# T: A Dialect of Lisp
# or, LAMBDA: The Ultimate Software Tool

Jonathan A. Rees
Norman I. Adams IV
Department of Computer Science
Yale University
New Haven, Connecticut

## Abstract

The T project is an experiment in language design and implementation. Its purpose is to test the thesis developed by Steele and Sussman in their series of papers about the Scheme language: that Scheme may be used as the basis for a practical programming language of exceptional expressive power; and, that implementations of Scheme could perform better than other Lisp systems, and competitively with implementations of programming languages, such as C and Bliss, which are usually considered to be inherently more efficient than Lisp on conventional machine architectures. We are developing a portable implementation of T, currently targeted for the VAX under the Unix and VMS operating systems and for the Apollo, a MC68000-based workstation.

## 1. Motivation

During the spring of 1981 the Yale Computer Science Department was faced with the problem of deciding how it would compute during the next several years. Cheap and powerful large address space machines existed, but language, implementation, or availability problems made the candidate Lisp systems unattractive. Instead of waiting for some other university to fill this need, the department decided to take advantage of the authors' eagerness to implement a production quality Lisp. The resulting project is a relentless compromise (or more optimistically speaking, a successful marriage) between the practical necessity of making a working system available soon for people to use, and the implementors' idealism. The opportunity was before us to determine just what kind of language we really wanted, and to proceed to implement that.

Rather than transport any existing Lisp system to the machines we were interested in, or even using an existing language specification (such as NIL) as a foundation on which to build, we decided to design a language and implementation from the ground up. This section attempts to summarize the considerations which led to this decision and which helped shape our design.



### 1.1. Lisp

It almost goes without saying that the basic Lisp paradigm seemed a very good place to start. Lisps in general provide the following desirable features:

- *Program as data.* A simple, standard internal representation is used to represent source programs.
- *General-purpose syntax.* Lisp provides a standard external syntax for representing programs and data which is unbiased with respect to application.
- *Incremental procedure redefinition.* The fact that edits to source code are reflected immediately in the running environment greatly speeds debugging.
- *Powerful debugging tools.* Lisp systems usually supply integrated tools to aid interactive debugging.
- *Storage management.* A garbage collector relieves the programmer of the chore of storage management, thus promoting conciseness and readability and eliminating many sources of error.
- *Typed data.* Lisp is dynamically typed rather than statically typed. This promotes generality and abstractness and helps reduce the "noise level" in programs by eliminating the need for declarations.
- *Not a police state.* Lisp systems provide the user with complete control over the system and general access to low-level representations.

Lisp's use of symbolic list structure to represent programs internally really has no parallel in other languages. The provision of a standard internal representation for source code is responsible for the ease with which evaluators, compilers, translators, and other program synthesizers and analyzers can be written. Lisp's macro facility provides a powerful means by which the language can be extended and new notations and abstractions developed. Their generality, far exceeding that of macro facilities in languages like Bliss and PL/1, owes to the fact that the macro expansion routines are written in Lisp itself, and that source code is represented in a standard way.

Related to the use of standard internal representation is the use of a standard external representation. Lisp's "Cambridge Polish" notation is unbiased with respect to its interpretation. It sacrifices conciseness in certain special cases for generality, uniformity, and simplicity. One need not worry about low-level syntax issues when dealing with extensions and imbedded languages. The computer science literature's obsession with lexical and syntactic analysis becomes a mystery to those familiar with Lisp.

Incremental redefinition effectively eliminates the compile and load phases in the traditional edit/compile/load/test debugging cycle; this is absolutely essential when dealing with very large programs, where the time required to recompile a module (and its clients), invoke a linker, start a process, initialize, and get into a state where one can proceed to locate the next bug in the program, would make any other approach absurd.

Lisps have traditionally paid close attention to interactive debugging, providing language-level single-stepping, call tracing, breakpoints, control stack and environment inspection, data structure inspectors and editors, and error recovery. These tools have usually not been wired into the system but rather have been built by user communities on top of general underlying primitives.

Dynamic typing aids in modularity and orthogonality; routines can be trivially defined which work on any kind of data, or do generic dispatches in a variety of ways. Type declarations are optional. This enhances program conciseness and readability. When included, type declarations provide a compiler with information that can be used to improve code quality. In an interpreter, they can be used for consistency checking.

Programmers are free to manipulate the system in potentially dangerous ways, if they need to be able to do so. Language designers and implementors should not presume that a programmer will never need to do something "dangerous" or "unportable". Security and consistency checks should exist as an aid to the programmer, not as a hindrance (as in Pascal, for example). [Kernighan 81]

## 1.2. Modern Lisps

In addition to these standard Lisp features, a number of other important issues have been addressed in various production Lisp implementations. More and more is becoming expected of modern production-quality Lisps and their implementations. [Teitelman 78, Weinreb 81] Some of the issues we consider important are outlined below.

Maclisp's popularity owes much to the quality of its compiler's object code and the years of effort devoted to system tuning. [Moon 74, Steele 77a, Steele 77b] The belief that Lisp is inherently inefficient is still prevalent, though recent work has begun to clear up that misconception. Some recent large-system Lisp efforts have argued that the amount of effort required to produce effective compilers for conventional-architecture machines, and the potential for complexity and error, are so great that the best approach is to develop special-purpose processors better suited to the task of running Lisp programs. [Bawden 77] While there is much to be said for that, the authors believe that the verdict is not yet in on the need for hardware support.

As Lisp's popularity grows and its use in developing useful, production programs increases, portability becomes important. However, a perhaps stronger argument for portability is the need to take advantage of new technology and architectures as they become available. Franz Lisp, Portable Standard Lisp, Interlisp, and Common Lisp have each addressed portability issues in different ways. [Steele 81, Foderaro 80, Griss 82, Moore 79]

Object-oriented languages have been implemented often in Lisp, but the Lisp Machine was the first production Lisp of which the authors are aware which made the message-passing idiom an integral part of the system. [Weinreb 81] The Lisp Machine "flavor" system has proved to work quite successfully as a tool to achieve modularity and is a standard and popular part of the programmer's repertoire.

Many Lisps implement some form of functional closure, though they are often neither completely general nor well integerated.

Modern Lisps provide general CATCH and THROW constructs for performing non-local exits. Another related feature is sometimes called UNWIND-PROTECT: the ability to force some sort of clean-up to happen even if during the computation of some expression there is a THROW out of it. For example, a routine which manipulates an open file would like to make sure the file gets closed even if the routine is aborted or otherwise exits abnormally. UNWIND-PROTECT can be thought of as a generalization of shallow dynamic binding, where variables which have been "temporarily" assigned values are always reset to their previous values even if the form which binds them exits abnormally. Coroutines and multitasking are also desirable features, particularly where the machine's operating system is Lisp itself.

As the size of a system grows the potential for naming conflicts increases. When one tries to run several large programs in a single Lisp environment, as on the Lisp Machine where the editor, compiler, "operating system," and different application programs all run in the same address space, using the traditional flat, global namespace is out of the question; multiple namespaces are necessary.

In order to develop correct and compilable programs, it is important that the compiler and interpreter implement the same language. Usually, Lisps have been dynamically scoped at heart, but when code is compiled, bound variables not declared to be "special" become lexically scoped. This use of lexical scoping has been considered a concession to efficient implementation, not a language feature. When a program runs well enough that a user thinks about trying to compile it, the user usually must make a pass over the program, adding declarations and even changing code and file organization, in order to make it compilable. Some programs (especially those making heavy use of FEXPR's) may never run compiled, while others (especially those making heavy use of functional arguments) will not run interpreted. In Lisp Machine Lisp some important language features are implemented only in compiled code, not in interpreted code.

One approach to this problem is to sacrifice efficiency by forcing the compiler to implement the same dynamic binding semantics as the interpreter. To the authors' knowledge, MIT's New Implementation of Lisp (NIL) is the only production Lisp system which endorses lexical scoping and employs an interpreter which correctly handles lexical binding as well as "special" declarations.

### 1.3. Scheme

Scheme is a lexically scoped variant of Lisp which has had broad influence in Lisp circles, in spite of the fact that no production implementations are generally available. [Steele 78a] Scheme is appealing for a number of reasons.

Scheme is a small language. It is easy to learn and straitforward to implement. In Scheme, a few powerful concepts combine well to provide a rich and expressive environment.

Lexical scoping is not merely a compromise to make the language compilable, as some dynamic binding partisans will attest. It aids the programmer as well as the compiler. The case for lexical binding is presented eloquently throughout Steele and Sussman's work, especially in [Steele 78b]. Essentially their point is that dynamic binding violates *referential transparency*, the requirement that "the meanings of parts of a program be apparent and not change, so that such meanings can be reliably depended upon." Because of its global effect on the meaning of names, dynamic binding can be an obstacle to the development of modular programs. Note that the choice between lexical and dynamic binding does not need to be either/or; in Scheme, lexical binding is the default, but one obtains dynamic binding where its is explicitly requested. Each discipline has its own justifications in different cases, and both are necessary to writing modular programs.

In Scheme, procedures are "first-class citizens"; they may be stored in data structures and returned as the value of other procedures. Procedures are created by evaluating LAMBDA-expressions, which are closed in the current lexical environment; that is, free variables obtain their values from the environment in effect at the time the LAMBDA-expression is evaluated. This simple and obvious feature not only has remarkably broad applicability, but can be implemented efficiently. Lexical closures can be used to implement lazy evaluation, procedurally defined data, and complex control structures. They facilitate the use of combinators and functional programming style. [Steele 76a, Steele 76b, Steele 77c]

The general consensus in the Lisp community appears to be that FEXPR's and NLAMBDA's are harmful and unnecessary, and should be removed from the language. Their use precludes structural analysis of programs by compilers, people, and other program understanders. [Pitman 80, Steele 82] Macros serve the notational and modularity needs that FEXPR's attempt to address, often more elegantly and correctly. This idea was probably first articulated by Steele and Sussman, and has since found its way into NIL and Common Lisp. [Steele 81, White 79]

## 2. Language

This section outlines the way in which the T language presents itself to users.

T is based on Scheme, and as such its most prominent characteristics are lexical scoping and full closures. (But see the discussion of dynamic binding, below.) Unlike closures in dynamically scoped Lisps, lexical closures may be efficiently implemented, and their existence incurs no overhead where they are not used.

T centers around a small core language, free of complicated features, thus easy to learn. We have sought to replace features which introduce what we considered to be undue semantic or implementation difficulties with acceptable alternatives. Where this was not possible, we have refrained from supporting features that we didn't feel completely right about. T's omissions are important: we have avoided the complicated argument list syntax of Common Lisp, keyword options, and multiple functionality overloaded on single functions. It's far easier to generalize on something later than to implement something now that one might later regret. All features have been carefully considered for stylistic purity and generality.

In designing T, we decided not to constrain ourselves by adhering to Lisp tradition. Instead, we have sought to regularize the language even if incompatibilities are introduced. One way in which we have done this is in our choice of names for primitive procedures and special forms. Examples of naming conventions are: predicates end in question-mark (ATOM?, NULL?, SYMBOL?); destructive variants of non-destructive procedures end in exclamation mark (APPEND! instead of NCONC); the consistent use of hyphens to separate words (READ-LINE instead of READLINE); and use of asterisks around global variable names (*PRINT-LEVEL* instead of PRINLEVEL). We have avoided all but the most obvious abbreviations. These conventions have made the language much easier to learn and remember.

Another way we have sought regularity is in argument passing conventions. Accessors take aggregate arguments first, and selector arguments following (like array referencing, and unlike Maclisp's NTH). Assignment routines take the value to be stowed as their last argument. Optional arguments go last.

T attempts to provides components which compose well. For example, NIL and Common Lisp provide a procedure FILL which destructively stores a given object as consecutive elements of a sequence (list, vector, or string). To obtain full generality, FILL accepts optional arguments which specify the range of a sequence to be filled. The problem with this is that all other routines which manipulate sequences must also accept and process such optional arguments to obtain similar generality; this introduces undue complexity in specification and implementation, especially where two or more sequences are involved. T's equivalent routine FILL! accepts only the object and the sequence, and if a subsequence is to be selected, that is done with a call to SLICE which returns a shared subsequence. We rely on the compiler to make sure this strategy incurs no efficiency penalty.

As in Common Lisp, T provides a rich set of data types, including characters, strings, bit vectors, arrays, record structures, and queues.

In T, as in object-oriented languages like Smalltalk, [Goldberg 76] the *type* of an object is defined by its behavior. Unlike more conventional Lisp systems, T has no notion of "the" type of an object. Types *per se* are only represented by type predicates, and objects may answer true to several predicates; for example, 35 answers true to both INTEGER? and NUMBER?.

In T one might write the above example as follows:

```
(DEFINE (KONS THE-KAR THE-KDR)
  (OBJECT NIL
          ((KAR SELF) THE-KAR)
          ((KDR SELF) THE-KDR)
          ((PARE? SELF) T)))
(DEFINE-OPERATION (KAR OBJECT) (ERROR))
(DEFINE-OPERATION (KDR OBJECT) (ERROR))
(DEFINE-OPERATION (PARE? OBJECT) NIL)
```

The clauses in the OBJECT form consist of pattern/behavior pairs. Calls on generic operations where the first argument is the result of a call to KONS are matched against the patterns in the OBJECT-expression. This is accomplished not by calling the object directly, as in the Scheme version of the example, but by calling its handler.

DEFINE-OPERATION closely resembles DEFINE. The body of the definition provides the operation's default method. In the case of the PARE? operation defined above, this works out nicely because PARE? behaves as a predicate which returns true for objects which specifically handle the PARE? operation by returning true, and returns false (NIL) otherwise. Presumably objects like 3 and APPEND will not handle the PARE? operation, so (PARE? APPEND) will return false as expected.

T provides a simple, standardized mechanism for defining *generic operations* and for creating objects which handle such operations in idiosyncratic ways. Generic operations are procedures which may elicit distinguishing behavior from an object. Such operations may also have *default methods* for dealing with objects which do not otherwise handle the operation. This system for creating objects and defining generic operations allows users to define types abstractly, and forms the basis of T's type system.

The standard protocol by which objects handle operations is isomorphic to the Scheme technique of implementing data structures procedurally. The difference in T is that all objects participate in this type system, not just those procedures which are prepared to handle the protocol. In effect, all objects have two ways in which they may be called: normally as a procedure (this is not always permissible; for example 35 is not callable), or alternately to handle generic operations (this alternate entry point, or *handler*, is always present).

For example, in Scheme one might write

```
(DEFINE (KONS THE-KAR THE-KDR)
  (LAMBDA (REQUEST)
    (COND ((EQ? REQUEST 'KAR) (LAMBDA () THE-KAR))
          ((EQ? REQUEST 'KDR) (LAMBDA () THE-KDR))
          ((EQ? REQUEST 'PARE?) (LAMBDA () T)))))
(DEFINE (KAR OBJECT) ((OBJECT 'KAR)))
(DEFINE (KDR OBJECT) ((OBJECT 'KDR)))
(DEFINE (PARE? OBJECT) ((OBJECT 'PARE?)))
```

One problem with the above example is that given an arbitrary object, there is no way to determine whether or not it is a PARE. For example, to determine whether the procedure APPEND is a PARE, it is not sufficient to say (PARE? APPEND) because this would wind up doing ((APPEND 'PARE?)), which is inappropriate and would cause an error.

This generic operation protocol is used by the T system for several purposes, most notably for input and ouput, generalized assignment, and debugging. Because generic operations are used these aspects of the system are inherently user-extensible. The following example illustrates the use of OBJECT in defining a new I/O stream. MAKE-PREFIXED-STREAM is defined to be a routine which returns an I/O stream which behaves like a given I/O stream, except that each output line is prefixed with a given string:

```
(DEFINE (MAKE-PREFIXED-STREAM STREAM PREFIX)
  (OBJECT NIL
          ((WRITEC SELF CHARACTER)
           (WRITEC STREAM CHARACTER))
          ((NEWLINE SELF)
           (NEWLINE STREAM)
           (WRITES STREAM PREFIX))))
```

WRITEC, WRITES and NEWLINE are generic operations which are intended to write a character, write a string, and start a newline, respectively.

Ordinary procedures may have handlers distinct from the code invoked when the procedure is called. This permits operations on procedures other than invocation, a feature not present in Scheme. For example, SETTER is a generic operation used for general assignment, a feature similar to SETF in Lisp Machine Lisp.

(SETTER *access-procedure*)

should return an assignment procedure which "inverts" *access-procedure*; (SETTER CAR) returns the procedure used to alter a pair's CAR (SET-CAR, that is, RPLACA). Similarly, a debugging system may obtain a procedure's name, expected number of arguments, documentation, or the name of the file in which it is defined, by invoking generic operations.

The decision between lexical and dynamic binding is not an either/or question; as in Scheme, T provides both, with lexical binding the default. In T, however, no distinct syntactic form (FLUID) is required to access dynamic variables, since dynamic binding consists merely of a temporary assignment to a lexical variable (which is usually global).[1]

Reliably optimized tail-recursion permits direct implementation of iteration constructs using recursion, simplifying the language's implementation and semantics. This also enhances extensibility and expressiveness by permitting easy specification of new or unusual control structures.

CATCH implements a non-local exit facility similar to that in Lisp. Within the *body* of

(CATCH *variable body*)

the *variable* is bound to an *escape procedure*. The body is evaluated and its value is the value of the CATCH-expression. If the escape procedure is called the CATCH-expression returns immediately and its value is the argument passed to the escape procedure. This differs from CATCH in Lisp in that the escape procedure is itself an independent object unrelated to the variable whose value it is. In Lisp the user does not have access to the escape procedure and its name is dynamically (globally) bound. Scheme's lexical CATCH may be used to implement constructs like Lisp's RETURN and C's break, although it is more similar to Bliss's more general leave because it is named. Because the identifier is lexically bound, and all references to it may be detected statically by the compiler, calls to the escape procedure can often compile as simple jumps.

As a concession to efficient implementation on standard architectures, escape procedures are not valid outside the dynamic extent of the CATCH-expression which creates them; this ensures that the control stack behaves in a stack-like way, unlike in Scheme, where the control stack must be heap-allocated.

Some aspects of the language, particularly lexical scoping and optimized tail-recursion, present special problems for debugging tools. A program's execution history is lost early, and variable environments of interest are not always easily accessible. We have designs for mitigating these inconveniences.

T is a portable language, one whose implementation neither requires nor precludes any particular "conventional" or special-purpose hardware. This is a kind of insurance against the future. By minimizing one's reliance on particular hardware, one obtains good vendor support for hardware and software, the ability to take advantage quickly of new technologies as soon they become available, and low cost.

On the other hand, users have access to special machine and environment features where desired. On the Apollo one may call arbitrary operating system code, providing easy low-level access to the Apollo's display, window system, graphics, process and file system control, and inter-process communication.

T supports full compatibility between interpreted and compiled code. The interface between compiled and interpreted code is transparent to the user. Code which runs compiled will run interpreted, and vice versa. Problems that users encounter of the form "my code runs interpreted but not compiled" are either an interpreter or compiler bug. As discussed in the previous section, most Lisps have suffered from a schizophrenia where interpreted and compiled code implement different default binding rules: dynamic binding in the interpreter, lexical in the compiler.

T provides support for building large systems. Lisps have usually only a single flat global namespace. Lisp Machine Lisp provides a hierarchy of namespaces to aid modularity and avoid name conflicts; this system operates using multiple hash arrays in which symbols are "interned"; thus names are resolved when programs are translated to S-expression by READ. The method has problems with forward references, shadowing, and import/export. Our solution was inspired by, and is being worked out with the help of, Gerry Sussman and colleagues at MIT. We eliminate the distinction between global and local variables, and consider all names as bound in some lexical contour. Some of these lexical contours have the property that they are editable; that is, new variable bindings may be added or old ones removed. Language-level primitives are provided for manipulating these contours in various ways and for accessing names not in lexical environments other than the current one. Such editable contours behave something like file directories, and subsume the role of the global environment.

Open-coding and block compilation are techniques whereby users may sacrifice security and incremental redefinition to obtain faster execution and more compact code. When a user permits the compiler to open-code +, CAR, or structure field accessors, for example, he is asserting that he is willing to put up with the loss of argument type checking (probably because he believes his code to be correct) and a high cost for redefining the open-coded routines — he may have to recompile all of his code or perhaps even the entire T system itself if CAR changes. T decouples open-coding from runtime consistency checking, and users always have independent control over each on a per-call, per-procedure, or per-module basis. If incremental redefinition conflicts with previous early binding decisions the user is warned and provided with an opportunity to regain consistency either by backing out or by recompiling the offending code.

## 3. Implementation

Most of T is currently implemented for the VAX-11[2] under Unix[3] and VMS and for the Motorola MC68000-based Apollo Domain[4] workstation. [Apollo 82] Internally these implementations are structured quite similarly and share most of their code, to the extent that sometimes we refer to "the VAX/68000 implementation." This section describes these implementations, although much of what is described will apply to future implementations on different machines as well. Many pieces of the VAX/68000 implementation will be directly usable in other architectures.

Many specific implementation techniques, for example the way type codes work, the means by which position-independent code is achieved, and the idea of a global system constants table, have been borrowed from the VAX NIL implementation. [White 79]

We have written as much code as possible in T itself. We exploit the compiler and its ability to generate good code quite heavily. In fact, in terms of lines of source code, the compiler comprises at the moment well more than half of the implementation. In this way T's character resembles that of more traditional compiled languages, where the runtime library is relatively unimportant compared to the compiler, so one tends to think of the language as implemented by the compiler rather than by the runtime system. However, this is not to say that the compiler knows many specific things, but rather, it knows a lot about a small number of things (like LAMBDA).

The compiler is a substantial modification of one written by Guy Steele, the S-1 Common Lisp compiler, which is described in detail elsewhere in these proceedings. [Gabriel 82] It is based on principles similar to the RABBIT compiler, and incorporates a register allocation algorithm similar to that in the Bliss-11 compiler. [Steele 78c, Wulf 75] In addition to changes in the surface language and code generator, we extended the compiler to implement lexical closures. Environments are allocated using generalizations of the methods used in RABBIT. Heap-allocated environments are represented not by lists as in RABBIT but by chained vectors. Environment consing is postponed until the latest possible point, and contours are collapsed where appropriate to obtain more compact representations.

In January 1982 we decided to suspend work on a critical subphase of the code generator, to get a chance to work on the rest of the system. As a result the object code is much more verbose than it will be when this subphase is working. We expect about a factor of two improvement in the overall speed of the system after we have a chance to work on the code generator again. We anticipate no problems in making this change.

Because most of the system is written in T, its machine-language kernel is small. For example, it does not include the evaluator or garbage collector. However, it is bigger than we would like it to be. The reason for its size is mostly that it was generally more expeditious to introduce new assembly-language routines than new code generators and compiler features; in practical terms, the choice between these two alternatives is often a toss-up. We expect the size of the kernels to decrease as time passes.

By requiring that all objects be quadword aligned, the low three bits of a T pointer can be used to encode the low-level, internal type of the object represented by the pointer. The internal type *escape* indicates that the actual internal type can be obtained from a standard place in the object itself. The pointer tags are used to encode the same internal types on both the VAX and 68000, though this need not be the case. There is, in fact, some variation between the two implementations in the mapping of tag to type. Internal types that are currently represented directly by pointer tags are:

| | |
|---|---|
| *fixnum* | immediate integer, 29 bits |
| *pair* | a single list cell |
| *flonum* | floating point number |
| *string* | character string header |
| *miscellaneous* | () and characters |
| *template* | type descriptor |

The use of the one unassigned tag will soon be decided based on the results of system metering.

The representations have been engineered to minimize the pointer calculation needed in order to manipulate the objects represented. For example, a pair (cons cell) consists of two 32-bit pointers for the cell's CAR and CDR. Suppose the type tag used for pointers to pairs is 5 (this is a system parameter). If R0 is a register which holds such a pointer, then the assembly operands -5(R0) and -1(R0) fetch the CAR and CDR, respectively. Fixnum arithmetic can be compiled in line efficiently because the fixnum tag is 0. Two fixnums can be added or subtracted in a single instruction. Multiplication requires that one of the fixnums be pre-normalized with a 3 bit shift right; the result of division must be post-normalized with a 3 bit shift left.

All objects have *templates*, which are low-level type descriptors.[5] These roughly correspond to *classes* in Smalltalk, or *flavors* in Lisp Machine Lisp. In the case of objects whose type is represented directly in the pointer's tag field, the template is obtained by a table lookup (except that the *miscellaneous* type is multiplexed, requiring further disambiguation); otherwise, the pointer's tag field is the escape tag, and the template is found by following the pointer (similarly to CAR, described above). An object's template provides information for the garbage collector, and contains the procedure used to handle generic operations applied to the object.

In the case of procedures, the template also gives the machine code to run when the procedure is called, the procedure's name, number of arguments, and other information. Templates thus correspond to source-level LAMBDA-expressions or "open procedures".

Invoking unknown procedures is uniform and cheap; one can jump directly to the called procedure's template. In Maclisp the FUNCALL primitive is considerably more expensive than a call to a known procedure, because a type dispatch must be performed, and different calling sequences must be handled. The T implementation has a single standard calling sequence and one representation for procedures, so its equivalent of FUNCALL has overhead more similar to Maclisp's SUBRCALL. The implementation permits dynamic alteration, on a per-module basis, of the amount of consistency checking (*e.g.* number of arguments) performed on each procedure call.

Stack frames are valid language-level objects at no extra cost. A stack frame's template is precisely its return address. This has allowed considerable simplification and generality in those parts of the system which are concerned with stacks, for example, THROW, stack inspection in the debugger, and the garbage collector. The correspondence between stack frames and the implicit continuation discussed in [Steele 76b] is reflected in the implementation, to the extent that returning a value — that is, invoking an implicit continuation — behaves nearly the same as procedure calling.

We open-code consing (alloction) in one or two instructions; the top-of-heap pointer is contained in a register. A simple copying garbage collector is employed. When necessary or desirable, the system stops and traces all active pointers, making a compact copy of the active heap. Objects may also be statically allocated if desired, in which case they are traced but not copied. The garbage collector is data-driven; each low-level type descriptor (template) effectively contains a pointer to the routine to be used to copy instances of the type.

As in Maclisp, asynchronous interrupts (for example, timer or user keyboard interrupts) may occur between any two instructions. Since interrupt handlers consist of arbitrary code, they may potentially invoke the garbage collector, which might involve relocating the interrupted object code itself. Getting this right requires some care if consing is interrupted.

One register is used to point into a global system constants table. This is a structure, whose format is known by the compiler, which holds frequently-used quantities of various kinds, like (), pointers to important procedures, and bit masks. This is important because all code must be position-independent.

The implementation supports position-independent sharable code. This has advantages in the virtual memory multiprocessing architectures for which T was designed. Multiple T processes (perhaps invoked by different users) may dynamically load object files and obtain sharing, using an exit-vector strategy similar to that used in the Multics and Apollo operating systems. (Unfortunately, we can't make use of this under Unix, due to its per-process limit on the number of open files.) Code which is loaded into the active storage area (heap) is moved during garbage collections, and if all pointers into a module have been dropped, its storage is reclaimed.

A compiler option permits generation of modules in such a way that they can loaded using native operating system linkers, such as ld in Unix. Because the T dynamic loader need not process these modules we call them "pre-cooked." Along with the object code for pre-cooked modules, the compiler also produces the skeleton of an initial heap, which consists of the symbols and value cells required by those modules. Only the dynamic loader, and the support it requires, need be pre-cooked; when the system starts up it dynamically loads all of the remaining system. Actually, the dynamic loader's loading (file I/O) and relocation phases are independent. This allows ordinary (not pre-cooked) modules to be linked into the system with the operating system's linker; they are automatically Lisp-relocated in the system initialization sequence.

During summer 1982 we plan to generalize the garbage collector to be able to copy the entire current state of the system (including heap and control stack) to a file, which can later be invoked to recover this state, like SUSPEND in Maclisp. Unlike the situation with Maclisp on the PDP-10, however, the operating systems under which we must run T do not provide any support for a suspension facility; in fact, on the Apollo, we are constrained to use a position-independent representation for suspended systems. This representation will be similar to the object file representation, and we will use the same relocation program to resume after the suspension.

We provide general interfaces to arbitrary non-Lisp code; usually C on the VAX and Pascal on the Apollo. The implementations use these interfaces to do basic I/O functions and other system control. This is one major means by which we have avoided writing code in assembly language.

Portability is achieved in the conventional way by a well modularized compiler and (small) machine dependent runtime system that handles, among other things, machine interrupts, memory management, and low level input and output. The compiler's register allocation and optimization phases are machine independent. Both are driven by information which may be machine specific, for example, the number of registers, and restrictions on the types of instruction operands permissible in open-coded procedures such as CAR and FIXNUM-ADD.

Because the language design is still experimental, we have been careful to avoid premature optimization. The emphasis has been on providing functionality first and working on performance later; correctness and completeness have taken priority over efficiency.

## 4. Status

The most important components of the system are already in place. There is a preliminary manual [Adams 82], and users at Yale are developing new applications programs in T, as well as transporting code previously developed in other Lisp dialects. The system is now in good enough shape that work can begin transporting the T compiler, which currently runs in Tops-20 Maclisp, to T on the VAX and Apollo. We expect no major problems doing this, as most of the compiler is written in a common subset of Maclisp and T, using a simple T compatibility mode developed for Maclisp.

Work on the project began in June 1981 with a staff of three. During this initial period an interpreter for Scheme was developed in which many language and system design issues were explored; this interpreter ran in DEC-20 Maclisp. We also converted the compiler, which when we obtained it generated code for the S-1 computer, to generate code for the VAX and the 68000, and adjusted its source language to be more Scheme-like than Lisp-like.

From September 1981 to May 1982 the project was staffed with one and a quarter full-time people. About half this time was devoted to further work on the compiler to support closures and dynamic loading. The other half was spent writing and debugging system code: interpreter, garbage collector, I/O system, and so forth.

During summer 1982 two of us will be working full-time. Most of our effort will be directed towards improving performance. Inspired in part by reported experience with implementing Interlisp [Burton 80], we intend to develop and begin using general tools for profiling and metering the system as soon as possible. Debugging aids and user interface will also receive much attention.

Plans for future development include the following.

*Arithmetic.* We intend to support rational and arbitrary-precision integer arithmetic, and possibly interval and complex arithmetic. This area has so far received comparatively little attention. This is not because we don't consider it important, but because we've had so many other things to worry about.

*Early binding and module interfaces.* The early binding facilities outlined in section 2 are designed and partially coded. We need to develop more complete tools for source code control and module interface specification.

*Declarations and type propagation.* We want to be able to statically transform generic to specific operations with a minimum of hints (in the form of assertions or declarations on the types of expressions and variables) from the user; for example, transforming generic + to open-coded fixnum or flonum addition based on knowledge of the types of operands and result. This is conceptually similar to constant folding. Ideally this would be done with a general partial evaluation strategy similar to that already in the compiler.

*Open-coded references to procedural data.* For example, if a variable F is known to be a closure over a particular lambda-expression (LAMBDA () Y), where Y is a local variable, then calls to F may be open-coded as references to Y's position in F's environment structure. This problem is discussed in [Steele 76b]. Once this is possible, the current structure package, which is currently distinct from the rest of the type system for efficiency reasons, may be implemented using closures.

*General stack-allocated objects.* In particular, the ability to create closures on the stack allows one to pass general "downward funargs" to programs without heap-allocating environments.

*Multiple control stacks.* We need to re-introduce some kind of coroutine mechanism, to compensate for our restriction on the use of escape procedures to enforce stack-like behavior. With this facility we could easily implement multiple tasks within a single T image, with a simple scheduler. We anticipate no problems extending the garbage to be able to reclaim stacks and coroutines to which there are no active pointers.

*Portability.* We need to make the transportation process smoother, in preparation for moves to other machines, which will inevitably differ from the VAX and 68000 to a greater extent than those machines differ from each other. The interfaces between components of the compiler which deal with more or less machine-independent must be made more explicit than they are now. We are considering beginning an IBM 370 implementation this summer.

*Common Lisp compatibility.* We believe that T will be general enough to be useful as an implementation vehicle for Common Lisp. We expect programs which adhere to the Common Lisp specification to run transparently in a Common Lisp subsystem implemented within T. Determining the difficulty of this task will need to wait until the Common Lisp language specification has stabilized.

In the long run, we would like to continue to improve efficiency towards a degree competitive with, say, Bliss. We feel that there is ultimately no reason that programmers should have to resort languages besides Lisp (that is, T) to obtain desired performance levels.

## 5. Conclusions

The T project has benefited from having not been constrained by the need for compatibility with other Lisps. We have made every effort to ensure that this incompatibility is not gratuitous.

We believe our methodology has proved effective. Our approach has emphasized the following principles:

- Avoid premature optimization.
- Approach ends not monolithically but by progressive approximation.
- If there's a problem, generalize it, then solve the more general problem.

This approach has permitted a small number of people to design and implement a powerful system in a relatively short amount of time. Much remains to be done, but it can be said that at one year old T is already a practical system.

T's emphasis and dependence on an optimizing compiler may be a new idea in the Lisp world, especially today when everyone seems to be opting for special hardware, but it is accepted practice outside the Lisp world. Early experience with T seems to justify our belief that conventional machine architectures can be used effectively to implement a production Lisp system.

T is a synthesis of those concepts and features we have seen elsewhere and liked. But we believe that in spite of its diverse influences we have built a clean, coherent, and powerful programming language.

## Acknowledgements

Kent Pitman's generous contributions to this project are much appreciated.

The authors wish to thank John Ellis, Drew McDermott, Nathaniel Mishkin, John O'Donnell, and Robert Nix, for their support during the preparation of this paper.

The work of Guy L. Steele Jr. and Gerald J. Sussman has been a continuing inspiration.

## References

[Adams 82] Norman I. Adams and Jonathan A. Rees.
The T Manual.
Yale University, Department of Computer Science, New Haven, Connecticut, 1982.
Pre-release Edition.

[Apollo 82] *System Programmer's Reference Manual.*
Apollo Computer, Inc., 19 Alpha Road, Chelmsford, Massachusetts 01824, 1982.

[Bawden 77] Alan Bawden, Richard Greenblatt, Jack Holloway, Tom Knight, David Moon, and Daniel Weinreb.
Lisp Machine progress report.
AI Memo 444, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, August 1977.

[Burton 80] Richard R. Burton, L. M. Masinter, Daniel G. Bobrow, Willie Sue Haugeland, Ronald M. Kaplan, and B. A. Sheil.
Overview and status of Doradolisp.
In *Conference Record of the 1980 Lisp Conference*, pages 179-187. Stanford University, Computer Science Department, August 1980.

[Foderaro 80] John K. Foderaro.
The Franz Lisp manual: A document in four movements.
Computer Science Research Group, University of California at Berkeley, 1980.

[Gabriel 82] Richard P. Gabriel, Rodney A. Brooks and Guy L. Steele.
S-1B Common Lisp implementation.
In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming.* Association for Computing Machinery, August 1982.
To appear.

[Goldberg 76] Adele Goldberg and Alan Kay.
Smalltalk-72 instruction manual.
Technical Report, Xerox Palo Alto Research Center, March 1976.

[Griss 82] Martin L. Griss.
Portable Standard LISP: A brief overview.
Operating Note 58, University of Utah, Department of Computer Science, January 1982.

[Kernighan 81] Brian W. Kernighan.
Why Pascal is not my favorite programming language.
Computing Science Technical Report 100, Bell Laboratories, Murray Hill, New Jersey, July 1981.

[Moon 74] David A. Moon.
Maclisp reference manual, revision 0.
Project MAC, MIT, Cambridge, Massachusetts, 1974.

[Moore 79] J. Strother Moore II.
The Interlisp Virtual Machine specification.
Technical Report CSL 76-5, Xerox Palo Alto Research Center, March 1979.

[Pitman 80] Kent M. Pitman.
Special forms in Lisp.
In *Conference Record of the 1980 Lisp Conference*, pages 179-187. Stanford University, Computer Science Department, August 1980.

[Steele 76a] Guy Lewis Steele, Jr. and Gerald Jay Sussman.
Lambda, the ultimate imperative.
AI Memo 353, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, March 1976.

[Steele 76b] Guy Lewis Steele, Jr.
Lambda, the ultimate declarative.
AI Memo 379, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, November 1976.

[Steele 77a] Guy Lewis Steele, Jr.
Data representation in PDP-10 Maclisp.
AI Memo 420, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, September 1977.

[Steele 77b] Guy Lewis Steele, Jr.
Fast arithmetic in Maclisp.
AI Memo 421, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, September 1977.

[Steele 77c] Guy Lewis Steele, Jr.
Debunking the expensive procedure call myth, or, procecdure call implementations considered harmful, or, lambda: The ultimate goto.
AI Memo 443, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, October 1977.

[Steele 78a] Guy Lewis Steele, Jr. and Gerald Jay Sussman.
The revised report on Scheme, a dialect of Lisp.
AI Memo 452, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, January 1978.

[Steele 78b] Guy Lewis Steele, Jr. and Gerald Jay Sussman.
The art of the interpreter or, the modularity complex (parts zero, one, and two).
AI Memo 453, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 1978.

[Steele 78c] Guy Lewis Steele, Jr.
Rabbit: A compiler for Scheme (a study in compiler optimization).
Technical Report 454, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 1978.

[Steele 81] Guy L. Steele, Jr. and Scott E. Fahlman.
Common Lisp reference manual.
Spice Document S061, Computer Science Department, Carnegie-Mellon University, September 1981.

[Steele 82] Guy L. Steele.
Report on the 1980 Lisp Conference, Stanford Universty, August 25-27, 1980.
*ACM SIGPLAN Notices* 17(3):22-35, March 1982.

[Teitelman 78] Warren Teitelman.
*Interlisp Reference Manual.*
Xerox Palo Alto Research Center, Palo Alto, California, 1978.

[Weinreb 81] Daniel Weinreb and David Moon.
*Lisp Machine Manual.*
Third edition, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1981.

[White 79] Jon L. White.
Nil: A perspective.
In *1979 Macsyma Users' Conference Proceedings.* Macsyma User's Conference, Washington, D.C., June 1979.

[Wulf 75] William Wulf, Richard K. Johnsson, Charles B. Weinstock, Steven O. Hobbs, Charles M. Geschke.
*The Design of an Optimizing Compiler.*
Elsevier North-Holland, New York, 1975.

[1]Shallow binding works only in uniprocessor architectures. If T were implemented in a multiprocessor architecture we would need to go to some sort of deep-binding strategy for dynamic variables. Thanks to Guy Steele for this insight.

[2]VAX is a trademark of Digital Equipment Corporation.

[3]UNIX is a trademark of Bell Laboratories.

[4]Domain is a trademark of Apollo Computer, Inc.

[5]This is not strictly true in the current system, but it might as well be.