# An Optimizing Compiler for Lexically Scoped LISP

Rodney A. Brooks
*Massachusetts Institute of Technology*
*Cambridge, Massachusetts 02139*

Richard P. Gabriel
*Stanford University*
*Stanford, California 94305*
*and*
*Lawrence Livermore National Laboratory*
*University of California*
*Livermore, California 94550*

Guy L. Steele Jr.
*Carnegie-Mellon University*
*Pittsburgh, Pennsylvania 15213*

## Abstract

We are developing an optimizing compiler for a dialect of the LISP language. The current target architecture is the S-1, a multiprocessing supercomputer designed at Lawrence Livermore National Laboratory. While LISP is usually thought of as a language primarily for symbolic processing and list manipulation, this compiler is also intended to compete with the S-1 PASCAL and FORTRAN compilers for quality of compiled numerical code. The S-1 is designed for extremely high-speed signal processing as well as for symbolic computation; it provides primitive operations on vectors of floating-point and complex numbers. The LISP compiler is designed to exploit the architecture heavily.

The compiler is structurally and conceptually similar to the BLISS-11 compiler and the compilers produced by PQCC. In particular, the TNBIND technique has been borrowed and extended.

Particularly interesting properties of the compiler are:

- Extensive use of source-to-source transformations.
- Use of an intermediate form that is expression-oriented rather than statement-oriented.
- Exploitation of tail-recursive function calls to represent complex control structures.
- Efficient compilation of code that can manipulate procedural objects that require heap-allocated environments.
- Smooth run-time interfacing between the "numerical world" and "LISP pointer world", including automatic stack allocation of objects that ordinarily must be heap-allocated.

Each of these techniques has been used before, but we believe their synthesis to be original and unique.

The compiler is table-driven to a great extent, more so than BLISS-11 but less so than a PQCC compiler. We expect to be able to to redirect the compiler to other target architectures such as the VAX or PDP-10 with relatively little effort.



## 1 Introduction

LISP has historically been thought of as an odd language, out of the mainstream of programming language design (this mainstream consisting primarily of ALGOL 60 and its descendants):

> LISP is unusual, in the sense that it clearly deviates from every other type of programming language that has ever been developed. [37]

The primary applications for LISP have historically been artificial intelligence research and symbolic computation. These applications as a rule do not require large amounts of numerical computation, and so for most LISP implementations little effort is invested in making numerical computations efficient. As an understandable consequence, many people have come to assume that the inefficiency of LISP in performing numerical computation is inherent in the language, rather than simply the result of lack of attention in the implementations.

Eventually there arose an application for LISP that required fairly large amounts of numerical computation in addition to powerful symbolic manipulation: the MACSYMA symbolic algebra system [31]. In 1969 to 1972 the needs of MACSYMA induced the implementors of MACLISP to invest the effort necessary to construct a LISP compiler that could generate high-quality numerical code [57, 44]. An experiment by Fateman in 1973 [17] established that the resulting compiler compared favorably with the FORTRAN compiler for the same machine (the Digital Equipment Corporation PDP-10 [14]).

The compiler described here, the S-1 LISP compiler, combines compilation techniques from three sources:

1. Techniques from "mainstream" compiler technology that have never before been systematically applied to the LISP language. These include the TNBIND register allocation techniques [28]developed for BLISS-11 [58] and PQCC [29]. Here we report that these techniques can be profitably applied to the LISP language.
2. Techniques developed specifically for LISP. These include the methods of the MACLISP compiler for interfacing symbolic and numerical computations. Here we report on a systematic generalization of these techniques that considerably simplifies the compiler.
3. Techniques of a heretofore largely theoretical nature. These include the reliance on source-to-source transformation techniques that exploit the ability of the source language to treat procedures as first-class data objects [45, 49]; in effect, the denotational-semantic

definitions of language constructs can be used *directly* by this compiler to compile those constructs *efficiently*, because the compiler performs sufficiently elaborate special-case analyses of lambda-expressions. Here we report that these techniques have been applied successfully in a real application environment.

The remainder of the paper gives an overview of the language dialect, an overview of the target architecture, and a description of the overall compiler compiler structure We then discuss in detail two aspects of particular interest: source-level program transformations and generation of good numerical code. Finally, the entire compilation process is briefly described for a small example program.

## 2 Language Overview

The language compiled is a dialect of LISP (and is in fact an extension of COMMON LISP, a descendant of MACLISP [32] and Lisp Machine LISP [56]). Properties of LISP that distinguish it from the ALGOL family (such as PASCAL and ADA) and that affect compilation strategy:

- Types are attached to run-time data objects, rather than to compile-time named variables. Programs can explicitly test the types of objects. Moreover, the type-checking prerequisite to some type-specific operation (such as making sure that multiplication is performed on numbers and not on strings) must in principle be done at run time, although suitable declarations and compile-time analysis may permit compile-time type analysis in some situations.
- All values, not only those in variables but those computed by expressions, are pointers to objects, not the objects themselves. In ADA parlance, every user-visible LISP data type is an access type, and every LISP object is an access value. (Because dereferencing is in effect present *everywhere*, notation for the dereferencing can be everywhere suppressed; thus the ^ operator of PASCAL [27], the `.all` selector of ADA [1, 2], or the VAL operator of ECL [55] is not required.)
- Variables may be dynamically scoped (the LISP term for this is "special"). This means that a routine may refer to variables bound by its caller. (Dynamic scoping tends to arise in those languages, and only those languages, that have interactive implementations. The best example of this besides LISP is APL [25, 16].)

Properties of this particular LISP dialect that distinguish it from other dialects of LISP:

- Functions are full-fledged data objects of the language. (In LISP parlance, one may pass around the function itself (compiled or interpreted), rather than a symbol whose `expr` or `subr` property is the function.)
- This dialect supports lexically scoped variables *as well as* dynamically scoped variables. By "lexical scoping" we essentially mean ALGOL-like scope rules. However, because it is permissible for a function to return as its value another function, sometimes environment structures must be heap-allocated rather than stack-allocated. See [33, 51, 42, 50]. This is in contrast with the rules for ALGOL 60 [34] and ALGOL 68 [54].
- While the language is "compile-time typeless", the user may optionally provide type declarations, which can be checked for correctness by the interpreter and treated as advice by the compiler.
- The language has "tail-recursive" semantics: recursive procedures of a certain form have iterative behavior. For example, the following procedure behaves iteratively (it cannot produce stack overflow no matter how large n is):

```
(defun expt1 (x n a)     ;Compute a*x^n by
  (cond ((zerop n) a)    ; repeated squaring.
        ((oddp n)
         (expt1 (* x x)
                (floor (/ n 2))
                (* a x)))
        (t (expt1 (* x x)
                  (floor (/ n 2))
                  a))))
```

A rendering of this example into PASCAL would be:

```
procedure expt1(x: real, n: integer, a: integer);
begin
  if n=0 then expt1 := a
  else if odd(n) then expt1 := expt1(x*x, n div 2, a*x)
  else expt1 := expt1(x*x, n div 2, a*x)
end
```

The procedure `expt1` calls itself only in places where such a call is the *last thing* to be done by the current procedure activation; a procedure call in this case is more akin to a parameter-passing **goto** than to a recursive call, and can be implemented as such, as a simple unconditional branch. See [7, 23, 24, 51, 43, 42, 46, 47].

- Procedures can accept varying numbers of arguments; the procedure header may specify default computations for missing arguments, or may ask to receive a list of any and all arguments beyond a certain point. (The parameter defaulting here is more general than that in ADA; a default-value expression may perform any computation, and may refer to other parameters occurring earlier in the same formal parameter set. The parameter syntax was adopted indirectly from MDL [19]; compare this with [18]. An example appears later in this paper.)
- A rich set of numerical data types is provided, including integers of indefinite size, rational numbers, floating-point numbers of several precisions, and complex numbers.

## 3 Target Architecture Overview

The S-1 computer is a so-called "supercomputer" [30], designed for extremely high-speed numerical computations such as signal processing and numerical simulations. While most of the properties of the compiler described here are not tied to peculiarities of the S-1, the S-1 architecture [12] does have some unusual features (as well as some ordinary ones) that a good compiler should strive to exploit:

- Words are 36 bits, quarter-word addressible (bytes are 9 bits, not 8).
- There are 32 general-purpose registers, each 36 bits wide. A few of these have special purposes (for example, the "RT" registers described below) or special restrictions (for example, some of them cannot be be used for short indexing).

- Virtual addresses are 31 bits plus a five-bit tag. Nine of the 32 possible tags have special meaning to the architecture (to implement MULTICS-like ring protection [39, 35], among other things); the others may be used freely as user data-type tags. (S-1 LISP of course uses most of these tags to indicate LISP data types. Some care is needed, however, not to run afoul of the ring protection mechanism.)

- Rather complex addressing modes are provided. For example, one may take a general-purpose register (call it Rb), add a 26-bit signed offset *bo* to it, fetch the word addressed, and call it *b*; take another general-purpose register Rn, add a 6-bit signed offset *no* (left-shifted two bits) to it, fetch the word addressed, and call in *n*; then left-shift *n* by *sh* bits (for *sh*=0, 1, 2, or 3), then add that to *b* to produce the final operand address. (The computation performed can be expressed by the BLISS expression

  .(.(.Rb+*bo*)+(.(.Rn+(*no*^2))^*sh*)

  where "^" is a left-shift operator and "." is the "contents-of" operator. Such an addressing mode can, in one operand, fetch from a record a component that is a pointer to an array, fetch an index from a local variable in the stack, adjust the index for the array's element size, and fetch the selected array element.)

- Instructions occupy one, two, or three 36-bit words. Every operation has a one-word format, consisting of a 12-bit opcode and two 12-bit operand specifiers. The second and third words are needed only for the more complex addressing modes, one extra word for each of two operands. (In this the S-1 vaguely resembles the DEC PDP-11 [13].)

- Most arithmetic instructions for binary operations are "2 1/2 address". Only two general operand calculations are available to the instruction; a clever encoding scheme multiplexes these among two sources and a destination. The three operands to ADD (for example) may be in three distinct places, provided that one of them is one of the two registers named RTA and RTB (general-purpose registers 4 and 6). If the destination and one source are identical, then both addresses may be general memory locations (as in the PDP-11). As an example, these patterns are permissible for the "subtract" instruction (M1 and M2 are arbitrary memory or register addresses):

```
SUB M1,M2              ;M1 := M1-M2
SUB RTA,M1,M2          ;RTA := M1-M2
SUB RTB,M1,M2          ;RTB := M1-M2
SUB M1,RTA,M2          ;M1 := RTA-M2
SUBV M1,M2             ;M1 := M2-M1
SUBV M1,RTA,M2         ;M1 := M2-RTA
```

  For commutative operations such as ADD, only the first four formats are provided because the last two are redundant.

- A variant of IEEE proposed standard floating-point arithmetic [10, 26] is provided, including special "overflow", "underflow", and "undefined" values. (The arithmetic does not conform to the IEEE proposed standard partly because it was designed before the standard was announced and partly because of the 36-bit word size; however, it was developed with the help of some of the designers of the IEEE proposed standard.)

- There are sixteen rounding modes for floating-point operations (including all the rounding modes mandated by [26]) and for integer division (thus `floor`, `ceiling`, `truncate`, `round`, `mod`, and `rem` are all primitive instructions).

- There are single instructions for complex arithmetic, SIN, COS, EXP, LOG, SQRT, ATAN, and so on.

- There are vector processing instructions to perform component-wise arithmetic, vector dot product, matrix transposition, convolution, Fast Fourier Transform, and string processing. While a compiler may not output the FFT instruction every day, the vector and string-processing instructions are more frequently useful.

- The standard configuration is a multiprocessor; synchronization instructions are available to the user. (These are in turn made available to the LISP user. Moreover, the run-time system, and especially the garbage collector, has been written with multiprocessing in mind.)

## Compiler Structure

The S-1 LISP compiler is similar in structure to the BLISS-11 compiler [58]. The source program is converted to an internal tree format whose structure reflects the expression structure of the program. (The tree contains extra cross-links that effectively make it a general graph, but it is useful to think of it as an augmented tree to emphasize its structural relationship to the source program.) Each node of the tree has extra data slots; these are filled in by successive phases of the compiler. Occasionally the tree is transformed. Finally, after many phases have performed their analysis, code is generated in a single pass (a postorder tree walk) over the internal tree. The highest-level decomposition of the compiler is shown in Table 1. A few remarks on this structure are in order here.

### 4.1 Preliminary Conversion

The preliminary phase creates the internal tree structure. Each node of the tree corresponds quite directly to one of a small number of source-level constructs, such as constants, variable references, lambda-expressions, and if-then-else conditionals; the set of constructs used is shown in Table 2. All other program constructs are expanded as macros or otherwise re-expressed in terms of the small basic set. Therefore the internal tree can always be back-translated into valid source code, equivalent to, though not necessarily identical to, the original source. (Such a back-translation facility has been written as a debugging aid for the compiler writers. Examples of its output appear later in this paper.) For example, this input program:

```
(defun quadratic (a b c)
  (let ((d (- (* b b) (* 4.0 a c))))
    (cond ((< d 0) '())
          ((= d 0) (list (/ (- b) (* 2.0 a))))
          (t (let ((2a (* 2.0 a))
                   (sd (sqrt d)))
               (list (/ (+ (- b) sd) 2a)
                     (/ (- (- b) sd) 2a)))))))
```

would be transformed into an internal tree that would back-translate into:

- **Preliminary.** Syntax checking. Resolving of variable references. Expansion of macro calls. Very simple program transformations. Conversion to internal tree form.
- **Source-program analysis.** Tree decoration required for transformations.
  - **Environment analysis.** For each subtree, determine the sets of variables read and written within that subtree. For each variable binding, attach a list of all referent nodes.
  - **Side-effects analysis.** For each subtree, classify the possible side-effects produced by its execution, and the side-effects that might adversely affect such execution.
  - **Complexity analysis.** Make a preliminary estimate of the size of the object code for each subtree. (This is primarily to aid the optimizer in deciding whether to substitute copies of the initializing expression for several occurrences of a variable).
  - **Tail-recursion analysis.** For each node, make a list of other nodes that potentially generate its value.
  - **[Data-type analysis.** Processing of optional user-specified type declarations, and deduction of types of intermediate values.]
- **Source-level optimization.** Transform the tree in ways that can be back-translated to source-level code. (Many optimizations usually performed on object code are taken care of here.)
- **[Common subexpression elimination.** This too can be expressed as tree transformations that can be back-translated into source-level `let` constructs.]
- **Machine-dependent annotation.** Tree decoration preceding code generation.
  - **Special variable lookups.** Determine when to search for special variable bindings (for a deep-binding implementation).
  - **Binding annotation.** Determine how each internal lambda-expression is to be compiled. Determine which variables may be stack-allocated and which require heap allocation.
  - **Representation annotation.** Determine, for every variable and every temporary value, the machine representation to be used for that value.
  - **Pdl number annotation.** Determine which numerical quantities may be stack-allocated rather than heap-allocated, despite passing pointers to them to other procedures. (The term "pdl" (pronounced "piddle") is an abbreviation for "push-down list", which is an approximate synonym for "stack".)
  - **Target annotation.** The TNBIND and PACK phases of BLISS-11 and PQCC.
- **Code generation.** Generate code in a single pass over the tree. Code is generated in forwards order. This is partly procedural and partly table-driven.
- **[Peephole optimizer.** Perform cross-jumping and branch tensioning.]

(Square brackets indicate portions not yet coded or coded only in preliminary form.)

**Table 1:** Phase Structure of the S-1 LISP Compiler

```
(defun quadratic (a b c)
  ((lambda (d)
     (if (< d (quote 0))
         (quote ())
         (if (= d (quote 0))
             (list (/ (- b) (* (quote 2.0) a)))
             ((lambda (2q sd)
                (list (/ (+ (- b) sd) 2a)
                      (/ (- (- b) sd) 2a)))
              (* (quote 2.0) a)
              (sqrt d)))))
   (- (* b b) (* (quote 4.0) a c))))
```

The `let` constructs are expressed in terms of function calls to explicitly appearing lambda-expressions, and the `cond` construct is expressed in terms of multiple if-then-else forms. (Internally `if` is used rather than `cond` because it is simpler and symmetric, making program transformations easier.) Also, all constants are internally explicitly quoted for uniformity, as shown above (but for readability the back-translator actually omits `quote`-forms around numbers).

There is no single, central "symbol table" data structure. Instead, this information is effectively spread throughout the internal tree. With every distinct variable (two variables with the same name may be distinct because of scoping rules) is associated a little data structure; the construct that binds the variable (if any) and all references to the variable all point to the data structure, which has back-pointers to the binding and all the references.

### 4.2 Analysis and Optimization

The next two phases (source-program analysis and source-level optimization) are actually executed in a complicated co-routining manner for efficiency. Conceptually the analysis is performed first and the results made available to the optimizer. However, optimization can alter the program, requiring re-analysis. A system of flags, one per node to indicate which nodes require re-analysis, effectively permits re-analysis to be performed incrementally and only where necessary.

### 4.3 Common Sub-expression Elimination

Common sub-expression elimination has not yet been implemented, because preliminary experiments indicate that its contribution to program speed will be smaller than the other techniques that have been implemented. Like the source-level optimization phase, its use is completely optional, for it only affects the efficiency of the resulting code and can be expressed as a source-level transformation [49] using lambda-expressions.

| | |
|---|---|
| `literal` | Constants (LISP `quote` construct). |
| `variable` | Variable reference. |
| `caseq` | A case statement. |
| `catcher` | Analogous to the MACLISP `catch` construct, which is a target for non-local exits. |
| `go` | Goto statement. |
| `if` | An if-then-else construct. |
| `lambda` | A lambda-expression. The value of a lambda-expression is a function (a lexical closure). |
| `progbody` | A construct that contains tagged statements. A `go` can jump to a tag, and a `return` can exit the construct. The usual LISP `prog` construct translates into a `let` (which takes care of the variable bindings, and is further translated into a `call` of a `lambda`) containing a `progbody` (which takes care of `go` and `return`). |
| `progn` | Sequential execution, the equivalent of a **begin-end** block. |
| `return` | This can exit from a surrounding `progbody`. |
| `setq` | Assignment to a variable. |
| `call` | Function invocation. This has three special cases of interest: calling a lambda-expression (`let`), calling a known primitive operation (to be compiled in-line), and calling a user- or system-defined function. |

Each node of the compiler's internal tree structure corresponds to one of these source-level constructs.

**Table 2:** Basic Internal Constructs

(The elimination of common sub-expressions could be performed as a part of the source-level optimization phase. It is designed as a separate phase primarily to avoid a thrashing problem. It is very important that the source-level optimization be able to perform common sub-expression *introduction* (substituting for occurrences of a variable the expression to whose value the variable is initialized). Having separate phases avoids the possibility of an endless cycle of introductions and eliminations. Right now the heuristics for introduction are relatively conservative, and as a result some optimizations that would be triggered by such introductions are missed. These heuristics can be made more liberal when the source-level optimizer can count on the common sub-expression elimination phase to reverse its poor decisions.)

### 4.4 Machine-dependent Annotation

The previous phases are in principle completely machine-independent (though some of the source-level transformations are slanted toward the S-1: they would be benign but useless for certain other architectures). From this point on the data collected and added to the tree is machine-dependent, at least in the sense that it depends on certain aspects of the implementation techniques rather than only on the language semantics.

The S-1 LISP implementation uses the *deep-binding* technique for implementing dynamically scoped variables; this technique has also been used in INTERLISP [6, 52] (but INTERLISP has since converted to a shallow-binding technique [53]). Deep binding calls for binding a variable by pushing its name and new value onto a stack. This allows for fast context switching among processes with different sets of bindings (all that is required is to switch stack pointers), but in general requires a linear search when accessing a variable. This is in contrast with shallow binding, in which the current value of a variable is maintained in a fixed location, and a variable is bound by pushing its name and *old* value onto a stack and then installing its new value in the fixed location. This allows constant-time access, but for a context switch an arbitrarily large number of variables may have to be changed. (For a discussion of deep and shallow binding techniques and the trade-offs involved, see [5].)

The S-1 LISP compiler uses the same trick formerly used in INTERLISP to reduce this search overhead: on entry to a function, all the special variables needed by that function are searched for *once* and pointers to the relevant stack locations are cached in the function's local activation frame; from then on each special variable can be accessed indirectly through a cached pointer in constant time. The S-1 LISP compiler actually generalizes the trick further. For each variable the smallest subtree that contains all the references is determined; the lookup and pointer caching for that variable is performed before execution of that smallest subtree. This may avoid a lookup if the subtree is in an arm of a conditional. The trick is further refined to take loops into account.

The binding annotation phase examines each lambda-expression in the tree and determines how that lambda-expression is to be compiled. In the most general case, a closure object must be explicitly constructed at run time, containing the current lexical environment and a pointer to the code. (Such an object is effectively an *actor* [42], and can be used to implement object-oriented programming in the SMALLTALK [21] sense.) However, in many special cases this is not necessary. If through compile-time analysis all the places can be found where the lambda-expression may be invoked, then it may be possible to compile all such calls as, in effect, parameter-passing **goto** statements, and no closure need be constructed at run time. If not all calls to the lambda-expression are tail-recursive, it may be appropriate to compile the lambda-expression using a special (fast) subroutine linkage that can avoid error checks such as on the number of arguments passed and can even use special register conventions; this is feasible because all calls can be found and compiled for the same special-purpose calling convention. The binding analysis also determines which variables can be stack-allocated and which must (because they are referred to by closures) be heap-allocated.

The representation annotation phase determines how each intermediate quantity of the (transformed) source program is to be represented internally. In principle, arbitrarily complex decisions can be made on the basis of the operations to be performed on each quantity. In practice, the most important decisions are as to whether a numerical quantity should be represented in LISP pointer format or in "raw machine form". (This issue is discussed in greater detail below.)

The pdl number annotation phase determines, for those situations where a number in "raw machine format" must be converted to LISP pointer format, whether stack allocation of the number will provide a sufficient lifetime or whether the general heap-allocation of a number is required. (This issue is discussed in greater detail below.)

The target annotation phase performs all register and memory allocation.

### 4.5 Code Generation

Code generation is performed during a single tree walk over the decorated program tree. There is nothing particularly special about this phase. It is largely coded procedurally and frequently but not systematically table-driven.

Currently there is no peephole optimizer. Experimentation has shown that every time we have been tempted to add a peephole optimizer to eliminate some terrible-looking circumlocution in the generated code, a little thought has led to an insight resulting in an improvement to either the source-level transformation phase or the target annotation phase. Most global improvements to the object code have had some means of expression in terms of source-level constructs, and so lend themselves to source-level transformations (it is these we mean by transformations useful to the S-1 (say) and benign but useless for some, though not all, other architectures). Most local improvements to the object code involve reduction of data movement, and so generally fall prey to sharper analysis in the register allocator.

The one optimization for which we may need to add a peephole optimizer is branch tensioning. It is very difficult to express the elimination of branches to branch instructions at the source level, because branch instructions do not appear in the internal tree, but rather are artifacts of the embedding of the tree into a linear instruction stream. Rather than building a peephole optimizer, however, we have in mind experimenting with a global process for packing linear blocks that would handle branch tensioning, perform "loop rotations" that put the conditional test for a loop at the bottom rather than the top, take frequency information for conditional branches into account, and perhaps also account for the fact that some branch instructions have limited ranges. (A version of such a process is used in PQCC, primarily to solve the loop-rotation problem; for that purpose it works rather well.)

## 5 Source-Language Transformations

In general, all source-program constructs outside a certain small set are re-expressed as combinations of constructs within the set, and this small set of constructs is a subset of the source language; therefore the final transformed tree can be converted back into a source program. Each construct outside the set is expanded in a uniform and general way; for the most part the compiler relies on a small set of general optimization techniques to produce special-case efficiencies. In this respect the S-1 LISP compiler resembles the "General Purpose Compiler" [38, 8, 22]. It differs in that the fundamental set of constructs is expression-oriented rather than statement-oriented. As an example of the distinction, where most compilers might translate a complex control structure into a network of tags and **goto** statements within a **begin-end** block, the S-1 LISP compiler will translate the same structure into an arrangement of procedure definitions and calls. (The tail-recursive language semantics are crucial here.) As an example, the short-circuiting of boolean expressions in an `if` is handled as follows:

```
(if (and a (or b c)) expression1 expression2)
```

expands by a series of transformations roughly as follows. First **and** and **or** are re-expressed in terms of `if`.

```
(if (if a (if b b c) false) expression1 expression2)
```

(To be precise, the subexpression `(or b c)` is actually translated into

```
((lambda (v f) (if v v (f))) b (lambda () c))
```

to avoid evaluating `b` twice, and then simplified, but we ignore these details here.) Next the transformation

```
      (if (if x y z) v w)
==>
      ((lambda (f g)
         (if f (if y (f) (g)) (if z (f) (g))))
       (lambda () v)
       (lambda () w))
```

is applied twice. The functions `f` and `g` are introduced to avoid space-wasting duplication of the code for `v` and `w`. This transformation is the essence of the boolean short-circuiting idea; all the rest is "merely" simplification. (Variations of this idea have of course appeared before [40, 41, 4] but none of these mentions the use of functions to avoid code duplication. The transformation of nested `if` forms given here first appeared in [45, 49].)

```
((lambda (f1 f2)
   (if a
       (if (if b b c) (f1) (f2))
       (if false (f1) (f2))))
 (lambda () expression1)
 (lambda () expression2))
```

which then becomes

```
((lambda (f1 f2)
   (if a
       ((lambda (f3 f4)
          (if b (if b (f3) (f4))
                (if c (f3) (f4))))
        (lambda () (f1))
        (lambda () (f2)))
       (if false (f1) (f2))))
 (lambda () expression1)
 (lambda () expression2))
```

Simplification of conditionals (including realizing that `b` is *true* in the inner `if` by virtue of the test in the outer one) yields:

```
((lambda (f1 f2)
   (if a
       ((lambda (f3 f4)
          (if b (f3) (if c (f3) (f4))))
        (lambda () (f1))
        (lambda () (f2)))
       (f2)))
 (lambda () expression1)
 (lambda () expression2))
```

Integration of procedures that are referred to in only one place yields:

```
    ((lambda (f2)
       (if a
           ((lambda (f3)
              (if b (f3) (if c (f3) (f2))))
            (lambda () expression1))
           (f2)))
     (lambda () expression2))
```

Here the tail-recursive semantics come into play. The calls to the functions `f2` and `f3` are tail-recursive, and so are compiled as simple

jump instructions; in effect such calls represent simple goto's. Given this we may render the above program as:

```
    if a then
       if b then goto g
          else if c then goto g else goto f
       else goto f;
  g: expression1;
     goto h;
  f: expression2;
  h:
```

which does indeed express the desired short-circuiting. The S-1 LISP compiler does compile the calls to `f2` and `f3` as jumps, and the resulting code is identical to what you would expect from a good compiler for boolean short-circuiting.

Of course, all this would be a foolish and roundabout method merely to implement boolean short-circuiting. The point is that boolean short-circuiting falls out as a special case of the application of such more general and powerful techniques as procedure integration, constants folding, and a general transformation on nested `if` expressions.

The three most important transformations used by the compiler are as follows. First,

`((lambda () body))` ==> *body*

That is, a call with no arguments to a manifest lambda-expression with no parameters can be replaced by the body of the lambda-expression. Second,

```
((lambda (v1 v2 ... vj ... vn) body)
 a1 a2 ... aj ... an)
==>
((lambda (v1 v2 ...  ... vn) body)
 a1 a2 ...  ... an)
```

provided that *vj* is not referenced in *body* and execution of *aj* has no side effects (except possibly heap-allocation, which is a side effect that may be eliminated but must not be duplicated). Third, in

```
((lambda (v1 v2 ... vj ... vn) body)
 a1 a2 ... aj ... an)
```

the expression *aj* may be used to replace some or all occurrences of *vj* in *body* provided that certain complicated conditions regarding side effects are satisfied [3, 20, 58, 40].

The third rule creates an exception to the second. If the variable corresponding to an argument with a side effect has one reference, it *may* be possible to substitute the argument expression for the variable using the third rule, *provided* that the second rule is immediately applied to eliminate the argument, lest the expression be evaluated twice after all! This requires some collusion.

The third rule expresses the fact that `lambda` can be viewed as a renaming operator [42], and together the three rules constitute the lambda-calculus rule of beta-conversion [9]: if the third rule can be applied to all arguments, then the second rule will be applicable to all arguments, and then the first rule will eventually apply. It is useful to have three distinct rules, however, so that partial beta-conversion can be done. (This is equivalent to noting that LISP allows lambda-expressions of more than one argument, while in pure lambda-calculus each lambda-expression binds exactly one variable.)

Certain optimization techniques widely known by other names [3] effectively fall out as special cases of beta-conversion. *Constant propagation* (*subsumption*) obviously is one. Another is *procedure integration*; this occurs as the special case where the expression providing the value of a variable is a lambda-expression. If a (tail-)recursive procedure definition is used to achieve iteration (as in the example procedure `expt1` above), then integration of the procedure within itself achieves *loop unrolling*. (The heuristics of the S-1 LISP compiler are so conservative as to avoid loop unrolling completely in the name of avoiding indefinite regress; however, all that is needed is a more discriminating decision procedure, as the compiler already contains the necessary procedure integration machinery.) That these special cases easily fall out from the more general transformations on lambda-expressions is a direct reflection on the power, simplicity, and universality of the lambda-calculus.

Other transformations used by the S-1 LISP compiler include compile-time expression evaluation (invoking primitive functions known to be free of side effects on constant operands, a very convenient thing to do in LISP with the `apply` operator!), dead code elimination (simplification of `if` and `caseq` constructs with constant predicate or key expressions), and certain manipulations of associative and commutative operators (such as table-driven elimination of identity operands). A few transformations are in themselves not useful, such as

```
(if (progn a b ... p q) x y)
==>
(progn a b ... p (if q x y))
```

and

```
(if ((lambda (v1 v2 ...) body) a1 a2 ...) x y)
==>
((lambda (v1 v2 ...) (if body x y)) a1 a2 ...)
```

(the latter being valid only because all variables, particularly *v1*, *v2*, etc., have effectively been uniformly renamed to prevent scoping problems). These are included because they tend to drive the tree toward a semi-canonical form upon which detection of other transformations may depend, simplifying the optimization task.

The number of internal node types is kept small primarily because the number of such trivial semi-canonicalizing transformations tends to grow as the square of the number of node types. The one violation of this principle may be seen in the inclusion of special node types for `go`, `return`, and `progbody` (see Table 2). In the RABBIT compiler [49], definitions in the style of denotational semantics were used to transform `prog` and `go` and `return` into a rather complex melange of lambda-expressions and invocations thereof, from which good code could be generated but only at great expenditure of compile-time analysis. The `go`, `return`, and `progbody` node types were introduced into the S-1 compiler to gain compilation speed. In retrospect, this may have been a mistake. Jonathan Rees [36] has modified an early version of the S-1 LISP compiler to produce code for the DEC VAX [15]. (The language compiled by Rees' compiler is not the same, being more closely related to SCHEME [50] than to MACLISP.) In the process he eliminated the special node types for `go`, `return`, and `progbody`, thereby simplifying all parts of the compiler, and reports no particular problems.

## 6 Generating Good Numerical Code

The high quality of numerical code produced by the S-1 LISP compiler is attributable to three techniques:

1. Register allocation by the TNBIND technique.
2. Representation analysis, to allow different quantities of the same type to have different run-time representations.
3. Conventions and special-case analysis for stack allocation of objects that in the general case must be heap-allocated.

Register allocation is done by the TNBIND technique of BLISS-11 [58, 28]. By "register allocation" we actually mean the compile-time determination of storage locations for all computational quantities, whether such storage allocators be in registers, static memory, stack frames, or the heap. We would use the more general term "storage allocator" or "mmeory allocator" did we not reserve those terms to refer to the run-time heap manager.

### 6.1 Register Allocation

In the TNBIND technique a TN (this term means "temporary name", and refers to a small data structure) is assigned to every computational quantity in the program, both user variables and intermediate results. Each TN is annotated on the basis of the context of its use as to the costs associated with allocating it to one or another kind of storage location (memory, stack slot, register, ...) and the costs associated with maintaining or failing to maintain certain relationships between it and other TN's (for example, two TN's might be forbidden to occupy the same place because their lifetimes overlap, while two others might desirably be allocated to the same place because one is logically copied to the other at some point). After all TN's have been annotated, a global packing process assigns each TN to a specific run-time storage location. Compilation time can be traded for run-time efficiency here by making the packing process more or less clever; for example, a packing method that backtracks can potentially produce better packings than one that does not.

The "2 1/2 address" arithmetic instructions of the S-1 present a peculiar problem in code generation because many (though not all) arithmetic operations must pass through one of the two special registers RTA and RTB. These are sometimes (unofficially and humorously) referred to as the "bottleneck registers", although in practice, given proper register allocation, they are not a performance limitation at all, and indeed RTB is not often needed. Nevertheless, for the best code a clever dance is often needed. Consider this assignment statement:

```
Z[I,K] := A[I,J] * B[J,K] + C[I,K] + D
```

For simplicity suppose the array-subscripts to be zero-origin, and suppose the array dimensions for A to be stored in locations A1 and A2 (and similarly for the other arrays). Then the S-1 code might look like this:

```
MULT RTA,I,A1              ;Prepare subscript for A in RTA
ADD RTA,J
MULT RTB,J,B1              ;Prepare subscript for B in RTB
ADD RTB,K
FMULT RTA,A(RTA),B(RTB)    ;Floating-point multiply to RTA
MULT RTB,I,C1              ;Prepare subscript for C in RTB
ADD RTB,K
FADD RTA,C(RTB)            ;Floating-point add to RTA
MULT RTB,I,Z1              ;Prepare subscript for Z in RTB
ADD RTB,K
FADD Z(RTB),RTA,D          ;Floating-point add to Z[I,K]
```

At each point two RT registers just barely suffice for the job. In some cases, however, they become a bottleneck. The superficially simpler statement

```
Z[I,K] := A[I,J] * B[J,K] + C[I,K]
```

is much more difficult to compile optimally:

```
MULT RTA,I,Z1              ;Prepare subscript for Z in TEMP
ADD TEMP,RTA,K
MULT RTA,I,A1              ;Prepare subscript for A in RTA
ADD RTA,J
MULT RTB,J,B1              ;Prepare subscript for B in RTB
ADD RTB,K
FMULT RTA,A(RTA),B(RTB)    ;Floating-point multiply to RTA
MULT RTB,I,C1              ;Prepare subscript for C in RTB
ADD RTB,K
FADD Z(TEMP),RTA,C(RTB)    ;Floating-point add to Z[I,K]
```

Note that here the subscript for Z cannot be computed at the "obvious" point in the code because there are not enough RT registers to go around. However, computing it ahead allows the subscript computation to dance into RTA and then out again into TEMP. Thus no MOV instructions are required; each instruction performs useful arithmetic.

The RT-register architecture is a very clever way of encoding three operands into two operand specifiers. Nearly all of the time it is *possible* (but not always easy!) to generate code for arithmetic and subscripting expressions that requires no MOV instructions. While the S-1 LISP compiler currently does not always choose the best possible code sequence, it does quite well on typical code, and this is largely attributable to the good performance of the TNBIND method in selecting which TN's should be assigned to RT registers. This compiler represents a confirmation of the ability of the general TNBIND technique to adapt to new and unanticipated difficulties.

### 6.2 Representation Analysis

The second technique for generating good numerical code is representation analysis, to determine for each numerical quantity in the program (a variable or an intermediate temporary) whether it is more efficient to represent it as a true LISP object (a pointer or "access object" [1, 2]) or as a "raw machine number". The distinction may be understood as follows. Suppose that a numeric quantity is to be kept in register 8. If it is stored there in pointer form, it will occupy exactly 36 bits, as all pointers do; register 8 contains a 31-bit address and five tag bits indicating the type of the number. The number itself, whatever its size (and this may be determined from the tag bits), resides in memory at the address indicated by the pointer in register 8. If, on the other hand, it is stored in register 8 as a raw number, then register 8 contains the numeric value itself (and indeed, if it is a double-word number, the value occupies both registers 8 and 9).

The pointer representation is the standard one for LISP data, but raw numbers are more desirable for machine computation. The primary goal is to avoid needless conversion between these two representations. Conversion from a pointer to a raw number requires dereferencing, a simple indirection operation that can often be expressed as an addressing mode (though often a run-time data-type check is also needed). Conversion from a raw number back to pointer format, however, may entail allocation of new storage and consequent garbage-collection overhead, and so this conversion is more to be avoided.

As an example, in the expression `(cons (+ (* a 3) b) 'foo)` the arguments `a` and `b` would most desirably be raw numbers, because that is more convenient for the + and * operators. The result of * is naturally a raw number, and should remain one for use by +. The result of +, however, must become a heap object because `cons` will create another heap object that points to it. The representation analysis for this expression would determine that the result of + must change representations, determine that the result of * should not be changed, and annotate `a` and `b` as to the costs associated with the representation choices for those variables. In this way the "pointer world" of LISP objects and the "number world" that ordinary number-crunching computers are designed to handle best are interfaced at least cost.

(Minor technical point: in S-1 LISP, arithmetic operations such as + are generic, as in most algebraic languages. Unlike most algebraic languages, however, LISP is not strongly typed, and so the compiler cannot necessarily determine the types of all arguments to a generic operator. Presently S-1 LISP provides a set of type-specific operators in much the same way that MACLISP does [44]; thus "+&" indicates addition of 32-bit integers, "`+$f`" and "`+$d`" indicate single-precision and double-precision floating-point addition, and so on. Use of these operators results in rather ugly code. A system of optional type declarations for variables will eventually allow the compiler to make the usual type deductions without requiring every operation to be type-annotated, but this has not yet been implemented. We will use the type-specific operator names in the remainder of this paper.)

The representation analysis is carried out in two passes. The first pass is top-down; every internal tree node is annotated with a *desired* representation, called the WANTREP for the node. The WANTREP for a node is determined by its context within its parent node and by the WANTREP of the parent. As an example, for an `if` expression (`if` *p x y*), the WANTREP for the expression *p* is JUMP; that is, we would prefer that the result of calculating *p* be a conditional jump rather than an actual value. The WANTREP for *x* and for *y* is the same as the WANTREP for the `if` expression. For the expression (`+$f` *x y*), the WANTREP for *x* and for *y* is `SWFLO` (single-word floating-point "raw machine number"). A list of all representations currently used by the compiler appears in Table 3.

The second pass is bottom-up; every internal tree node is annotated with a *deliverable* representation, called the ISREP for the node. This is calculated for the node on the basis of the ISREP information for its descendants and the operation performed by the node itself. For example, the ISREP for the expression (`+$f` *x y*) is always `SWFLO`; the operation is always compiled as a floating-point ADD instruction, and so a "raw machine number" is always produced, to be converted to a pointer only if necessary.

| | |
|---|---|
| `SWFIX` | 36-bit integer. |
| `DWFIX` | 72-bit integer. |
| `HWFLO` | 18-bit floating-point number. |
| `SWFLO` | 36-bit floating-point number. |
| `DWFLO` | 72-bit floating-point number. |
| `TWFLO` | 144-bit floating-point number. |
| `HWCPLX` | 36-bit complex floating-point number. |
| `SWCPLX` | 72-bit complex floating-point number. |
| `DWCPLX` | 144-bit complex floating-point number. |
| `TWCPLX` | 288-bit complex floating-point number. |
| `POINTER` | LISP pointer. |
| `BIT` | 1-bit integer. |
| `JUMP` | Conditional jump. |
| `NONE` | Don't care (value not used). |

Table 3: Internal Object Representations

For an `if` expression (`if` *p x y*) the situation is more complicated. If the WANTREP for the expression is `NONE`, then the ISREP is also `NONE`; no value is required. If *x* and *y* have the same ISREP, then that same ISREP will do for the `if` expression. Suppose, however, that the ISREP of *x* differs from the ISREP of *y*. If the ISREP of, say, *x* matches the WANTREP of the `if` expression, then it would save code on the *x* arm not to have to make a conversion, provided that the ISREP for `y` can be converted to that WANTREP. For example, consider this code:

```
(+$f (if p (sqrt$f q) (car r)) 3.0)
```

The result of `(car r)` is necessarily a pointer, and so the ISREP for that expression is `POINTER`. However, the ISREP for `(sqrt$f q)` is `SWFLO`, as is the WANTREP for the `if` expression because it is an argument to `+$f`. The internal representation `POINTER` can be converted to `SWFLO` at run time (by dereferencing, possibly preceded by a type check on the pointer), and so the ISREP of the `if` expression will be `SWFLO`. Therefore when the conditional succeeds, no conversion of the result of `sqrt$f` will be necessary; when the conditional fails, the result of `car` will merely be dereferenced. This is better than the ultimate default strategy of letting the ISREP of an `if` expression be `POINTER` if the ISREP's of the two arms disagree; that would cause the result of `sqrt$f` to be converted to pointer form and then immediately dereferenced.

The clean top-down/bottom-up nature of the process is spoiled by variables, which introduce loops into the otherwise tree-like representation analysis. To produce the very best analysis in general, solutions must be found to simultaneous equations over the discrete domain of internal types. In practice, a little heuristic guesswork suffices; if not all the references to a variable agree as to what type is desirable for it, the type `POINTER` can always be used.

This process of representation analysis is reminiscent of the resolving of overloaded operators in such languages as ADA. One can think of `+$f` as really being an overloaded operator with eight definitions:

```
(+$f POINTER POINTER) => POINTER
(+$f POINTER SWFLO) => POINTER
(+$f SWFLO POINTER) => POINTER
(+$f SWFLO SWFLO) => POINTER
(+$f POINTER POINTER) => SWFLO
(+$f POINTER SWFLO) => SWFLO
(+$f SWFLO POINTER) => SWFLO
(+$f SWFLO SWFLO) => SWFLO
```

Actually, there need to be many more than eight definitions, because other types such as NONE may be involved. The task of representation analysis is to find particular definitions for each node such that the internal types match at the node interfaces. This differs from ADA overloading resolution, however, in that for ADA the resolution must be unique, lest the user program be ambiguous. For S-1 LISP, however, the choices are among internal representations, and so there may be many resolutions, all of which preserve user program semantics. For the S-1 COMMON LISP compiler, the problem is not to determine uniqueness, but to choose one of least cost from among many permissible solutions.

It was stated above that for register allocation purposes, a TN is associated with every internal tree node to stand for the intermediate value of that node. The full truth is that a node may instead have two TN's (called the WANTTN and ISTN) if the WANTREP and ISREP differ. (Each node could always have two TN's, but to save storage space the same one is used for both purposes if the representations agree.) In this way the storage necessary for both representations at the time of conversion is represented explicitly as TN's to the register allocator. In effect, the compiler is prepared to do a type coercion on *every* intermediate value of the program (even fetches of variables)! Therefore a single TN does mark the boundary between the input of one operation and the output of the next, to represent the storage needed for the intermediate value; it is just that a potential coercion is interposed between every pair of user operations, and so two TN's appear to be needed at such a juncture. (Once again, however, we emphasize that these coercions are internal and do not affect user-level program semantics.)

### 6.3 Pdl Numbers

The third technique for generating good numerical code tries to reduce the cost of conversion from raw number to pointer format. A lifetime analysis of those numerical quantities that must be converted to pointer form determines when stack allocation may be used rather than heap allocation. To provide a uniform procedure interface, all arguments to user functions must be in pointer format; however, the convention is that such pointers may point into the stack and are therefore "unsafe" in a certain sense. Arguments to compiled procedures are guaranteed to be valid during execution of the procedure, but may be invalidated when the procedure exits. Pointers obtained from any other source (composite objects containing pointers, global variables, values returned by procedures, etc.) are guaranteed "safe"; such pointers never point into the stack.

Operations are also classified as "safe" and "unsafe". For example, checking the type of a pointer is safe, as is passing a pointer to a procedure. However, storing a pointer into a global variable or into a heap object (as with `rplaca`) is unsafe.

The rule, then, is that to perform an operation on a pointer *either* the pointer *or* the operation must be safe. Before an unsafe operation is performed on a potentially unsafe pointer, the pointer must be "certified", either by determining at run time that the pointer is safe (does not point into the stack) or, if that fails, by copying the stack-allocated object into the heap. (The semantics of the user-level language are carefully defined to permit such copying. The operation `eq` is not guaranteed to work on numbers. Another predicate, `eql`, does "work" as an object identity predicate for all objects, because it compares addresses only for non-numeric objects, and compares values for numeric objects.)

This technique was first used in the MACLISP compiler [44]. However, the lifetime analysis in the MACLISP compiler is rather complex and *ad hoc*, and still contains a few known but hard-to-fix bugs. The S-1 LISP compiler confines the analysis to a separate simple phase (three pages of code) whose results are then thrown into TNBIND, which does the rest of the work (allocating the necessary stack slots) automatically.

In more detail, the pdl number annotation phase performs a two-pass top-down/bottom-up analysis of the tree (though it is implemented as a single pass in the form of an "outorder" tree walk, a combination of preorder and postorder). For each node two flags are calculated.

The PDLOKP flag indicates whether the node's parent is willing to accept a pdl number (unsafe pointer) as the result of this node. For example, in the context `(+$f x y)`, the node for *x* is permitted to produce a pdl number; this is no problem because `+$f` is a safe operation. On the other hand, in the context `(rplaca x y)`, *y* may not produce a pdl number because `rplaca` is an unsafe operation.

Actually, the PDLOKP property is more than a flag. If not false, it points to the node that originally authorized the use of a pdl number. For example, in `(atan (if p x y) 3.0)`, *x* has a non-false PDLOKP property that points to the `atan` node, not the `if` node. This indicates that the lifetime of the pdl number must extend at least until execution of the `atan` node. (The processing of an `if` node simply passes the PDLOKP authorization of its parent down to the two arms of the conditional. On the other hand, it always of itself authorizes the predicate computation to produce a pdl number, because the conditional test performed by `if` is a safe operation.)

The PDLNUMP flag indicates whether the node itself might be inclined to produce a pdl number. For example, the result of `(+$f x y)` might well be a pdl number if a pointer result is required. On the other hand, the result of `(car x)` is never a pdl number.

The TNBIND phase was then modified to attach an extra TN to a node when all of the following conditions hold:

- The PDLOKP flag is true.
- The PDLNUMP flag is true.
- The WANTREP for the node is `POINTER`.
- The ISREP for the node is one of `SWFLO`, `DWFLO`, `TWFLO`, `HWCPLX`, `SWCPLX`, `DWCPLX`, or `TWCPLX`. (These are the internal numeric types that have corresponding user-visible, heap-allocated pointer representations.)

This TN represents a piece of storage of the correct size to hold the internal numeric quantity. This TN is further annotated in the following ways:

- It must be allocated to the scratch (non-pointer) region of the stack, not to a register.
- It is desirable for the result of the node's calculation to be delivered directly to this stack location if possible.

- The lifetime of the TN must extend at least as far as the lifetime of the program node in the PDLOKP slot of this node (this is the node that originally authorized creation of a pdl number).

All this takes only a dozen extra lines of code in TNBIND. The TN packing process then allocates the necessary stack locations for stack allocation of pdl numbers.

What is a rather complex process in the MACLISP compiler is therefore relatively simple in the S-1 LISP compiler because it leans heavily on the already-existing internal representation analysis and TNBIND phases. All that remains is a few routines in the code generator: pointer certification checks must be inserted at critical points, and the code-generation routine that handles implicit internal type coercions must check for the presence of a pdl number TN to decide whether to stack-allocate or heap-allocate when converting from a raw number to a pointer.

## 7 An Example

Here we give a complete example of the compilation of a simple function:

```
(defun testfn (a &optional (b 3.0) (c a))
  (let ((d (+$f a b c))
        (e (*$f a b c)))
    (let ((q (sin$f e)))
      (frotz d e (max$f d e))
      q)))
```

This function was designed to illustrate these points:

- Handling of optional arguments. If `testfn` is called with three arguments, then they provide values for parameters a, b, and c as usual. If only two arguments are provided, then parameter c defaults to the value of a, that is, the first argument. If only one argument is provided, then b defaults to `3.0` and c defaults to the value of a.
- Source-level transformations, particularly substitution of expressions for variables.
- Representation analysis.
- Pdl number generation.

The following output is taken directly from the compiler's debugging transcript file. The program fragments that appear are the result of back-translating the internal tree structure.

First the compiler expands the `let` forms and introduces `progn` explicitly where necessary:

```
(lambda (a &optional (b 3.0) (c a))
  ((lambda (d e)
     ((lambda (q)
        (progn (frotz d e (max$f d e)) q))
      (sinc$f (*$f e 0.159154942))))
   (+$f a b c)
   (*$f a b c)))
```

Another transformation is from `sin$f` (the sine function with argument in radians) to `sin$c` (the sine function with argument in cycles). This transformation is machine-independent but machine-inspired: the S-1 `SIN` instruction assumes its argument to be in cycles. The conversion factor is a floating-point approximation to $1/2\pi$.

The optimizer then performs the following transformations:

```
;**** Optimizing this form:
(+$f a b c)
;*** to be this form:
(+$f (+$f c b) a)
;**** courtesy of META-EVALUATE-ASSOC-COMMUT-CALL

;**** Optimizing this form:
(*$f a b c)
;*** to be this form:
(*$f (*$f c b) a)
;**** courtesy of META-EVALUATE-ASSOC-COMMUT-CALL
```

Most associative operations with more than two arguments are reduced to compositions of two-argument calls. This all is machine-independent but helpful to the code generator. This transformation is completely table-driven. If the operation is also commutative, the arguments may be rearranged, primarily to get constants to the head of the argument list to promote compile-time expression evaluation. (The user-level semantics for such operators explicitly permits such re-association *within a single argument list*, even though floating-point operations are not truly associative. If such deviations concern the user, he may prescribe the association explicitly by writing an appropriate composition of two-argument calls himself, and the compiler is not permitted to re-associate such compositions.)

```
;**** Optimizing this form:
(*$f e 0.159154942)
;*** to be this form:
(*$f 0.159154942 e)
;**** courtesy of CONSIDER-REVERSING-ARGUMENTS
```

By convention constant arguments are put first where possible.

```
;**** 1 substitution for the variable  q  by
(sinc$f (*$f 0.159154942 e))
;**** courtesy of META-SUBSTITUTE
```

Although the (presumably user-defined) function `frotz` may produce side effects, it cannot affect the variable e because e is lexically scoped, nor can it affect the operations `*$f` and `sinc$f`, which are known to the compiler to be immutable mathematical functions. Therefore motion of this code past the call to `frotz` is permissible.

```
;**** Optimizing this form:
((lambda ()
   (progn (frotz d e (max$f d e))
          (sinc$f (*$f 0.159154942 e)))))
;*** to be this form:
(progn (frotz d e (max$f d e))
       (sinc$f (*$f 0.159154942 e)))
;**** courtesy of META-CALL-LAMBDA
```

Once the defining expression for q had been substituted for all occurrences, the variable q could be eliminated and the program simplified. The resulting program is:

```
(lambda (a &optional (b 3.0) (c a))
  ((lambda (d e)
     (progn (frotz d e (max$f d e))
            (sinc$f (*$f 0.159154942 e))))
   (+$f (+$f c b) a)
   (*$f (*$f c b) a)))
```

At this point the compiler offers to print several pages of information about how it performed the register allocation, but we omit this here!

```
        ((JMPZ NEQ Q) RTA-Q0 ( (REF SQ *:SQ-WRONG-TYPE-OF-FUNCTION)))
                              ;Jump if other than single pointer value desired.
        ((JMPZ GTR) (? -3 RTA) (SQ *:SQ-WRONG-NUMBER-OF-ARGUMENTS))
                                                  ;Jump if more than 3 arguments.
        (MOV R2 (REF (DATA 0 (- L0024 L0025) (- L0022 L0025)
                             (- L0020 L0025) (- L0018 L0025)) RTA S))
                                                  ;Dispatch on number of arguments.
L0025   (JMPA () (REF PC 0 R2 Q))
L0024   (JMPA () (SQ *:SQ-WRONG-NUMBER-OF-ARGUMENTS))   ;Wrong number of arguments.
L0022   ;; Come here if one argument was supplied.
        ((PUSH UP D) SP (SQ *:SQ-NIL))            ;Push slots for parameters B and C
        ((ALLOC 1) (? 0) (SP 4))                  ;Allocate 1 word of pointer memory
        ((ALLOC 1) (? (POINTER *:DTP-GC 12)) (SP 16))   ;Allocate 3 words scratch memory
        ((MOVP P A) TP (SP 108))                  ;Set up TP to point to temporaries
L0023   ;; Calculate default value for parameter 1 [B].
        (MOV (FP -100) (QUOTE 3.0))
        (JMPA () L0021)
L0020   ;; Come here if two arguments were supplied.
        ((PUSH UP) SP (SQ *:SQ-NIL))              ;Push slot for parameter C
        ((ALLOC 1) (? 0) (SP 4))                  ;Allocate 1 word of pointer memory
        ((ALLOC 1) (? (POINTER *:DTP-GC 12)) (SP 16))   ;Allocate 3 words scratch memory
        ((MOVP P A) TP (SP 108))                  ;Set up TP to point to temporaries
L0021   ;; Calculate default value for parameter 2 [C].
        (MOV (FP -96) (FP -104))
        (JMPA () L0019)
L0018   ;; Come here if three arguments were supplied.
        ((ALLOC 1) (? 0) (SP 4))                  ;Allocate 1 word of pointer memory
        ((ALLOC 1) (? (POINTER *:DTP-GC 12)) (SP 16))   ;Allocate 3 words scratch memory
        ((MOVP P A) TP (SP 108))                  ;Set up TP to point to temporaries
L0019   ((FADD S) RTA (REF (FP -96) 0) (REF (FP -100) 0))       ;(+$F C B)
        ((FADD S) RTA RTA (REF (FP -104) 0))                    ;(+$F (+$F C B) A)
        (MOV (REF TP -112) RTA)          ;Install value for PDL-allocated number.
        ((MOVP *:DTP-SINGLE-FLONUM A) A (REF TP -112))          ;Pointer to PDL slot.
        ((FMULT S) RTA (REF (FP -96) 0) (REF (FP -100) 0))      ;(*$F C B)
        ((FMULT S) RTA RTA (REF (FP -104) 0))                   ;(*$F (*$F C B) A)
        (MOV (REF TP -116) RTA)          ;Install value for PDL-allocated number.
        ((MOVP *:DTP-SINGLE-FLONUM A) (TP -128) (REF TP -116))  ;Pointer to PDL slot.
        (%SETUP FROTZ)                            ;Set up to call FROTZ
        ((PUSH UP) SP A)
        ((PUSH UP) SP (TP -128))
        ((FMAX S) RTA (A 0) (REF (TP -128) 0))                  ;(MAX$F D E)
        (MOV (REF TP -120) RTA)          ;Install value for PDL-allocated number.
        ((MOVP *:DTP-SINGLE-FLONUM A) A (REF TP -120))          ;Pointer to PDL slot.
        ((PUSH UP) SP A)
        (%CALL 0 3)                               ;Call for (FROTZ D E (MAX$F D E))
        ((MOVP P A) SP (SP -4))                   ;Throw away returned value.
        ;; 17088374285. = #o177242763015 = singleword 0.159154942
        ((FMULT S) RTA (? 17088374285) (REF (TP -128) 0))       ;(*$F 0.159154942 E)
        ((FSIN S) I RTA)                                        ;(SINC$F (*$F 0.159154942 E))
        (JSP T2 (@ (REF SQ *:SQ-SINGLE-FLONUM-CONS)))  ;Generate new number object.
        (MOV (@ (SP -4)) I)                       ;Install value in new number object.
        ((POP UP) (FP *:FRAME-RETURN-VALUE) SP)
        ((MOV D D) T1 CP)                         ;Function exit:
        ((MOVMS 3) TP (FP *:FRAME-OLD-TP))        ; restore TP, CP, and FP for caller,
        ((RETSR T2 (T2 *:FRAME-OLD-TP))           ; and return.
```

**Table 4:** Code Produced by S-1 LISP Compiler

The code generated by the compiler (in "parenthesized assembly language") is shown in Table 4. The comments on the code are generated by the compiler (as might be inferred from their repetitious and stilted style).

On entry to the function register RTA contains some procedure interface information, including the number of arguments passed, for error-checking purposes. The first two instructions check this information. The third instruction performs a four-way dispatch on whether there were zero, one, two, or three arguments. Zero arguments is an error. For each of the other cases there is code customized to the number of arguments to set up the stack frame and initialize parameters for which no arguments were passed. This code is rather bulky, but high speed is a relatively more important goal for the S-1 LISP compiler than compact code. It might appear that the frame-allocation code does not need to be replicated for each case, and that is true in this example; but it must be replicated in general, because the initialization for an optional parameter may be any LISP computation whatsoever, and such computations may require a properly set up frame to run in. The compiler could reduce the code size in special cases, but that improves only code size, not

speed, and so we consider that optimization to be of relatively low priority.

In interpreting the code, keep in mind that register SP is the stack pointer, FP the frame pointer (relative to which arguments may be accessed), and TP the temporaries pointer (relative to which local quantities may be accessed). (These respectively correspond roughly but not precisely to the registers named SP, AP, and FP in the DEC VAX architecture [15].)

The meat of the code begins at label L0019. Two floating-point additions are performed and the result saved in a stack slot. (This should have been accomplished in two instructions, for the MOV is unnecessary; more fine-tuning of the register allocator with respect to the TN's for pdl numbers is needed.) A pointer to the stack slot (a pdl number pointer) becomes the value for variable d, which is saved in register A. (The MOVP instruction creates a pointer to its second operand, installing the indicated type (*:DTP-SINGLE-FLONUM) in the tag field.) In a similar manner two multiplications are then performed and the result made into a pdl number. The pointer then becomes the value of variable e, which has been assigned a location on the stack rather than in a register like d. The reason for this is that TNBIND determined that e must survive the call to frotz, while d need not; calls to other procedures by convention may destroy nearly all registers.

An argument frame is then set up for calling frotz. The pointer in register A for variable d is pushed; the pointer in location -128(TP) for variable e is pushed; then the max$f operation is performed (note the indirections in the operands to the FMAX instruction), the result made into a pdl number, and that pointer pushed. Finally frotz is called. By convention, its result is returned as a pointer on top of the stack; the MOVP instruction discards the value.

Next the value of e is multiplied by a constant (notice the comment by the compiler concerning its value) and the FSIN instruction used to calculate the sine function. A storage allocation routine is called to heap-allocate a new floating-point number (returning a value from a procedure is not a "safe" operation, so a pdl number may not be used), and the value is installed in it. Finally this object is popped into a place where the caller can find it, and a three-instruction ritual exits the procedure.

## 8 Conclusions

An effective, production-quality compiler for LISP is being constructed for the S-1 computer. For optimization and generation of high-quality code it depends primarily on three techniques:

- Source-level program transformations primarily based on the lambda-calculus. In effect, denotational-semantic style definitions of program constructs can be used directly and manipulated by the compiler.
- Register allocation techniques developed for the BLISS-11 compiler. The TNBIND method appears to provide good register allocation with a minimum of difficulty, even when the S-1's strange RT registers are involved.
- Stack-allocation of normally heap-allocated objects, primarily numbers, as developed for MACLISP. Given a general, global register allocation technique such as TNBIND to lean on, and analysis of feasible alternatves for internal representations, good numerical code can be generated that interfaces cleanly to the "pointer world" of LISP.

We believe that with additional tuning the S-1 LISP compiler will be competitive with the S-1 PASCAL and FORTRAN compilers for numerical applications, as well as providing the support for symbolic algorithms traditionally associated with LISP. While LISP is unlikely to displace these other languages completely, we feel that this LISP implementation will be valuable for certain artificial intelligence applications, such as speech recognition and visual scene analysis, that presently require a mixture of symbolic heuristic calculations and intense numerical crunching.

## References


1. United States Department of Defense. "Preliminary ADA Reference Manual." *SIGPLAN Notices* 14, 6A (June 1979).

2. *Reference Manual for the ADA Programming Language Proposed Standard Document.* United States Department of Defense (1980).

3. Allen, Frances E., and Cocke, John. "A Catalogue of Optimizing Transformations." In *Design and Optimization of Compilers*, Rustin, Randall (Ed.),Prentice-Hall (Englewood Cliffs, N.J., 1972), 1-30.

4. Arsac, Jacques J. "Syntactic Source to Source Transforms and Program Manipulation." *Comm. ACM* 22, 1 (Jan. 1979), 43-54.

5. Baker, Henry B., Jr. "Shallow Binding in LISP 1.5." *Comm. ACM* 21, 7 (July 1978), 565-569.

6. Bobrow, Daniel G. and Wegbreit, Ben. "A Model and Stack Implementation of Multiple Environments." *Comm. ACM* 16, 10 (Oct. 1973), 591-603.

7. Burstall, R.M., and Darlington, John. "A Transformation System for Developing Recursive Programs." *J. ACM* 24, 1 (Jan. 1977), 44-67.

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

9. Church, Alonzo. *Annals of Mathematics Studies.* Volume 6: *The Calculi of Lambda Conversion.* Princeton University Press (Princeton, 1941). Reprinted by Klaus Reprint Corp. (New York, 1965).

10. Coonen, Jerome T. "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic." *Computer* 13, 1 (Jan. 1980), 68-79. Errata for this paper appeared as [11].

11. Coonen, Jerome T. "Errata for 'An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic'." *Computer* 14, 3 (March 1981), 62. These are errata for [10].

12. Correll, Steven. "S-1 Uniprocessor Architecture (SMA-4)." In *The S-1 Project 1979 Annual Report*, Lawrence Livermore Laboratory (Livermore, California, 1979), Chapter 4.

13. *PDP-11 Handbook.* Digital Equipment Corporation (Maynard, Massachusetts, 1969).

14. *DecSystem 10 Assembly Language Handbook (third edition).* Digital Equipment Corporation (Maynard, Massachusetts, 1973).

15. *VAX Architecture Handbook.* Digital Equipment Corporation (Maynard, Massachusetts, 1981).

16. Falkoff, A.D., and Orth, D.L. "Development of an APL Standard." *APL 79 Conference Proceedings.* ACM SIGPLAN/STAPL (Rochester, New York, June 1979), 409-453. Proceedings published as *APL Quote Quad* 9, 4 (June 1979).

17. Fateman, Richard J. "Reply to an Editorial." *ACM SIGSAM Bulletin* 25 (March 1973), 9-11.

18. Ford, Gary, and Hansche, Brian. "Optional, Repeatable, and Varying Type Parameters." *SIGPLAN Notices* 17, 2 (Feb. 1982), 41-48.

19. Galley, S.W. and Pfister, Greg. *The MDL Language.* Programming Technology Division Document SYS.11.01, MIT Project MAC (Cambridge, Massachusetts, Nov. 1975).

20. Geschke, Charles M. *Global Program Optimizations.* Ph.D. Th., Carnegie-Mellon University (Pittsburgh, Oct. 1972).

21. Goldberg, Adele, and Kay, Alan. *Smalltalk-72 Instruction Manual.* Learning Research Group, Xerox Palo Alto Research Center (Palo Alto, California, March 1976).

22. Harrison, William. "A New Strategy for Code Generation: The General Purpose Optimizing Compiler." *Proceedings of the Fourth Symposium on Principles of Programming Languages.* Association for Computing Machinery (Los Angeles, Jan. 1977), 29-37.

23. Hewitt, Carl. "Viewing Control Structures as Patterns of Passing Messages." *Artificial Intelligence* 8, 3 (June 1977), 323-364. A comment on this paper appeared as [24].

24. Hewitt, Carl. "Comments on 'Viewing Control Structures as Patterns of Passing Messages'." *Artificial Intelligence* 10, 3 (Nov. 1978), 317-318. This is a comment on [23].

25. *APL\360 User's Manual.* International Business Machines Corporation (1968).

26. IEEE Computer Society Standard Committee, Microprocessor Standards Subcommittee, Floating-Point Working Group. "A Proposed Standard for Binary Floating-Point Arithmetic." *Computer* 14, 3 (March 1981), 51-62.

27. Jensen, Kathleen, and Wirth, Niklaus. *Pascal User Manual and Report.* Springer-Verlag (New York, 1974).

28. Johnsson, Richard Karl. *An Approach to Global Register Allocation.* Ph.D. Th., Carnegie-Mellon University (Pittsburgh, Dec. 1975).

29. Leverett, Bruce W.; Cattell, Roderic G.G.; Hobbs, Steven O.; Newcomer, Joseph M.; Reiner, Andrew H.; Schatz, Bruce R.; and Wulf, William A. *An Overview of the Production Quality Compiler-Compiler Project.* Tech. Rept. CMU-CSD-79-105, Carnegie-Mellon University Computer Science Department (Pittsburgh, Feb. 1979).

30. Levine, Ronald D. "Supercomputers." *Scientific American* 246, 1 (Jan. 1982), 118-135.

31. The Mathlab Group. *MACSYMA Reference Manual (Version Nine).* MIT Lab. for Computer Science (Cambridge, Massachusetts, 1977).

32. Moon, David. *MacLISP Reference Manual, Revision 0.* M.I.T. Project MAC (Cambridge, Massachusetts, April 1974).

33. Moses, Joel. *The Function of FUNCTION in LISP.* AI Memo 199, MIT Artificial Intelligence Lab. (Cambridge, Massachusetts, June 1970).

34. Naur, Peter (ed.), et al. "Revised Report on the Algorithmic Language ALGOL 60." *Comm. ACM* 6, 1 (Jan. 1963), 1-20.

35. Organick, Elliot I.. *The Multics System: An Examination of Its Structure.* MIT Press (Cambridge, Massachusetts, 1972).

36. Rees, Jonathan. Private communication.

37. Sammet, Jean E.. *Programming Languages: History and Fundamentals.* Prentice-Hall (Englewood Cliffs, New Jersey, 1969).

38. Schatz, Bruce R. *Algorithms for Optimizing Transformations in a General Purpose Compiler: Propagation and Renaming.* Tech. Rept. RC 6232 (#26773), IBM Thomas J. Watson Research Center (Yorktown Heights, New York, Oct. 1976).

39. Schroeder, Michael D., and Saltzer, Jerome H. "A Hardware Architecture for Implementing Protection Rings." *Comm. ACM* 15, 3 (March 1972), 157-170.

40. Standish, T.A.; Harriman, D.C.; Kibler, D.F.; and Neighbors, J.M. *The Irvine Program Transformation Catalogue.* University of California (Irvine, California, Jan. 1976).

41. Standish, Thomas A.; Kibler, Dennis F.; and Neighbors, James M. "Improving and Refining Programs by Program Manipulation." *Proceedings of the ACM National Conference.* Association for Computing Machinery (Houston, Texas, Oct. 1976), 509-516.

42. Steele, Guy Lewis Jr. *LAMBDA: The Ultimate Declarative.* AI Memo 379, MIT Artificial Intelligence Lab. (Cambridge, Massachusetts, Nov. 1976).

43. Steele, Guy Lewis Jr., and Sussman, Gerald Jay. *LAMBDA: The Ultimate Imperative.* AI Memo 353, MIT Artificial Intelligence Lab. (Cambridge, Massachusetts, March 1976).

44. Steele, Guy Lewis Jr. "Fast Arithmetic in MacLISP." *Proceedings of the 1977 MACSYMA Users' Conference.* NASA Scientific and Technical Information Office (Washington, D.C., July 1977), 215-224. Also published as [48].

45. Steele, Guy Lewis Jr. *Compiler Optimization Based on Viewing LAMBDA as Rename plus Goto.* Master Th., MIT (May 1977). Published as [49].

46. Steele, Guy Lewis Jr. "Debunking the 'Expensive Procedure Call' Myth; or, Procedure Call Implementations Considered Harmful; or, LAMBDA: The Ultimate GOTO." *Proceedings of the ACM National Conference.* Association for Computing Machinery (Seattle, Oct. 1977), 153-162. Revised version published as [47].

47. Steele, Guy Lewis Jr. *Debunking the 'Expensive Procedure Call' Myth; or, Procedure Call Implementations Considered Harmful; or, LAMBDA: The Ultimate GOTO.* AI Memo 443, MIT Artificial Intelligence Lab. (Cambridge, Massachusetts, Oct. 1977). This is a revised version of [46].

48. Steele, Guy Lewis Jr. *Fast Arithmetic in MacLISP.* AI Memo 421, MIT Artificial Intelligence Lab. (Cambridge, Massachusetts, Sept. 1977). Also appeared as [44].

49. Steele, Guy Lewis Jr. *RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization).* Tech. Rept. 474, MIT Artificial Intelligence Lab. ( May 1978). This is a revised version of the author's master's thesis [45].

50. Steele, Guy Lewis Jr., and Sussman, Gerald Jay. *The Revised Report on SCHEME: A Dialect of LISP.* AI Memo 452, MIT Artificial Intelligence Lab. (Cambridge, Massachusetts, Jan. 1978).

51. Sussman, Gerald Jay, and Steele, Guy Lewis Jr. *SCHEME: An Interpreter for Extended Lambda Calculus.* Tech. Rept. 349, MIT Artificial Intelligence Lab. (Cambridge, Massachusetts, Dec. 1975).

52. Teitelman, Warren, et al. *InterLISP Reference Manual.* Xerox Palo Alto Research Center (Palo Alto, California, 1975). Second revision.

53. Teitelman, Warren, et al. *InterLISP Reference Manual.* Xerox Palo Alto Research Center (Palo Alto, California, 1978). Third revision.

54. van Wijngaarden, A.; Mailloux, B.J.; Peck, J.E.L.; Koster, C.H.A.; Sintzoff, M.; Lindsey, C.H.; Merrtens, L.G.L.T.; and Fisker, R.G. (eds.). "Revised Report on the Algorithmic Language ALGOL 68." *SIGPLAN Notices* 12, 5 (May 1977), 1-70.

55. Wegbreit, Ben; Holloway, Glenn; Spitzen, Jay; and Townley, Judy. *ECL Programmer's Manual.* Tech. Rept. 23-74, Harvard University Center for Research in Computing Technology (Cambridge, Massachusetts, Dec. 1974).

56. Weinreb, Daniel, and Moon, David. *LISP Machine Manual, Fourth Edition.* MIT Artificial Intelligence Lab. (Cambridge, Massachusetts, July 1981).

57. White, Jon L. "LISP: Program is Data: A Historical Perspective on MacLISP." *Proceedings of the 1977 MACSYMA Users' Conference.* NASA Scientific and Technical Information Office (Washington, D.C., July 1977), 181-189.

58. Wulf, William; Johnsson, Richard K.; Weinstock, Charles B.; Hobbs, Steven O.; and Geschke, Charles M.. *Programming Language Series.* Volume 2: *The Design of an Optimizing Compiler.* American Elsevier (New York, 1975).