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DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
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-passinq 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 <u>interpreters</u> 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 _combinations_, 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 <u>list</u> 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 nearly 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 <u>continuation</u>, 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 <u>all</u> 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 <u>not</u> 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 pos[illegible] 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 symbols to decide which variable 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)
```