Lisp in 99 lines of C and how to write one yourself

Dr. Robert A. van Engelen

July 9, 2022; updated July 14, 2022

"In 1960, John McCarthy published a remarkable paper in which he did for programming something like what Euclid did for geometry. He showed how, given a handful of simple operators and a notation for functions, you can build a whole programming language. He called this language Lisp, for "List Processing," because one of his key ideas was to use a simple data structure called a list for both code and data." – Paul Graham [1]

1 Introduction

McCarthy's paper [2] not only showed how a programming language can be built entirely from lists as code and data, he also showed a function in Lisp that acts like an interpreter for Lisp itself. This function, called `eval`, takes as an argument a Lisp expression and returns its value. It was a remarkable discovery that Lisp can be written in Lisp itself. Lisp also introduced the concept of functions as first-class objects (*closures*) with *static scoping*[1], *runtime typing* and *garbage collection*. Features we now take for granted but were radical at that time. To put this into context, other programming languages at that time were Fortran (1957) and Algol (1958). Many new programming languages have appeared since. Most are still "Algol-like", or as some say, "C-like." The Lisp model of computation has regained momentum over the past decade. Contemporary programming languages now include Lisp-like "lambda functions." Lambda functions are syntactic materializations of *lambda abstractions* from the *lambda calculus* [3] introduced by Alonzo Church in 1944. It was lambda calculus that inspired McCarthy's to write Lisp.

In honor of the contributions made by Church and McCarthy, I wrote this article to show how anyone can write a tiny Lisp interpreter in a few lines of C or any "C-like" programming language. I attempted to preserve the original meaning and flavor of Lisp as much as possible. As a result, the C code in this article is strongly Lisp-like in compact form. Despite being small, these tiny Lisp interpreters in C include 20 built-in Lisp primitives, garbage collection and REPL, which makes them a bit more practical than a toy example. If desired, more Lisp features can be easily added with a few more lines of C as explained in this article with examples that are ready for you to try.

I encourage anyone to explore other Lisp implementations and their code. Many are cool with lots of features. Some are actually incorrect. Sometimes little nuggets surface when digging deeper to achieve perfection. This appears to be the case when writing this article, as it turns out that the list dot operator plays an important and useful role in lambda variable lists and arguments lists. In this case, no special forms are needed to achieve the same in pure Lisp. *I hope you will enjoy reading this article as much as I did writing it!*

[1] Originally *dynamic scoping*, which has some drawbacks.

Contents

Biography

Dr. Robert A. van Engelen is the CEO/CTO of Genivia.com, a US technology company he founded in 2003. He is a professor in Computer Science and Scientific Computing and worked for 20 years in the department of Computer Science at the Florida State University, where he also served as department chair. Van Engelen received the B.S. and the M.S. in Computer Science from Utrecht University, the Netherlands, in 1994 and the Ph.D. in Computer Science from the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University, the Netherlands, in 1998. His research interests include High-Performance Computing, Programming Languages and Compilers, Problem-Solving Environments for Scientific Computing, Cloud Computing, Services Computing, Machine Learning, and Bayesian Networks. Van Engelen?s research has been recognized with awards and research funding from the US National Science Foundation and the US Department of Energy. He published over 70 peer-reviewed technical publications in reputable international conferences and journals, He served as a member of the editorial board on the IEEE Transactions on Services Computing journal and has served on over 40 technical program committees for international conferences/workshops. Van Engelen is a senior member of the ACM and IEEE professional societies.

2 Understanding Lisp

Lisp programs are composed of anonymous functions written in the form of a `( )`-delimited list

(`lambda` *variables expression*)

where *variables* is a list of names denoting the function parameters and *expression* is the body of the function. Just like any other normal math function, lambdas don't do anything until we apply them to arguments. Application of a lambda to arguments is written as a list

(*function arguments*)

The application is performed in two steps. First, we bind the *variables* to the values of the corresponding *arguments*. Then the *expression* is evaluated. The *expression* may reference the function's *variables* by their name. The value of *expression* is "returned" as the result of the application.

Note that "return" is an imperative concept. Lisp has no imperative keywords. The entire Lisp language is built from functions and function applications, all using lists as syntax. Besides lists, Lisp also has symbols (names) for variables, primitives, functions (closures) and numbers. Lisp dialects may also include strings.

Function application is perhaps best illustrated with an example. Consider the function

```
(lambda (x y) (/ (- y x) x))
```

This function returns the value of $\frac{y-x}{x}$ for numeric arguments x and y. The first thing we note is that all arithmetic operations are written in functional form in Lisp. There is no need for any specific rules for operator precedence and associativity in Lisp. To apply our lambda to arguments, say 3 and 9, we write the list

```
((lambda (x y) (/ (- y x) x)) 3 9)
```

Spacing in Lisp is immaterial, so let's add some more spacing

```
( ( lambda (x y)
    (/ (- y x) x)
  )
  3 9
)
```

The Lisp interpreter binds `x` to 3 and `y` to 9, then evaluates the function body `(/ (- y x) x))` to compute 2 as the result of the application. Nice, isn't it?

But our function is not stored anywhere. What if we want to reuse it? After all, programs are composed of functions and those functions should be stored as part of a program to use them[2]. We can save our function by giving it a name using a `define`

```
(define subdiv (lambda (x y) (/ (- y x) x)))
```

and then apply `subdiv` to 3 and 9 with

```
(subdiv 3 9)
```

[2]The beauty of lambda calculus is that this is not an absolute requirement: lambda calculus is Turing-complete without named functions.

which displays 2. A defined name is not required to be alphabetic. Names are syntactically symbolic forms in Lisp. We could have named our lambda `-/` for example. Any sequence of characters can be used as a name, as long as it is distinguishable from a number and doesn't use parenthesis, quotes and whitespace characters.

The power and simplicity of Lisp's lambdas is better justified when we take a closer look at *closures*. A closure combines a function (a lambda) with an *environment*. An environment defines a set of name-value bindings. An environment is created (or extended) when the variables of a lambda are bound to the argument values in a lambda application. An environment provides a concrete mechanism to create a local scope of variables for the function body. Because lambdas are first-class objects and can therefore be returned as values by lambdas, environments play a crucial role to scope nested lambdas properly through *static scoping*. Consider for example the `adder` lambda that takes an `x` to return a new lambda that takes a `y` and adds them together:

```
(define make-adder (lambda (x) (lambda (y) (+ x y))))
```

Applying `make-adder` to 5 returns a closure in which the environment includes a binding of `x` to 5. When this closure is applied to 2 it returns 7 as expected:

```
> (define make-adder (lambda (x) (lambda (y) (+ x y))))
> ((make-adder 5) 2)
7
> (define add5 (make-adder 5))
> (add5 2)
7
```

Note that `make-adder` returns a closure that combines `(lambda (y) (+ x y))` with an environment in which `x` is bound to 5. The `x` in the lambda is not modified, it is Lisp code after all. When the closure is applied, the environment is extended to include a binding of `y` to 2. With this environment the function body `(+ x y)` evaluates to 7.

A handful of programming languages both correctly and safely implement the semantics of closures with static scoping. The implementation requires *unlimited extent* of non-local variables in scope to store bindings. Otherwise, non-local variables are "gone" as their values are removed from memory. This requires environments and garbage collection to remove them safely after the work is done. It doesn't suffice that functions can be syntactically nested within other functions.

Lisp also has a collection of built-in *primitives*. These are functions like `+` and *special forms* like `define`. A special form is a function that selectively evaluates its arguments rather than all of its arguments as in lambda applications. For example, `define` does not evaluate its *name* argument. Otherwise the value of the *name* would end up being used by `define` or an error is produced when *name* is not yet defined, which is more likely.

An overview of Lisp is not complete without a presentation of the basic primitives introduced in McCarthy's paper. We list them here and also include two more primitives `if` and `let` since these are often used in Lisp[3]. Some of the primitives listed below are special forms, namely `quote`, `cond`, `if` and `let`:

- `(quote` x`)` returns x unevaluated, "as is". Abbreviated `'`x.

[3] The `if` and `let` can be defined as macros, but we keep our Lisp interpreter simple and omit macro processing.

```
> (quote a)
a
> 'a
a
> '(a b c)
(a b c)
```

- `(cons` x y`)` returns the pair `(`x `.` y`)` where the dot is displayed if y is not the empty list `()`.

```
> (cons 'a 'b)
(a . b)
> (cons 'a ())
(a)
> (cons 'a (cons 'b (cons 'c ())))
(a b c)
```

- `(car` x`)` (*"Contents of the Address part of Register"*) returns the first element of the pair or list x.

```
> (car (cons 'a 'b))
a
> (car (cons 'a (cons 'b (cons 'c ()))))
a
```

- `(cdr` x`)` (*"Contents of tbe Decrement part of Register"*, pronounced "coulder") returns the second element of the pair x. When x is a list, the rest of the list is returned after the first element.

```
> (cdr (cons 'a 'b))
b
> (cdr (cons 'a (cons 'b (cons 'c ()))))
(b c)
```

- `(eq?` x y`)` returns the atom `#t` (representing true) if the values of x and y are identical. Otherwise returns `()` representing false.

```
> (eq? 2 2)
#t
> (eq? 2 3)
()
> (eq? 'a 'a)
#t
```

- `(cond (`x_1 y_1`) (`x_2 y_2`) ... (`x_n y_n`))` evaluates x_i from left to right until x_i is not the empty list (i.e. is true), then returns the corresponding value of y_i.

```
> (cond ((eq? 'a 'b) 1) ((eq? 'b 'b) 2))
2
> (cond (() 1) (#t 2))
2
```

- `(if` x t e`)` if x is not the empty list (i.e. is true), then the value of t is returned else the value of e is returned. `(if` x t e`)` is a shorthand for `(cond (`x t`) (#t` e`))`.

```
> (if 'a 1 2)
1
> (if () 1 2)
2
> (if (eq? 'a 'a) 'ok 'fail)
ok
```

- `(let (`v_1 x_1`) (`v_2 x_2`) ... (`v_n x_n`)` y`)` evaluates x_i from left to right and binds each variable v_i to the value of x_i to extend the environment to the body y, then returns the value of y. The same is accomplished with `((lambda (`v_1 v_2 `...` v_n`)` y`)` x_1 x_2 `...` x_n`)`.

```
> (let (x 3) (y 9) (/ (- y x)))
2
```

Lisp implementations include more primitives, notably for arithmetic, logic and runtime type checking. Most Lisp implementation define additional Lisp primitives in Lisp itself.

3 Lisp Expressions as Tagged Structures

Lisp expressions are composed of numbers, atoms (names and symbols), strings (when implemented), primitives, cons pairs and closures. A Lisp expression type can be conveniently defined in C as a tagged union:

```
struct Expr {
  enum { NMBR, ATOM, STRG, PRIM, CONS, CLOS, NIL } tag;
  union {
    double number;              /* NMBR: double precision float number */
    const char *atom;           /* ATOM: pointer to atom name on the heap */
    const char *string;         /* STRG: pointer to string on the heap */
    struct Expr (*fn)(struct Expr, struct Expr);    /* PRIM: built-in primitive */
    struct Expr *cons;          /* CONS: pointer to (car,cdr) pair on the heap */
    struct Expr *closure;       /* CLOS: pointer to closure pairs on the heap */
  } value;
};
```

However, rather than storing this information elaborately in a structure, we can exploit *NaN boxing* to store this information in an IEEE-754 single or double precision float because all structure members are pointers that are essentially unsigned integer offsets from a base address.

3.1 NaN Boxing

The idea behind NaN boxing is that floating point NaN *Not-a-Number* values are not unique. A double precision NaN allows up to 52 bits to be arbitrarily used to stuff any information we want into a double precision NaN:

$$| \; s \; \overbrace{b\;b\;b \mid b\;b\;b\;b \mid b\;b\;b\;b}^{\textit{2's complement exponent}} \; | \overbrace{b\; \underbrace{b\;b\;b}_{\textit{tag}} \mid \underbrace{\cdots \mid b\;b\;b\;b}_{\textit{also available}}}^{\textit{fraction}} \; |$$

where

s is the sign bit of the float, $s = 1$ for negative numbers

2's complement exponent consists of 11 bits to represent binary exponents -1022 to 2023, when the bits are all 1 the value is NaN or INF

fraction consists of 52 bits with an invisible 1 bit as the leading digit of the mantissa

The *tag* and other data can be stored in the freely available 52 bit fraction part of a NaN. However, we want to use *quiet NaNs* which means that the first bit of the fraction (the bit before the *tag* bits) must be 1, leaving 51 bits and the sign bit for both the *tag* and other data to our disposal. This is plenty of space in a NaN-boxed double precision float to store a tag to identify atoms, strings, primitives, cons pairs, and closures together with their pointers and/or integer indices. In all, 48 bits are available to store an integer, or 49 bits when including the sign bit.

In this article I will also describe a Lisp implementation for the Sharp PC-G850 vintage pocket computer, which poses a bit of a challenge since it does not use IEEE 754 floating point representations. Instead, floating point values are represented internally in BCD (Binary Coded Decimal). This raises the question: *can we use similar tricks as NaN boxing with BCD floats?* Let's find out.

3.2 BCD Float Boxing

To understand if and how we can exploit BCD floats to store information other than floating point numbers, we will take a closer look at the PC-G850's internal decimal floating point representation. A decimal floating point value is stored in 8 bytes. The first 2 bytes store the exponent in BCD and the sign of the number. The next 5 bytes store the 10 digit BCD mantissa followed by a zero byte:

$$| \; \overbrace{b\; \underbrace{b\;b\;b}_{\textit{tag}} \mid b\;b\;b\;b \mid b\;b\;b\;b}^{\textit{10's complement BCD exponent}} \; | \; \overbrace{s\;d\;u\;u}^{\textit{control}} \; | \; \overbrace{b\;b\;b\;b \mid \cdots \mid b\;b\;b\;b}^{\textit{10 digit BCD mantissa}} \; | \; \overbrace{0\;0\;0\;0 \mid 0\;0\;0\;0}^{\textit{2 BCD guard digits}} \; |$$

where

s is the sign bit of the float, $s = 1$ for negative numbers

d is the degree bit, $d = 1$ to display degrees in $D^oM'S.S$" format in BASIC

u is an unused bit, likely a mantissa carry bit used by the system

10's complement BCD exponent consists of 12 bits for 3 BCD digits to represent exponents -99 (901 BCD) to 99 (099 BCD)

10 digit BCD mantissa consists of 10 BCD digits with the normalized mantissa, the leading BCD digit is nonzero unless all mantissa digits are zero

2 BCD guard digits are always zero after internal rounding to 10 significant digits

To explore opportunities to exploit *BCD float boxing*[4] to store information other than decimal floating point numbers, we can write some C code for testing. Unfortunately, we cannot use the mantissa or its trailing *guard digits* to store extra information. The mantissa is always normalized to BCD and the guard digits are always reset to zero when passing floats through functions, even when no arithmetic operations are applied to the float. Our target bits to box a *tag* with data in a float are the three bits in the upper half of the leading byte of the float. These three bits of the float remain unmodified when passing the floating point value through functions as arguments and as return values. A quick test confirms our hypothesis, with some caveats:

```
double func(double x) { return x; }
int main() {
  double x,y,z; char *p = (char*)&x,*q = (char*)&y; int i;
  scanf("%lg",&z);                          /* input a value z to check */
  for (i = 0x20; i <= 0x70; i += 0x10) {
    x = z; *p |= i;                         /* set x to z and set its tag bits */
    x = func(x);                            /* pass x through func() */
    y = z; *q |= i;                         /* set y to z and set its tag bits */
    if (x != y)                             /* x and y should be equal */
      printf("fail %x\n",i);
  }
}
```

The first caveat is that tag 000 (`i==0x00`) cannot be used. This tagged value is indistinguishable from a normal float value. Second, tag 001 (`i==0x10`) cannot be used because all tagged floats appear to fail with an arithmetic error when passed to a function, perhaps because the tagged value corresponds to a non-normalized carry digit in the exponent. Third, all tagged floats with values $|x| < 10$ are normalized to zero and thus fail this test. This type of failure happens when the two-digit BCD exponent is zero and the third highest order BCD exponent digit nonzero, thus representing a non-normalized two-digit zero BCD exponent.

After confirming our hypothesis by observation, we conclude that we have six possible tag bit patterns 010 to 111 to our disposal, as long as we box integers as tagged floats $|x| \geq 10$. This is not a problem, because we can simply multiply a value by 10 before boxing and divide by 10 after unboxing. When boxing unsigned integers, such as pointers and array indices, it suffices to add 10 before boxing and subtract 10 after unboxing. So we will use this boxing method.

3.3 Types of Lisp Expressions

We define two types[5] in C that we exclusively use in our Lisp interpreter: a nice and simple type `I` for unsigned integers and a type `L` for Lisp expressions defined as floats for NaN or BCD float boxing:

[4] I think the term "BCD float boxing" nails it, but Google shows sports gear.

[5] PC-G850 C does not support `typedef`. We use `#define` instead.

```
#define I unsigned
#define L double
```

We define five tags for ATOMs, PRIMitives, CONStructed pairs, CLOSures and NIL (the empty list):

```
  /*** with NaN Boxing ***/
I ATOM=0x7ff8,PRIM=0x7ff9,CONS=0x7ffa,CLOS=0x7ffb,NIL=0x7ffc;
  /*** with BCD boxing ***/
I ATOM=32,PRIM=48,CONS=64,CLOS=80,NIL=96;
```

To access the tag bits of a tagged float we cast the pointer to the float to a `char*` using a nice short and sweet `T`:

```
  /*** with NaN Boxing ***/
#define T(x) *(uint64_t*)&x>>48
  /*** with BCD boxing ***/
#define T *(char*)&
```

We use this to set a tag with `T(x) =` *tag* and to retrieve a tag with `T(x)` for Lisp expression `L x`. Instead of `uint64_t` to cast the 64 bit double in `T(x)` we can use `unsigned long long` instead, which is typically 64 bits. We also need the following two functions to manipulate tagged floats and their ordinal content:

```
  /*** with NaN Boxing ***/
L box(I t,I i) { L x; *(uint64_t*)&x = (uint64_t)t<<48|i; return x; }
I ord(L x) { return *(uint64_t*)&x; }
  /*** with BCD boxing ***/
L box(I t,I i) { L x = i+10; T(x) = t; return x; }
I ord(L x) { T(x) &= 15; return (I)x-10; }
```

The `box` function returns a float tagged with the specified tag `t` as `ATOM`, `PRIM`, `CONS`, `CLOS` or `NIL` and by boxing unsigned integer `i` as ordinal content. For BCD boxing we must add 10 to `i` to avoid the aforementioned caveat when boxing values in BCD floats. The `ord` function unboxes the unsigned integer (ordinal) of a tagged float. For BCD boxing we first untag the float with `T(x) &= 15` then subtract 10 to return the boxed ordinal content of the tagged float.

We should be able to perform arithmetic on floats in our Lisp interpreter. To do so, we could simply assume that the arguments and operands to arithmetic operations are always untagged floats. However, to make sure we aren't applying arithmetic operations on tagged floats by accident, we should define a new function `num` to clear two of the three tag bits first, before applying arithmetic operations:

```
  /*** with NaN Boxing ***/
L num(L n) { return n; }
  /*** with BCD boxing ***/
L num(L n) { T(n) &= 159; return n; }
```

With NaN boxing the number is returned "as is", but we could check if `n` is a NaN and take some action. For now, we just pass NaNs to perform arithmetic on, which results in a NaN. For BCD boxing we clear two bits of the tag with `T(n) &= 159` (`0x9f`) since negative BCD exponents are represented in BCD 9*dd*. The high-order digit 9 (binary 1001) should be preserved.

Checking if two values are equal is performed with the `equ` function. Because equality comparisons `==` with NaN values always produces false, we just need to compare the 64 bits of the values for equality:

```
  /*** with NaN Boxing ***/
I equ(L x,L y) { return *(uint64_t*)&x == *(uint64_t*)&y; }
  /*** with BCD boxing ***/
I equ(L x,L y) { return x == y; }
```

Note that BCD boxing does not require `equ`, but we include it here since this may depend on the BCD arithmetic performed by the machine.

Checking if a Lisp expression is `nil` (the empty list) only requires checking its tag for `NIL`:

```
I not(L x) { return T(x) == NIL; }
```

The `not` function comes in handy later when we implement conditionals, since `nil` is considered *false* in Lisp. Anything else is implicitly *true* in Lisp.

The C functions we defined here are the only ones specific to NaN or BCD float boxing. The rest of the Lisp interpreter is independent of the tagging method used.

4 Constructing Lisp Expressions

Lisp expressions are composed of atoms (also called symbols in Lisp), primitives, cons pairs, closures and `nil`. The `nil` constant represents the empty list `()` in Lisp. The `nil` constant is also considered *false* in Lisp conditionals. Furthermore, we have two pre-defined atoms, namely `#t` and `ERR`. The `#t` atom will be used as an explicit *true* in Lisp, although any value other than `nil` is implicitly *true* in Lisp conditionals. The `ERR` atom represents an error and is returned to the user when an expression evaluates to an error. These three constants are globally declared since we will often use them in the internals of our Lisp interpreter. They are initialized[6] in the `main` function as follows:

```
L nil,tru,err;
...
int main() {
  ...
  nil = box(NIL,0); tru = atom("#t"); err = atom("ERR");
  ...
}
```

where we used the `box` function defined in Section 3.2. The `atom` function returns an `ATOM`-tagged float that is globally unique. The `atom` function checks if the atom name already exists on the heap

[6] We initialize globals in `main` since PC-G850 C does not support initialization of globals with non-constants, e.g. function calls cannot be used.

and returns the heap index corresponding to the atom name boxed in the `ATOM`-tagged float. If the atom name is new, then additional heap space is allocated to copy the atom name into the heap as a string. The heap index of the new atom name is boxed in the `ATOM`-tagged float and returned by the `atom` function:

```
L atom(const char *s) {
  I i = 0; while (i < hp && strcmp(A+i,s)) i += strlen(A+i)+1;
  if (i == hp && (hp += strlen(strcpy(A+i,s))+1) > sp-1 << 3) abort();
  return box(ATOM,i);
}
```

where `hp` is the *heap pointer* pointing to free bytes available on the heap, `A` is the starting byte address of the heap and `sp` is the *stack pointer* pointing to the top of the stack of Lisp values (tagged and untagged floats `L`):

```
#define A (char*)cell
#define N 1024
L cell[N];
I hp = 0,sp = N;
```

The `cell[N]` array of 1024 (tagged) floats contains both the heap and the stack. The value of `N` can be increased to pre-allocate more memory.

Note that the `atom` function searches the heap at addresses `A+i` up to `i==hp` until a matching atom name is found to return `box(ATOM,i)`. If the atom name is new, then `i==hp` and space for the atom's string name is allocated at `A+hp`, then copied into this space with a terminating zero byte. In this way, atoms constructed with `box(ATOM,i)` are globally unique. The method by which we construct them by looking them up in a pool of names is often referred to as *interning*.

The heap grows upward towards the stack. The stack grows downward, as stacks usually do. The remaining free space is available between the heap and stack. For example, the table below depicts the memory configuration after pushing a pair of cells `box(ATOM,0)` and `nil` on the stack and storing two atoms `#t` and `ERR` in the heap:

`cell[1023]`:	`box(ATOM,0)`	*stack*
`cell[sp=1022]`:	`nil`	↓
	free	
`A+hp=A+7`:	*space*	
`A+3`:	"`ERR\0`"	↑
`A+0`:	"`#t\0`"	*heap*

What's actually on the stack here is the Lisp list `(#t)` containing one element `#t` in the list. This list is represented by `box(CONS,1022)` where 1022 is the stack index of the `cdr` cell of this list pair. The cell above it on the stack contains the `car` of this list pair. Lisp uses linked lists with the `car` of a list node (a cons pair) containing the list element and `cdr` pointing to the next cons pair in the list.

With this memory configuration in mind, constructing cons pairs is easy. We just allocate two cells on the stack, copy the values therein and return `box(CONS,sp)` since `sp` points to the `cdr` cell:

```
L cons(L x,L y) {
  cell[--sp] = x;
  cell[--sp] = y;
  if (hp > sp-1 << 3) abort();
  return box(CONS,sp);
}
```

If the `hp` and `sp` pointers meet we ran out of memory and we should `abort` or take some other action. Note that `hp` points to bytes whereas `sp` points to 8-byte floats. Therefore, the out-of-memory condition is checked in the `atom` and `cons` functions by scaling `sp` by a factor 8 in the conditional `if (hp > sp-1 << 3) abort()`.

Deconstructing a cons pair is trivial. We just need to get the `cell` of the `car` or `cdr` indexed by `i` in `box(CONS,i)` by retrieving it with the `ord` function:

```
L car(L p) { return (T(p) & ~(CONS^CLOS)) == CONS ? cell[ord(p)+1] : err; }
L cdr(L p) { return (T(p) & ~(CONS^CLOS)) == CONS ? cell[ord(p)] : err; }
```

where `ord(p)` is the cell index of the `cdr` of the cons pair `p` with the `car` cell located just above it. However, we do not trust the argument `p` to be a cons pair. The condition `T(p) & ~(CONS^CLOS)) == CONS` guards valid `car` and `cdr` function calls on cons pairs. The condition is true if `p` is a cons pair or a closure pair. Closure pairs are just cons pairs tagged as closures. The `~(CONS^CLOS)` mask is an efficient way to check for both `CONS` and `CLOS` tags in one comparison, because the tag values of `CONS` and `CLOS` were carefully chosen to differ by only one bit.

For example, suppose `p = box(CONS,sp)` representing the list `(#t)` with the cells on the stack depicted in the memory configuration shown previously. Then `car(p)` returns `box(ATOM,0)` representing `#t` and `cdr(p)` returns `nil`.

Closures and environment lists are constructed with the `cons` function applied twice, first to construct the name-value pair, then to add the pair in front of the environment list. We define a `pair` function for this purpose:

```
L pair(L v,L x,L e) { return cons(cons(v,x),e); }
```

Closures are actually `CONS`-tagged pairs representing instantiations of Lisp functions of the form `(lambda v x)` with either a single atom `v` as a variable referencing a list of arguments passed to the function, or `v` is a list of atoms as variables, each referencing the corresponding argument passed to the function. Closures include their static scope as an environment `e` to reference the bindings of their parent functions, if functions are nested, and to reference the global static scope:

```
L closure(L v,L x,L e) { return box(CLOS,ord(pair(v,x,equ(e,env) ? nil : e))); }
```

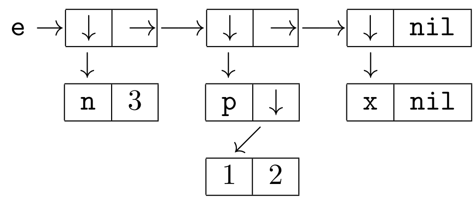
The conditional `equ(e,env) ? nil : e` forces the scope of a closure to be `nil` if `e` is the global environment `env`. Later, when we apply the closure, we check if its environment is `nil` and use the current global environment. This permits recursive calls and the calling of forward-defined functions, because the current global environment includes the latest global definitions.

An *environment* in Lisp is implemented as a list of name-value associations, where names are Lisp atoms. Environments are searched with the `assoc` function given an atom `a` and an environment `e`:

```
L assoc(L a,L e) {
  while (T(e) == CONS && !equ(a,car(car(e)))) e = cdr(e);
  return T(e) == CONS ? cdr(car(e)) : err;
}
```

The `assoc` function returns the Lisp expression associated with the specified atom in the specified environment `e` or returns `err` if not found.

Consider for example the environment `e = ((n . 3) (p . (1 . 2)) (x . nil))` where the Lisp expression `(1 . 2)` constructs a pair[7] of 1 and 2 instead of a list. This example environment `e` is represented in memory as follows with boxes for cells and arrows for `CONS` indices pointing to cells:



Note that a `CONS` index points to two cells on the stack, the `car` and `cdr` cells.

5 Evaluating Lisp Expressions

A Lisp expression is either a number, an atom, a primitive, a cons pair, a closure, or nil. Numbers, primitives, closures and nil are constant and returned by `eval` as is. Atoms are evaluated by returning their associated value from the environment using the `assoc` function, see Section 4. The environment includes function parameters and global definitions. Lists are evaluated by applying the first element in the list as a function to the rest of the list as arguments passed to that function:

```
L eval(L x,L e) {
  return T(x) == ATOM ? assoc(x,e) :
         T(x) == CONS ? apply(eval(car(x),e),cdr(x),e) :
         x;
}
```

Note that Lisp expression `x` evaluates to the value `assoc(x,e)` of `x` when `x` is an atom, or evaluates to the result of a function application `apply(eval(car(x),e),cdr(x),e)` if `x` is a list, or evaluates to `x` itself otherwise. A function application requires evaluating the function `eval(car(x),e)` first before applying it, because `car(x)` may be an expression that returns a function such as an atom associated with a Lisp primitive or the closure constructed for a `lambda`. The `apply` function applies the primitive or the closure `f` to the list of arguments `t` in environment `e`:

```
L apply(L f,L t,L e) {
  return T(f) == PRIM ? prim[ord(f)].f(t,e) :
         T(f) == CLOS ? reduce(f,t,e) :
         err;
}
```

[7] Basically this pair is `p = cons(1,2)` whereas a list always ends in `nil` such as `cons(1,cons(2,nil))`.

The `prim[]` array contains all pre-defined primitives. A primitive is stored in a structures with its name as a string and a function pointer. To call the primitive we invoke the function pointer with `prim[ord(f)].f(t,e)`. See also Section 6. If `f` is a closure then `reduce`[8] is called to apply closure `f` to the list of arguments `t`:

```
L reduce(L f,L t,L e) {
  return eval(cdr(car(f)),bind(car(car(f)),evlis(t,e),not(cdr(f)) ? env : cdr(f)));
}
```

where `f` is the closure and `t` is the list of arguments. The outer `eval` evaluates the body of closure `f` retrieved by `cdr(car(f))`. The evaluation is performed with an extended environment to include parameter bindings to the evaluated arguments. The arguments are evaluated with the `evlis` function:

```
L evlis(L t,L e) {
  return T(t) == CONS ? cons(eval(car(t),e),evlis(cdr(t),e)) :
         eval(t,e);
}
```

where `evlis` recursively traverses the list `t` to create a new list with values. Recursion bottoms out at a non-`CONS` `t`, which is evaluated. It is important to handle the dot in argument lists correctly, such as `(f x . args)` by calling `eval(t,e)` to evaluate `args`.

The variable-argument bindings for `apply` are constructed as a list of pairs with the `bind` function originally called `pairlis`:

```
L bind(L v,L t,L e) {
  return T(v) == NIL ? e :
         T(v) == CONS ? bind(cdr(v),cdr(t),pair(car(v),car(t),e)) :
         pair(v,t,e);
}
```

where `v` is the list of variables or a variable (an atom), and `t` is the list of evaluated arguments. When recursion bottoms out, we either have a `NIL` or a name that is bound with `pair(v,t,e)`. The latter happens when a single variable is used like `(lambda args args)` and when a dot is used like `(lambda (x . args) args)`.

6 Lisp Primitives

The Lisp primitives are defined in an array `prim[]` of structures containing the name of the primitive as string `s` and the function pointer `f` pointing to the implementation in C. The function implementing the primitive takes the list of Lisp arguments as the first parameter and the Lisp environment as its second parameter:

[8] Viz. lambda calculus *beta reduction* involves a *contraction* step $(\lambda v.x)\, y \Rightarrow x[v := y]$ where y may or may not be evaluated first before the contraction. This models strict and lazy evaluation, respectively.

```
struct { const char *s; L (*f)(L,L); } prim[] = {
{"eval",  f_eval},
{"quote", f_quote},
{"cons",  f_cons},
{"car",   f_car},
{"cdr",   f_cdr},
{"+",     f_add},
{"-",     f_sub},
{"*",     f_mul},
{"/",     f_div},
{"int",   f_int},
{"<",     f_lt},
{"eq?",   f_eq},
{"or",    f_or},
{"and",   f_and},
{"not",   f_not},
{"cond",  f_cond},
{"if",    f_if},
{"let*",  f_leta},
{"lambda",f_lambda},
{"define",f_define},
{0}};
```

The `main` program initializes the global environment `env` with `#t` to return itself `#t`, followed by the Lisp primitives:

```
int main() {
  ...
  env = pair(tru,tru,nil);
  for (i = 0; prim[i].s; ++i) env = pair(atom(prim[i].s),box(PRIM,i),env);
  ...
}
```

Lisp includes so-called *special forms*, which are functions that do not evaluate all arguments passed to them. For example, the `if` special form evaluates the test. If the test is true, then the then-expression is evaluated and returned. Otherwise the else-expression is evaluated and returned.

6.1 `eval`

(`eval` *expr*) evaluates an expression. This primitive is also called "unquote" since *expr* is typically a quoted Lisp expression:

```
L f_eval(L t,L e) { return eval(car(evlis(t,e)),e); }
```

All arguments to `eval` are evaluated with `evlis(t,e)` but `eval` only applies to one argument or the first argument when more than one is specified. Example: `(eval (quote (+ 1 2)))` gives 3.

6.2 `quote`

`(quote` *expr*`)` quotes an expression to keep it unevaluated. The Lisp parser also accepts `'`*expr*. Note that *expr* may be a list which means that the list and all of its elements remain unevaluated.

```
L f_quote(L t,L _) { return car(t); }
```

`quote` only applies to one argument or the first argument when more than one is specified, hence `car(t)` is returned. Example: `(quote (1 2 3))` and `'(1 2 3)` give `(1 2 3)`

6.3 `cons`

`(cons` $expr_1$ $expr_2$`)` constructs a new pair `(`$expr_1$ `.` $expr_2$`)`. Typically $expr_2$ is a list to construct a list with $expr_1$ at its head.

```
L f_cons(L t,L e) { return t = evlis(t,e),cons(car(t),car(cdr(t))); }
```

Example: `(cons 1 ())` gives `(1)`.

6.4 `car` and `cdr`

`(car` *pair*`)` and `(car` *pair*`)` give the first and second element of a *pair*, respectively. This means that `(car` *list*`)` and `(car` *list*`)` give the element at the front of the *list* and the rest of the *list*, respectively.

```
L f_car(L t,L e) { return car(car(evlis(t,e))); }
L f_cdr(L t,L e) { return cdr(car(evlis(t,e))); }
```

6.5 Arithmetic

The four basic arithmetic operations are variadic functions:

```
L f_add(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n += car(t); return num(n); }
L f_sub(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n -= car(t); return num(n); }
L f_mul(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n *= car(t); return num(n); }
L f_div(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n /= car(t); return num(n); }
```

Example: `(+ 1 2 3 4)` gives 10, `(- 3 2)` gives 1, and `(- 3)` gives 3, not -3 (some other Lisp may give -3, which can be implemented by checking if only one argument is passed to the operator).

6.6 `int`

`(int` *expr*`)` truncates *expr* to an integer.

```
L f_int(L t,L e) { L n = car(evlis(t,e)); return n-1e9 < 0 && n+1e9 > 0 ? (long)n : n; }
```

6.7 Comparison

(< $expr_1$ $expr_2$) and (eq? $expr_1$ $expr_2$) compare $expr_1$ and $expr_2$ to give `#t` when true or `()` when false.

```
L f_lt(L t,L e) { return t = evlis(t,e),car(t) - car(cdr(t)) < 0 ? tru : nil; }
L f_eq(L t,L e) { return t = evlis(t,e),equ(car(t),car(cdr(t))) ? tru : nil; }
```

Equality for pairs and lists is only true if the lists are the same objects in memory. Otherwise the pairs or lists are not equal, even when they contain the same elements. Example: `(eq? 'a 'a)` gives `#t` and `(eq? '(a) '(a))` gives `()`.

6.8 Logic

(not *expr*) gives `#t` when *expr* is `()` (the empty list) and `()` otherwise. (or $expr_1$ $expr_2$ ... $expr_n$) gives `#t` if any of the $expr_i$ is true (i.e. not `()`) and `()` otherwise if all $expr_1$ are false (i.e. `()`). (and $expr_1$ $expr_2$ ... $expr_n$) gives `()` if any of the $expr_i$ is false (i.e. `()`) and `#t` otherwise if all $expr_1$ are true (i.e. not `()`).

```
L f_not(L t,L e) { return not(car(t = evlis(t,e))) ? tru : nil; }
L f_or(L t,L e) { for (; T(t) != NIL; t = cdr(t)) if (!not(eval(car(t),e))) return tru; return nil; }
L f_and(L t,L e) { for (; T(t) != NIL; t = cdr(t)) if (not(eval(car(t),e))) return nil; return tru; }
```

Only the first arguments to `or` and `and` are evaluated to determine the result. Example: `(and #t ())` gives `()` and `(and 1 2)` gives `#t` since numbers are not `()`.

6.9 `cond`

(cond ($test_1$ $expr_1$) ($test_2$ $expr_2$) ... ($test_n$ $expr_n$)) evaluates the tests from the first to the last until $test_i$ evaluates to true and then returns $expr_i$.

```
L f_cond(L t,L e) {
  while (T t != NIL && not(eval(car(car(t)),e))) t = cdr(t);
  return eval(car(cdr(car(t))),e);
}
```

Example: `(cond ((eq? 'a 'b) 1) ((< 2 1) 2) (#t 3))` gives 3.

6.10 `if`

(if *test* $expr_1$ $expr_2$) evaluates and tests if *test* is true or false. If true (i.e. not `()`) then $expr_1$ is evaluated and returned. Else $expr_2$ is evaluated and returned.

```
L f_if(L t,L e) { return eval(car(cdr(not(eval(car(t),e)) ? cdr(t) : t)),e); }
```

Example: `(if (eq? 'a 'a) 1 2)` gives 1.

6.11 `let*`

`(let* (`var_1 $expr_1$`) (`var_2 $expr_2$`) ... (`var_n $expr_n$`)` *expr*`)` defines a set of bindings of variables in the scope of the body *expr* by evaluating each $expr_i$ sequentially and associating var_i with the result.

```
I let(L x) { return T(x) != NIL && (x = cdr(x),T(x) != NIL); }
L f_leta(L t,L e) {
  while (let(t)) e = pair(car(car(t)),eval(car(cdr(car(t))),e),e),t = cdr(t);
  return eval(car(t),e);
}
```

The loop runs over the pairs in the `let*`. Each iteration extends the environment `e` with a pair that binds var_i (i.e. `car(car(t))`) to the value of $expr_i$ (i.e. `eval(car(cdr(car(t))),e)`. Example: `(let* (a 3) (b (* a a)) (+ a b))` gives 12.

6.12 `lambda`

`(lambda` *var* *expr*`)` and `(lambda (`var_1 var_1 `...` var_n`)` *expr*`)` create a closure, i.e. an anonymous function. The first form associates *var* with the list of arguments passed to the closure when the closure is applied. The second form associates each var_i with the corresponding argument passed to the closure when the closure is applied. Also the list dot may be used `(lambda (`var_1`_`var_2`)` *expr*`)` to specify the remaining arguments to be passed as a list in `var`$_2$.

```
L f_lambda(L t,L e) { return closure(car(t),car(cdr(t)),e); }
```

Example: `((lambda (x) (* x x)) 3)` gives 9 and `((lambda (x y . args) args) 1 2 3 4)` gives `(3 4)`.

6.13 `define`

`(define` *var* *expr*`)` globally defines *var* and associates it with the evaluated *expr*.

```
L f_define(L t,L e) { env = pair(car(t),eval(car(cdr(t)),e),env); return car(t); }
```

Globally defined functions may be (mutually) recursive. Example: after `(define pi 3.14)` the value of `pi` is 3.14, after `(define square (lambda (x) (* x x)))` the application `(square 3)` gives 9 and after `(define factorial (lambda (n) (if (< 1 n) (* n (factorial (- n 1))) 1)))` the application `(factorial 5)` gives 120.

We have not defined an `apply` primitive often found in Lisp implementations, because `apply` is not needed. To apply a function to a list of arguments `(f . args)` suffices, but only if `args` is a variable associated with a list of arguments[9]. Otherwise, use a `(let (args` x`) (f . args))`.

[9] If we place a list after the dot like `(f . (g args))`, then the arguments passed to `f` are actually `g` and `args`.

7 Reading and Parsing Lisp Expressions

A Lisp tokenizer scans the input for tokens to return to the parser. A token is a single parenthesis, the quote character, or a white space delimited sequence of characters up to 39 characters long. The `scan` function populates `buf[40]` with the next token scanned from standard input. The token is stored as a 0-terminated string in `buf[]`:

```
char buf[40],see = ' ';
void look() { see = getchar(); }
I seeing(char c) { return c == ' ' ? see > 0 && see <= c : see == c; }
char get() { char c = see; look(); return c; }
char scan() {
  char *tok = buf;
  while (seeing(' ')) look();
  if (seeing('(') || seeing(')') || seeing('\'')) *tok++ = get();
  else do *tok++ = get(); while (!seeing('(') && !seeing(')') && !seeing(' '));
  return *tok = 0,*buf;
}
```

This implementation of `scan` does not support Lisp comments. Lisp comments begin with a semi-colon and end at the next line. To support comments, we can change the second line of the `scan` function by adding a label[10] `s:` and by adding a line to skip comments:

```
s:while (seeing(' ')) look();
  if (seeing(';') { do look(); while (!seeing('\n')); goto s; }
```

Note that EOF is not checked. A nice trick is to look for an EOF and then reopen standard input to read from the terminal:

```
void look() {
  int c = getchar();
  if (c == EOF) freopen("/dev/tty", "r", stdin), c = ' ';
  see = c;
}
```

By changing `look` this way, we can now read a collection of Lisp definitions from a file before the interactive session starts using the Linux/Unix `cat` utility:

```
bash$ cat common.lisp list.lisp math.lisp | ./tinylisp
```

This sends the `common.lisp`, `list.lisp` and `math.lisp` files (see Section E) to the interpreter. After EOF of `cat`, the Lisp interpreter is ready to accept input again, this time from the terminal.

To parse Lisp expressions we employ a *recursive-descent parsing* technique. The `read` function returns a Lisp expression parsed from the input by invoking the `scan` and `parse` functions:

[10] Goto's are not a bad to use if the code is actually *more readable* with them than without them, despite Edsgar Dijkstra's aversion to goto, there is no harm in using gotos appropriately.

```
L read() { return scan(),parse(); }
```

where the `parse` function parses a list, a quoted expression, or an atomic expression (an atom or a number):

```
L parse() { return *buf == '(' ? list() : *buf == '\'' ? quote() : atomic(); }
```

The `list` function recursively parses and constructs a list up to the closing parenthesis `)`. In addition, a dot in a list creates a pair:

```
L list() {
  L x;
  return scan() == ')' ? nil :
         !strcmp(buf, ".") ? (x = read(),scan(),x) :
         (x = parse(),cons(x,list()));
}
```

Note that `x = parse()` must be called before the parsed value is used in `cons(x,list())`, because argument evaluation order in C is undefined, i.e. not necessarily left-to-right. You may have noticed by now that I use the C comma operator a lot, including for this specific purpose and to keep the code compact. The C comma operator has a cousin in Lisp `(begin` $expr_1$ $expr_2$ `...` $expr_n$`)` that returns the value of the last expression $expr_n$.

Parsing the list `(x y z)`, for example, results in the list construction `(cons x (cons y (cons z nil)))` and parsing `(x y . args)` results in the construction `(cons x (cons y args))`. Parsing a quoted expression `'`*expr* produces `(quote` *expr*`)`:

```
L quote() { return cons(atom("quote"),cons(read(),nil)); }
```

Parsing an atomic expression produces a number if the token is numeric and an atom otherwise:

```
L atomic() { L n; I i; return sscanf(buf,"%lg%n",&n,&i) && !buf[i] ? n : atom(buf); }
```

A token must be numeric to convert it to a number. If it is not, then an atom with the specified tokenized name is returned. But the PC-G850 requires this function:

```
L atomic() {
  L n; I i = strlen(buf);
  return isdigit(buf[*buf == '-']) && sscanf(buf,"%lg%n",&n,&i) && !buf[i] ? n : atom(buf);
}
```

where `i` must be initialized. Because `sscanf` on the PC-G850 simply returns 0 for incomplete numeric forms such as a single character `-`. So we should check if the token begins with a digit after an optional minus sign.

8 Printing Lisp Expressions

Displaying Lisp expression on screen or on a printer requires a few lines of code. This code should be self-explanatory:

```
void print(L x) {
  if (T(x) == NIL) printf("()");
  else if (T(x) == ATOM) printf("%s",A+ord(x));
  else if (T(x) == PRIM) printf("<%s>",prim[ord(x)].s);
  else if (T(x) == CONS) printlist(x);
  else if (T(x) == CLOS) printf("{%u}",ord(x));
  else printf("%.10lg",x);
}
```

Function `printlist` iterates over the list to display its elements in order, including a dot for the last cons pair if the list does not end in `nil`:

```
void printlist(L t) {
  putchar('(');
  while (1) {
    print(car(t));
    if (not(t = cdr(t))) break;
    if (T(t) != CONS) { printf(" . "); print(t); break; }
    putchar(' ');
  }
  putchar(')');
}
```

Note that `not(t = cdr(t))` changes `t` to the next list pair. Then, if `t` is `nil` we break from the loop. Otherwise, if the next `t` is not a cons pair, then we display a dot followed by the value of `t`. The dot visually separates the pair's values. The dot is also used to construct pairs, see Section 7.

9 Garbage Collection

To keep our Lisp interpreter code small, we should implement a very simple form of garbage collection to delete all temporary cells from the stack. We should preserve all globally-defined names and functions listed in `env`, To delete all temporary cells and keep `env` intact, it suffices to restore the stack pointer to the point on the stack where the free space begins, which is right below the global environment `env` cell on the stack:

```
void gc() { sp = ord(env); }
```

Why does this work? After the last `env = pair(`*name*`,`*expr*`,env)` call was made to define *name* globally, we know for sure that *expr* is already stored higher up in the stacked cells, i.e. in cells above the last `env` pair on the stack. These cells are not removed by `gc` when setting `sp` to the cell indexed by `env`.

One possible caveat of this approach is that we cannot support interactive use of the Lisp special forms `setq` (modifies an association in an environment), `set-car!` (overwrites the `car` of a pair) and `set-cdr!` (overwrites the `cdr` of a pair), since these may change previously-defined expressions in the global environment, If the modified global environment references temporary expressions, then `gc` corrupts the global environment. In principle we could support `setq`, `set-car!` and `set-cdr!` if we somehow limit their use to prevent this problem from occurring.

10 The Read-Eval-Print Loop

After the `main` program initialized the static variables `nil`, `tru`, and `err` (see Section 4) and populated the environment with `#t` and other primitives (see Section 6), the `main` program executes the so-called Lisp *read-eval-print loop* (REPL):

```
int main() {
  ...
  while (1) { printf("\n%u>",sp-(hp>>3)); print(eval(read(),env)); gc(); }
}
```

The prompt in the REPL displays the number of cells freely available, i.e. the space between the heap pointer `hp` and stack pointer `sp`. Note that `hp` points to bytes and `sp` points to 8-byte floats, so `hp` is scaled down by a factor 8. Garbage collection is performed in the REPL after the results are displayed.

This completes Lisp in 99 lines of C. See Appendix A and B for the complete listings with NaN and BCD boxing, respectively.

11 Additional Lisp Primitives

The following Lisp primitives are not included in the 99 line C program, since these are not absolutely required to write Lisp programs.

11.1 `assoc` and `env`

`(assoc` *var environment*`)` gives the expression associated with *var* in the specified *environment*. `(env)` returns the current environment in which `(env)` is evaluated.

```
L f_assoc(L t,L e) { return t = evlis(t,e),assoc(car(t),car(cdr(t))); }
L f_env(L _,L e) { return e; }
... prim[] = { ... {"assoc",f_assoc},{"env",f_env} ... };
```

Note that *var* should be quoted when passed to `assoc` since its arguments are evaluated first. Example: `(assoc 'b '((a 1) (b 2) (c 3))` gives 2.

11.2 `let` and `letrec*`

The `let` special form is similar to the `let*` special form, but evaluates all expressions first before binding the values to the variables.

```
L f_let(L t,L e) {
  L d = e;
  while (let(t)) d = pair(car(car(t)),eval(car(cdr(car(t))),e),d),t = cdr(t);
  return eval(car(t),d);
}
... prim[] = { ... {"let",f_let} ... };
```

The `letrec*` special form is similar to the `let*` special form, but allows for local recursion where the name may also appear in the value of a `letrec*` name-value pair.

```
L f_letreca(L t,L e) {
  while (let(t)) {
    e = pair(car(car(t)),err,e);
    cell[sp+2] = eval(car(cdr(car(t))),e);
    t = cdr(t);
  }
  return eval(car(t),e);
}
... prim[] = { ... {"letrec*",f_letreca} ... };
```

This implementation updates the environment `e` first with a new variable-`err` binding, then sets the `err` cell on the stack to the expression evaluated within the updated scope `e`. Example:

```
> (letrec* (f (lambda (n) (if (< 1 n) (* n (f (- n 1))) 1))) (f 5))
120
```

11.3 `setq`

The `setq` special form sets the value of a variable as a side-effect with `(setq` *var expr*`)`:

```
L f_setq(L t,L e) {
  L a = car(t),x = eval(car(cdr(t)),e);
  while (T(e) == CONS && a != car(car(e))) e = cdr(e);
  return T(e) == CONS ? cell[ord(car(e))] = x : err;
}
... prim[] = { ... {"setq",f_setq} ... };
```

This function is dangerous, because garbage collection after `setq` may corrupt the stack if the new value assigned to a global variable is a temporary list (all interactively constructed lists are temporary). On the other hand, `setq` is safe to use to assign local variables of a `lambda` and a `let`.

11.4 `set-car!` and `set-cdr!`

The `set-car!` primitive sets the value of the `car` cell of a cons pair as a side-effect. The `set-cdr!` primitive sets the value of the `cdr` cell of a cons pair as a side-effect.

```
L f_setcar(L t,L e) {
  L p = car(t = evlis(t,e)),x = car(cdr(t));
  if (T(p) == CONS) cell[ord(p)+1] = x;
  return x;
}
L f_setcdr(L t,L e) {
  L p = car(t = evlis(t,e)),x = car(cdr(t));
  if (T(p) == CONS) cell[ord(p)] = x;
  return x;
}
... prim[] = { ... {"set-car!",f_setcar},{"set-cdr!",f_setcdr} ... };
```

Like `setq`, these functions are dangerous.

11.5 `macro`

Macros allow Lisp to be syntactically extended. A `macro` is similar to a `lambda`, except that its arguments are not evaluated when the macro is applied. Typically a macro constructs Lisp code when the macro is applied, thereby *expanding* and evaluating the Lisp code in place.

To add `macro` is easy and only requires a few lines of code to define a new `MACR` tag, add new the functions `macro`, `f_macro` and `expand`, and make some minor changes to the existing functions `car`, `cdr` and `apply`. First we add a new tag `MACR`:

```
  /*** with NaN Boxing ***/
I ATOM=0x7ff8,PRIM=0x7ff9,CONS=0x7ffa,CLOS=0x7ffb,MACR=0x7ffc,NIL=0x7ffd;
  /*** with BCD boxing ***/
I ATOM=32,PRIM=48,CONS=64,CLOS=80,MACR=96,NIL=112;
```

The `car` and `cdr` functions should be modified to make them applicable to `MACR`-tagged floats which are essentially cons pairs containing the list of *variables* of the macro as `car` cell and *expression* as the `cdr` cell:

```
L car(L p) { return T(p) == CONS || T(p) == CLOS || T(p) == MACR ? cell[ord(p)+1] : err; }
L cdr(L p) { return T(p) == CONS || T(p) == CLOS || T(p) == MACR ? cell[ord(p)] : err; }
```

We add a constructor `macro` with the corresponding Lisp primitive `f_macro`:

```
L macro(L v,L x) { return box(MACR,ord(cons(v,x))); }
L f_macro(L t,L e) { return macro(car(t),car(cdr(t))); }
... prim[] = { ... {"macro", f_macro} ... };
```

Application of macros is similar to lambdas, but they expand instead which is performed by a new `expand` function:

```
L expand(L f,L t,L e) { return eval(eval(cdr(f),bind(car(f),t,env)),e); }
L apply(L f,L t,L e) {
  return T(f) == PRIM ? prim[ord(f)].f(t,e) :
```

```
          T(f) == CLOS ? reduce(f,t,e) :
          T(f) == MACR ? expand(f,t,e) :
          err;
}
```

The macro is evaluated in the global environment `env`. This typically constructs Lisp code that is then evaluated in the current environment `e`.

Example: the Lisp *delayed evaluation* primitives `delay` and `force` implemented as a macro:

```
> (define list (lambda args args))
> (define delay (macro (x) (list 'lambda () x)))
> (define force (lambda (f) (f)))
```

The `delay` macro is used for *lazy evaluation* or *call by need* of arguments, by passing them unevaluated to a function together with their environment as a closure called a *promise*. Hence, `(list 'lambda () x)` constructs the Lisp code `(lambda () x)` where `x` is the unevaluated argument passed to `delay`. The `force` function evaluates a promise:

```
> (force (delay (+ 1 2)))
3
```

More information and example on `delay` and `force` can be found in Lisp textbooks and manuals.

11.6 `read` and `print`

Since our Lisp implementation already includes `read` and `print` functions, we can add them as primitives as follows:

```
L f_read(L _,L e) { L x; char c = see; see = ' '; x = read(); see = c; return x; }
L f_print(L t,L e) { print(car(evlis(t,e))); return nil; }
... prim[] = { ... {"read", f_read},{"print", f_print} ... };
```

Example: `(read)` gives the Lisp expression typed in (unevaluated) and `(print 'hello)` displays `hello`.

With a few more lines of C code, other Lisp IO primitives can be added to complete the IO support.

12 Conclusions

This article demonstrated how a fully-functional Lisp interpreter with 20 Lisp primitives, garbage collection and REPL can be written in 99 lines of C or less. The concepts and implementation presented largely follow the original ideas and discoveries made by McCarthy in his 1960 paper. Given the material included in this articel, it should not be difficult to expand the Lisp interpreter to support additional features and experiment with alternative syntax and semantics of a hybrid Lisp or a completely new language.

Any overlap or resemblance to any other Lisp implementations is coincental. I wrote the article from scratch based on McCarthy's paper and based on my 20 years of experience teaching programming language courses that include Lisp/Scheme design and programming.

13 Bibliography

References

[1] Graham, Paul (2002). "The Roots of Lisp" `http://www.paulgraham.com/rootsoflisp.html` retrieved July 9, 2022.

[2] McCarthy, John. "Recursive functions of symbolic expressions and their computation by machine, part I." *Communications of the ACM* 3.4 (1960): 184-195.

[3] Church, Alonzo. "The calculi of lambda-conversion." Bull. Amer. Math. Soc 50 (1944): 169-172.

A Tiny Lisp Interpreter with NaN boxing: 99 Lines of C

Lisp in 99 lines of C without comments:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define I unsigned
#define L double
#define T(x) *(unsigned long long*)&x>>48
#define A (char*)cell
#define N 1024
I hp=0,sp=N,ATOM=0x7ff8,PRIM=0x7ff9,CONS=0x7ffa,CLOS=0x7ffb,NIL=0x7ffc;
L cell[N],nil,tru,err,env;
L box(I t,I i) { L x; *(unsigned long long*)&x = (unsigned long long)t<<48|i; return x; }
I ord(L x) { return *(unsigned long long*)&x; }
L num(L n) { return n; }
I equ(L x,L y) { return *(unsigned long long*)&x == *(unsigned long long*)&y; }
L atom(const char *s) {
 I i = 0; while (i < hp && strcmp(A+i,s)) i += strlen(A+i)+1;
 if (i == hp && (hp += strlen(strcpy(A+i,s))+1) > sp-1<<3) abort();
 return box(ATOM,i);
}
L cons(L x,L y) { cell[--sp] = x; cell[--sp] = y; if (hp > sp-1<<3) abort(); return box(CONS,sp); }
L car(L p) { return (T(p)&~(CONS^CLOS)) == CONS ? cell[ord(p)+1] : err; }
L cdr(L p) { return (T(p)&~(CONS^CLOS)) == CONS ? cell[ord(p)] : err; }
L pair(L v,L x,L e) { return cons(cons(v,x),e); }
L closure(L v,L x,L e) { return box(CLOS,ord(pair(v,x,equ(e,env) ? nil : e))); }
L assoc(L a,L e) { while (T(e) == CONS && !equ(a,car(car(e)))) e = cdr(e); return T(e) == CONS ? cdr(car(e)) : err; }
I not(L x) { return T(x) == NIL; }
I let(L x) { return T(x) != NIL && (x = cdr(x),T(x) != NIL); }
L eval(L,L),parse();
L evlis(L t,L e) { return T(t) == CONS ? cons(eval(car(t),e),evlis(cdr(t),e)) : eval(t,e); }
L f_eval(L t,L e) { return eval(car(evlis(t,e)),e); }
L f_quote(L t,L _) { return car(t); }
L f_cons(L t,L e) { return t = evlis(t,e),cons(car(t),car(cdr(t))); }
L f_car(L t,L e) { return car(car(evlis(t,e))); }
L f_cdr(L t,L e) { return cdr(car(evlis(t,e))); }
L f_add(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n += car(t); return num(n); }
L f_sub(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n -= car(t); return num(n); }
L f_mul(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n *= car(t); return num(n); }
L f_div(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n /= car(t); return num(n); }
L f_int(L t,L e) { L n = car(evlis(t,e)); return n<1e16 && n>-1e16 ? (long long)n : n; }
L f_lt(L t,L e) { return t = evlis(t,e),car(t) - car(cdr(t)) < 0 ? tru : nil; }
L f_eq(L t,L e) { return t = evlis(t,e),equ(car(t),car(cdr(t))) ? tru : nil; }
L f_not(L t,L e) { return not(car(t = evlis(t,e))) ? tru : nil; }
L f_or(L t,L e) { for (; T(t) != NIL; t = cdr(t)) if (!not(eval(car(t),e))) return tru; return nil; }
L f_and(L t,L e) { for (; T(t) != NIL; t = cdr(t)) if (not(eval(car(t),e))) return nil; return tru; }
L f_cond(L t,L e) { while (T(t) != NIL && not(eval(car(car(t)),e))) t = cdr(t); return eval(car(cdr(car(t))),e); }
L f_if(L t,L e) { return eval(car(cdr(not(eval(car(t),e)) ? cdr(t) : t)),e); }
L f_leta(L t,L e) { while (let(t)) e = pair(car(car(t)),eval(car(cdr(car(t))),e),e),t = cdr(t); return eval(car(t),e); }
L f_lambda(L t,L e) { return closure(car(t),car(cdr(t)),e); }
L f_define(L t,L e) { env = pair(car(t),eval(car(cdr(t)),e),env); return car(t); }
struct { const char *s; L (*f)(L,L); } prim[] = {
{"eval",f_eval},{"quote",f_quote},{"cons",f_cons},{"car", f_car}, {"cdr",   f_cdr},   {"+",     f_add},   {"-",  f_sub},
{"*",   f_mul}, {"/",    f_div},  {"int", f_int}, {"<",    f_lt},  {"eq?",   f_eq},    {"or",    f_or},    {"and",f_and},
{"not", f_not}, {"cond", f_cond}, {"if",  f_if},  {"let*",f_leta},{"lambda",f_lambda},{"define",f_define},{0}};
L bind(L v,L t,L e) { return T(v) == NIL ? e : T(v) == CONS ? bind(cdr(v),cdr(t),pair(car(v),car(t),e)) : pair(v,t,e); }
L reduce(L f,L t,L e) { return eval(cdr(car(f)),bind(car(car(f)),evlis(t,e),not(cdr(f)) ? env : cdr(f))); }
L apply(L f,L t,L e) { return T(f) == PRIM ? prim[ord(f)].f(t,e) : T(f) == CLOS ? reduce(f,t,e) : err; }
L eval(L x,L e) { return T(x) == ATOM ? assoc(x,e) : T(x) == CONS ? apply(eval(car(x),e),cdr(x),e) : x; }
char buf[40],see = ' ';
void look() { see = getchar(); }
I seeing(char c) { return c == ' ' ? see > 0 && see <= c : see == c; }
```

```
char get() { char c = see; look(); return c; }
char scan() {
 char *tok = buf;
 while (seeing(' ')) look();
 if (seeing('(') || seeing(')') || seeing('\'')) *tok++ = get();
 else do *tok++ = get(); while (!seeing('(') && !seeing(')') && !seeing(' '));
 return *tok = 0,*buf;
}
L read() { return scan(),parse(); }
L list() { L x; return scan() == ')' ? nil : !strcmp(buf, ".") ? (x = read(),scan(),x) : (x = parse(),cons(x,list())); }
L quote() { return cons(atom("quote"),cons(read(),nil)); }
L atomic() { L n; I i; return sscanf(buf,"%lg%n",&n,&i) && !buf[i] ? n : atom(buf); }
L parse() { return *buf == '(' ? list() : *buf == '\'' ? quote() : atomic(); }
void print(L);
void printlist(L t) {
 putchar('(');
 while (1) {
  print(car(t));
  if (not(t = cdr(t))) break;
  if (T(t) != CONS) { printf(" . "); print(t); break; }
  putchar(' ');
 }
 putchar(')');
}
void print(L x) {
 if (T(x) == NIL) printf("()");
 else if (T(x) == ATOM) printf("%s",A+ord(x));
 else if (T(x) == PRIM) printf("<%s>",prim[ord(x)].s);
 else if (T(x) == CONS) printlist(x);
 else if (T(x) == CLOS) printf("{%u}",ord(x));
 else printf("%.10lg",x);
}
void gc() { sp = ord(env); }
int main() {
 I i; printf("tinylisp");
 nil = box(NIL,0); err = atom("ERR"); tru = atom("#t"); env = pair(tru,tru,nil);
 for (i = 0; prim[i].s; ++i) env = pair(atom(prim[i].s),box(PRIM,i),env);
 while (1) { printf("\n%u>",sp-(hp>>3)); print(eval(read(),env)); gc(); }
}
```

B Tiny Lisp Interpreter with BCD boxing: 99 Lines of C

Lisp for the PC-G850 consists of 99 lines of C without comments:

```
#define I unsigned
#define L double
#define T *(char*)&
#define A (char*)cell
#define N 1024
I hp=0,sp=N,ATOM=32,PRIM=48,CONS=64,CLOS=80,NIL=96;
L cell[N],nil,tru,err,env;
L box(I t,I i) { L x = i+10; T(x) = t; return x; }
I ord(L x) { T(x) &= 15; return (I)x-10; }
L num(L n) { T(n) &= 159; return n; }
I equ(L x,L y) { return x == y; }
L atom(const char *s) {
 I i = 0; while (i < hp && strcmp(A+i,s)) i += strlen(A+i)+1;
 if (i == hp && (hp += strlen(strcpy(A+i,s))+1) > sp-1<<3) abort();
 return box(ATOM,i);
}
L cons(L x,L y) { cell[--sp] = x; cell[--sp] = y; if (hp > sp-1<<3) abort(); return box(CONS,sp); }
L car(L p) { return (T(p)&~(CONS^CLOS)) == CONS ? cell[ord(p)+1] : err; }
L cdr(L p) { return (T(p)&~(CONS^CLOS)) == CONS ? cell[ord(p)] : err; }
L pair(L v,L x,L e) { return cons(cons(v,x),e); }
L closure(L v,L x,L e) { return box(CLOS,ord(pair(v,x,equ(e,env) ? nil : e))); }
L assoc(L a,L e) { while (T(e) == CONS && !equ(a,car(car(e)))) e = cdr(e); return T(e) == CONS ? cdr(car(e)) : err; }
I not(L x) { return T(x) == NIL; }
I let(L x) { return T(x) != NIL && (x = cdr(x),T(x) != NIL); }
L eval(L,L),parse();
L evlis(L t,L e) { return T(t) == CONS ? cons(eval(car(t),e),evlis(cdr(t),e)) : eval(t,e); }
L f_eval(L t,L e) { return eval(car(evlis(t,e)),e); }
L f_quote(L t,L _) { return car(t); }
L f_cons(L t,L e) { return t = evlis(t,e),cons(car(t),car(cdr(t))); }
L f_car(L t,L e) { return car(car(evlis(t,e))); }
L f_cdr(L t,L e) { return cdr(car(evlis(t,e))); }
L f_add(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n += car(t); return num(n); }
L f_sub(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n -= car(t); return num(n); }
L f_mul(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n *= car(t); return num(n); }
L f_div(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n /= car(t); return num(n); }
L f_int(L t,L e) { L n = car(evlis(t,e)); return n-1e9 < 0 && n+1e9 > 0 ? (long)n : n; }
L f_lt(L t,L e) { return t = evlis(t,e),car(t) - car(cdr(t)) < 0 ? tru : nil; }
L f_eq(L t,L e) { return t = evlis(t,e),equ(car(t),car(cdr(t))) ? tru : nil; }
L f_not(L t,L e) { return not(car(t = evlis(t,e))) ? tru : nil; }
L f_or(L t,L e) { for (; T(t) != NIL; t = cdr(t)) if (!not(eval(car(t),e))) return tru; return nil; }
L f_and(L t,L e) { for (; T(t) != NIL; t = cdr(t)) if (not(eval(car(t),e))) return nil; return tru; }
L f_cond(L t,L e) { while (T(t) != NIL && not(eval(car(car(t)),e))) t = cdr(t); return eval(car(cdr(car(t))),e); }
L f_if(L t,L e) { return eval(car(cdr(not(eval(car(t),e)) ? cdr(t) : t)),e); }
L f_leta(L t,L e) { while (let(t)) e = pair(car(car(t)),eval(car(cdr(car(t))),e),e),t = cdr(t); return eval(car(t),e); }
L f_lambda(L t,L e) { return closure(car(t),car(cdr(t)),e); }
L f_define(L t,L e) { env = pair(car(t),eval(car(cdr(t)),e),env); return car(t); }
struct { const char *s; L (*f)(L,L); } prim[] = {
{"eval",f_eval},{"quote",f_quote},{"cons",f_cons},{"car", f_car}, {"cdr",   f_cdr},   {"+",     f_add},   {"-",  f_sub},
{"*",   f_mul}, {"/",    f_div},  {"int", f_int}, {"<",    f_lt},  {"eq?",    f_eq},    {"or",    f_or},    {"and",f_and},
{"not", f_not}, {"cond", f_cond}, {"if",  f_if},  {"let*",f_leta},{"lambda",f_lambda},{"define",f_define},{0}};
L bind(L v,L t,L e) { return T(v) == NIL ? e : T(v) == CONS ? bind(cdr(v),cdr(t),pair(car(v),car(t),e)) : pair(v,t,e); }
L reduce(L f,L t,L e) { return eval(cdr(car(f)),bind(car(car(f)),evlis(t,e),not(cdr(f)) ? env : cdr(f))); }
L apply(L f,L t,L e) { return T(f) == PRIM ? prim[ord(f)].f(t,e) : T(f) == CLOS ? reduce(f,t,e) : err; }
L eval(L x,L e) { return T(x) == ATOM ? assoc(x,e) : T(x) == CONS ? apply(eval(car(x),e),cdr(x),e) : x; }
char buf[40],see = ' ';
void look() { see = getchar(); }
I seeing(char c) { return c == ' ' ? see > 0 && see <= c : see == c; }
char get() { char c = see; look(); return c; }
char scan() {
 char *tok = buf;
```

```
 while (seeing(' ')) look();
 if (seeing('(') || seeing(')') || seeing('\'')) *tok++ = get();
 else do *tok++ = get(); while (!seeing('(') && !seeing(')') && !seeing(' '));
 return *tok = 0,*buf;
}
L read() { return scan(),parse(); }
L list() { L x; return scan() == ')' ? nil : !strcmp(buf, ".") ? (x = read(),scan(),x) : (x = parse(),cons(x,list())); }
L quote() { return cons(atom("quote"),cons(read(),nil)); }
L atomic() {
  L n; I i = strlen(buf);
  return isdigit(buf[*buf == '-']) && sscanf(buf,"%lg%n",&n,&i) && !buf[i] ? n : atom(buf);
}
L parse() { return *buf == '(' ? list() : *buf == '\'' ? quote() : atomic(); }
void print(L);
void printlist(L t) {
 putchar('(');
 while (1) {
  print(car(t));
  if (not(t = cdr(t))) break;
  if (T(t) != CONS) { printf(" . "); print(t); break; }
  putchar(' ');
 }
 putchar(')');
}
void print(L x) {
 if (T(x) == NIL) printf("()");
 else if (T(x) == ATOM) printf("%s",A+ord(x));
 else if (T(x) == PRIM) printf("<%s>",prim[ord(x)].s);
 else if (T(x) == CONS) printlist(x);
 else if (T(x) == CLOS) printf("{%u}",ord(x));
 else printf("%.10lg",x);
}
void gc() { sp = ord(env); }
int main() {
 I i; printf("lisp850");
 nil = box(NIL,0); err = atom("ERR"); tru = atom("#t"); env = pair(tru,tru,nil);
 for (i = 0; prim[i].s; ++i) env = pair(atom(prim[i].s),box(PRIM,i),env);
 while (1) { printf("\n%u>",sp-(hp>>3)); print(eval(read(),env)); gc(); }
}
```

C Optimized Lisp Interpreter with NaN bpxing

The following version of the Lisp interpreter is aggressively optimized for speed and reduced memory usage at runtime.

```
/* tinylisp-opt.c by Robert A. van Engelen 2022 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define I unsigned
#define L double
#define T(x) *(unsigned long long*)&x>>48
#define A (char*)cell
#define N 1024
I hp=0,sp=N,ATOM=0x7ff8,PRIM=0x7ff9,CONS=0x7ffa,CLOS=0x7ffb,NIL=0x7ffc;
L cell[N],nil,tru,err,env;
L box(I t,I i) { L x; *(unsigned long long*)&x = (unsigned long long)t<<48|i; return x; }
I ord(L x) { return *(unsigned long long*)&x; }
L num(L n) { return n; }
I equ(L x,L y) { return *(unsigned long long*)&x == *(unsigned long long*)&y; }
L atom(const char *s) {
 I i = 0; while (i < hp && strcmp(A+i,s)) i += strlen(A+i)+1;
 if (i == hp && (hp += strlen(strcpy(A+i,s))+1) > sp-1<<3) abort();
 return box(ATOM,i);
}
L cons(L x,L y) { cell[--sp] = x; cell[--sp] = y; if (hp > sp-1<<3) abort(); return box(CONS,sp); }
L car(L p) { return (T(p)&~(CONS^CLOS)) == CONS ? cell[ord(p)+1] : err; }
L cdr(L p) { return (T(p)&~(CONS^CLOS)) == CONS ? cell[ord(p)] : err; }
L pair(L v,L x,L e) { return cons(cons(v,x),e); }
L closure(L v,L x,L e) { return box(CLOS,ord(pair(v,x,equ(e,env) ? nil : e))); }
L assoc(L a,L e) { while (T(e) == CONS && !equ(a,car(car(e)))) e = cdr(e); return T(e) == CONS ? cdr(car(e)) : err; }
I not(L x) { return T(x) == NIL; }
I let(L x) { return T(x) != NIL && (x = cdr(x),T(x) != NIL); }
L eval(L,L),parse();
L evlis(L t,L e) { return T(t) == CONS ? cons(eval(car(t),e),evlis(cdr(t),e)) : eval(t,e); }
L f_eval(L t,L e) { return eval(car(evlis(t,e)),e); }
L f_quote(L t,L _) { return car(t); }
L f_cons(L t,L e) { return t = evlis(t,e),cons(car(t),car(cdr(t))); }
L f_car(L t,L e) { return car(car(evlis(t,e))); }
L f_cdr(L t,L e) { return cdr(car(evlis(t,e))); }
L f_add(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n += car(t); return num(n); }
L f_sub(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n -= car(t); return num(n); }
L f_mul(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n *= car(t); return num(n); }
L f_div(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n /= car(t); return num(n); }
L f_int(L t,L e) { L n = car(evlis(t,e)); return n<1e16 && n>-1e16 ? (long long)n : n; }
L f_lt(L t,L e) { return t = evlis(t,e),car(t) - car(cdr(t)) < 0 ? tru : nil; }
L f_eq(L t,L e) { return t = evlis(t,e),equ(car(t),car(cdr(t))) ? tru : nil; }
L f_not(L t,L e) { return not(car(t = evlis(t,e))) ? tru : nil; }
L f_or(L t,L e) { for (; T(t) != NIL; t = cdr(t)) if (!not(eval(car(t),e))) return tru; return nil; }
L f_and(L t,L e) { for (; T(t) != NIL; t = cdr(t)) if (not(eval(car(t),e))) return nil; return tru; }
L f_cond(L t,L e) { while (T(t) != NIL && not(eval(car(car(t)),e))) t = cdr(t); return eval(car(cdr(car(t))),e); }
L f_if(L t,L e) { return eval(car(cdr(not(eval(car(t),e)) ? cdr(t) : t)),e); }
L f_leta(L t,L e) { while (let(t)) e = pair(car(car(t)),eval(car(cdr(car(t))),e),e),t = cdr(t); return eval(car(t),e); }
L f_lambda(L t,L e) { return closure(car(t),car(cdr(t)),e); }
L f_define(L t,L e) { env = pair(car(t),eval(car(cdr(t)),e),env); return car(t); }
struct { const char *s; L (*f)(L,L); } prim[] = {
{"eval",f_eval},{"quote",f_quote},{"cons",f_cons},{"car", f_car}, {"cdr",   f_cdr},   {"+",     f_add},   {"-",  f_sub},
{"*",   f_mul}, {"/",     f_div},  {"int", f_int}, {"<",    f_lt},  {"eq?",   f_eq},    {"or",    f_or},    {"and",f_and},
{"not", f_not}, {"cond", f_cond}, {"if",  f_if},  {"let*",f_leta},{"lambda",f_lambda},{"define",f_define},{0}};
L eval(L x,L e) {
 L f,v,d;
 while (1) {
  if (T(x) == ATOM) return assoc(x,e);
  if (T(x) != CONS) return x;
```

```
  f = eval(car(x),e),x = cdr(x);
  if (T(f) == PRIM) return prim[ord(f)].f(x,e);
  if (T(f) != CLOS) return err;
  v = car(car(f)),d = cdr(f);
  if (T(d) == NIL) d = env;
  while (T(v) == CONS && T(x) == CONS) d = pair(car(v),eval(car(x),e),d),v = cdr(v),x = cdr(x);
  if (T(v) == CONS) { x = eval(x,e); while (T(v) == CONS) d = pair(car(v),car(x),d),v = cdr(v),x = cdr(x); }
  else if (T(x) == CONS) x = evlis(x,e);
  if (T(v) != NIL) d = pair(v,x,d);
  x = cdr(car(f)),e = d;
 }
}
char buf[40],see = ' ';
void look() { see = getchar(); }
I seeing(char c) { return c == ' ' ? see > 0 && see <= c : see == c; }
char get() { char c = see; look(); return c; }
char scan() {
 char *tok = buf;
 while (seeing(' ')) look();
 if (seeing('(') || seeing(')') || seeing('\'')) *tok++ = get();
 else do *tok++ = get(); while (!seeing('(') && !seeing(')') && !seeing(' '));
 return *tok = 0,*buf;
}
L read() { return scan(),parse(); }
L list() {
 L t = nil,*p = &t;
 while (1) {
  if (scan() == ')') return t;
  if (*buf == '.' && !buf[1]) return *p = read(),scan(),t;
  *p = cons(parse(),nil),p = cell+sp;
 }
}
L parse() {
 L n; I i;
 if (*buf == '(') return list();
 if (*buf == '\'') return cons(atom("quote"),cons(read(),nil));
 if (sscanf(buf,"%lg%n",&n,&i) && !buf[i]) return n;
 return atom(buf);
}
void print(L);
void printlist(L t) {
 putchar('(');
 while (1) {
  print(car(t));
  if (not(t = cdr(t))) break;
  if (T(t) != CONS) { printf(" . "); print(t); break; }
  putchar(' ');
 }
 putchar(')');
}
void print(L x) {
 if (T(x) == NIL) printf("()");
 else if (T(x) == ATOM) printf("%s",A+ord(x));
 else if (T(x) == PRIM) printf("<%s>",prim[ord(x)].s);
 else if (T(x) == CONS) printlist(x);
 else if (T(x) == CLOS) printf("{%u}",ord(x));
 else printf("%.10lg",x);
}
void gc() { sp = ord(env); }
int main() {
 I i; printf("tinylisp");
 nil = box(NIL,0); err = atom("ERR"); tru = atom("#t"); env = pair(tru,tru,nil);
 for (i = 0; prim[i].s; ++i) env = pair(atom(prim[i].s),box(PRIM,i),env);
 while (1) { printf("\n%u>",sp-(hp>>3)); print(eval(read(),env)); gc(); }
}
```

D Optimized Lisp Interpreter with BCD bpxing

The following version of the Lisp interpreter for the PC-G850 is aggressively optimized for speed and reduced memory usage at runtime.

```
#define I unsigned
#define L double
#define T *(char*)&
#define A (char*)cell
#define N 1024
I hp=0,sp=N,ATOM=32,PRIM=48,CONS=64,CLOS=80,NIL=96;
L cell[N],nil,tru,err,env;
L box(I t,I i) { L x = i+10; T x = t; return x; }
I ord(L x) { T x &= 15; return (I)x-10; }
L num(L n) { T n &= 159; return n; }
L atom(const char *s) {
 I i = 0; while (i < hp && strcmp(A+i,s)) i += strlen(A+i)+1;
 if (i == hp && (hp += strlen(strcpy(A+i,s))+1) > sp-1<<3) abort();
 return box(ATOM,i);
}
L cons(L x,L y) { cell[--sp] = x; cell[--sp] = y; if (hp > sp-1<<3) abort(); return box(CONS,sp); }
L car(L p) { return (T p&224) == CONS ? cell[T p &= 15,(I)p-9] : err; }
L cdr(L p) { return (T p&224) == CONS ? cell[T p &= 15,(I)p-10] : err; }
L pair(L v,L x,L e) { return cons(cons(v,x),e); }
L assoc(L a,L e) { while (T e == CONS && a != car(car(e))) e = cdr(e); return T e == CONS ? cdr(car(e)) : err; }
I not(L x) { return T x == NIL; }
I let(L x) { return T x != NIL && (x = cdr(x),T x != NIL); }
L eval(L,L),parse();
L evlis(L t,L e) {
 L s = nil,*p = &s;
 while (T t == CONS) *p = cons(eval(car(t),e),nil),t = cdr(t),p = cell+sp;
 if (T t != NIL) *p = eval(t,e);
 return s;
}
L f_eval(L t,L e) { return eval(car(evlis(t,e)),e); }
L f_quote(L t,L _) { return car(t); }
L f_cons(L t,L e) { return t = evlis(t,e),cons(car(t),car(cdr(t))); }
L f_car(L t,L e) { return car(car(evlis(t,e))); }
L f_cdr(L t,L e) { return cdr(car(evlis(t,e))); }
L f_add(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n += car(t); return num(n); }
L f_sub(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n -= car(t); return num(n); }
L f_mul(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n *= car(t); return num(n); }
L f_div(L t,L e) { L n = car(t = evlis(t,e)); while (!not(t = cdr(t))) n /= car(t); return num(n); }
L f_int(L t,L e) { return t = car(evlis(t,e)),t-1e9 < 0 && t+1e9 > 0 ? (long)t : t; }
L f_lt(L t,L e) { return t = evlis(t,e),car(t) - car(cdr(t)) < 0 ? tru : nil; }
L f_eq(L t,L e) { return t = evlis(t,e),car(t) == car(cdr(t)) ? tru : nil; }
L f_not(L t,L e) { return not(car(t = evlis(t,e))) ? tru : nil; }
L f_or(L t,L e) { for (; T t == CONS; t = cdr(t)) if (!not(eval(car(t),e))) return tru; return nil; }
L f_and(L t,L e) { for (; T t == CONS; t = cdr(t)) if (not(eval(car(t),e))) return nil; return tru; }
L f_cond(L t,L e) { while (T t == CONS && not(eval(car(car(t)),e))) t = cdr(t); return eval(car(cdr(car(t))),e); }
L f_if(L t,L e) { return eval(car(cdr(not(eval(car(t),e)) ? cdr(t) : t)),e); }
L f_leta(L t,L e) { while (let(t)) e = pair(car(car(t)),eval(car(cdr(car(t))),e),e),t = cdr(t); return eval(car(t),e); }
L f_lambda(L t,L e) { return box(CLOS,ord(pair(car(t),car(cdr(t)),e == env ? nil : e))); }
L f_define(L t,L e) { env = pair(car(t),eval(car(cdr(t)),e),env); return car(t); }
struct { const char *s; L (*f)(L,L); } prim[] = {
{"eval",f_eval},{"quote",f_quote},{"cons",f_cons},{"car", f_car}, {"cdr",   f_cdr},   {"+",     f_add},   {"-",  f_sub},
{"*",   f_mul}, {"/",     f_div},  {"int", f_int}, {"<",    f_lt},  {"eq?",    f_eq},    {"or",    f_or},    {"and",f_and},
{"not", f_not}, {"cond", f_cond}, {"if",  f_if},  {"let*",f_leta},{"lambda",f_lambda},{"define",f_define},{0}};
L eval(L x,L e) {
 L f,v,d;
 while (1) {
  if (T x == ATOM) return assoc(x,e);
  if (T x != CONS) return x;
  f = eval(car(x),e),x = cdr(x);
```

```
  if (T f == PRIM) return prim[T f &= 15,(I)f-10].f(x,e);
  if (T f != CLOS) return err;
  v = car(car(f)),d = cdr(f);
  if (T d == NIL) d = env;
  while (T v == CONS && T x == CONS) d = pair(car(v),eval(car(x),e),d),v = cdr(v),x = cdr(x);
  if (T v == CONS) { x = eval(x,e); while (T v == CONS) d = pair(car(v),car(x),d),v = cdr(v),x = cdr(x); }
  else if (T x == CONS) x = evlis(x,e);
  if (T v != NIL) d = pair(v,x,d);
  x = cdr(car(f)),e = d;
 }
}
char buf[40],see = ' ';
void look() { see = getchar(); }
I seeing(char c) { return c == ' ' ? see > 0 && see <= c : see == c; }
char get() { char c = see; look(); return c; }
char scan() {
 char *tok = buf;
 while (seeing(' ')) look();
 if (seeing('(') || seeing(')') || seeing('\'')) *tok++ = get();
 else do *tok++ = get(); while (!seeing('(') && !seeing(')') && !seeing(' '));
 return *tok = 0,*buf;
}
L read() { return scan(),parse(); }
L list() {
 L t = nil,*p = &t;
 while (1) {
  if (scan() == ')') return t;
  if (*buf == '.' && !buf[1]) return *p = read(),scan(),t;
  *p = cons(parse(),nil),p = cell+sp;
 }
}
L parse() {
 L n; I i;
 if (*buf == '(') return list();
 if (*buf == '\'') return cons(atom("quote"),cons(read(),nil));
 i = strlen(buf);
 if (isdigit(buf[*buf == '-']) && sscanf(buf,"%lg%n",&n,&i) && !buf[i]) return n;
 return atom(buf);
}
void print(L);
void printlist(L t) {
 putchar('(');
 while (1) {
  print(car(t));
  if (not(t = cdr(t))) break;
  if (T t != CONS) { printf(" . "); print(t); break; }
  putchar(' ');
 }
 putchar(')');
}
void print(L x) {
 if (T x == NIL) printf("()");
 else if (T x == ATOM) printf("%s",A+ord(x));
 else if (T x == PRIM) printf("<%s>",prim[ord(x)].s);
 else if (T x == CONS) printlist(x);
 else if (T x == CLOS) printf("{%u}",ord(x));
 else printf("%.10lg",x);
}
void gc() { sp = ord(env); }
int main() {
 I i; printf("lisp850");
 nil = box(NIL,0); err = atom("ERR"); tru = atom("#t"); env = pair(tru,tru,nil);
 for (i = 0; prim[i].s; ++i) env = pair(atom(prim[i].s),box(PRIM,i),env);
 while (1) { printf("\n%u>",sp-(hp>>3)); print(eval(read(),env)); gc(); }
}
```

E Example Lisp Functions

E.1 Standard Lisp Functions

The following functions should be self-explanatory, for details see further below:

```
(define null? not)
(define err? (lambda (x) (eq? x 'err)))
(define number? (lambda (x) (eq? (* 0 x) 0)))
(define pair? (lambda (x) (not (err? (cdr x)))))
(define symbol?
    (lambda (x)
        (and
            (not (err? x))
            (not (number? x))
            (not (pair? x)))))
(define atom?
    (lambda (x)
        (or
            (not x)
            (symbol? x))))
(define list?
    (lambda (x)
        (if (not x)
            #t
            (if (pair? x)
                (list? (cdr x))
                ()))))
(define equal?
    (lambda (x y)
        (or
            (eq? x y)
            (and
                (pair? x)
                (pair? y)
                (equal? (car x) (car y))
                (equal? (cdr x) (cdr y))))))
(define negate (lambda (n) (- 0 n)))
(define > (lambda (x y) (< y x)))
(define <= (lambda (x y) (not (< y x))))
(define >= (lambda (x y) (not (< x y))))
(define = (lambda (x y) (eq? (- x y) 0)))
(define list (lambda args args))
(define cadr (car (cdr x)))
(define caddr (car (cdr (cdr x))))
(define begin (lambda (x . args) (if args (begin . args) x)))
```

Explanation:

- `equal?` tests equality of two values recursively (`eq?` tests exact equality only)
- `list` returns a list of the values of the arguments. For example, `(list 1 2 (+ 1 2))` gives `(1 2 3)`.
- `begin` returns the last value of its last argument. For example, `(begin 1 2 (+ 1 2))` gives 3. This function is often used as a code block in Lisp, to evaluate a sequence of expressions,

which only makes sense if the expressions have side effects, such as `setq` to change the value of a variable.

E.2 Math Functions

The following functions should be self-explanatory:

```
(define abs
    (lambda (n)
        (if (< n 0)
            (- 0 n)
            n)))
(define frac (lambda (n) (- n (int n))))
(define truncate int)
(define floor
    (lambda (n)
        (int
            (if (< n 0)
                (- n 1)
                n))))
(define ceiling (lambda (n) (- 0 (floor (- 0 n)))))
(define round (lambda (n) (+ (floor n) 0.5)))
(define mod (lambda (n m) (- n (* m (int (/ n m))))))
(define gcd
    (lambda (n m)
        (if (eq? m 0)
            n
            (gcd m (mod n m)))))
(define lcm (lambda (n m) (/ (* n m) (gcd n m))))
(define even? (lambda (n) (eq? (mod n 2) 0)))
(define odd? (lambda (n) (eq? (mod n 2) 1)))
```

E.3 List Functions

The following functions should be self-explanatory, for details see further below:

```
(define length
    (lambda (t)
        (if t
            (+ 1 (length (cdr t)))
            0)))
(define append
    (lambda (s t)
        (if s
            (cons (car s) (append (cdr s) t))
            t)))
(define rev1
    (lambda (r t)
        (if t
            (rev1 (cons (car t) r) (cdr t))
            r)))
(define reverse (lambda (t) (rev1 () t)))
(define member
    (lambda (x t)
        (if t
```

```
            (if (equal? x (car t))
                t
                (member x (cdr t)))
            t)))
(define foldr
    (lambda (f x t)
        (if t
            (f (car t) (foldr f x (cdr t)))
            x)))
(define foldl
    (lambda (f x t)
        (if t
            (foldl f (f (car t) x) (cdr t))
            x)))
(define min
    (lambda args
        (foldl
            (lambda (x y)
                (if (< x y)
                    x
                    y))
            9.999999999e99
            args)))
(define max
    (lambda args
        (foldl (lambda (x y)
            (if (< x y)
                y
                x))
        -9.999999999e99
        args)))
(define filter
    (lambda (f t)
        (if t
            (if (f (car t))
                (cons (car t) (filter f (cdr t)))
                (filter f (cdr t)))
            ())))
(define all?
    (lambda (f t)
        (if t
            (if (f (car t))
                (all? f (cdr t))
                ())
            #t)))
(define any?
    (lambda (f t)
        (if t
            (if (f (car t))
                #t
                (any? f (cdr t)))
            ())))
(define map1
    (lambda (f t)
        (if t
            (cons (f (car t)) (map1 f (cdr t)))
```

```
            ()))))
(define map
    (lambda (f . args)
        (if (any? null? args)
            ()
            (let
                (mapcar (map1 car args))
                (mapcdr (map1 cdr args))
                (cons (f . mapcar) (map f . mapcdr))))))
(define zip (lambda args (map list . args)))
(define range
    (lambda (n m . args)
        (if args
            (if (< 0 (* (car args) (- m n)))
                (cons n (range (+ n (car args)) m (car args)))
                ())
            (if (< n m)
                (cons n (range (+ n 1) m))
                ())))))
```

Explanation:

- `member` checks list membership and returns the rest of the list where `x` was found.
- `foldr` and `foldl` return the value of right- and left-folded lists `t` using an operator `f` and initial value: `x (foldl` $\oplus$ x_0 `'(`x_1 x_2 `...` x_n`))` $= (\cdots((x_0 \oplus x_1) \oplus x_2) \oplus \cdots x_n)$ and `x (foldr` $\oplus$ x_0 `'(`x_1 x_2 `...` x_n`))` $= (x_1 \oplus (x_2 \oplus (\cdots (x_n \oplus x_0))))$.
- `filter` returns a list with elements x from the list `t` for which `(f` x`)` is true.
- `all?` returns `#t` if all elements x of the list `t` satisfy `(f` x`)` is true, and returns `()` otherwise.
- `any?` returns `#t` if any one of the elements x of the list `t` satisfy `(f` x`)` is true, and returns `()` otherwise.
- `map` applies `f` to the elements of a list or to n lists for n-ary function `f`.
- `zip` takes n lists of length m to return a list of length m with lists of length n.
- `range` generates a list of successive values `n` up to but not including `m` with an optional step value.

E.4 Higher-Order Functions

The following functions that take one or more functions as arguments to construct new functions are explained further below:

```
(define curry (lambda (f x) (lambda args (f x . args))))
(define compose (lambda (f g) (lambda args (f (g . args)))))
(define Y (lambda (f) (lambda args ((f (Y f)) . args)))
```

Explanation:

- `curry` takes a function `f` and an argument `x` and returns a function that applies `f` to `x` and the given arguments.
- `compose` takes two function `f` and `g` and returns a function that applies `g` to the arguments followed by the application of `f` to the result.
- `Y` is the Y combinator that takes a function `f` to return a function that applies `f` to `(Y f)` that is a copy of itself and the given arguments. The Y combinator can be used for recursion without naming the function. For example the factorial of 5 is 120:

```
> ((Y (lambda (f) (lambda (k) (if (< 1 k) (* k (f (- k 1))) 1)))) 5)
120
```