CNODE\CFORM (CAR (CCOMBINATION\ARGS CFM))))
                           (CONT (CNODE\CFORM (CADR (CCOMBINATION\ARGS CFM)))))
                          (MAPANALYZE (CDDR (CCOMBINATION\ARGS CFM))
                                      RNL
                                      PROGNAME
                                      BLOCKFNS
                                      FNS
                                      (LAMBDA (ARGS FNS)
                                              (EQCASE (TYPE CONT)
                                                     (CONTINUATION
                                                      (PRODUCE-TRIVFN-COMBINATION-CONTINUATION
                                                       CNODE RNL PROGNAME BLOCKFNS CENV
                                                       FNS C CFM FN CONT ARGS))
                                                     (CVARIABLE
                                                      (PRODUCE-TRIVFN-COMBINATION-CVARIABLE
                                                       CNODE RNL PROGNAME BLOCKFNS CENV
                                                       FNS C CFM FN CONT ARGS)))))))))

(DEFINE PRODUCE-TRIVFN-COMBINATION-CONTINUATION
        (LAMBDA (CNODE RNL PROGNAME BLOCKFNS CENV FNS C CFM FN CONT ARGS)
                (BLOCK (AND (CONTINUATION\CLOSEREFS CONT)
                            (ERROR '|CONTINUATION for TRIVFN has CLOSEREFS| CNODE 'FAIL-ACT))
                       (OR (EQ (CONTINUATION\FNP CONT) 'NOCLOSE)
                           (ERROR '|Non-NOCLOSE CONTINUATION for TRIVFN| CNODE 'FAIL-ACT))
                       (COMP-BODY (CONTINUATION\BODY CONT)
                                  (IF (CONTINUATION\TVARS CONT)
                                      (CONS (CONS (CAR (CONTINUATION\TVARS CONT))
                                                  (TEMPLOC (CONTINUATION\DEP CONT)))
                                            (ENVCARCDR CENV RNL))
                                      (ENVCARCDR CENV RNL))
                                  PROGNAME
                                  BLOCKFNS
                                  CENV
                                  FNS
                                  (LAMBDA (BODY FNS)
                                          (C (LAMBDACATE
                                              (LIST (CONTINUATION\VAR CONT))
                                              (CONTINUATION\TVARS CONT)
                                              (CONTINUATION\DEP CONT)
                                              (LIST "(,(VARIABLE\VAR (NODE\FORM (TRIVIAL\NODE FN)))
                                                      @ARGS))
                                              (REMARK-ON (CADR (CCOMBINATION\ARGS CFM)))
                                              '**ENV**
                                              BODY)
                                             FNS))))))
```

PRODUCE-TRIVFN-COMBINATION-CVARIABLE handles a case like (CONS CONT-43 arg1 arg2), where the continuation for a trivial function call is a CVARIABLE. In this situation the continuation is given to ANALYZE to generate MacLISP code for referring to it; there are then two cases, depending on whether the CVARIABLE has a KNOWN-FUNCTION property. (Note that before the decision is made, VAL names the piece of MacLISP code for calling the trivial function on the arguments.)

If the CVARIABLE denotes a KNOWN-FUNCTION, then it should be possible to invoke it by adjusting the environment, setting up the arguments in registers, and jumping to the code. First the environment adjustment is computed; ADJUST-KNOWNFN-CENV generates a piece of MacLISP code which will at run time compute the correct new environment in which the continuation will expect to run. There are then two subcases, depending on whether the KNOWN-FUNCTION is of type NOCLOSE or not. If it is, then LAMBDACATE is used to set up the arguments in the appropriate registers (the last argument of NIL indicates that there is no "body", but rather that the caller of LAMBDACATE takes the responsibility of jumping to the code). If it is not, then PSETQIFY is used, because the value will always go in **ONE** (which is the car of **ARGUMENT-REGISTERS**). In either case, a GO is generated to jump to the code (within the current module, of course) for the continuation.

If the continuation is not a KNOWN-FUNCTION, then the standard function linkage mechanism is used: the continuation is put into **FUN**, the value into **ONE**, and then (RETURN NIL) exits the module to request the SCHEME run-time interface to invoke the continuation in whatever manner is appropriate.

```
(DEFINE PRODUCE-TRIVFN-COMBINATION-CVARIABLE
        (LAMBDA (CNODE RNL PROGNAME BLOCKFNS CENV FNS C CFM FN CONT ARGS)
                (ANALYZE
                 (CADR (CCOMBINATION\ARGS CFM))
                 RNL
                 PROGNAME
                 BLOCKFNS
                 FNS
                 (LAMBDA (CONTF FNS)
                         (LET ((KF (GET (CVARIABLE\VAR CONT) 'KNOWN-FUNCTION))
                               (VAL "(,(VARIABLE\VAR (NODE\FORM (TRIVIAL\NODE FN))) @ARGS)))
                              (IF KF
                                  (LET ((KCFM (CNODE\CFORM KF)))
                                       (LET ((ENVADJ
                                              (ADJUST-KNOWNFN-CENV CENV
                                                                   (CVARIABLE\VAR CONT)
                                                                   CONTF
                                                                   (CONTINUATION\FNP KCFM)
                                                                   (APPEND
                                                                    (CONTINUATION\CLOSEREFS KCFM)
                                                                    (CONTINUATION\CONSENV KCFM)))))
                                            (C "(PROGN
                                                 @(IF (EQ (CONTINUATION\FNP KCFM)
                                                          'NOCLOSE)
                                                      (DEPROGNIFY
                                                       (LAMBDACATE (LIST (CONTINUATION\VAR KCFM))
                                                                   (CONTINUATION\TVARS KCFM)
                                                                   (CONTINUATION\DEP KCFM)
                                                                   (LIST VAL)
                                                                   (REMARK-ON KF)
                                                                   ENVADJ
                                                                   NIL))
                                                      (PSETQIFY (LIST ENVADJ VAL)
                                                                (LIST '**ENV**
                                                                      (CAR **ARGUMENT-REGISTERS**))))
                                                 (GO ,(CONTINUATION\NAME KCFM)))
                                               FNS)))
                                  (C "(PROGN (SETQ **FUN** ,CONTF)
                                             (SETQ ,(CAR **ARGUMENT-REGISTERS**) ,VAL)
                                             (RETURN NIL))
                                     FNS)))))))
```

PRODUCE-COMBINATION handles combinations whose function positions contain neither TRIVFNs nor CLAMBDAs. All of the arguments, including the function position itself and the continuation, are given to MAPANALYZE, resulting in a list FORM of pieces of MacLISP code. There are then two cases. If the function position is a VARIABLE (within a TRIVIAL - not a CVARIABLE!), then PRODUCE-COMBINATION-VARIABLE is used. Otherwise code is generated to use the standard SCHEME run-time interface: first set **FUN** to the function, then set up the arguments in the standard argument registers (PSETQ-ARGS generates the code for this), then set **NARGS** to the number of arguments (this does not include the continuation), and exit the module with (RETURN NIL).

PRODUCE-COMBINATION-VARIABLE first determines whether the variable has a KNOWN-FUNCTION property. If so, then the approach is very much as in TRIVFN-COMBINATION-CVARIABLE: first the environment adjustment is computed, then either LAMBDACATE or PSETQ-ARGS-ENV is used to adjust the environment and set up the arguments, and finally a GO to the piece of code for the KNOWN-FUNCTION is generated.

If the variable is not a KNOWN-FUNCTION, then it may still be in the list BLOCKFNS (which, recall, is a list of user functions included in this module). If so, the effect on the code generation strategy is roughly as if it were a KNOWN-FUNCTION. The environment adjustment is done differently, but a GO is generated to the piece of code for the called function.

In any other case, the standard interface is used. **FUN** is set to the function, the arguments are set up, **NARGS** is set to the number of arguments, and (RETURN NIL) exits the module.

```
(DEFINE PRODUCE-COMBINATION
         (LAMBDA (CNODE RNL PROGNAME BLOCKFNS CENV FNS C)
                 (MAPANALYZE (CCOMBINATION\ARGS (CNODE\CFORM CNODE))
                             RNL
                             PROGNAME
                             BLOCKFNS
                             FNS
                             (LAMBDA (FORM FNS)
                                     (C (LET ((F (CNODE\CFORM (CAR (CCOMBINATION\ARGS
                                                                   (CNODE\CFORM CNODE))))))
                                             (IF (AND (EQ (TYPE F) 'TRIVIAL)
                                                      (EQ (TYPE (NODE\FORM (TRIVIAL\NODE F)))
                                                          'VARIABLE))
                                                 (LET ((V (VARIABLE\VAR
                                                           (NODE\FORM (TRIVIAL\NODE F)))))
                                                      (PRODUCE-COMBINATION-VARIABLE
                                                       CNODE RNL PROGNAME BLOCKFNS CENV
                                                       FNS C FORM V (GET V 'KNOWN-FUNCTION)))
                                                 "(PROGN (SETQ **FUN** ,(CAR FORM))
                                                        @(PSETQ-ARGS (CDR FORM))
                                                         (SETQ **NARGS** ',(LENGTH (CDDR FORM)))
                                                         (RETURN NIL))))
                                        FNS)))))

(DEFINE PRODUCE-COMBINATION-VARIABLE
         (LAMBDA (CNODE RNL PROGNAME BLOCKFNS CENV FNS C FORM V KFN)
                 (IF KFN
                     (LET ((ENVADJ
                            (ADJUST-KNOWNFN-CENV CENV
                                                 V
                                                 (CAR FORM)
                                                 (CLAMBDA\FNP (CNODE\CFORM KFN))
                                                 (APPEND (CLAMBDA\CLOSEREFS (CNODE\CFORM KFN))
                                                         (CLAMBDA\CONSENV (CNODE\CFORM KFN))))))
                          (OR (EQ (TYPE (CNODE\CFORM KFN)) 'CLAMBDA)
                              (ERROR '|Known function not CLAMBDA| CNODE 'FAIL-ACT))
                          "(PROGN @(IF (EQ (CLAMBDA\FNP (CNODE\CFORM KFN)) 'NOCLOSE)
                                       (DEPROGNIFY
                                        (LAMBDACATE (CLAMBDA\VARS (CNODE\CFORM KFN))
                                                    (CLAMBDA\TVARS (CNODE\CFORM KFN))
                                                    (CLAMBDA\DEP (CNODE\CFORM KFN))
                                                    (CDR FORM)
                                                    (REMARK-ON KFN)
                                                    ENVADJ
                                                    NIL))
                                       (PSETQ-ARGS-ENV (CDR FORM) ENVADJ))
                                 (GO ,(CLAMBDA\NAME (CNODE\CFORM KFN)))))
                     (IF (ASSQ V BLOCKFNS)
                         "(PROGN @(PSETQ-ARGS (CDR FORM))
                                 @(IF (NOT (EQUAL (CLAMBDA\CONSENV
                                                   (CNODE\CFORM
                                                    (CADR (ASSQ V BLOCKFNS))))
                                                  CENV))
                                      "((SETQ **ENV** (CDDDR ,(CAR FORM))))))
                                 (GO ,(CLAMBDA\NAME (CNODE\CFORM (CADR (ASSQ V BLOCKFNS))))))
                         "(PROGN (SETQ **FUN** ,(CAR FORM))
                                 @(PSETQ-ARGS (CDR FORM))
                                 (SETQ **NARGS** ',(LENGTH (CDDR FORM)))
                                 (RETURN NIL))))))
```

ADJUST-KNOWNFN-CENV computes a piece of code for adjusting the environment. CENV is the internal representation (as a list of variable names) of the environment in which the generated code will be used. VAR is the name of the variable which names the function to be invoked, and for whose sake the environment is to be adjusted. VARREF is a piece of MacLISP code by which the run-time value of VAR may be accessed. FNP is the FNP type of the KNOWN-FUNCTION denoted by VAR. LCENV is the representation of the environment for the function. Thus, the generated code should compute LCENV given CENV.

The two easy cases are when LCENV=CENV, in which case the environment does not change, and when LCENV=NIL, in which case the run-time environment will also be NIL. Otherwise it breaks down into three cases on FNP.

For FNP=NOCLOSE, it must be true that LCENV is some tail of CENV; that is, there is a stack-like discipline for NOCLOSE functions, and so CENV was constructed by adding things to LCENV. The piece of code must therefore consist of some number of CDR operations on **ENV**. If this operation does not in fact produce LCENV, then there is an inconsistency in the compiler.

For FNP=EZCLOSE, then VARREF can be used to reference the run-time "closure"; this may require a CDR operation if the function is an EZCLOSE LABELS-FUNCTION (see PRODUCE-LABELS).

For FNP=NIL, then VARREF will refer to a full closure; the CDDDR of this closure is the environment.

PRODUCE-CONTINUATION-RETURN is, mutatis mutandis, identical to PRODUCE-LAMBDA-COMBINATION. This is a good example of the fact that much code was duplicated because of the early design decision to treat COMBINATION and RETURN as distinct data types.

```
(DEFINE ADJUST-KNOWNFN-CENV
        (LAMBDA (CENV VAR VARREF FNP LCENV)
                (COND ((EQUAL LCENV CENV) '**ENV**)
                      ((NULL LCENV) 'NIL)
                      (T (EQCASE FNP
                                 (NOCLOSE
                                  (DO ((X CENV (CDR X))
                                       (Y '**ENV** "(CDR ,Y))
                                       (I (- (LENGTH CENV) (LENGTH LCENV)) (- I 1)))
                                      ((< I 1)
                                       (IF (EQUAL X LCENV)
                                           (DECARCDRATE Y)
                                           (ERROR '|Cannot recover environment for known function|
                                                  VAR
                                                  'FAIL-ACT)))))
                                 (EZCLOSE
                                  (IF (GET VAR 'LABELS-FUNCTION)
                                      "(CDR ,VARREF)
                                      VARREF))
                                 (NIL "(CDDDR ,VARREF))))))

(DEFINE PRODUCE-CONTINUATION-RETURN
        (LAMBDA (CNODE RNL PROGNAME BLOCKFNS CENV FNS C)
                (LET ((CFM (CNODE\CFORM CNODE)))
                     (LET ((FN (CNODE\CFORM (RETURN\CONT CFM))))
                          (AND (CONTINUATION\CLOSEREFS FN)
                               (ERROR '|Functional CONTINUATION has CLOSEREFS| CNODE 'FAIL-ACT))
                          (OR (EQUAL CENV (CONTINUATION\CONSENV FN))
                              (ERROR '|Environment disagreement| CNODE 'FAIL-ACT))
                          (OR (EQ (CONTINUATION\FNP FN) 'NOCLOSE)
                              (ERROR '|Non-NOCLOSE CONTINUATION in function position|
                                     CNODE
                                     'FAIL-ACT))
                          (COMP-BODY (CONTINUATION\BODY FN)
                                     (IF (CONTINUATION\TVARS FN)
                                         (CONS (CONS (CAR (CONTINUATION\TVARS FN))
                                                     (TEMPLOC (CONTINUATION\DEP FN)))
                                               (ENVCARCDR CENV RNL))
                                         (ENVCARCDR CENV RNL))
                                     PROGNAME
                                     BLOCKFNS
                                     CENV
                                     FNS
                                     (LAMBDA (BODY FNS)
                                             (ANALYZE (RETURN\VAL CFM)
                                                      RNL
                                                      PROGNAME
                                                      BLOCKFNS
                                                      FNS
                                                      (LAMBDA (VAL FNS)
                                                              (C (LAMBDACATE
                                                                  (LIST (CONTINUATION\VAR FN))
                                                                  (CONTINUATION\TVARS FN)
                                                                  (CONTINUATION\DEP FN)
                                                                  (LIST VAL)
                                                                  (REMARK-ON (RETURN\CONT CFM))
                                                                  '**ENV**
                                                                  BODY)
                                                                 FNS)))))))))
```

PRODUCE-RETURN and PRODUCE-RETURN-1 together are almost identical to PRODUCE-COMBINATION and PRODUCE-COMBINATION-VARIABLE, except that the division between the two parts is different, and the BLOCKFNS trick is not applicable to RETURN.

PRODUCE-RETURN merely calls ANALYZE on each of the continuation and the value, and calls PRODUCE-RETURN-1.

PRODUCE-RETURN-1 checks to see whether the continuation is a KNOWN-FUNCTION. If so, the environment adjustment is computed, and code is generated in a way similar to previous routines. If not, the standard interface (involving (RETURN NIL)) is used. Notice the check to see if VAL is in fact **ONE** (the car of **ARGUMENT-REGISTERS**); if so, the redundant code (SETQ **ONE** **ONE**) is suppressed.

```
(DEFINE PRODUCE-RETURN
        (LAMBDA (CNODE RNL PROGNAME BLOCKFNS CENV FNS C)
                (LET ((CFM (CNODE\CFORM CNODE)))
                     (ANALYZE (RETURN\VAL CFM)
                              RNL
                              PROGNAME
                              BLOCKFNS
                              FNS
                              (LAMBDA (VAL FNS)
                                      (ANALYZE (RETURN\CONT CFM)
                                               RNL
                                               PROGNAME
                                               BLOCKFNS
                                               FNS
                                               (LAMBDA (CONT FNS)
                                                       (PRODUCE-RETURN-1
                                                        CNODE RNL PROGNAME BLOCKFNS
                                                        CENV FNS C CFM VAL CONT))))))))

(DEFINE PRODUCE-RETURN-1
        (LAMBDA (CNODE RNL PROGNAME BLOCKFNS CENV FNS C CFM VAL CONT)
                (IF (AND (EQ (TYPE (CNODE\CFORM (RETURN\CONT CFM))) 'CVARIABLE)
                         (GET (CVARIABLE\VAR (CNODE\CFORM (RETURN\CONT CFM)))
                              'KNOWN-FUNCTION))
                    (LET ((KCFM (CNODE\CFORM
                                 (GET (CVARIABLE\VAR
                                       (CNODE\CFORM (RETURN\CONT CFM)))
                                      'KNOWN-FUNCTION))))
                         (OR (EQ (TYPE KCFM) 'CONTINUATION)
                             (ERROR '|Known function not CONTINUATION| CNODE 'FAIL-ACT))
                         (LET ((ENVADJ
                                (ADJUST-KNOWNFN-CENV CENV
                                                     (CVARIABLE\VAR (CNODE\CFORM (RETURN\CONT CFM)))
                                                     CONT
                                                     (CONTINUATION\FNP KCFM)
                                                     (APPEND
                                                      (CONTINUATION\CLOSEREFS KCFM)
                                                      (CONTINUATION\CONSENV KCFM)))))
                              (C "(PROGN @(IF (EQ (CONTINUATION\FNP KCFM) 'NOCLOSE)
                                              (DEPROGNIFY
                                               (LAMBDACATE (LIST (CONTINUATION\VAR KCFM))
                                                           (CONTINUATION\TVARS KCFM)
                                                           (CONTINUATION\DEP KCFM)
                                                           (LIST VAL)
                                                           (REMARK-ON
                                                            (GET (CVARIABLE\VAR
                                                                  (CNODE\CFORM (RETURN\CONT CFM)))
                                                                 'KNOWN-FUNCTION))
                                                           ENVADJ
                                                           NIL))
                                              (PSETQIFY (LIST ENVADJ VAL)
                                                        (LIST '**ENV**
                                                              (CAR **ARGUMENT-REGISTERS**))))
                                         (GO ,(CONTINUATION\NAME KCFM)))
                                 FNS)))
                    (C "(PROGN (SETQ **FUN** ,CONT)
                               @(IF (NOT (EQ VAL (CAR **ARGUMENT-REGISTERS**)))
                                    "((SETQ ,(CAR **ARGUMENT-REGISTERS**) ,VAL)))
                               (RETURN NIL))
                       FNS))))
```

LAMBDACATE generates code for invoking a NOCLOSE KNOWN-FUNCTION. It arranges for the arguments to be evaluated and put in the proper registers, and also performs some optimizations.

VARS is a list of the variables which are to be bound. TVARS is a list of those variables (a subset of VARS) which will actually be passed through registers, as specified by the TVARS slot of the CLAMBDA or CONTINUATION; this is used for a consistency check on the optimizations of LAMBDACATE. DEP is the register depth of the function (the DEP slot). ARGS is a list of pieces of MacLISP code which have been generated for the arguments to the function. REM is a comment (usually one generated by REMARK-ON) to be included in the generated code for debugging purposes; this comment typically details the state of the environment and what variables are being passed through registers at this point. ENVADJ is a piece of MacLISP code (usually generated by ADJUST-KNOWNFN-CENV) to whose value **ENV** is to be set, to adjust the environment. BODY may be a list of pieces of MacLISP code which constitute the body of the known function, to be executed after the arguments are set up (typically because of a combination like ((LAMBDA ...) ...)), or it may be NIL, implying that the caller of LAMBDACATE intends to generate a GO to the code.

LAMBDACATE divides ARGS into three classes: (1) arguments which are themselves NOCLOSE KNOWN-FUNCTIONs -- such arguments actually have no actual run-time representation as a MacLISP data object, and so are not passed at all; (2) arguments whose corresponding variables are never referenced -- these are accumulated in EFFARGS, a list of arguments to be evaluated for effect only (presumably the optimizer eliminated those unreferenced arguments which had no side effects); and (3) arguments whose values are needed and are to be passed through the registers -- these are accumulated in REALARGS, and the corresponding variables in REALVARS.

When this loop is done, (the reverse of) REALVARS should equal TVARS, for it is the set of actually passed arguments.

The generated code first evaluates all the EFFARGS (if any), then sets all the proper registers to the REALARGS (this code is generated by PSETQ-TEMPS), then (after the remark REM) executed the BODY (which, if NIL, is empty).

For example, consider generating code for:

```
((LAMBDA (F A B) ... (F A) ...)
 (LAMBDA (X) ...)
 (CONS X Y)
 (PRINT Z))
```

where F denotes a NOCLOSE KNOWN-FUNCTION, and B is never referred to. Then the call to LAMBDACATE might look like this:

```
;;; HANDLE CASE OF INVOKING A KNOWN NOCLOSE FUNCTION OR CONTINUATION.
;;; FOR AN EXPLICIT ((LAMBDA ... BODY) ...), BODY IS THE BODY.
;;; OTHERWISE, IT IS NIL, AND SOMEONE WILL DO AN APPROPRIATE GO LATER.

(DEFINE LAMBDACATE
        (LAMBDA (VARS TVARS DEP ARGS REM ENVADJ BODY)
                (LABELS ((LOOP
                           (LAMBDA (V A REALVARS REALARGS EFFARGS)
                                    ;;REALVARS IS COMPUTED PURELY FOR ERROR-CHECKING
                                    (IF (NULL A)
                                        (LET ((B `(PROGN @(PSETQ-TEMPS (NREVERSE REALARGS) DEP ENVADJ)
                                                         ,REM
                                                         @(DEPROGNIFY BODY)))
                                              (RV (NREVERSE REALVARS)))
                                             (IF (NOT (EQUAL RV TVARS))
                                                 (ERROR '|TVARS screwup in LAMBDACATE|
                                                        `((VARS = ,VARS)
                                                          (TVARS = ,TVARS)
                                                          (REALVARS = ,RV))
                                                        'FAIL-ACT))
                                             (IF EFFARGS
                                                 `(PROGN @EFFARGS @(DEPROGNIFY B))
                                                 B))
                                        (COND ((LET ((KFN (GET (CAR V) 'KNOWN-FUNCTION)))
                                                    (AND KFN
                                                         (EQ (EQCASE (TYPE (CNODE\CFORM KFN))
                                                                     (CLAMBDA
                                                                      (CLAMBDA\FNP
                                                                       (CNODE\CFORM KFN)))
                                                                     (CONTINUATION
                                                                      (CONTINUATION\FNP
                                                                       (CNODE\CFORM KFN))))
                                                             'NOCLOSE)))
                                               (LOOP (CDR V) (CDR A) REALVARS REALARGS EFFARGS))
                                              ((OR (GET (CAR V) 'READ-REFS)
                                                   (GET (CAR V) 'WRITE-REFS))
                                               (LOOP (CDR V)
                                                     (CDR A)
                                                     (CONS (CAR V) REALVARS)
                                                     (CONS (CAR A) REALARGS)
                                                     EFFARGS))
                                              (T (LOOP (CDR V)
                                                       (CDR A)
                                                       REALVARS
                                                       REALARGS
                                                       (CONS (CAR A) EFFARGS))))))))
                         (LOOP VARS ARGS NIL NIL NIL))))
```

```
(LAMBDACATE '(F A B)
            '(A)
            12
            '(<illegal> (CONS X43 Y69) (PRINT Z91))
            <remark>
            **ENV**
            <body>)
```

where <illegal> is an object that should never be looked at (see ANALYZE-CLAMBDA); X43, Y69, and Z91 are pieces of code which refer to the variables X, Y, and Z; <remark> is some remark; the environment adjustment is assumed to be trivial; and <body> is the code for the body of the LAMBDA. The generated code would look something like this:

```
(PROGN (PRINT Z91)
       (SETQ -12- (CONS X43 Y69))
       <remark>
       <body>)
```

Notice that LAMBDACATE explicitly takes advantage of the fact that the execution of arguments for a combination may be arbitrarily reordered.

The various PSETQ... routines generate code to perform Parallel SETQs, i.e. the simultaneous assignment of several values to several values. The parallel nature is important, because some of the values may refer to other registers being assigned to, and a sequential series of assignments might not work.

The main routine here is PSETQIFY, which takes a list of arguments (pieces of MacLISP code which will generate values when executed at run-time) and a list of corresponding registers. One of two different methods is used depending on the number of values involved. Method 2 produces better code (this is obvious only when one understands the properties of the MacLISP compiler which will compile the MacLISP code into PDP-10 machine language). Unfortunately, it happened that when RABBIT was written there was a bug in the MacLISP compiler such that it often found itself unable to compile the code generated by Method 2. Moreover, the primary maintainer of the MacLISP compiler was on leave for a year. For this reason Method 3 was invented, which always works, but is considerably more expensive in terms of the PDP-10 code produced. (I concerned myself with this low level of detail only for this routine, because the code it produces is central to the whole code generator, and so its efficiency is of the greatest importance.) In order to achieve the best code, I determined empirically that Method 2 never failed as long as fewer than five values were involved. I might also add that a Method 1 was once used, which happened to provoke a different bug in the MacLISP compiler; Method 2 was invented in an attempt to circumvent that first bug! Now that the maintainer of the MacLISP compiler (Jon L White) has returned, it may soon be possible to remove Method 3 from RABBIT; but I think this story serves as an excellent example of pragmatic engineering to get around immediate obstacles (also known as a "kludge").

```
;;; GENERATE PARALLEL SETQ'ING OF REGISTERS TO ARGS.
;;; RETURNS A LIST OF THINGS; ONE WRITES @(PSETQIFY ...) WITHIN ".

(DEFINE PSETQIFY
        (LAMBDA (ARGS REGISTERS)
                (IF (< (LENGTH ARGS) 5)
                    (PSETQIFY-METHOD-2 ARGS REGISTERS)
                    (PSETQIFY-METHOD-3 ARGS REGISTERS))))


(DEFINE PSETQIFY-METHOD-2
        (LAMBDA (ARGS REGISTERS)
                (LABELS ((PSETQ1
                          (LAMBDA (A REGS QVARS SETQS USED)
                                  (IF (NULL A)
                                      (IF (NULL SETQS)
                                          NIL
                                          (IF (NULL (CDR SETQS))
                                              "((SETQ ,(CADAR SETQS) ,(CAR USED)))
                                              ;;IMPORTANT: DO NOT NREVERSE THE SETQS!
                                              ;;MAKES MACLISP COMPILER WIN BETTER.
                                              "(((LAMBDA ,(NREVERSE QVARS) @SETQS)
                                                 @(NREVERSE USED)))))
                                      (IF (EQ (CAR A) (CAR REGS))       ;AVOID USELESS SETQ'S
                                          (PSETQ1 (CDR A)
                                                  (CDR REGS)
                                                  QVARS
                                                  SETQS
                                                  USED)
                                          ((LAMBDA (QV)
                                                   (PSETQ1 (CDR A)
                                                           (CDR REGS)
                                                           (CONS QV QVARS)
                                                           (CONS "(SETQ ,(CAR REGS) ,QV) SETQS)
                                                           (CONS (CAR A) USED)))
                                           (GENTEMP 'Q)))))))
                        (PSETQ1 ARGS REGISTERS NIL NIL NIL))))
```

Method 2 essentially uses local MacLISP LAMBDA variables to temporarily name the values before assignment to the registers, while Method 3 uses global variables. (Method 2 produces better code because the MacLISP compiler can allocate the local variables on a stack, one by one, and then pop them off in reverse order into the "registers".) Both methods perform two peephole optimizations: (1) If a value-register pair calls for setting the register to its own contents, that SETQ is eliminated. (2) If this elimination reduces the number of SETQs to zero or one, then NIL or a single SETQ is produced, rather than the more complicated and general piece of code.

As examples, (PSETQIFY '(-12- -12- (CDR -13-)) '(-11- -12- -13-)) would produce:

```
((LAMBDA (Q-43 Q-44)
         (SETQ -13- Q-44)
         (SETQ -11- Q-43))
 -12-
 (CDR -13-))
```

(note that (SETQ -12- -12-) was eliminated), and

```
(PSETQIFY '(-23- -21- -24- -25- -22-) '(-21- -22- -23- -24- -25-))
```

would produce:

```
(PROG () (DECLARE (SPECIAL -21--TEMP -22--TEMP -23--TEMP -24--TEMP -25--TEMP)
      (SETQ -25--TEMP -22-)
      (SETQ -24--TEMP -25-)
      (SETQ -23--TEMP -24-)
      (SETQ -22--TEMP -21-)
      (SETQ -21--TEMP -23-)
      (SETQ -25- -25--TEMP)
      (SETQ -24- -24--TEMP)
      (SETQ -23- -23--TEMP)
      (SETQ -22- -22--TEMP)
      (SETQ -21- -21--TEMP))
```

The only reason for using PROG is so that the DECLARE form could be included for the benefit of the MacLISP compiler.

The examples here are slightly incorrect; PSETQIFY actually produces a list of MacLISP forms, so that when no SETQs are produced the resulting NIL is interpreted as no code at all.

In principle the elimination of redundant SETQs should be performed before choosing which method to use, so that there will be a maximal chance of using the more efficient Method 2. I chose not to only so that the two methods would remain distinct pieces of code and thus easily replaceable.

```
(DEFINE PSETQIFY-METHOD-3
        (LAMBDA (ARGS REGISTERS)
                (LABELS ((PSETQ1
                          (LAMBDA (A REGS QVARS SETQS USED)
                                  (IF (NULL A)
                                      (IF (NULL SETQS)
                                          NIL
                                          (IF (NULL (CDR SETQS))
                                              "((SETQ ,(CADAR SETQS) ,(CADDR (CAR USED))))
                                              "((PROG () (DECLARE (SPECIAL @QVARS)) @USED @SETQS) )))
                                      (IF (EQ (CAR A) (CAR REGS))       ;AVOID USELESS SETQ'S
                                          (PSETQ1 (CDR A)
                                                  (CDR REGS)
                                                  QVARS
                                                  SETQS
                                                  USED)
                                          ((LAMBDA (QV)
                                                   (PSETQ1 (CDR A)
                                                           (CDR REGS)
                                                           (CONS QV QVARS)
                                                           (CONS "(SETQ ,(CAR REGS) ,QV) SETQS)
                                                           (CONS "(SETQ ,QV ,(CAR A)) USED)))
                                           (CATENATE (CAR REGS) '|-TEMP|))))))))
                         (PSETQ1 ARGS REGISTERS NIL NIL NIL))))
```

PSETQ-ARGS is a handy routine which calls PSETQ-ARGS-ENV with an ENVADJ of **ENV**, knowing that later the redundant "(SETQ **ENV** **ENV**)" will be eliminated.

PSETQ-ARGS-ENV takes a set of arguments and an environment adjustment, and arranges to call PSETQIFY so as to set up the standard argument registers. Recall that how this is done depends on whether the number of arguments exceeds **NUMBER-OF-ARG-REGS**; if it does, then a list of the arguments (except the continuation) is passed in **ONE**. **ENV+CONT+ARG-REGS** is the same as **ARGUMENT-REGISTERS**, except that both the names **ENV** and **CONT** are adjoined to the front. It can be quite critical that **ENV** and the argument registers be assigned to in parallel, because the computation of the argument values may well refer to variables in the environment, whereas the environment adjustment may be taken from a closure residing in one of the argument registers.

PSETQ-TEMPS is similar to PSETQ-ARGS-ENV, but is used on registers other than the standard argument-passing registers. It takes ARGS and ENVADJ as before, but also a depth DEP which is the number of the first register to be assigned to. TEMPLOC is used to generate the register names, then **ENV** is tacked on and PSETQIFY does the real work.

MAPANALYZE is a simple loop which maps over a list of cnode-trees and calls ANALYZE on each. A list of the results returned by ANALYZE is given to C. Also, FNS is chained through the calls to ANALYZE, so that all functions to be compiled later will have been accumulated properly.

```
(DEFINE PSETQ-ARGS
        (LAMBDA (ARGS)
                (PSETQ-ARGS-ENV ARGS '**ENV**)))

(DEFINE PSETQ-ARGS-ENV
        (LAMBDA (ARGS ENVADJ)
                (IF (> (LENGTH ARGS) (+ **NUMBER-OF-ARG-REGS** 1))
                    (PSETQIFY (LIST ENVADJ (CAR ARGS) (CONS 'LIST (CDR ARGS)))
                              **ENV+CONT+ARG-REGS**)
                    (PSETQIFY (CONS ENVADJ ARGS) **ENV+CONT+ARG-REGS**))))

(DEFINE PSETQ-TEMPS
        (LAMBDA (ARGS DEP ENVADJ)
                (DO ((A ARGS (CDR A))
                     (J DEP (+ J 1))
                     (R NIL (CONS (TEMPLOC J) R)))
                    ((NULL A)
                     (PSETQIFY (CONS ENVADJ ARGS)
                               (CONS '**ENV** (NREVERSE R)))))))


(DEFINE MAPANALYZE
        (LAMBDA (FLIST RNL PROGNAME BLOCKFNS FNS C)
                (LABELS ((LOOP
                          (LAMBDA (F Z FNS)
                                  (IF (NULL F)
                                      (C (NREVERSE Z) FNS)
                                      (ANALYZE (CAR F)
                                               RNL
                                               PROGNAME
                                               BLOCKFNS
                                               FNS
                                               (LAMBDA (STUFF FNS)
                                                       (LOOP (CDR F)
                                                             (CONS STUFF Z)
                                                             FNS)))))))
                        (LOOP FLIST NIL FNS))))
```

ANALYZE is the routine called to compile a piece of code which is expected to produce a value. ANALYZE itself is primarily a dispatch to specialists. For the "simple" case of a "trivial" form, TRIVIALIZE is used to generate the code. For the simple case of a CVARIABLE, ANALYZE simply uses LOOKUPICATE to get the code for the variable reference.

ANALYZE-CLAMBDA has three cases based on FNP. For type NIL, code is generated to create a full closure of the form (CBETA <value of progname> <tag> . <environment>). CONS-CLOSEREFS generates the code to add the CLOSEREFS to the existing consed environment for making this closure. For type EZCLOSE, just the environment part is created, again using CONS-CLOSEREFS. For type NOCLOSE, the generated "code" should never be referenced at all -- it is not even passed as an argument as such -- and so a little message to the debugger is returned as the "code", which of course must not appear in the final code for the module. For all three cases, the code for the function is added to the FNS list for later compilation.

ANALYZE-CONTINUATION is essentially identical to ANALYZE-CLAMBDA.

```
(DEFINE ANALYZE
        (LAMBDA (CNODE RNL PROGNAME BLOCKFNS FNS C)
                (LET ((CFM (CNODE\CFORM CNODE)))
                     (EQCASE (TYPE CFM)
                             (TRIVIAL
                              (C (TRIVIALIZE (TRIVIAL\NODE CFM) RNL) FNS))
                             (CVARIABLE
                              (C (LOOKUPICATE (CVARIABLE\VAR CFM) RNL) FNS))
                             (CLAMBDA
                              (ANALYZE-CLAMBDA CNODE RNL PROGNAME BLOCKFNS FNS C CFM))
                             (CONTINUATION
                              (ANALYZE-CONTINUATION CNODE RNL PROGNAME BLOCKFNS FNS C CFM))
                             (CIF
                              (ANALYZE-CIF CNODE RNL PROGNAME BLOCKFNS FNS C CFM))
                             (CLABELS
                              (ANALYZE-CLABELS CNODE RNL PROGNAME BLOCKFNS FNS C CFM))
                             (CCOMBINATION
                              (ANALYZE-CCOMBINATION CNODE RNL PROGNAME BLOCKFNS FNS C CFM))
                             (RETURN
                              (ANALYZE-RETURN CNODE RNL PROGNAME BLOCKFNS FNS C CFM))))))

(DEFINE ANALYZE-CLAMBDA
        (LAMBDA (CNODE RNL PROGNAME BLOCKFNS FNS C CFM)
                (EQCASE (CLAMBDA\FNP CFM)
                        (NIL
                         (C "(CONS 'CBETA
                                   (CONS ,PROGNAME
                                         (CONS ',(CLAMBDA\NAME CFM)
                                                ,(CONS-CLOSEREFS (CLAMBDA\CLOSEREFS CFM)
                                                                RNL))))
                            (CONS (LIST PROGNAME CNODE NIL) FNS)))
                        (EZCLOSE
                         (C (CONS-CLOSEREFS (CLAMBDA\CLOSEREFS CFM) RNL)
                            (CONS (LIST PROGNAME CNODE NIL) FNS)))
                        (NOCLOSE
                         (C '|Shouldn't ever be seen - NOCLOSE CLAMBDA|
                            (CONS (LIST PROGNAME CNODE RNL) FNS))))))

(DEFINE ANALYZE-CONTINUATION
        (LAMBDA (CNODE RNL PROGNAME BLOCKFNS FNS C CFM)
                (EQCASE (CONTINUATION\FNP CFM)
                        (NIL
                         (C "(CONS 'CBETA
                                   (CONS ,PROGNAME
                                         (CONS ',(CONTINUATION\NAME CFM)
                                                ,(CONS-CLOSEREFS (CONTINUATION\CLOSEREFS CFM)
                                                                RNL))))
                            (CONS (LIST PROGNAME CNODE NIL) FNS)))
                        (EZCLOSE
                         (C (CONS-CLOSEREFS (CONTINUATION\CLOSEREFS CFM) RNL)
                            (CONS (LIST PROGNAME CNODE NIL) FNS)))
                        (NOCLOSE
                         (C '|Shouldn't ever be seen - NOCLOSE CONTINUATION|
                            (CONS (LIST PROGNAME CNODE RNL) FNS))))))
```

ANALYZE-CIF merely calls ANALYZE recursively on the predicate, consequent, and alternative, and then uses CONDICATE to construct a MacLISP COND form.

ANALYZE-CLABELS calls ANALYZE recursively on the body of the CLABELS, and then calls PRODUCE-LABELS to do the rest. (Unlike the other PRODUCE- functions, PRODUCE-LABELS does not depend on generating code which does not produce a value. It accepts an already-compiled body, and builds around that the framework for constructing the mutually referent functions. If the body was compiled using COMP-BODY, then the code generated by PRODUCE-LABELS will produce no value; but if the body was compiled using ANALYZE, then it will produce a value.)

```
(DEFINE ANALYZE-CIF
        (LAMBDA (CNODE RNL PROGNAME BLOCKFNS FNS C CFM)
                (ANALYZE (CIF\PRED CFM)
                         RNL
                         PROGNAME
                         BLOCKFNS
                         FNS
                         (LAMBDA (PRED FNS)
                                 (ANALYZE (CIF\CON CFM)
                                          RNL
                                          PROGNAME
                                          BLOCKFNS
                                          FNS
                                          (LAMBDA (CON FNS)
                                                  (ANALYZE (CIF\ALT CFM)
                                                           RNL
                                                           PROGNAME
                                                           BLOCKFNS
                                                           FNS
                                                           (LAMBDA (ALT FNS)
                                                                   (C (CONDICATE PRED CON ALT)
                                                                      FNS)))))))))

(DEFINE ANALYZE-CLABELS
        (LAMBDA (CNODE RNL PROGNAME BLOCKFNS FNS C CFM)
                (ANALYZE (CLABELS\BODY CFM)
                         (ENVCARCDR (APPEND (CLABELS\FNENV CFM)
                                            (CLABELS\CONSENV CFM))
                                    RNL)
                         PROGNAME
                         BLOCKFNS
                         FNS
                         (LAMBDA (LBOD FNS)
                                 (PRODUCE-LABELS CNODE LBOD RNL PROGNAME BLOCKFNS FNS C)))))
```

ANALYZE-CCOMBINATION requires the function to be a CLAMBDA (for if it were not, then something too complicated for continuation-passing style is going on). ANALYZE is called on the body of the CLAMBDA, and then on all the arguments (using MAPANALYZE); finally LAMBDACATE is used to generate the code. (LAMBDACATE is much like PRODUCE-LABELS, in that it is handed a body, and whether the generated code produces a value depends only on whether the body does.)

ANALYZE-RETURN is essentially just like ANALYZE-CCOMBINATION.

```
(DEFINE
 ANALYZE-CCOMBINATION
 (LAMBDA (CNODE RNL PROGNAME BLOCKFNS FNS C CFM)
         (LET ((FN (CNODE\CFORM (CAR (CCOMBINATION\ARGS CFM)))))
              (IF (EQ (TYPE FN) 'CLAMBDA)
                  (ANALYZE (CLAMBDA\BODY FN)
                           (ENVCARCDR (CLAMBDA\ASETVARS FN)
                                      (REGSLIST FN T (ENVCARCDR (CLAMBDA\CONSENV FN) RNL)))
                           PROGNAME
                           BLOCKFNS
                           FNS
                           (LAMBDA (BODY FNS)
                                   (MAPANALYZE
                                    (CDR (CCOMBINATION\ARGS CFM))
                                    RNL
                                    PROGNAME
                                    BLOCKFNS
                                    FNS
                                    (LAMBDA (ARGS FNS)
                                            (C (LAMBDACATE (CLAMBDA\VARS FN)
                                                           (CLAMBDA\TVARS FN)
                                                           (CLAMBDA\DEP FN)
                                                           ARGS
                                                           (REMARK-ON (CAR (CCOMBINATION\ARGS CFM)))
                                                           '**ENV**
                                                           (SET-UP-ASETVARS BODY
                                                                            (CLAMBDA\ASETVARS FN)
                                                                            (REGSLIST FN NIL NIL)))
                                               FNS)))))
                  (ERROR '|Non-trivial Function in ANALYZE-CCOMBINATION| CNODE 'FAIL-ACT)))))

(DEFINE ANALYZE-RETURN
        (LAMBDA (CNODE RNL PROGNAME BLOCKFNS FNS C CFM)
                (LET ((FN (CNODE\CFORM (RETURN\CONT CFM))))
                     (IF (EQ (TYPE FN) 'CONTINUATION)
                         (ANALYZE (CONTINUATION\BODY FN)
                                  (IF (CONTINUATION\TVARS FN)
                                     (CONS (CONS (CAR (CONTINUATION\TVARS FN))
                                                 (TEMPLOC (CONTINUATION\DEP FN)))
                                           (ENVCARCDR (CONTINUATION\CONSENV FN) RNL))
                                     (ENVCARCDR (CONTINUATION\CONSENV FN) RNL))
                                  PROGNAME
                                  BLOCKFNS
                                  FNS
                                  (LAMBDA (BODY FNS)
                                          (ANALYZE (RETURN\VAL CFM)
                                                   RNL
                                                   PROGNAME
                                                   BLOCKFNS
                                                   FNS
                                                   (LAMBDA (ARG FNS)
                                                           (C (LAMBDACATE
                                                                (LIST (CONTINUATION\VAR FN))
                                                                (CONTINUATION\TVARS FN)
                                                                (CONTINUATION\DEP FN)
                                                                (LIST ARG)
                                                                (REMARK-ON (RETURN\CONT CFM))
                                                                '**ENV**
                                                                BODY)
                                                               FNS)))))
                         (ERROR '|Non-trivial Function in ANALYZE-RETURN| CNODE 'FAIL-ACT)))))
```

LOOKUPICATE (I make no apology for the choice of the name of this or any other function; suffice it to say that a function named LOOKUP already existed in the SCHEME interpreter) takes a variable name VAR and a rename list RNL, and returns a piece of MacLISP code for referring to that variable. If an entry is in RNL for the variable, that entry contains the desired code. Otherwise a global variable reference must be constructed. This will simply be a reference to the MacLISP variable, unless it is the name of a TRIVFN. In this case a GETL form is constructed. (This is a big kludge which does not always work, and is done this way as a result of a rather unclean hack in the SCHEME interpreter which interfaces MacLISP functions with SCHEME functions.)

CONS-CLOSEREFS constructs a piece of MacLISP code which will cons onto the value of **ENV** all the variables in the set CLOSEREFS. This is a simple loop which uses LOOKUPICATE to generate code, and constructs a chain of calls to CONS. For example, (CONS-CLOSEREFS '(A B C) NIL) would produce:

```
(CONS A (CONS B (CONS C **ENV**)))
```

Notice the use of REVERSE to preserve an order assumed by other routines.

OUTPUT-ASET takes two pieces of code: VARREF, which refers to a variable, and BODY, which produces a value to be assigned to the variable. From the form of VARREF a means of assigning to the variable is deduced. (This implies that OUTPUT-ASET knows about all forms of code which might possibly be returned by LOOKUPICATE and, a fortiori, which might appear in a RNL.) For example, if the reference is (CADR (CDDDDR **ENV**)), OUTPUT-ASET would generate (RPLACA (CDR (CDDDDR **ENV**)) <body>).

```
(DEFINE LOOKUPICATE
        (LAMBDA (VAR RNL)
                ((LAMBDA (SLOT)
                         (IF SLOT (CDR SLOT)
                             (IF (TRIVFN VAR)
                                 "(GETL ',VAR '(EXPR SUBR LSUBR))
                                 VAR)))
                 (ASSQ VAR RNL))))

(DEFINE CONS-CLOSEREFS
        (LAMBDA (CLOSEREFS RNL)
                (DO ((CR (REVERSE CLOSEREFS) (CDR CR))
                     (X '**ENV** "(CONS ,(LOOKUPICATE (CAR CR) RNL) ,X)))
                    ((NULL CR) X))))

(DEFINE OUTPUT-ASET
        (LAMBDA (VARREF BODY)
                (COND ((ATOM VARREF)
                       "(SETQ ,VARREF ,BODY))
                      ((EQ (CAR VARREF) 'CAR)
                       "(CAR (RPLACA ,(CADR VARREF) ,BODY)))
                      ((EQ (CAR VARREF) 'CADR)
                       "(CAR (RPLACA (CDR ,(CADR VARREF)) ,BODY)))
                      ((EQ (CAR VARREF) 'CADDR)
                       "(CAR (RPLACA (CDDR ,(CADR VARREF)) ,BODY)))
                      ((EQ (CAR VARREF) 'CADDDR)
                       "(CAR (RPLACA (CDDDR ,(CADR VARREF)) ,BODY)))
                      (T (ERROR '|Unknown ASET discipline - OUTPUT-ASET| VARREF 'FAIL-ACT)))))
```

CONDICATE takes the three conponents of an IF conditional, and constructs a MacLISP COND form. It also performs a simple peephole optimization:

```
(COND (a b)
      (T (COND (c d) ...)))
```

becomes:

```
(COND (a b) (c d) ...)
```

Also, DEPROGNIFY is used to take advantage of the fact that MacLISP COND clauses are implicitly PROGN forms. Thus:

```
(CONDICATE '(NULL X) '(PROGN (PRINT X) Y) '(COND ((NULL Y) X) (T FOO)))
```

would produce:

```
(COND ((NULL X) (PRINT X) Y) ((NULL Y) X) (T FOO))
```

DECARCDRATE is a peephole optimizer which attempts to collapse CAR/CDR chains in a piece of MacLISP code to make it more readable. For example:

```
(CAR (CDR (CDR (CAR (CDR (CAR (CDR (CDR (CDR (CDR X))))))))))
```

would become:

```
(CADDR (CADR (CADDDR (CDR X))))
```

The arbitrary heuristic is that "A" should appear only initially in a "C...R" composition. DECARCDRATE also knows that MacLISP ordinarily has defined CAR/CDR compositions up to four long.

```
;;; CONDICATE TURNS AN IF INTO A COND; IN SO DOING IT TRIES TO MAKE THE RESULT PRETTY.

(DEFINE CONDICATE
        (LAMBDA (PRED CON ALT)
                (IF (OR (ATOM ALT) (NOT (EQ (CAR ALT) 'COND)))
                    "(COND (,PRED @(DEPROGNIFY CON))
                           (T @(DEPROGNIFY ALT)))
                    "(COND (,PRED @(DEPROGNIFY CON))
                           @(CDR ALT)))))


;;; DECARCDRATE MAKES CAR-CDR CHAINS PRETTIER.

(DEFINE DECARCDRATE
        (LAMBDA (X)
                (COND ((ATOM X) X)
                      ((EQ (CAR X) 'CAR)
                       (IF (ATOM (CADR X))
                           X
                           (LET ((Y (DECARCDRATE (CADR X))))
                                (COND ((EQ (CAR Y) 'CAR) "(CAAR ,(CADR Y)))
                                      ((EQ (CAR Y) 'CDR) "(CADR ,(CADR Y)))
                                      ((EQ (CAR Y) 'CDDR) "(CADDR ,(CADR Y)))
                                      ((EQ (CAR Y) 'CDDDR) "(CADDDR ,(CADR Y)))
                                      (T "(CAR ,Y))))))
                      ((EQ (CAR X) 'CDR)
                       (IF (ATOM (CADR X))
                           X
                           (LET ((Y (DECARCDRATE (CADR X))))
                                (COND ((EQ (CAR Y) 'CDR) "(CDDR ,(CADR Y)))
                                      ((EQ (CAR Y) 'CDDR) "(CDDDR ,(CADR Y)))
                                      ((EQ (CAR Y) 'CDDDR) "(CDDDDR ,(CADR Y)))
                                      (T "(CDR ,Y))))))
                      (T X))))
```

TRIVIALIZE is the version of ANALYZE which handles trivial forms. Recall that these are represented as pass-1 node-trees rather than as pass-2 cnode-trees. The task of TRIVIALIZE is to take such a node-tree and generate value-producing code. Recall that the subforms of a trivial form must themselves be trivial.

For a CONSTANT, a quoted copy of the value of the constant is generated.

For a VARIABLE, a reference to the variable is generated using LOOKUPICATE.

For an IF, the components are recursively given to TRIVIALIZE and then CONDICATE is used to generate a MacLISP COND form.

For an ASET, a reference to the ASET variable is generated using LOOKUPICATE, and code for the body is generated by calling TRIVIALIZE recursively; then OUTPUT-ASET generates the code for the ASET.

For a COMBINATION, the function must be either a TRIVFN or a LAMBDA-expression. For the former, a simple MacLISP function call is generated, after generating code for all the arguments. For the latter, TRIV-LAMBDACATE is invoked after generating code for the arguments and the LAMBDA body.

TRIV-LAMBDACATE is, so to speak, a trivial version of LAMBDACATE. The arguments are divided into two classes, those which are referenced and those which are not (the possibility of a referenced argument which is a KNOWN-FUNCTION cannot arise). When this is done, a MacLISP ((LAMBDA ...) ...) form is generated, preceded by any unreferenced arguments (which presumably have side-effects). For example:

```
    (TRIV-LAMBDACATE '(V1 V2 V3)
                     '((CAR X) (PRINT Y) (CDR Z))
                     '(PROGN (PRINT V1) (LIST V1 V3)))
```

ought to produce:

```
    (PROGN (PRINT Y)
           ((LAMBDA (V1 V3)
                    (COMMENT (VARS = (A C)))
                    (PRINT V1)
                    (LIST V1 V3))
            (CAR X)
            (CDR Z)))
```

Note that a MacLISP LAMBDA body is an implicit PROGN. TRIV-LAMBDACATE also takes advantage of the ability to arbitrarily reorder the execution of arguments to a combination.

```
(DEFINE TRIVIALIZE
        (LAMBDA (NODE RNL)
                (LET ((FM (NODE\FORM NODE)))
                     (EQCASE (TYPE FM)
                             (CONSTANT "',(CONSTANT\VALUE FM))
                             (VARIABLE (LOOKUPICATE (VARIABLE\VAR FM) RNL))
                             (IF (CONDICATE (TRIVIALIZE (IF\PRED FM) RNL)
                                            (TRIVIALIZE (IF\CON FM) RNL)
                                            (TRIVIALIZE (IF\ALT FM) RNL)))
                             (ASET
                              (OUTPUT-ASET (LOOKUPICATE (ASET\VAR FM) RNL)
                                           (TRIVIALIZE (ASET\BODY FM) RNL)))
                             (COMBINATION
                              (LET ((ARGS (COMBINATION\ARGS FM)))
                                   (LET ((FN (NODE\FORM (CAR ARGS))))
                                        (IF (AND (EQ (TYPE FN) 'VARIABLE)
                                                 (VARIABLE\GLOBALP FN)
                                                 (TRIVFN (VARIABLE\VAR FN)))
                                            (CONS (VARIABLE\VAR FN)
                                                  (AMAPCAR (LAMBDA (A) (TRIVIALIZE A RNL))
                                                           (CDR ARGS)))
                                            (IF (EQ (TYPE FN) 'LAMBDA)
                                                (TRIV-LAMBDACATE
                                                 (LAMBDA\VARS FN)
                                                 (AMAPCAR (LAMBDA (A) (TRIVIALIZE A RNL))
                                                          (CDR ARGS))
                                                 (TRIVIALIZE (LAMBDA\BODY FN) RNL))
                                                (ERROR '|Strange Trivial Function - TRIVIALIZE|
                                                       NODE
                                                       'FAIL-ACT)))))))))))

(DEFINE TRIV-LAMBDACATE
        (LAMBDA (VARS ARGS BODY)
                (LABELS ((LOOP
                          (LAMBDA (V A REALVARS REALARGS EFFARGS)
                                  (IF (NULL A)
                                     (LET ((RV (NREVERSE REALVARS)))
                                          (OR (NULL V)
                                              (ERROR '|We blew it in TRIV-LAMBDACATE| V 'FAIL-ACT))
                                          (LET ((B (IF RV
                                                       "((LAMBDA ,RV
                                                                 (COMMENT
                                                                  (VARS = ,(MAP-USER-NAMES RV)))
                                                                 @(DEPROGNIFY BODY))
                                                         @(NREVERSE REALARGS))
                                                       BODY)))
                                               (IF EFFARGS
                                                   "(PROGN @EFFARGS @(DEPROGNIFY B))
                                                   B)))
                                     (IF (OR (GET (CAR V) 'READ-REFS)
                                             (GET (CAR V) 'WRITE-REFS))
                                         (LOOP (CDR V)
                                               (CDR A)
                                               (CONS (CAR V) REALVARS)
                                               (CONS (CAR A) REALARGS)
                                               EFFARGS)
                                         (LOOP (CDR V)
                                               (CDR A)
                                               REALVARS
                                               REALARGS
                                               (CONS (CAR A) EFFARGS)))))))
                        (LOOP VARS ARGS NIL NIL NIL))))
```

We have examined the entire code generator, and now turn to high-level control routines. COMPILATE-ONE-FUNCTION is the highest-level entry to the code generator, called by COMPILE. It takes a code-tree and the user-name for the function, and returns a complete piece of MacLISP code constituting a module for the user function. It generates a global name for use as the module name (PROGNAME), and invokes COMPILATE-LOOP (which really ought to have been a LABELS function, but was too big to fit on the paper that way). The last argument is a list of two MacLISP forms; one causes a SCHEME compiled closure form (a CBETA list) to be put in the value cell of the user-name, so that it will be a globally defined SCHEME function, and the other creates a property linking the PROGNAME with the USERNAME for debugging purposes.

COMPILATE-LOOP repeatedly calls COMPILATE, giving it the next function on the FNS list. Of course, the invocation of COMPILATE may cause new entries to appear on the FNS list. COMPILATE-LOOP iterates until the FNS list converges to emptiness. As it does so it takes each piece of generated code and strings it together as PROGBODY. It also calculates in TMAX the maximum over all MAXDEP slots; this is the only place where the MAXDEP slot is ever used.

When FNS is exhausted, a module is constructed. This contains a comment, a MacLISP DEFUN form for defining a MacLISP function, a SETQ form to put the SUBR pointer in the value cell of the PROGNAME (for the benefit of code which creates CBETA forms), and extra "stuff". TMAX is used to generate a list of all temporary variables used in the module; a MacLISP SPECIAL declaration is created to advise the MacLISP compiler.

USED-TEMPLOCS takes a TMAX value and generates the names of all temporary registers (whose names are of the form -nn-; standard argument registers are not included) up to that number.

```
(DEFINE COMPILATE-ONE-FUNCTION                                  ;COMPLICATE-ONE-FUNCTION?
        (LAMBDA (CNODE USERNAME)
                (LET ((PROGNAME (GEN-GLOBAL-NAME)))
                     (COMPILATE-LOOP USERNAME
                                     PROGNAME
                                     (LIST (LIST USERNAME CNODE))
                                     (LIST (LIST PROGNAME CNODE NIL))
                                     NIL
                                     0
                                     (LIST "(SETQ ,USERNAME
                                                  (LIST 'CBETA
                                                        ,PROGNAME
                                                        ',(CLAMBDA\NAME (CNODE\CFORM CNODE))))
                                           "(DEFPROP ,PROGNAME ,USERNAME USER-FUNCTION))))))

(DEFINE COMPILATE-LOOP
        (LAMBDA (USERNAME PROGNAME BLOCKFNS FNS PROGBODY TMAX STUFF)
                (IF (NULL FNS)
                    "(PROGN 'COMPILE
                            (COMMENT MODULE FOR FUNCTION ,USERNAME)
                            (DEFUN ,PROGNAME ()
                                   (PROG ()
                                         (DECLARE (SPECIAL ,PROGNAME @(USED-TEMPLOCS TMAX)))
                                         (GO (PROG2 NIL
                                                    (CAR **ENV**)
                                                    (SETQ **ENV** (CDR **ENV**))))
                                         @(NREVERSE PROGBODY)))
                            (SETQ ,PROGNAME (GET ',PROGNAME 'SUBR))
                            @STUFF)
                    (COMPILATE (CAR (CAR FNS))
                               (CADR (CAR FNS))
                               (CADDR (CAR FNS))
                               BLOCKFNS
                               (CDR FNS)
                               (LAMBDA (CODE NEWFNS)
                                       (LET ((CFM (CNODE\CFORM (CADR (CAR FNS)))))
                                            (COMPILATE-LOOP
                                             USERNAME
                                             PROGNAME
                                             BLOCKFNS
                                             NEWFNS
                                             (NCONC (REVERSE (DEPROGNIFY1 CODE T))
                                                    (CONS (REMARK-ON (CADR (CAR FNS)))
                                                          (CONS (EQCASE (TYPE CFM)
                                                                        (CLAMBDA
                                                                         (CLAMBDA\NAME CFM))
                                                                        (CONTINUATION
                                                                         (CONTINUATION\NAME CFM)))
                                                                PROGBODY)))
                                             (MAX TMAX
                                                  (EQCASE (TYPE CFM)
                                                          (CLAMBDA
                                                           (CLAMBDA\MAXDEP CFM))
                                                          (CONTINUATION
                                                           (CONTINUATION\MAXDEP CFM))))
                                             STUFF)))))))

(DEFINE USED-TEMPLOCS
        (LAMBDA (N)
                (DO ((J (+ **NUMBER-OF-ARG-REGS** 1) (+ J 1))
                     (X NIL (CONS (TEMPLOC J) X)))
                    ((> J N) (NREVERSE X)))))
```

REMARK-ON takes a cnode for a CLAMBDA or CONTINUATION and generates a comment containing pertinent information about invoking that function. This comment will presumably be inserted in the output code to guide the debugging process.

MAP-USER-NAMES takes a list of internal variable names and returns a list of the corresponding user names for the variables, as determined by the USER-NAME property. (If a variable is an internally generated one, e.g. for a continuation, then it will have no USER-NAME property, and the internal name itself is used.)

```
(DEFINE REMARK-ON
        (LAMBDA (CNODE)
                (LET ((CFM (CNODE\CFORM CNODE)))
                     (LABELS ((REMARK1
                               (LAMBDA (DEP FNP VARS ENV)
                                       "(COMMENT (DEPTH = ,DEP)
                                                 (FNP = ,FNP)
                                                 @(IF VARS "((VARS = ,(MAP-USER-NAMES VARS))))
                                                 @(IF ENV "((ENV = ,(MAP-USER-NAMES ENV))))))))
                             (EQCASE (TYPE CFM)
                                     (CLAMBDA
                                      (REMARK1 (CLAMBDA\DEP CFM)
                                               (CLAMBDA\FNP CFM)
                                               (IF (EQ (CLAMBDA\FNP CFM) 'NOCLOSE)
                                                   (CLAMBDA\TVARS CFM)
                                                   (CLAMBDA\VARS CFM))
                                               (APPEND (CLAMBDA\CLOSEREFS CFM)
                                                       (CLAMBDA\CONSENV CFM))))
                                     (CONTINUATION
                                      (REMARK1 (CONTINUATION\DEP CFM)
                                               (CONTINUATION\FNP CFM)
                                               NIL      ;NEVER INTERESTING ANYWAY
                                               (APPEND (CONTINUATION\CLOSEREFS CFM)
                                                       (CONTINUATION\CONSENV CFM)))))))))


(DEFINE MAP-USER-NAMES
        (LAMBDA (VARS)
                (AMAPCAR (LAMBDA (X) (OR (GET X 'USER-NAME) X)) VARS)))
```

The next few pages contain routines which deal with files. COMFILE takes a file name, and compiles all the code in that file, producing an output file of MacLISP code suitable for giving to the MacLISP compiler. It also computes the CPU time required to compile the file.

FN gets the given file name, processed and defaulted according to ITS/MacLISP standard conventions. RT and GCT save runtime and gc-runtime information.

IFILE and OFILE get MacLISP "file objects" created by the OPEN function, which opens the file specified by its first argument. (The output file names are initially " RABB OUTPUT", conforming to an ITS standard. These will later be renamed.)

GLOBAL-GEN-PREFIX is initialized as a function of the file name, to "directory=firstname". This is to guarantee that the global symbols generated for two different compiled files of SCHEME code will not conflict should the two files be loaded into the same SCHEME together. (Notice the use of SYMEVAL. This is necessary because MacLISP allows names to be used in two different kinds of contexts, one meaning the "functional" value, and the other meaning the "variable" value. SCHEME does not make this distinction, and tries to make the functional value available, but does not do this consistently. This is a problem which results from a fundamental difference in semantics between SCHEME and MacLISP. For such variables as DEFAULTF and TYO, which in MacLISP are used for both purposes, it is necessary to use SYMEVAL to specify that the variable, rather than the function, is desired.)

(SYMEVAL 'TYO) refers to the file object for the terminal; this is used to print out messages to the user while the file is being compiled. Various information is also printed to the file, including identification and a timestamp. The DECLARE form printed to the output file contains the names of the standard argument registers, and also **ENV**, **FUN**, and **NARGS**. (This is why USED-TEMPLOCS need not generate names of standard argument registers -- this single global declaration covers them.) The second DECLARE form defines to the MacLISP compiler a function called DISPLACE for obscure reasons having to do with the implementation of SCHEME macros.

TRANSDUCE does the primary work of processing the input file. When it is done, another timestamp is printed to the output file, so that the real time consumed can be determined; then the runtime statistics are calculated and printed, along with the number of errors if any occurred. The output file is then renamed as "firstname LISP" and closed. The statistics message is returned so that it will be printed as the last message on the terminal.

```
(DEFINE COMFILE
        (LAMBDA (FNAME)
                (LET ((FN (DEFAULTF (MERGEF FNAME '(* )))))
                      (RT (RUNTIME))
                      (GCT (STATUS GCTIME)))
                     (LET ((IFILE (OPEN FN 'IN))
                           (OFILE (OPEN (MERGEF '(_RABB_ OUTPUT) FN) 'OUT)))
                          (SET' *GLOBAL-GEN-PREFIX*
                                (CATENATE (CADAR (SYMEVAL 'DEFAULTF))
                                         '|=|
                                         (CADR (SYMEVAL 'DEFAULTF))))
                          (LET ((TN (NAMESTRING (TRUENAME IFILE))))
                               (PRINT "(COMMENT THIS IS THE RABBIT LISP CODE FOR ,TN) OFILE)
                               (TIMESTAMP OFILE)
                               (TERPRI OFILE)
                               (TERPRI (SYMEVAL 'TYO))
                               (PRINC '|;Beginning RABBIT compilation on | (SYMEVAL 'TYO))
                               (PRINC TN (SYMEVAL 'TYO)))
                          (PRINT "(DECLARE (SPECIAL @**CONT+ARG-REGS** **ENV** **FUN** **NARGS**))
                                 OFILE)
                          (PRINT '(DECLARE (DEFUN DISPLACE (X Y) Y)) OFILE)
                          (ASET' *TESTING* NIL)
                          (ASET' *ERROR-COUNT* 0)
                          (ASET' *ERROR-LIST* NIL)
                          (TRANSDUCE IFILE
                                     OFILE
                                     (LIST NIL)
                                     (CATENATE '|INIT-| (CADR (TRUENAME IFILE))))
                          (TIMESTAMP OFILE)
                          (LET ((X (*QUO (- (RUNTIME) RT) 1.0E6))
                                (Y (*QUO (- (STATUS GCTIME) GCT) 1.0E6)))
                               (LET ((MSG "(COMPILE TIME: ,X SECONDS
                                                   (GC TIME ,Y SECONDS)
                                                   (NET ,(-$ X Y) SECONDS)
                                                   @(IF (NOT (ZEROP *ERROR-COUNT*))
                                                        "((,*ERROR-COUNT* ERRORS))))))
                                    (PRINT "(COMMENT ,MSG) OFILE)
                                    (RENAMEF OFILE
                                             (MERGEF (LIST (CADR FN) 'LISP)
                                                     FN))
                                    (CLOSE OFILE)
                                    MSG))))))
```

TRANSDUCE processes forms from IFILE, one by one, calling PROCESS-FORM to do the real work on each one. PROCESS-FORM may print results on OFILE, and may also return a list of "random forms" to be saved.

The business of "random forms" has to do with the fact that the file being compiled may contains forms which are not function definitions. The expectation is that when the file is loaded these forms will be evaluated during the loading process, and this is indeed true if the interpreter loads the original file of source forms.

Now MacLISP provides a facility for evaluating random forms within a compiled file, but they are evaluated as MacLISP forms, not SCHEME forms. To get around this whole problem, I chose another solution. All the random forms in the file are accumulated, and then compiled as the body of a function named "INIT-firstname". In this way, once the compiled code is loaded, the user is expected to say (INIT-firstname) to cause the random forms to be evaluated.

This idea would have worked perfectly except that files typically have a large number of random forms in them (macro definitions create one or two random forms as well as the definition of the macro-function). Putting all the random forms together in a single function often creates a function too big for RABBIT to compile, given PDP-10 memory limitiations. The four lines of code in TRANSDUCE for this have therefore been commented out with a ";" at the beginning of each line.

The final solution was to compile each random form as its own function, and arrange for all these little functions to be chained; each one executes one random form and then calls the next. A call to INIT-firstname starts the chain going.

This, then, is the purpose of the big DO loop in TRANSDUCE: to construct all the little functions for the random forms. The third argument to PROCESS-FORM may be NIL, which suppresses the printing of any messages on the terminal; this spares the user having to see tens or hundreds of uninteresting messages concerning the compilation of these initialization functions. However, so that the user will not be dismayed at the long pause, a message saying how many random forms there were is printed first.

READIFY implements the MacLISP convention that if the value of the variable READ is non-nil, then that value is the read-in function to use; while if it is NIL, then the function READ is the read-in function. (This "hook" is the method by which CGOL works, for example.)

```
(DEFINE TRANSDUCE
        (LAMBDA (IFILE OFILE EOF INITNAME)
                (LABELS ((LOOP
                          (LAMBDA (FORM RANDOM-FORMS)
                                  (IF (EQ FORM EOF)
                                      (DO ((X (GENTEMP INITNAME) (GENTEMP INITNAME))
                                           (Y NIL X)
                                           (Z RANDOM-FORMS (CDR Z)))
                                          ((NULL Z)
                                           (IF RANDOM-FORMS
                                               (PRINT "(,(LENGTH RANDOM-FORMS)
                                                       RANDOM FORMS IN FILE TO COMPILE)
                                                     (SYMEVAL 'TYO)))
                                           (IF Y (PROCESS-FORM "(DECLARE (SPECIAL ,Y))
                                                              OFILE
                                                              T))
                                           (PROCESS-FORM "(DEFINE ,INITNAME
                                                                  (LAMBDA () ,(IF Y (LIST Y) NIL)))
                                                         OFILE
                                                         T))
                                          (IF Y (PROCESS-FORM "(DECLARE (SPECIAL ,Y))
                                                             OFILE
                                                             NIL))
                                          (PROCESS-FORM "(DEFINE ,X
                                                                 (LAMBDA ()
                                                                         (BLOCK ,(CAR Z)
                                                                                ,(IF Y
                                                                                     (LIST Y)
                                                                                     NIL))))
                                                        OFILE
                                                        NIL))
;                                     (PROCESS-FORM
;                                         "(DEFINE ,INITNAME
;                                                 (LAMBDA () (BLOCK @RANDOM-FORMS NIL NIL)))
;                                         OFILE)
                                      (LET ((X (PROCESS-FORM FORM OFILE T)))
                                           (LOOP (READIFY IFILE EOF) (NCONC X RANDOM-FORMS)))))))
                        (LOOP (READIFY IFILE EOF) NIL))))


(DEFINE READIFY                  ;FUNNY MACLISP CONVENTION - READIFY'LL DO THE JOB!
        (LAMBDA (IFILE EOF)
                (IF (SYMEVAL 'READ)
                    (APPLY (SYMEVAL 'READ) IFILE EOF)
                    (READ IFILE EOF))))
```

PROCESS-FORM takes a form, an output file, and a switch saying whether to be "noisy". The form is broken down into one of many special cases and processed accordingly. The returned value is a list of "random forms" for TRANSDUCE to save for later handling.

An atom, while pretty useless, is transduced directly to the output file.

A DEFINE-form, which defined a function to be compiled, is given to PROCESS-DEFINE-FORM. This is the interesting case, which we will discuss on the next page.

A special hack handed down from MacLISP is that a form (PROGN 'COMPILE ...) (and, for SCHEME, the analogous (BLOCK 'COMPILE ...)) should be treated as if all the subforms of the PROGN (or BLOCK) after the first should be processed as if they had been read as "top-level" forms from the file. This allows a macro call to generate more than one form to be compiled, for example. It is necessary to accumulate all the results of the calls to PROCESS-FORM so that they may be collectively returned.

A PROCLAIM form contains a set of forms to be evaluated by RABBIT at compile time. The evaluation is accomplished by constructing a LAMBDA form and using the SCHEME primitive ENCLOSE to create a closure, and then invoking the closure. As a special service, the variable OFILE is made apparent to the evaluated form so that it can print information to the output file if desired.

A DECLARE form is meant to be seen by the MacLISP compiler, and so it is passed on directly.

A COMMENT form is simply eliminated. (It could be passed through directly with no harm.)

A DEFUN form is passed directly, for compilation by the MacLISP compiler.

A form which is a macro call is expanded and re-processed. As a special hack, those which are calls to DEFMAC, SCHMAC, or MACRO are also evaluated (MacLISP evaluation serves), so that the defined macro will be available for compiling calls to it later in the file.

Any other form is considered "random", and is returned to TRANSDUCE _provided_ *BUFFER-RANDOM-FORMS* is non-NIL. This switch is provided in case it is necessary to force a random form (e.g. an ALLOC form) to be output early in the file. In this case any random forms must be MacLISP-evaluable as well as SCHEME-evaluable. (This requirement is the reason RABBIT has random forms like "(SET' FOO ...)"; SETQ is unacceptable to SCHEME, while ASET' is unacceptable to MacLISP, but SET' happens to work in both languages for setting a global variable.) RABBIT itself sets *BUFFER-RANDOM-FORMS* to NIL on page 1 in a PROCLAIM form.

```
(SET' *OPTIMIZE* T)

(SET' *BUFFER-RANDOM-FORMS* T)

(DEFINE PROCESS-FORM
        (LAMBDA (FORM OFILE NOISYP)
                (COND ((ATOM FORM)
                       (PRINT FORM OFILE)
                       NIL)
                      ((EQ (CAR FORM) 'DEFINE)
                       (PROCESS-DEFINE-FORM FORM OFILE NOISYP)
                       NIL)
                      ((AND (MEMQ (CAR FORM) '(BLOCK PROGN))
                            (EQUAL (CADR FORM) ''COMPILE))
                       (DO ((F (CDDR FORM) (CDR F))
                            (Z NIL (NCONC Z (PROCESS-FORM (CAR F) OFILE NOISYP))))
                           ((NULL F) Z)))
                      ((EQ (CAR FORM) 'PROCLAIM)
                       (AMAPC (LAMBDA (X) ((ENCLOSE "(LAMBDA (OFILE) ,X)) OFILE))
                              (CDR FORM))
                       NIL)
                      ((EQ (CAR FORM) 'DECLARE)
                       (PRINT FORM OFILE)
                       NIL)
                      ((EQ (CAR FORM) 'COMMENT)
                       NIL)
                      ((EQ (CAR FORM) 'DEFUN)
                       (PRINT FORM OFILE)
                       NIL)
                      ((AND (ATOM (CAR FORM))
                            (EQ (GET (CAR FORM) 'AINT) 'AMACRO)
                            (NOT (EQ (GET (CAR FORM) 'AMACRO) 'AFSUBR)))
                       (IF (MEMQ (CAR FORM) '(DEFMAC SCHMAC MACRO))
                           (EVAL FORM))
                       (PROCESS-FORM (MACRO-EXPAND FORM) OFILE NOISYP))
                      (T (COND (*BUFFER-RANDOM-FORMS* (LIST FORM))
                               (T (PRINT FORM OFILE) NIL))))))
```

PROCESS-DEFINE-FORM disambiguates the three DEFINE formats permitted in SCHEME:

```
(DEFINE FOO (LAMBDA (X Y) ...))
(DEFINE FOO (X Y) ...)
(DEFINE (FOO X Y) ...)
```

and constructs an appropriate LAMBDA-expression in standard form.

PROCESS-DEFINITION takes this LAMBDA-expression and compiles it, after some error checks. It then cleans up, and if desired prints a message on the console to the effect that the function was compiled successfully.

CLEANUP is used to clear out garbage left around by the compilation process which is no longer needed (but is useful during the compilation, whether for compilation proper or only for debugging should the compilation process fail).

REPLACE has to do with macros which displace calls to them with the expanded forms, but retain enough information to undo this. REPLACE undoes this and throws away the saved information. (The DISPLACE facility is normally turned off anyway, and is used only to make the compiler run faster when it itself is being run under the SCHEME interpreter. This was of great use when RABBIT wasn't running well enough to compile itself!)

GENFLUSH removes from the MacLISP OBARRAY all the temporary generated symbols created by GENTEMP.

The MAPATOMS form removes from every atom on the OBARRAY the properties shown. This takes more time but less space than recording exactly which atoms had such properties created for them.

```
(DEFINE PROCESS-DEFINE-FORM
        (LAMBDA (FORM OFILE NOISYP)
                (COND ((ATOM (CADR FORM))
                       (PROCESS-DEFINITION FORM
                                           OFILE
                                           NOISYP
                                           (CADR FORM)
                                           (IF (NULL (CDDDR FORM))
                                               (CADDR FORM)
                                               "(LAMBDA ,(CADDR FORM)
                                                        (BLOCK . ,(CDDDR FORM))))))
                      (T (PROCESS-DEFINITION FORM
                                             OFILE
                                             NOISYP
                                             (CAADR FORM)
                                             "(LAMBDA ,(CDADR FORM)
                                                      (BLOCK . ,(CDDR FORM))))))))

(DEFINE PROCESS-DEFINITION
        (LAMBDA (FORM OFILE NOISYP NAME LAMBDA-EXP)
                (COND ((NOT (EQ (TYPEP NAME) 'SYMBOL))
                       (WARN |Function Name Not SYMBOL| NAME FORM))
                      ((OR (NOT (EQ (CAR LAMBDA-EXP) 'LAMBDA))
                           (AND (ATOM (CADR LAMBDA-EXP))
                                (NOT (NULL (CADR LAMBDA-EXP)))))
                       (WARN |Malformed LAMBDA-expression| LAMBDA-EXP FORM))
                      (T (PRINT (COMPILE NAME
                                         LAMBDA-EXP
                                         NIL
                                         *OPTIMIZE*)
                                OFILE)
                         (CLEANUP)
                         (IF NOISYP
                             (PRINT (LIST NAME 'COMPILED)
                                    (SYMEVAL 'TYO)))))))

(DEFINE CLEANUP
        (LAMBDA ()
                (BLOCK (REPLACE)
                       (GENFLUSH)
                       (MAPATOMS '(LAMBDA (X)
                                          (REMPROP X 'READ-REFS)
                                          (REMPROP X 'WRITE-REFS)
                                          (REMPROP X 'NODE)
                                          (REMPROP X 'BINDING)
                                          (REMPROP X 'USER-NAME)
                                          (REMPROP X 'KNOWN-FUNCTION)
                                          (REMPROP X 'EASY-LABELS-FUNCTION))))))
```

SEXPRFY and CSEXPRFY are debugging aids which take a node-tree or cnode-tree and produce a fairly readable S-expression version of the code it represents. They are used by the SX and CSX macros defined earlier. The USERP switch for SEXPRFY specifies whether internal variables names or user variable names should be used in the construction.

```
;;; INVERSE OF ALPHATIZE.  USED BY SX, E.G., FOR DEBUGGING.

(DEFINE SEXPRFY
        (LAMBDA (NODE USERP)
                (LET ((FM (NODE\FORM NODE)))
                     (EQCASE (TYPE FM)
                             (CONSTANT "(QUOTE ,(CONSTANT\VALUE FM)))
                             (VARIABLE (IF (AND USERP (NOT (VARIABLE\GLOBALP FM)))
                                           (GET (VARIABLE\VAR FM) 'USER-NAME)
                                           (VARIABLE\VAR FM)))
                             (LAMBDA "(LAMBDA ,(IF USERP (LAMBDA\UVARS FM) (LAMBDA\VARS FM))
                                               ,(SEXPRFY (LAMBDA\BODY FM) USERP)))
                             (IF "(IF ,(SEXPRFY (IF\PRED FM) USERP)
                                      ,(SEXPRFY (IF\CON FM) USERP)
                                     ·,(SEXPRFY (IF\ALT FM) USERP)))
                             (ASET "(ASET' ,(IF (AND USERP (NOT (ASET\GLOBALP FM)))
                                                (GET (ASET\VAR FM) 'USER-NAME)
                                                (ASET\VAR FM))
                                           ,(SEXPRFY (ASET\BODY FM) USERP)))
                             (CATCH "(CATCH ,(IF USERP
                                                 (GET (CATCH\VAR FM) 'USER-NAME)
                                                 (CATCH\VAR FM))
                                            ,(SEXPRFY (CATCH\BODY FM) USERP)))
                             (LABELS "(LABELS ,(AMAPCAR (LAMBDA (V D) "(,(IF USERP
                                                                             (GET V 'USER-NAME)
                                                                             V)
                                                                        ,(SEXPRFY D USERP)))
                                                        (LABELS\FNVARS FM)
                                                        (LABELS\FNDEFS FM))
                                              ,(SEXPRFY (LABELS\BODY FM) USERP)))
                             (COMBINATION
                              (AMAPCAR (LAMBDA (A) (SEXPRFY A USERP))
                                       (COMBINATION\ARGS FM)))))))

(DEFINE CSEXPRFY
        (LAMBDA (CNODE)
                (LET ((CFM (CNODE\CFORM CNODE)))
                     (EQCASE (TYPE CFM)
                             (TRIVIAL "(TRIVIAL ,(SEXPRFY (TRIVIAL\NODE CFM) NIL)))
                             (CVARIABLE (CVARIABLE\VAR CFM))
                             (CLAMBDA "(CLAMBDA ,(CLAMBDA\VARS CFM)
                                                 ,(CSEXPRFY (CLAMBDA\BODY CFM))))
                             (CONTINUATION
                              "(CONTINUATION (,(CONTINUATION\VAR CFM))
                                              ,(CSEXPRFY (CONTINUATION\BODY CFM))))
                             (CIF "(CIF ,(CSEXPRFY (CIF\PRED CFM))
                                         ,(CSEXPRFY (CIF\CON CFM))
                                         ,(CSEXPRFY (CIF\ALT CFM))))
                             (CASET "(CASET' ,(CSEXPRFY (CASET\CONT CFM))
                                             ,(CASET\VAR CFM)
                                             ,(CSEXPRFY (CASET\BODY CFM))))
                             (CLABELS "(CLABELS ,(AMAPCAR (LAMBDA (V D) "(,V
                                                                          ,(CSEXPRFY D)))
                                                          (CLABELS\FNVARS CFM)
                                                          (CLABELS\FNDEFS CFM))
                                                 ,(CSEXPRFY (CLABELS\BODY CFM))))
                             (CCOMBINATION
                              (AMAPCAR CSEXPRFY (CCOMBINATION\ARGS CFM)))
                             (RETURN
                              "(RETURN ,(CSEXPRFY (RETURN\CONT CFM))
                                       ,(CSEXPRFY (RETURN\VAL CFM)))))))
```

CHECK-NUMBER-OF-ARGS is used by COMPILE and ALPHA-COMBINATION to make sure that function calls and definitions agree on the number of arguments taken by a function. If a mismatch is detacted, a warning is issued. This check frequently catches various typographical errors. The argument DEFP is NIL unless this call is on behalf of a definition rather than a call. The DEFINED property is used only so that a more comprehensive warning may be given.

*EXPR and *LEXPR are two special forms suitable for use in a PROCLAIM form for declaring that certain names refer to MacLISP functions rather than SCHEME functions. An example, for PRINT-SHORT, occurs on page 1 of RABBIT.

DUMPIT establishes a simple user interface for RABBIT. After loading a compiled RABBIT into a SCHEME run-time system, the call (DUMPIT) initializes the RABBIT, then suspends the MacLISP environment, with an argument which is an ITS command for dumping the core image. When this core image is later loaded and resumed, DUMPIT prints "FILE NAME:" and reads a line. All the user need do is typoe a file name and a carriage return to compile the file. When this is done, the call to QUIT kills the RABBIT job.

STATS prints out statistics accumulated in the counters listed in *STAT-VARS*. RESET-STATS resets all the counters.

```
(DEFINE CHECK-NUMBER-OF-ARGS
        (LAMBDA (NAME NARGS DEFP)
                (OR (GETL NAME '(*LEXPR LSUBR))
                    (LET ((N (GET NAME 'NUMBER-OF-ARGS)))
                         (IF N
                             (IF (NOT (= N NARGS))
                                 (IF DEFP
                                     (WARN |definition disagrees with earlier use on number of args|
                                           NAME
                                           NARGS
                                           N)
                                     (IF (GET NAME 'DEFINED)
                                         (WARN |use disagrees with definition on number of args|
                                               NAME
                                               NARGS
                                               N)
                                         (WARN |two uses disagree before definition on number of args|
                                               NAME
                                               NARGS
                                               N))))
                             (PUTPROP NAME NARGS 'NUMBER-OF-ARGS))
                         (IF DEFP (PUTPROP NAME 'T 'DEFINED))))))


(DEFUN *EXPR FEXPR (X)
       (MAPCAR '(LAMBDA (Y) (PUTPROP Y 'T '*EXPR)) X))

(DEFPROP *EXPR AFSUBR AMACRO) (DEFPROP *EXPR AMACRO AINT)

(DEFUN *LEXPR FEXPR (X)
       (MAPCAR '(LAMBDA (Y) (PUTPROP Y 'T '*LEXPR)) X))

(DEFPROP *LEXPR AFSUBR AMACRO) (DEFPROP *LEXPR AMACRO AINT)


(DEFINE DUMPIT
        (LAMBDA ()
                (BLOCK (INIT-RABBIT)
                       (SUSPEND '|:PDUMP DSK:SCHEME;TS RABBIT|)
                       (TERPRI)
                       (PRINC '|File name: |)
                       (COMFILE (READLINE))
                       (QUIT))))

(DEFINE STATS
        (LAMBDA ()
                (AMAPC (LAMBDA (VAR)
                               (BLOCK (TERPRI)
                                      (PRIN1 VAR)
                                      (PRINC '| = |)
                                      (PRIN1 (SYMEVAL VAR))))
                       *STAT-VARS*)))

(DEFINE RESET-STATS
        (LAMBDA () (AMAPC (LAMBDA (VAR) (SET VAR 0)) *STAT-VARS*)))
```

*	SIDE EFFECTS	017	005
*EXPR	FEXPR	071	026
*EXPR	PROPERTY	071	027
*EXPR	AMACRO	071	029
*EXPR	AINT	071	029
*LEXPR	FEXPR	071	031
*LEXPR	PROPERTY	071	032
*LEXPR	AMACRO	071	034
*LEXPR	AINT	071	034
+	SIDE EFFECTS	017	003
-	SIDE EFFECTS	017	004
/	SIDE EFFECTS	017	006
<	SIDE EFFECTS	017	008
=	SIDE EFFECTS	017	007
>	SIDE EFFECTS	017	009
ACCESSFN	@DEFINE	004	002
ACCESSFN	MACLISP MACRO	004	004
ADDPROP	SCHEME FUNCTION	006	004
ADJOIN	SCHEME FUNCTION	006	029
ADJUST-KNOWNFN-CENV	SCHEME FUNCTION	052	002
AINT	PROPERTY	071	029
AINT	PROPERTY	071	034
ALPHA-ASET	SCHEME FUNCTION	010	010
ALPHA-ATOM	SCHEME FUNCTION	009	032
ALPHA-BLOCK	SCHEME FUNCTION	011	011
ALPHA-CATCH	SCHEME FUNCTION	010	029
ALPHA-COMBINATION	SCHEME FUNCTION	011	037
ALPHA-IF	SCHEME FUNCTION	010	002
ALPHA-LABELS	SCHEME FUNCTION	010	040
ALPHA-LABELS-DEFN	SCHEME FUNCTION	011	002
ALPHA-LAMBDA	SCHEME FUNCTION	009	042
ALPHATIZE	SCHEME FUNCTION	009	005
AMACRO	PROPERTY	071	029
AMACRO	PROPERTY	071	034
ANALYZE	SCHEME FUNCTION	058	002
ANALYZE-CCOMBINATION	SCHEME FUNCTION	060	003
ANALYZE-CIF	SCHEME FUNCTION	059	002
ANALYZE-CLABELS	SCHEME FUNCTION	059	025
ANALYZE-CLAMBDA	SCHEME FUNCTION	058	023
ANALYZE-CONTINUATION	SCHEME FUNCTION	058	040
ANALYZE-RETURN	SCHEME FUNCTION	060	033
APPEND	SIDE EFFECTS	017	057
ASET	DATA TYPE	008	042
ASK	PDP-10 SCHEME MACRO	003	024
ASSQ	SIDE EFFECTS	017	059
ATOM	SIDE EFFECTS	017	045
BIGP	SIDE EFFECTS	017	052
BIND-ANALYZE	SCHEME FUNCTION	034	030
BIND-ANALYZE-CASET	SCHEME FUNCTION	035	040
BIND-ANALYZE-CCOMBINATION	SCHEME FUNCTION	037	002
BIND-ANALYZE-CIF	SCHEME FUNCTION	035	030
BIND-ANALYZE-CLABELS	SCHEME FUNCTION	036	002
BIND-ANALYZE-CLAMBDA	SCHEME FUNCTION	035	002
BIND-ANALYZE-CONTINUATION	SCHEME FUNCTION	035	016
BIND-ANALYZE-RETURN	SCHEME FUNCTION	036	033
BIND-CCOMBINATION-ANALYZE	SCHEME FUNCTION	037	043
CAAAAR	SIDE EFFECTS	017	024
CAAADR	SIDE EFFECTS	017	025
CAAAR	SIDE EFFECTS	017	016
CAADAR	SIDE EFFECTS	017	026
CAADDR	SIDE EFFECTS	017	027
CAADR	SIDE EFFECTS	017	017
CAAR	SIDE EFFECTS	017	012
CADAAR	SIDE EFFECTS	017	028
CADADR	SIDE EFFECTS	017	029
CADAR	SIDE EFFECTS	017	018

CADDAR	SIDE EFFECTS	017	030
CADDDR	SIDE EFFECTS	017	031
CADDR	SIDE EFFECTS	017	019
CADR	SIDE EFFECTS	017	013
CAR	SIDE EFFECTS	017	010
CASET	DATA TYPE	026	039
CATCH	DATA TYPE	008	046
CATENATE	MACLISP MACRO	002	016
CCOMBINATION	DATA TYPE	026	047
CDAAAR	SIDE EFFECTS	017	032
CDAADR	SIDE EFFECTS	017	033
CDAAR	SIDE EFFECTS	017	020
CDADAR	SIDE EFFECTS	017	034
CDADDR	SIDE EFFECTS	017	035
CDADR	SIDE EFFECTS	017	021
CDAR	SIDE EFFECTS	017	014
CDDAAR	SIDE EFFECTS	017	036
CDDADR	SIDE EFFECTS	017	037
CDDAR	SIDE EFFECTS	017	022
CDDDAR	SIDE EFFECTS	017	038
CDDDDR	SIDE EFFECTS	017	039
CDDDR	SIDE EFFECTS	017	023
CDDR	SIDE EFFECTS	017	015
CDR	SIDE EFFECTS	017	011
CENV-ANALYZE	SCHEME FUNCTION	032	018
CENV-CCOMBINATION-ANALYZE	SCHEME FUNCTION	033	028
CENV-TRIV-ANALYZE	SCHEME FUNCTION	033	004
CHECK-COMBINATION-PEFFS	SCHEME FUNCTION	016	002
CHECK-NUMBER-OF-ARGS	SCHEME FUNCTION	071	002
CIF	DATA TYPE	026	038
CL	PDP-10 SCHEME MACRO	007	049
CLABELS	DATA TYPE	026	040
CLAMBDA	DATA TYPE	026	017
CLEANUP	SCHEME FUNCTION	069	038
CLOBBER	MACLISP MACRO	004	025
CLOSE-ANALYZE	SCHEME FUNCTION	040	002
CNAME	MACLISP MACRO	004	015
CNODE	DATA TYPE	026	007
CNODIFY	SCHEME FUNCTION	027	002
COMBINATION	DATA TYPE	008	055
COMFILE	SCHEME FUNCTION	066	002
COMP-BODY	SCHEME FUNCTION	045	006
COMPILATE	SCHEME FUNCTION	041	018
COMPILATE-LOOP	SCHEME FUNCTION	064	017
COMPILATE-ONE-FUNCTION	SCHEME FUNCTION	064	002
COMPILE	SCHEME FUNCTION	007	010
COMPONENT-NAMES	PROPERTY	005	060
CONDICATE	SCHEME FUNCTION	062	004
CONS	SIDE EFFECTS	017	055
CONS-CLOSEREFS	SCHEME FUNCTION	061	011
CONSTANT	DATA TYPE	008	028
CONTINUATION	DATA TYPE	026	036
CONVERT	SCHEME FUNCTION	027	006
CONVERT-ASET	SCHEME FUNCTION	029	002
CONVERT-CATCH	SCHEME FUNCTION	029	024
CONVERT-COMBINATION	SCHEME FUNCTION	031	014
CONVERT-IF	SCHEME FUNCTION	028	024
CONVERT-LABELS	SCHEME FUNCTION	030	006
CONVERT-LAMBDA-FM	SCHEME FUNCTION	028	009
COPY-CODE	SCHEME FUNCTION	025	002
COPY-NODES	SCHEME FUNCTION	025	007
CSEXPRFY	SCHEME FUNCTION	070	036
CSX	MACLISP MACRO	003	029
CVARIABLE	DATA TYPE	026	015
CXR	SIDE EFFECTS	017	040
DECARCDRATE	SCHEME FUNCTION	062	015
DEFINE	@DEFINE	001	062

```
DEFMAC .................................. @DEFINE ................. 001 063
DEFTYPE ................................. @DEFINE ................. 005 002
DEFTYPE ................................. MACLISP MACRO ........... 005 008
DELPROP ................................. SCHEME FUNCTION ......... 006 012
DEPROGNIFY .............................. MACLISP MACRO ........... 042 005
DEPROGNIFY1 ............................. SCHEME FUNCTION ......... 042 009
DEPTH-ANALYZE ........................... SCHEME FUNCTION ......... 038 010
DISPLACE ................................ EXPR .................... 001 006
DUMPIT .................................. SCHEME FUNCTION ......... 071 037
EFFDEF .................................. @DEFINE ................. 016 039
EFFDEF .................................. MACLISP MACRO ........... 016 034
EFFECTLESS .............................. SCHEME FUNCTION ......... 023 036
EFFECTLESS-EXCEPT-CONS .................. SCHEME FUNCTION ......... 023 039
EFFS-ANALYZE ............................ SCHEME FUNCTION ......... 014 006
EFFS-ANALYZE-COMBINATION ................ SCHEME FUNCTION ......... 015 031
EFFS-ANALYZE-IF ......................... SCHEME FUNCTION ......... 015 010
EFFS-INTERSECT .......................... SCHEME FUNCTION ......... 023 028
EFFS-UNION .............................. SCHEME FUNCTION ......... 015 002
EMPTY ................................... SCHEME FUNCTION ......... 002 005
ENV-ANALYZE ............................. SCHEME FUNCTION ......... 012 018
ENVCARCDR ............................... SCHEME FUNCTION ......... 043 002
EQ ...................................... SIDE EFFECTS ............ 017 044
EQCASE .................................. MACLISP MACRO ........... 003 032
ERASE-ALL-NODES ......................... MACLISP MACRO ........... 018 005
ERASE-NODE .............................. MACLISP MACRO ........... 018 004
ERASE-NODES ............................. SCHEME FUNCTION ......... 018 007
FILTER-CLOSEREFS ........................ SCHEME FUNCTION ......... 039 030
FIXP .................................... SIDE EFFECTS ............ 017 050
FLOATP .................................. SIDE EFFECTS ............ 017 051
FN ...................................... FN-SIDE-EFFECTS ......... 016 035
FN ...................................... FN-SIDE-AFFECTED ........ 016 036
FN ...................................... OKAY-TO-FOLD ............ 016 037
FN-SIDE-AFFECTED ........................ PROPERTY ................ 016 036
FN-SIDE-EFFECTS ......................... PROPERTY ................ 016 035
GEN-GLOBAL-NAME ......................... SCHEME FUNCTION ......... 002 041
GENFLUSH ................................ SCHEME FUNCTION ......... 002 036
GENTEMP ................................. SCHEME FUNCTION ......... 002 030
HUNKFN .................................. @DEFINE ................. 004 028
HUNKFN .................................. MACLISP MACRO ........... 004 030
HUNKP ................................... SIDE EFFECTS ............ 017 049
IF ...................................... DATA TYPE ............... 008 038
INCREMENT ............................... MACLISP MACRO ........... 002 014
INTERSECT ............................... SCHEME FUNCTION ......... 006 039
LABELS .................................. DATA TYPE ............... 008 050
LAMBDA .................................. DATA TYPE ............... 008 034
LAMBDACATE .............................. SCHEME FUNCTION ......... 054 006
LIST .................................... SIDE EFFECTS ............ 017 056
LOOKUPICATE ............................. SCHEME FUNCTION ......... 061 002
MACRO ................................... @DEFINE ................. 001 065
MACRO-EXPAND ............................ SCHEME FUNCTION ......... 011 026
MAKE-RETURN ............................. SCHEME FUNCTION ......... 028 002
MAP-USER-NAMES .......................... SCHEME FUNCTION ......... 065 028
MAPANALYZE .............................. SCHEME FUNCTION ......... 057 023
MEMQ .................................... SIDE EFFECTS ............ 017 058
META-COMBINATION-LAMBDA ................. SCHEME FUNCTION ......... 021 007
META-COMBINATION-TRIVFN ................. SCHEME FUNCTION ......... 020 037
META-EVALUATE ........................... SCHEME FUNCTION ......... 019 007
META-IF-FUDGE ........................... SCHEME FUNCTION ......... 020 010
META-SUBSTITUTE ......................... SCHEME FUNCTION ......... 024 009
NAME .................................... MACLISP MACRO ........... 004 011
NAME .................................... ACCESS MACRO ............ 004 031
NAME .................................... COMPONENT-NAMES ......... 005 060
NAME .................................... SUPPRESSED-COMPONENT-NAMES 005 061
NODE .................................... DATA TYPE ............... 008 012
NODIFY .................................. SCHEME FUNCTION ......... 008 059
NOT ..................................... SIDE EFFECTS ............ 017 053
NULL .................................... SIDE EFFECTS ............ 017 054
```

```
NUMBERP ............................... SIDE EFFECTS ............ 017 046
OKAY-TO-FOLD .......................... PROPERTY ................ 016 037
OUTPUT-ASET ........................... SCHEME FUNCTION ......... 061 017
PAIRLIS     ........................... SCHEME FUNCTION ......... 007 002
PASS1-ANALYZE ......................... SCHEME FUNCTION ......... 007 041
PASSABLE .............................. SCHEME FUNCTION ......... 023 042
PRIN1 ................................. SIDE EFFECTS ............ 017 061
PRINC ................................. SIDE EFFECTS ............ 017 062
PRINT ................................. SIDE EFFECTS ............ 017 060
PRINT-SHORT ........................... EXPR .................... 003 016
PRINT-WARNING ......................... SCHEME FUNCTION ......... 003 005
PROCESS-DEFINE-FORM ................... SCHEME FUNCTION ......... 069 002
PROCESS-DEFINITION .................... SCHEME FUNCTION ......... 069 020
PROCESS-FORM .......................... SCHEME FUNCTION ......... 068 006
PRODUCE-ASET .......................... SCHEME FUNCTION ......... 046 003
PRODUCE-COMBINATION ................... SCHEME FUNCTION ......... 051 002
PRODUCE-COMBINATION-VARIABLE .......... SCHEME FUNCTION ......... 051 026
PRODUCE-CONTINUATION-RETURN ........... SCHEME FUNCTION ......... 052 023
PRODUCE-IF ............................ SCHEME FUNCTION ......... 045 042
PRODUCE-LABELS ........................ SCHEME FUNCTION ......... 047 003
PRODUCE-LAMBDA-COMBINATION ............ SCHEME FUNCTION ......... 048 003
PRODUCE-RETURN ........................ SCHEME FUNCTION ......... 053 002
PRODUCE-RETURN-1 ...................... SCHEME FUNCTION ......... 053 021
PRODUCE-TRIVFN-COMBINATION ............ SCHEME FUNCTION ......... 049 002
PRODUCE-TRIVFN-COMBINATION-CONTINUATION SCHEME FUNCTION ......... 049 023
PRODUCE-TRIVFN-COMBINATION-CVARIABLE .. SCHEME FUNCTION ......... 050 002
PSETQ-ARGS ............................ SCHEME FUNCTION ......... 057 002
PSETQ-ARGS-ENV ........................ SCHEME FUNCTION ......... 057 006
PSETQ-TEMPS ........................... SCHEME FUNCTION ......... 057 013
PSETQIFY .............................. SCHEME FUNCTION ......... 055 005
PSETQIFY-METHOD-2 ..................... SCHEME FUNCTION ......... 055 012
PSETQIFY-METHOD-3 ..................... SCHEME FUNCTION ......... 056 002
READ .................................. SIDE EFFECTS ............ 017 065
READIFY ............................... SCHEME FUNCTION ......... 067 042
REANALYZE1 ............................ SCHEME FUNCTION ......... 023 002
REFD-VARS ............................. SCHEME FUNCTION ......... 034 053
REGSLIST .............................. SCHEME FUNCTION ......... 044 004
REMARK-ON ............................. SCHEME FUNCTION ......... 065 002
REMOVE ................................ SCHEME FUNCTION ......... 006 047
RESET-STATS ........................... SCHEME FUNCTION ......... 071 055
RETURN ................................ DATA TYPE ............... 026 049
RPLACA ................................ SIDE EFFECTS ............ 017 041
RPLACD ................................ SIDE EFFECTS ............ 017 042
RPLACX ................................ SIDE EFFECTS ............ 017 043
SCHMAC ................................ @DEFINE ................. 001 064
SET-UP-ASETVARS ....................... SCHEME FUNCTION ......... 044 032
SETDIFF ............................... SCHEME FUNCTION ......... 006 058
SETPROP ............................... SCHEME FUNCTION ......... 006 018
SEXPRFY ............................... SCHEME FUNCTION ......... 070 004
STATS ................................. SCHEME FUNCTION ......... 071 046
SUBST-CANDIDATE ....................... SCHEME FUNCTION ......... 022 006
SUPPRESSED-COMPONENT-NAMES ............ PROPERTY ................ 005 061
SX .................................... MACLISP MACRO ........... 003 028
SYMBOLP ............................... SIDE EFFECTS ............ 017 048
TEMPLOC ............................... SCHEME FUNCTION ......... 042 029
TERPRI ................................ SIDE EFFECTS ............ 017 063
TEST-COMPILE .......................... SCHEME FUNCTION ......... 007 051
TRANSDUCE ............................. SCHEME FUNCTION ......... 067 002
TRIV-ANALYZE .......................... SCHEME FUNCTION ......... 013 012
TRIV-ANALYZE-FN-P ..................... SCHEME FUNCTION ......... 013 056
TRIV-LAMBDACATE ....................... SCHEME FUNCTION ......... 063 033
TRIVFN ................................ SCHEME FUNCTION ......... 002 009
TRIVIAL ............................... DATA TYPE ............... 026 013
TRIVIALIZE ............................ SCHEME FUNCTION ......... 063 002
TYI ................................... SIDE EFFECTS ............ 017 066
TYO ................................... SIDE EFFECTS ............ 017 064
TYPE .................................. HUNK ACCESS MACRO ....... 005 006
```

TYPEP	SIDE EFFECTS	017	047
UNION	SCHEME FUNCTION	006	033
USED-TEMPLOCS	SCHEME FUNCTION	064	059
VARIABLE	DATA TYPE	008	030
WARN	MACLISP MACRO	003	002