# The L Programming Language or Tcl for C Programmers

*Oscar Bonilla, Tim Daly, Jr., Larry McVoy*

BitMover, Inc.
300 Orchard City Drive, Suite 132
Campbell, CA 95008

*Jeffrey Hobbs*

ActiveState Software Inc.
1700-409 Granville Street
Vancouver, BC, Canada
V6C 1T2

`l@bitmover.com`

*ABSTRACT*

This paper describes a new programming language called L. L is a compiled-to-byte-code language with the unusual twist that it compiles to Tcl byte codes and by doing so leverages the entire Tcl runtime. L is designed to peacefully coexist with Tcl rather than replace Tcl. L functions may call Tcl procs and vice versa. They may also coexist in the same source file. L is a static weakly typed language with int, float, string, struct, array, and hash as first-class objects. The L syntax is reminiscent of C with a tiny bit of C++ thrown in.

The implementation consists primarily of a simple compiler that Tcl invokes whenever L source code is encountered. The L code is parsed by a Bison-generated parser into an abstract syntax tree (AST), which is type-checked and then translated into Tcl byte code. Upon its execution, L code is indistinguishable from Tcl code, which makes for easy interoperability.

L is open source software, and it is made available under the same license as Tcl/Tk with the hope that people will find it useful and it may encourage more people to join the Tcl/Tk community.

*"It's like perl without the nastiest bits."*

-- Donal K. Fellows (on the #tcl IRC channel)

## 1. Introduction

BitMover software is produced using a conservative development methodology. All development goes through a stringent process that relies heavily on peer review and extensive regression tests to ensure quality products.

Because of the stability requirements of our market, we read code much more than we write it. Spot checks indicate that we spend at least 10 times as much time reading and reviewing as we do writing. Naturally, we tend to optimize heavily for the read path rather than the write path.

For years we have used the Tcl/Tk system for our graphical user interfaces. We periodically consider the alternatives and have consistently found that short of doing native implementations, the Tcl/Tk system is still the best choice from a development cost point of view. Our estimate is that it would cost roughly six times as much to develop and maintain native GUIs instead of using a single Tcl source base for all platforms. However, the maintenance of our Tcl source base has recently become problematic because two things happened:

- Our Tcl source base grew past a manageable size (for us).
- Our peer review system could not handle Tcl code.

We have about 25,000 lines lines of Tcl, implementing about a dozen graphical interfaces for browsing code, checking in code, viewing changes, etc.[1] Maintaining and extending the Tcl source base has become unmanageable, and when the review process was added to the mix, the costs became too high.

This has been a problem for us for years and we were forced to come up with a better answer. We investigated the alternatives but in the end the Tcl runtime and the Tk widgets were too compelling. We solved our problems by marrying a language syntax we felt was well suited for fast reviewing and understanding with what we feel is the best GUI toolkit and runtime available today.

The rest of the paper is divided into sections that discuss the following topics: an overview of L, why the L approach is interesting, why other runtimes were not chosen, why not pure Tcl, why not native GUIs, L language details such as types, calling/return conventions, current status, features we have not yet done but want to do, licensing and availability, and a summary. There is an appendix with some small working program examples.

[1] This number is artificially low because we have been holding off on a number of GUIs until we had a better answer. Had we not been holding back, 100,000 lines is more likely where we would be.

## 2. L overview

L is actually a very small addition to the Tcl system. If we divide the Tcl system into logical parts this becomes obvious:

| Subsection | Percentage of Tcl/Tk 8.5 |
|---|---|
| Tcl parser/compiler | <= 1% |
| L parser/compiler | <= 1% |
| Tcl runtime | 48% |
| Tk | 51% |

The parser and compiler are quite small when compared to the code that implements the runtime and the libraries (in both Tcl and L it is less than 10K lines of code). Because the parser/compiler is such a small part of the system, it is reasonable to add an alternative parser/compiler to the system and let them both run side by side. That is L in a nutshell. It is the small amount of effort required to leverage a large amount of value embodied in the runtime and libraries.

The L compiler creates an abstract syntax tree from L source and compiles that to byte codes. The byte codes generated are standard Tcl byte codes, following Tcl call/return conventions and using Tcl variables. Because we are careful not to break any Tcl rules, L functions may call Tcl procs and vice versa. This allows L to use the extensive, mature Tcl/Tk runtime and libraries unmodified.

## 3. Unique design

As we dive deeper into the L syntax and semantics it would be easy to be drawn into a discussion of why L is better or why Tcl is better. To do so would be to miss an important point. Regardless of the merits of each language, the value of L is that it demonstrates a new way to leverage and reuse existing code. With a relatively small amount of effort, we have leveraged over 1.4 million lines of source making up the Tcl/Tk system plus some extensions.

The existence of L opens the door to any number of domain-specific languages being added to the Tcl runtime system.

For example, consider the GDB debugger. GDB lets users type C, C++, etc., at it and run the code. Doing so means GDB has to provide an interpreter and a runtime. Rather than building one, GDB could reuse the ideas and code pioneered by the L effort. Having a well maintained runtime with the option of creating an arbitrary syntax to use that runtime is useful for any sort of debugger or runtime inspector. L is just one example of a different syntax leveraging the Tcl/Tk system; we are confident there will be others.

## 4. Alternative runtimes

Once the idea of adding a different parser/compiler to a scripting language is understood, the question becomes: why Tcl rather than some other runtime such as Perl, Python, Ruby, Java, or others? We looked briefly at that question. Our need was for a well supported, mature runtime that supported scripting GUI interfaces and was extensible from C.

We dismissed Java because the runtime is too large and the GUI toolkits are weak, both in features and in performance. The other runtimes addressed the GUI issues mostly by providing Tk bindings (and in some cases Qt or Gtk bindings). Any system that is using Tk bindings is already dragging along a Tcl interpreter to run the Tk code. It seemed like a waste to have a different interpreter just for the GUIs. It has also been our experience that the only way to build robust software systems is to have the minimum number of ‘‘moving parts.’’ Having two interpreters is an unnecessary complication.

But even if there were a good runtime with a good GUI interface, there was another requirement we felt was only well addressed by Tcl. Tcl has been designed from the onset to be an extendable language. The original vision was that Tcl was glue and all the heavy lifting would be done by C extensions to the language. The internal Tcl code is fairly small and quite pleasant to use; adding extensions is straightforward and natural. We needed to take advantage of this feature of the Tcl system and other runtimes made this difficult.

## 5. L vs pure Tcl

Many in the Tcl community may question whether there is any value in an alternate syntax for the Tcl runtime. After all, Tcl is a powerful, dynamic language and many significant applications are based on Tcl.

We agree that Tcl is powerful, but that power comes at a cost. Tcl’s dynamic nature makes it impossible to detect even simple parse errors, such as typos, without running the program.

Although there are advantages to the dynamic approach in language design, there are also drawbacks:

**Data structures.** Probably the single largest problem we found with Tcl was the lack of a C-style struct, i.e., a centralized collection of variables with annotations indicating why they are there. These are commonly emulated in Tcl with associative arrays. That isn’t good enough because the ‘‘struct fields’’ are scattered all over the source base rather than being in one place, laid out with types and comments. To paraphrase Fred Brooks: *“Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won’t usually need your code; it’ll be obvious.”*[Brooks1975a]

**Lint.** It is impossible to write a syntax checker or a lint-like tool for Tcl that works 100% of the time unless that tool is actually running the program it is checking. Even an interpreter-based tool would have the problem that it is not practical to force the application through all possible code paths. It is worth noting that this problem is present in all dynamic languages and object-oriented languages have the same problem; you can’t just look at the code and know what it is doing.

**Reviewing.** As mentioned previously, at BitMover we do a lot of peer review as well as other forms of code reading. For the same reasons that it is difficult to write a lint-like tool for Tcl, it is difficult for a human to look at Tcl and understand what it is doing. The verbose style of basic operations in Tcl, e.g.,

```
lset fib $i \
    [expr \
    {[lindex $fib [expr {$i-1}]] +
     [lindex $fib [expr {$i-2}]]}]
```

vs

```
fib[i] = fib[i-1] + fib[i-2];
```

tend to obscure what is actually being said in the code.

**Optimization.** Optimizing Tcl is more challenging than optimizing a ‘‘weaker’’ language such as L. Many well understood optimization techniques could be applied to the compilation of L, resulting in a significant performance increase for some programs. As an example, due to the static type system of L, we believe it’s possible to make L immune to ‘‘shimmering.’’[Wiki2005a]

We tend to view Tcl more like assembly language on steroids. It is a powerful tool and when that power is needed it is appreciated. But most of the time we are doing fairly simplistic programming deliberately so it is easy to read, and we find that a static language with a static type system is much easier for us to read and easier for a compiler to optimize and check.

## 6. L vs native GUIs

This question gets raised at least once a year here: why not do native GUIs? It is certainly possible to do so. We have done implementations of several of our GUIs in other toolkits. The arguments for doing so are compelling: better look and feel, native behavior, etc.

The reasons for staying with Tcl/Tk are simple:

**Cost.** The cost of creating 2-4 different implementations of each GUI interface is probably 3 times what it took us to get where we are today. But the cost does not end there. The cost extends to testing the GUIs on each platform as well as putting processes in place to make sure that the GUIs march forward in sync, i.e., if the Java revtool gets a new feature, that same feature needs to be added to the Linux, Windows, and Aqua GUIs. When we add up all the costs, it looks more like 6 times the effort.

**Functionality.** Every time we go look at the other toolkits we find that they are not as powerful as the Tk toolkit. In particular, the canvas and text widgets are more useful than any others we have found.

That said, a large drawback of the Tk approach is the lack of a complete widget set in the core. In order to get the functionality needed, a ragtag group of extensions, with partially overlapping features, need to be combined into a Tcl/Tk ‘‘distribution.’’ We look forward to the day that this issue is resolved.

## 7. L language details

In this section we cover some of the differences from C, differences from Tcl, types, call/return conventions, expressions, and control flow.

### 7.1. Extensions to C

**Regex.** L uses Perl’s syntax for regular expressions in statements, but it uses Tcl’s regular expression engine. So you may say:

```
if (a =~ /${r}/) {...
```

to get the same results as Tcl’s

```
if {[regexp $r $a]} {...
```

**Associative arrays.** We call these hashes in L to distinguish them from traditional C-style arrays. The keys and values are strings.

**Arrays grow.** If you assign into an array past the last element the array grows as needed. Many constructs that would normally use C pointers, such as linked lists or trees, can be constructed with an array of structures linked via indices rather than pointers.

**defined().** A built-in that indicates if the variable passed is defined. The following tests for the existence of the field in the hash, and the existence of the array element, respectively.

```
defined(foo{"bar"})
defined(stuff[3])
```

**Strings.** Strings are first-class objects like any other base type. One implication of this is that unlike C strings, which are pointers, if you want to pass a reference to the string you must do so explicitly.

### 7.2. Unimplemented C features

L does not have bit fields, enums, unions, or C-style pointers. L currently does not have a C-like preprocessor, though one is planned.

### 7.3. Extensions to Tcl

**Type checking.** L has a weak static type system, which makes it possible to do type checking at compile time. Note that L’s type system is independent of Tcl’s runtime type system, although the two can interoperate. Variables in L may not change types, unlike Tcl variables, which are strings except when they’re not (as with floats, ints, lists, etc.)

**Structs.** C-style structs are part of L. A Tcl API is provided that supports getting and setting fields as well as introspection.

**References.** Pass by reference in Tcl is possible but awkward. Attempts have been made to improve it in Tcl[Wiki2005b] but they are unsatisfying. We think our syntax is cleaner and easier to read.

**Function prototypes.** Currently these are used to get type checking when calling Tcl built-ins. For example, we can prototype gets() as

```
extern int gets(FILE, string &);
```

to always require gets to be called with two arguments. We could also prototype gets() as

```
extern string gets(FILE);
```

to make it return a string. If prototypes are missing, L treats undefined functions as external Tcl functions that return poly and take a variable number of arguments of type poly.

## 7.4. Types

### 7.4.1. Simple types

**int.** Integer types in L are like C integers: they are sized to the machine's word size (at least 32 bits and possibly 64). Integers in L are initialized to 0, even for local variables.

```
int    a = 5;
int    b;          // defaults to 0
```

Any constant that looks like an int is typed as an int.

**float.** Floating-point numbers in L are at least double-precision IEEE 754. Floats are initialized to 0.0, even for local variables.

Any constant that looks like a fbat is typed as a fbat. Note that this means that assigning an integer to a fbat is only legal because of automatic type conversion.

```
float  f = 1;   // converts to 1.0
float  g;       // defaults to 0.0
float  pi = 3.14159265;
```

**string.** The string type is the same as a Tcl string but different from a C string. Strings are not null-terminated as they are in C, nor are they arrays of bytes. L strings are Tcl strings, which are UTF-8 encoded and have a known length. L strings are initialized to the empty string.

To iterate over each character in a string, use the defi ned() operator:

```
int    i;
string s = "a string";

for (i = 0; defined(s[i]); i++) {
    printf("s[%d]=%s\n", i, s[i]);
}
```

Note that there is no separate character type in L. When indexing into a string, each character is merely a string of length 1. This also means that there is no need to use special single-quoted syntax for character literals:

```
str[i] = "c";
```

L provides a special escape sequence, ${, which allows embedding code in strings. All the text from ${ to the matching } is collected and evaluated. Its value is then substituted into the string:

```
int i = 41;

printf("41 + 1 is ${i + 1}\n");
```

prints:

```
41 + 1 is 42
```

### 7.4.2. Tclish types

**poly.** This is a generic type that is like a Tcl variable on which no type checking is done. Normal variables cause compile-time errors if they attempt to change types; a poly variable suppresses the static type checking so that a variable can switch from one type to another, e.g. fbat to array or to int, etc. The following is legal code:

```
poly   unchecked;
string s;

unchecked = 1;
unchecked = "Hey there";
unchecked = 3.14;
// cast is required
s = (string)unchecked;
```

**var.** This is a compromise variable type. It is type-checked but the type is not set until the fi rst assignment. The type is determined from the assignment and may not change. The following throws an error:

```
var    late_binding;

late_binding = 1;
late_binding = "Hey there";
```

As we noted above, constant types are intuited. This might cause problems with *var* variables. For example, this throws an error:

```
var    f = 1;    // f is now an int

f = "pi";         // int/string error
```

but this works fi ne:

```
var    f = 1.0;

f += 3.14;
```

### 7.4.3. Magic

**:constant.** Many Tcl/Tk interfaces take key/value pairs that look like

```
text .t -bg white -fg black
```

which in L might look like

```
text(".t",
    "-bg", "white", "-fg", "black");
```

We wanted a way to make the −*whatever* stand out from the values being passed as an argument to −*whatever*. We decide to do that like this:

```
text(".t",
    :bg, "white", :fg, "black");
```

When the parser sees an identifi er in a function call that has a leading colon, L treats it as if it were a quoted string with the colon replaced by a dash.

#### 7.4.4. Compound types

**array.** Arrays are like C arrays in syntax but are implemented as Tcl lists under the covers. Array elements are homogeneous; all elements must share the same type. Array assignments in declarations are supported for globals and locals:

```
string foo[] = { "Hi", "there" };
int    bar[] = { 1, 2, 3, 4 };
int    i;
int    total = 0;

for (i = 0; defined(bar[i]); i++) {
    total += bar[i];
}
```

Arrays are dynamically grown and cannot be sparse.

```
int    a[2];

a[0] = 10;
a[100] = 20; // allowed
```

After the previous code has been executed, *a* has 101 elements. *a*[1] to *a*[99] have the value 0, which is the default initial value for integers.

The defined operator is an easy way to check if an index is outside the array bounds:

```
// prints 'no'
if (defined(a[101])) {
    printf("yes\n");
} else {
    printf("no\n");
}
```

**hash.** Hashes are associative arrays, indexed by strings and returning string values. They are implemented by Tcl dictionaries under the covers. Hash assignments in declarations are supported for globals and locals and follow the Perl syntax:

```
hash  h = { "key" => "val",
            "key2" => "val2" };

h{"foo"} = "bar";
if (defined(h{"blech"})) {
   printf("blech is not a key!\n");
}
```

The defined operator can also be used to check if a key is present in a hash:

```
// prints no
if (defined(foo{"k"})) {
    printf("yes\n");
} else {
    printf("no\n");
}
```

It is possible to iterate over each value in a hash using a foreach loop:

```
foreach (h as k => v) {
    printf("%s => %s\n", k, v);
}
```

**struct.** Structs are collections of typed variables, as in C. Declarations are the same as C declarations. Struct assignments in declarations are supported for globals and locals:

```
typedef struct {
    int     a;
    float   b;
    string  c;
} eg;

eg    s = { 1, 3.14, "hi there" };
```

Structures are implemented as Tcl lists just like L arrays. The names are translated into integer indices by the L compiler. Since it is just a Tcl list, an L structure can be passed to any Tcl proc that expects a list.

It is likely that we will extend the struct construct to have initializers, e.g.,

```
typedef struct {
    int     a = 1;
    float   b = 3.14;
    string  c = "hi there";
} eg;

eg    foo;
puts(foo.a);    // prints 1
```

### 7.5. Passing semantics

A C programmer, looking at Tcl, would think that the Tcl model is pass by value. While Tcl has no way to pass a C-style pointer to an object, it does have a way to fake it with something called *upvar*. L wants pass by value but it also wants to provide pass by reference. This section describes how we used the Tcl system to provide the L passing semantics. It amounts to a little syntactic sugar on top of *upvar*.

#### 7.5.1. By value

L obeys Tcl's semantics for pass by value. Parameter passing looks like it does in C:

```
int    i = 1234;

foo(i, 0xdeadbeef, "string");
```

L programs typically do not pass compound types by value to other L functions (but see the (*tcl*) cast below for how to pass them to Tcl procs).

### 7.5.2. By reference

The Tcl system has a way of passing by reference that might appear strange to C programmers.

```
proc foo {ref} {
    upvar $ref pointer

    set pointer 1
}
```

The *upvar* command creates a reference to the variable in the caller's context and places it in *pointer*. Assignments to *pointer* are the same as if the assignment were done in the caller's context (after evaluating the right-hand side).

We used this mechanism to emulate pass by reference in L. We call it ''pass by name'' because the caller is putting the name of the variable on the stack and the callee is doing an automatic *upvar* to create the reference. The syntax looks like:

```
void foo(int &ref)
{
    ref = 1234;
}

int   a = 19;

foo(&a);
puts(a);
```

and that prints

```
1234
```

Arrays and hashes do not take the ampersand because they are trying to behave like C arrays, i.e., they are already references.

```
void clear(int v[])
{
    int    i;

    for (i = 0; defined(v[i]); i++) {
        v[i] = 0;
    }
}

int   junk[] = { 1, 2, 3 };

clear(junk);// junk = { 0, 0, 0 }
```

Note that strings, unlike in C, are first-class objects and are **not** references. If you want to modify a string, you must pass it by reference. For example, to use the Tcl built-in for reading a line of input you have to do this:

```
string buf;

// buf is an out parameter
gets(stdin, &buf);
```

### 7.5.3. L pointers

While the *upvar* trick works nicely for many cases, there is still a need for real pointers. When creating a widget, such as an entry box, it would be natural to have a struct that contained all the things related to that widget such as its path, the variable that the entry box sets, etc., like so:

```
widgets(entry &e)
{
    e.frame = frame(".f");
    e.entry = entry("${top}.entry");
    e.entry("configure",
        :textvariable, &e.textvar);
}
```

Our trick of making an ampersand mean ''push the variable name on the stack'' does not work here for multiple reasons. First, the variable in this case is a structure field, which is an element of a Tcl list. There is currently no way to pass a list element as a –*variable* argument; Tcl does not support that. Second, –*variable* arguments must be accessible at the global scope. There is no guarantee that the name passed in makes sense at the global scope.

What is needed is a way to take an L variable and turn it into something that Tcl can find out of the event loop. The natural answer is some kind of pointer.

We created a new Tcl object type to hold all the information related to a pointer. The information looks like:

```
struct pointer {
    int    depth;  // upvar #depth
    string name;   // var pointed to
    string index;  // optional index
};
```

The depth field is used to get to the call frame where the variable being pointed at was declared. For GUI code like the example above, the depth is almost always 0, indicating a global. The string is the name of the variable to which the pointer refers. If the underlying type of the variable is a list (remember that structs are implemented as lists) then the index is the index into that list. The index is a string because in the future we intend to make pointers into hashes work.

There is a new Tcl command, *pointer*, which may be used to manipulate pointers from Tcl directly. The following code creates a pointer, points it at the last element of the list *l*, uses the pointer to get the value of the variable pointed at, and uses the pointer to set the value of the variable pointed at to *foo*. When we are done, $*l* contains *a b foo*.

```
set l [list a b c]
set p [pointer create l]
pointer index $p 2
pointer get $p          # prints c
pointer set $p foo
```

Let's now consider the widget example above, remembering that it had a variable reference &*e.textvar*. The compiler provides some magic to treat that construct as an L pointer. When the compiler sees a string constant of the form –.**variable*[2] and the next token is an L variable with a leading ampersand, the compiler automatically wraps the variable in an L pointer.

### 7.5.4. Return values

Because returns are by value in L, and Tcl also returns by value, no changes were required to make returns work in L.

It is worth noting, especially for C programmers, that there is a sneaky way to do an allocation. When a local variable is returned, the return bumps the reference count. Without that bump, the local variable in question would have been freed along with any other locals that were on the callee's stack. Tcl objects are reference counted so the variable will get freed when the caller is finished with it.

```
string[]
vector(int n)
{
    string  v[];

    // Allocate 0..n-1
    v[n - 1] = "";
    return (v);
}

string foo[] = v(100);
```

## 7.6. Casts

**(tcl).** There are times when we need to pass a compound object (array, hash) as a string. Any Tcl proc that expects to see a string on the stack will want this. The (*tcl*) cast is used to do this.

[2] Remember that : *foo* token is just syntactic sugar for "– *foo*."

```
string v[] = { "hi", "good day" };

puts((tcl)v);
```

prints

```
hi {good day}
```

**(L).** There may be times when a Tcl proc is returning a complex structure to us and we want to cast it from the Tcl list to our structure:

```
#lang(tcl)
proc demo {} {
    return [list {good day} sir]
}

#lang(L)
v = (L)demo();
printf("%s %s\n", v[0], v[1]);
```

prints

```
good day sir
```

Note: doing this sort of thing puts you at the mercy of the Tcl code which knows nothing about the L type system.

## 7.7. Operators

L supports most of the operators in the C programming language, as well as several of the most useful operators from Perl. In this section we do a quick run through all of the operators in L and discuss some of their more subtle aspects in depth.

Much of this section is cribbed from the C reference manual.[Kernighan1978a]

### 7.7.1. Arithmetic operators

The binary arithmetic operators in L are +, -, *, /, and % (modulus). They work as in C with the C precedence rules.

### 7.7.2. True vs. false

All of the relational and logical operators are part of an expression and that expression evaluates to either true or false.

In L, there is only one false value. This is different from Tcl, which allows many false values, such as the strings "false" and "off." The false value in L is 0, or, equivalently, "0". Any value other than 0 is considered true.

```
if (0) {
    printf("consequent\n");
} else {
    printf("alternative\n");
}
```

prints: `alternative`

#### 7.7.3. Numeric Comparison

These all work as in C with the C precedence rules.

**Relational operators**

*expr* `>` *expr*
*expr* `>=` *expr*
*expr* `<` *expr*
*expr* `<=` *expr*

**Equality operators**

*expr* `==` *expr*
*expr* `!=` *expr*

**Logical Operators**

The && and || operators short-circuit as in C.

*expr* `&&` *expr*
*expr* `||` *expr*
`!`*expr*

#### 7.7.4. Regular expression operators

Stolen from Perl, the first form is true if *regex* is a regular expression that matches *string*. The second form is true if *regex* is a regular expression that does not match *string*. The // construct is an alias for a double quoted string, which means that all or part of the string may be an interpolated variable (or expression). The *m*|| construct is also from perl; it means use the vertical bars instead of slashes (frequently useful when dealing with path names).

*string* `=~` `/`*regex*`/`
*string* `!~` `/`*regex*`/`
*string* `=~` `m|`*${expr}*`|`

#### 7.7.5. Increment and Decrement Operators

As in C, with the value returned either before or after the increment or decrement.

*lvalue*`++`
`++`*lvalue*
*lvalue*`--`
`--`*lvalue*

#### 7.7.6. Bitwise Operators

*expr* `&` *expr*
*expr* `|` *expr*
*expr* `^` *expr*
*expr* `<<` *expr*
*expr* `>>` *expr*
`~`*expr*

#### 7.7.7. Assignment Operators

*lvalue* `=` *expr*
*lvalue* `+=` *expr*
*lvalue* `-=` *expr*
*lvalue* `*=` *expr*
*lvalue* `/=` *expr*
*lvalue* `%=` *expr*
*lvalue* `<<=` *expr*
*lvalue* `>>=` *expr*
*lvalue* `&=` *expr*
*lvalue* `|=` *expr*
*lvalue* `^=` *expr*

#### 7.7.8. Ternary Operator

*expr* `?` *expr* `:` *expr*

### 7.8. Reserved Words

These are L's reserved words:

```
break case continue defined do
else float for foreach if int L
poly return string struct switch
tcl typedef unless until var void
while
```

### 7.9. Control flow

**Conditional statements**

`if (` *expr* `)` *statement*
`if (` *expr* `)` *statement* `else` *statement*
`unless (` *expr* `)` *statement*

In all cases *expr* is evaluated and if it returns anything other than zero, then the first **if** statement is executed. If it returns zero, then the **else** statement or the **unless** statement is executed.

**While/until statements**

`while (` *expr* `)` *statement*
`until (` *expr* `)` *statement*

The *expr* is evaluated and *statement* is executed repeatedly while *expr* is non-zero in the **while** case, or zero in the **until** case.

**do statements**

`do` *statement* `while (` *expr* `)`
`do` *statement* `until (` *expr* `)`

*statement* is executed repeatedly while *expr* is non-zero in the **while** case, or until non-zero in the **until** case.

**for statement**

for ( $exp1_{opt}$; $exp2_{opt}$; $exp3_{opt}$ ) *statement*

All expressions are optional. Other than the continue statement, which in this case executes *exp*3, this is the same as

```
exp1;
while ( exp2 ) {
        statement
        exp3;
}
```

**foreach statement**

```
foreach (h as key => val) statement
foreach (p in v) statement
```

The first statement iterates over each key/value pair in the hash *h*. The key/value pair is placed in *key* and *val* and then *statement* is executed. Behavior is undefined if keys are inserted or deleted in *h* in *statement*. The second statement sets *p* to each element of *v*, calling *statement* once per element.

**switch statement**

switch ( *expr* ) *statement*

*expr* must evaluate to an **int** or a **string**. Any statement within *statement* may contain one or more labeled statements of the form

```
case constant – expr: statement
case /constant – expr/: statement
case <constant – expr>: statement
```

There may be at most one statement of the form:

default: *statement*

When the **switch** statement is run, *expr* is evaluated and jumps to the **case** label that matches. Case labels may be double-quoted string constants, integer constants (not floats), constant regular expressions (/.*.[*ch*]/), or constant globs (< *.[*ch*] >). If no label matches, then if the **default** label exists, a jump to the **default** label occurs. As in C, control continues to flow past labels; see the "break statement" for exiting from a **switch**.

**break**

break ;

causes termination of the smallest enclosing **while**, **until**, **do**, **for**, or **switch** statement.

**continue**

continue ;

causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **until**, **do**, or **for** loop.

**return**

```
return;
return ( expr );
```

In the first case the return value is undefined. In the second, the return value is *expr*.

## 7.10. Changes to Tcl

In the course of implementing L, two small but important changes were made to Tcl that could affect all Tcl programs, although we don't expect the effects to be visible.

### 7.10.1. Top-level Compilation

Top-level code in Tcl, i.e., code that isn't contained in a proc body, is now passed to the byte-code compiler. We require this so that the L compiler can emit byte code for top-level L code. It could be useful in the future for saving Tcl byte code between invocations, similar to the TclPro compiler.

### 7.10.2. Changes to the Tcl Parser

The #*lang*(*tcl*) string forces the language to be Tcl, the #*lang*(*L*) forces the language to be L. It is allowed to have snippets of both L and Tcl in the same source file.

When Tcl starts up with a file argument, if the file ends in *.l* then #*lang*(*L*) is implicit. The default is to start up in Tcl mode.

Tcl's *Tcl_ParseCommand* has been modified to recognize a comment with a special form. Whenever the parser sees #*lang*(*L*) it stops normal parsing and inserts two tokens into the token stream. The first token is a call to the *LCompileCommand* function and the second is the text after the #*lang*(*L*) comment up to the next #*lang*(*tcl*) comment or end-of-file.

## 8. Status

The L language is under active development and the speed of development is increasing. Our expectation is that we will have a usable system in 1-2 months. Our goal is to be rewriting our GUI tools in L early in 2007. There is a mailing list, `l@bitmover.com`, and an IRC channel, `##l` on `Freenode`. People are welcome to join either.

## 9. Future work

### 9.1. Scoping

Like a C source file, a scope provides a container for private and/or public variables and/or functions. This could be used to provide a self-contained "class."

### 9.2. Pre-compiled modules

Imagine that each scope is a module and each module can be pre-compiled. The on-disk format is in sections: there is a byte-code section and a sort of table of contents which can be thought of as a header file containing function prototypes.

### 9.3. Optimizations

The dynamic nature of Tcl means that many traditional compiler optimization techniques cannot be used. L compiles the source to an abstract syntax tree and could take advantage of a number of well known optimizations. These include: constant subexpression elimination, dead code removal, strength reduction, loop invariant code motion, tail-call optimization, code hoisting, and others.[Muchnick1997a]

The lack of general C-like pointers in L greatly simplifies alias analysis and makes it possible to be more aggressive when applying optimizations.[Hind2001a, Marlowe1993a]

### 9.4. Debugging

The static nature of L code would make it possible to create a mapping between L source code and Tcl byte codes such that traditional debugging techniques could be used. One possible approach would be to instrument the generated byte code to invoke a debugger every time an L statement completes.

## 10. Licensing and availability

The license is the Tcl license; L is part of Tcl as far as we are concerned.

The source is maintained in a BitKeeper repository which is an import of the CVS Tcl repository. For the 3 people in the world who won’t use BK, we will do nightly tarballs and make them available on our FTP server.

## 11. Conclusion

This paper has described the L programming language. The L language is unique in that it is an alternate syntax which peacefully coexists with the Tcl/Tk system and leverages all of that system.

Over the course of the next year we expect to use L to rewrite our GUI systems. Given that L is a young language, we expect that it will continue to evolve as we use it. It is likely that we will publish an updated version of this paper after the language stabilizes.

## 12. Acknowledgements

The L language draws heavily from the C language. It’s hard to imagine that Brian, Dennis and Ken want any more pats on the back, but here is one more. We are definitely C fans.

Rob Netzer, Brian Griffin, and Mark Roseman were helpful in talking over various language problems and ideas.

John Ousterhout for Tcl/Tk, introduced in 1988 and still going strong.

Kennan Rossi for being there as always with editorial help.

This paper was typeset using groff and as always we thank Joe Ossana for troff and James Clark for groff.

## References

Brooks1975a.
Fred Brooks, *The Mythical Man Month* (1975).

Wiki2005a.
The Tcler’s Wiki, “shimmering,” *http://wiki.tcl.tk/3033* (Dec 2005).

Wiki2005b.
The Tcler’s Wiki, “use_ref,” *http://wiki.tcl.tk/15120* (Dec 2005).

Kernighan1978a.
Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc. (1978).

Muchnick1997a.
Steven S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann (1997).

Hind2001a.
Michael Hind, *Pointer Analysis: Haven’t We Solved This Problem Yet?"* (2001).

Marlowe1993a.
Thomas J. Marlowe, Jong-Deok Choi, William G. Landi, Michael G. Burke, Barbara G. Ryder, and Paul Carini, “Pointer-Induced Aliasing: A Clarification,” *ACM SIGPLAN Notices*, Volume 28, No. 9 (September 1993).

## Appendix - code samples

### A simple cat

```
int
main(int ac, string av[])
{
    int  i;
    FILE fd;

    if (ac == 1) {
        puts(:nonewline, read(stdin));
        return (0);
    }
    for (i = 1; defined(av[i]); i++) {
        fd = open(av[i], "r");
        puts(:nonewline, read(fd));
    }
}
```

### A simple grep

```
int
main(int ac, string av[])
{
    int     i, rc;
    string  regex;
    FILE    fd;

    if (ac < 2) {
        // Tcl’s [error]
        error("Not enough arguments.");
    }
    regex = av[1];
    ac--;
    if (ac == 1) {
        rc = grep(regex, stdin) ? 0 : 1;
        return (rc);
    } else {
     rc = 1;
        for (i = 2; i < ac; i++) {
            fd = open(av[i], "r");
            if (grep(regex, fd))) rc = 0;
            close(fd);
        }
        return (rc);
    }

}

int
grep(string regex, FILE in)
{
    string buf;
    int    matches = 0;

    while (gets(in, &buf) >= 0) {
        if (buf =~ /${regex}/) {
            printf("%s\n", buf);
         matches++;
        }
    }
    return (matches);
}
```

### Fibonacci

```
main()
{
    int fib[] = fib(100);

    for (i=0; defined(fib[i]); i++) {
        printf("%d\t%d\n", i, fib[i]);
    }
}

int[]
fib(int n)
{
    int    fib[] = { 0, 1 };
    int    i;

    for (i=2; i<n; i++) {
        fib[i] = fib[i-1] + fib[i-2];
    }
    return (fib);
}
```

### Quicksort

```
/*
 * qsort:
 * sort v[left]...v[right]
 * into increasing order.
 * From K&R C, verbatim.
 */
void qsort(int v[], int left, int right)
{
    int i, last;

    if (left >= right)
        return;
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i<= right; i++)
        if (v[i] < v[left])
            swap(v, ++last, i);
        swap(v, left, last);
        qsort(v, left, last-1);
        qsort(v, last+1, right);
}

/* swap: interchange v[i] and v[j] */
void swap(int v[], int i, int j)
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```