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

Dr. Robert A. van Engelen

July 9, 2022; updated January 8, 2024

"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 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 I will show 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. *Static scoping* is also known as *lexical scoping*.

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. He is the single author of the Ctadel code generation system and CAS for partial differential equations, the gSOAP toolkit for C/C++ web services, the ugrep search utility, the RE/flex lexical analyzer, Husky functional programming, Forth500 and many other projects in academia and industry, 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 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. 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 ordinary 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 `make-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 body 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 a `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 small without macro processing. To add macro processing, see Section 11.5.

```
> (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) x))
2
```

The `let` in this article do not require the binding list, so `(let* (x 3) (y 9) (/ (- y x) x))` works. Lisp implementations include more primitives, notably for arithmetic, logic and runtime type checking. Most Lisp implementations 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 IEEE 754 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:

$$| \; s \; \overbrace{b\;b\;b \,|\, b\;b\;b\;b \,|\, b\;b\;b\;b}^{\textit{exponent}} \;|\; \overbrace{b\;\underbrace{b\;b\;b}_{\textit{tag}} \,|\, \underbrace{\cdots \,|\, b\;b\;b\;b}_{\textit{also available}}}^{\textit{fraction}} \;|$$

where

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

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:

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

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 simpler 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 seems to think it's 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 `uint64_t*` or `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 will set a tag with `T(x) =` *tag* and 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[6]. 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 check or clear 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; }
```

[6] At least 64 bits, but we want 64 exactly. C is one of the oldest programming languages and had to accommodate systems with strange bit widths like 18 and 36 and so on, hence "at least".

With NaN boxing the number is returned "as is", but we could check if `n` is a NaN[7] 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 produce 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 really require defining an `equ` function. It just wraps `==`. But we include it here since we may have to redefine it depending on the BCD arithmetic performed by a specific 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 our 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[8] in the `main` function as follows:

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

[7]Checking if a value `n` is a NaN is easy, just do an equality check on itself: `if (n != n)` *NaN-action-here.*

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

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 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<<3) abort();
  return box(ATOM,i);
}
```

where `hp` is the *heap pointer* pointing to free bytes available on the heap. When `i==hp` we execute `strlen(strcpy(A+i,s))+1)` to copy the string `s` to the free heap address at `A+hp` and return the length of the string plus one to increase `hp`. `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.

The `atom` function searches the heap at addresses `A+i` until a matching atom name is found to return `box(ATOM,i)`. If the atom name is new, then space for the atom's string name is allocated and 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,4)` and `nil` on the stack and storing two atoms `ERR` and `#t` in the heap:

`cell[1023]:`	`box(ATOM,4)`	*stack*
`cell[sp=1022]:`	`nil`	↓
	free	
`A+hp=A+7:`	*space*	
`A+4:`	"`#t\0`"	↑
`A+0:`	"`ERR\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 or it is `nil` at the end of 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<<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<<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,4)` representing `#t` and `cdr(p)` returns `nil`, the empty list.

Closures and environment lists are constructed with the `cons` function applied twice, first to construct the name-value Lisp pair[9] `(v . x)`, then to place the pair in front of the Lisp environment list. We define a `pair` function for this purpose:

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

A closure is a `CLOS`-tagged `pair(v,x,e)` representing an instantiation of a Lisp `(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))); }
```

[9]The Lisp pair denoted with a dot `(v . x)` is constructed by the Lisp parser with `p = cons(v,x)`, whereas a Lisp list always ends in `nil`, such as `(1 2)` constructed by the Lisp parser with `cons(1,cons(2,nil))`.

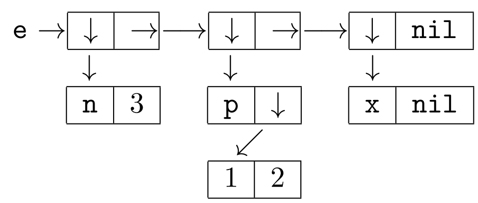
The conditional `equ(e,env) ? nil : e` forces the scope of a closure to be `nil` if `e` is the global environment `env`. Later, when we apply the closure, we check if its environment is `nil` and use the current global environment. This permits recursive calls and calls to forward-defined functions,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

```
L assoc(L v,L e) {
  while (T(e) == CONS && !equ(v,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 Lisp environment `e = ((n . 3) (p . (1 . 2)) (x . nil))` where the Lisp expression `(1 . 2)` constructs a pair 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;
}
```

The `prim[]` array contains all pre-defined primitives. A primitive is stored in a structure with its name as a string and a function pointer to its code. 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`[10] 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 with `cdr(car(f))`. The evaluation is performed with an extended environment produced with the `bind` function. This extended environment includes the bindings of the lambda's variables to the evaluated arguments. Remember that a closure was constructed for a `lambda` to include its static scope or `nil` as its environment? We retrieve its environment with `cdr(f)` and check it for `nil` with `not(cdr(f)) ? env : cdr(f)`, which gives the current global environment `env` or the lambda's local static environment `cdr(f)`. This environment is used with `bind` to bind the lambda variables `car(car(f))` with the evaluated arguments `evlis(t,e)`.

Arguments passed to a function or primitive 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)) :
         T(t) == ATOM ? assoc(t,e) :
         nil;
}
```

where `evlis` recursively traverses the list of expressions `t` to create a new list with their values. Recursion bottoms out at a non-`CONS` `t`. It is important to correctly handle the dot operator in a list of actual arguments in this way, 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 a 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. The name is bound to the rest of the list `t` with `pair(v,t,e)`. The latter happens when a single variable is used like `(lambda args args)` and when a dot is used in the list of variables like `(lambda (x . args) args)`.

[10] 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.

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:

```
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, 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)`, `(cons 1 2)` gives the pair `(1 . 2)` and `(cons 1 (cons 2 ()))` gives the list `(1 2)`.

6.4 `car` and `cdr`

`(car` *pair*`)` and `(cdr` *pair*`)` give the first and second element of a *pair*, respectively. This means that `(car` *list*`)` and `(cdr` *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); }
```

Note that the arithmetic functions expect at least one argument. They are undefined if no arguments are provided. 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 function. I will leave it to you to change the implementation to support this feature.

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 `()`. The less-than primitive compares numbers only and is always false for non-numeric arguments. Non-numeric types can be compared by comparing tags first and when equal comparing the `ord` values for example.

6.8 Logic

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

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

Only the first arguments to `or` and `and` are evaluated to determine the result. Example: `(and #t ())` gives `()` and `(and 1 2)` gives `2` since numbers are not `()`. Note that `and` returns the value of the *last expression* or `()`. Like the Lua programming language, the `or` and `and` combined produce an if-then-else of the form `(or (and` *test* *then*`)` *else*`)`. However, like Lua, this is not correct when *test* evaluates to true but *then* is nil (the `()` empty list.) In that case *else* is evaluated.

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[11] 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 t) { return T(t) != NIL && !not(cdr(t)); }
L f_leta(L t,L e) {
  for (; let(t); t = cdr(t)) e = pair(car(car(t)),eval(car(cdr(car(t))),e),e);
  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 in the list of variables `(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[12]. Otherwise, we shall use `(let* (args` x`) (f . args))` to ensure `args` is a variable bound to expression x.

[11] Other Lisp require pairs (*var expr*) to be placed in a list such as `(let* ((a 3) (b (* a a))) (+ a b))`, but I prefer the simpler form shown here. Of course, you can change this implementation in any way you like.

[12] The reason is that if we place a list after the dot like `(f . (g args))`, then the arguments passed to `f` are actually `g` and `args` which is not what we want.

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() { int c = getchar(); see = c; if (c == EOF) exit(0); }
I seeing(char c) { return c == ' ' ? see > 0 && see <= c : see == c; }
char get() { char c = see; look(); return c; }
char scan() {
  int i = 0;
  while (seeing(' ')) look();
  if (seeing('(') || seeing(')') || seeing('\'')) buf[i++] = get();
  else do buf[i++] = get(); while (i < 39 && !seeing('(') && !seeing(')') && !seeing(' '));
  return buf[i] = 0,*buf;
}
```

This implementation of `scan` does not support Lisp comments. I did this intentionally, since it is up to you to write your own implementation with the features that *you* want.

Lisp comments begin with a semicolon and end at the next line. To support comments, we can change the second line of the `scan` function to skip comments and continue scanning:

```
while (seeing(' ') || seeing(';')) if (get() == ';') while (!seeing('\n')) look();
```

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, by 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. *Beware that the number of cells* `N` *must be increased to import this many definitions!*

To parse Lisp expressions we employ a *recursive-descent parsing* technique. The `read` function[13] returns a Lisp expression parsed from the input by invoking the `scan` and `parse` functions:

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

[13] Not to be confused with the `unistd.h` `read` function. To avoid link failures, I've renamed it to `Read` in the source code of the tinylisp repository `https://github.com/Robert-van-Engelen/tinylisp`.

where the `parse` function parses a list, a quoted expression, or an atomic expression (a symbol 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*$_n$`)` sometimes called `progn` that returns the value of the last expression *expr*$_n$. So its use is sanctioned by Lisp to sequence expressions as statements with *side effects*[14].

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) > 0 && !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. Note that `sscanf` accepts `inf`, `-inf` and `nan` as numbers and hexadecimal `0xh...h`, all of which we get it for free. Since a NaN corresponds to the `ERR` atom with zero offset into the heap, `nan` is reported as `ERR`.

The PC-G850 requires function `atomic` to be modified as follows:

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

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

[14] Functions with side effects affect the state of the machine outside of their arguments and locals, such as through assignments to non-local variables and by performing IO operations. Pure functional programming bans them. For C and Lisp we must properly sequence functions with side effects to avoid undefined behavior.

8 Printing Lisp Expressions

Displaying Lisp expression 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. It prints a dot for the last cons pair if the list does not end in `nil`:

```
void printlist(L t) {
  for (putchar('('); ; putchar(' ')) {
    print(car(t));
    if (not(t = cdr(t))) break;
    if (T(t) != CONS) { printf(" . "); print(t); break; }
  }
  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 tiny, 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 stack in cells above the last `env` pair on the stack. These cells are not removed by `gc` when we set `sp = ord(env)`.

One caveat of this approach is that we cannot support interactive use of the Lisp special forms `setq` (modifies an association in an environment, Section 11.3), `set-car!` and `set-cdr!` (overwrites the `car` or `cdr` of a pair, Section 11.4), since these may change previously-defined expressions in the global environment. If the modified global environment references temporary lists, then `gc` corrupts the global environment by removing these lists. We can support `setq`, `set-car!` and `set-cdr!` if we only assign atomic values to globals. Locals are always assignable.

This simple one-line garbage collector does not remove unused symbols. Temporary atoms on the heap are kept. A minor addition suffices to remove unused atoms from the heap. We only need to find the max heap reference among the used `ATOM`-tagged cells and adjust `hp` accordingly:

```
void gc() {
  I i = sp = ord(env);
  for (hp = 0; i < N; ++i) if (T(cell[i]) == ATOM && ord(cell[i]) > hp) hp = ord(cell[i]);
  hp += strlen(A+hp)+1;
}
```

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/8); 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[15] 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.

[15] That is, a Lisp-like functional style of structured C. Lines are 55 columns wide on average and never wider than 120 columns for convenient editing.

11.2 `let` and `letrec*`

The `let` special form[16] 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;
  for (; let(t); t = cdr(t)) d = pair(car(car(t)),eval(car(cdr(car(t))),e),d);
  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) {
  for (; let(t); t = cdr(t)) {
    e = pair(car(car(t)),err,e);
    cell[sp+2] = eval(car(cdr(car(t))),e);
  }
  return eval(car(t),e);
}
... prim[] = { ... {"letrec*", f_letreca} ... };
```

This implementation adds new variable-`err` bindings[17] to the environment `e` , then overrides the `err` cell on the stack with 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 v = car(t),x = eval(car(cdr(t)),e);
  while (T(e) == CONS && !equ(v, 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, atomic values are always safe to assign and `setq` is safe to use to assign local variables in the scope of a `lambda` and a `let`.

[16] This `let` syntax differs from other Lisp: like our `let*` we don't need to put all the *var-expr* pairs in a list. Change the syntax as you like.

[17] you can pick `nil` instead of `err` if you've implemented error handling in Section 14

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

`(set-car!` *pair expr*`)` sets the value of the `car` cell of a cons *pair* to *expr* as a side-effect, `(set-cdr!` *pair expr*`)` sets the value of the `cdr` cell of a cons *pair* to *expr* as a side-effect.

```
L f_setcar(L t,L e) {
  L p = car(t = evlis(t,e));
  return (T(p) == CONS) ? cell[ord(p)+1] = car(cdr(t)) : err;
}
L f_setcdr(L t,L e) {
  L p = car(t = evlis(t,e));
  return (T(p) == CONS) ? cell[ord(p)] = car(cdr(t)) : err;
}
... 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 applied, thereby *expanding* and evaluating the Lisp code in place.

To add the `(macro` *variables expression*`)` primitive is easy. It 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;
```

We must modify the `car` and `cdr` functions to apply 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` and 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 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 closure of this `lambda` includes the proper environment in which `x` should be evaluated, thus respecting its static scope. The `force` function evaluates a promise:

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

More information and examples of `delay` and `force` can be found in Lisp textbooks and manuals.

With macros we can also define `defun` to define functions more easily without a `lambda`:

```
> (define defun (macro (f v x) (list 'define f (list 'lambda v x))))
> (defun square (n) (* n n))
> (square 3)
9
```

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 t,L e) { L x; char c = see; see = ' '; x = read(); see = c; return x; }
L f_print(L t,L e) {
  for (t = evlis(t,e); T(t) != NIL; t = cdr(t)) print(car(t));
  return nil;
}
L f_println(L t,L e) { f_print(t,e); putchar('\n'); return nil; }
... prim[] = { ... {"read",f_read},{"print",f_print},{"println",f_println} ... };
```

Example: `(read)` gives the Lisp expression typed in (unevaluated), `(print 'hello 123)` displays `hello123`, and `(println '(hello world))` displays `(hello world)`.

With a few more lines of C code, you can add your own Lisp primitives for a more complete IO implementation. You could store open file `FILE*` streams in an array to support `open` and `close` primitives on multiple streams in Lisp. Then index this array by a float number in Lisp to obtain the `FILE*` for an internal IO operation. Some Lisp implementations include a special *port* type for this, which is basically a `FILE*`.

12 Adding Readline with History

Command shells such as bash use GNU readline for interactive input. Readline plays nice with interactive input and provides a history mechanism to let us recall previous input. The readline library will be a nice addition to our Lisp. While we are at it, we might as well read a Lisp initialization file `init.lisp` with one or more Lisp definitions to import before the interactive prompt. First we need to include the usual C headers. There are two for readline:

```
#include <readline/readline.h>
#include <readline/history.h>
```

Besides the `buf` and `see` globals, we also need a `ptr` pointing at the current character in the `line` string returned by `readline`. We also add a prompt string `ps` to display the Lisp prompt and an `in` pointer to the open file with input, i.e. to read `init.lisp` when opened successfully:

```
char buf[40],see = ' ',*ptr = "",*line = NULL,ps[20];
FILE *in = NULL;
```

The `look` function is modified to read a character from `in` when `in` is not NULL, or to read a line of input with by `readline` when we reach the end of the last line read, i.e. when `see=='\n'`:

```
void look() {
  if (in) {
    int c = getc(in);
    see = c;
    if (c != EOF) return;
    fclose(in);
    in = NULL;
  }
  if (see == '\n') {
    if (line) free(line);
    while (!(ptr = line = readline(ps))) freopen("/dev/tty","r",stdin);
    add_history(line);
    strcpy(ps,"?");
  }
  if (!(see = *ptr++)) see = '\n';
}
```

To read `init.lisp` or a file specified on the command line, let's modify `main` as follows:

```
int main(int argc,char **argv) {
  ...
  in = fopen((argc > 1 ? argv[1] : "init.lisp"),"r");
  using_history();
  while (1) { putchar('\n'); snprintf(ps,20,"%u>",sp-hp/8); print(eval(read(),env)); gc(); }
}
```

Note that the modified REPL populates the prompt string `ps` with `snprintf` to display the number of remaining free cells. The prompt string is set to `?` in the `look` function when the interactive input is not yet complete. The `libreadline` library must be linked with our updated source:

```
bash$ cc -o tinylisp-opt.c -lreadline
```

Now you're all set to enjoy enhanced interactive input!

13 Tracing Lisp

Everything that is happing behind the scenes in our Lisp interpreter is essentially just the traversal of lists as code to return numbers, atoms, closures and lists as data. But what does that actually look like? Well, let's find out by tracing the evaluation steps in our Lisp. When we display every evaluation step performed by `eval` we see exactly what is happening in detail:

```
tinylisp
930>(define sq (lambda (n) (* n n)))
920: define => <define>
920: lambda => <lambda>
916: (lambda (n) (* n n)) => {916}
912: (define sq (lambda (n) (* n n))) => sq
sq
901>(sq 3)
908: sq => {916}
908: 3 => 3
908: () => ()
902: * => <*>
902: n => 3
902: n => 3
902: () => ()
898: (* n n) => 9
898: (sq 3) => 9
9
901>
```

A trace displays the Lisp code and data before `eval` followed by the Lisp code and data after `eval`. To implement tracing, we rename `eval` to `step` and add a new `eval` function to display the trace:

```
L step(L x,L e) { ... }    /* this is the old eval() function renamed to step() */
void print(L);             /* function prototype moved up since we need print() */
L eval(L x,L e) {
  L y = step(x,e);
  printf("%u: ",sp); print(x); printf(" => "); print(y);
  while (getchar() >= ' ') continue;
  return y;
}
```

The `while`-loop in `eval` waits until the RETURN key is pressed (or any special key or EOF). This always displays the trace of your Lisp. With a few more lines of C, you could make tracing an option enabled with a new `(trace` *state*`)` Lisp function that takes *state* 0, 1 or 2 to set a global variable. When this variable is set to 1 or 2, the trace output is displayed in `eval` and when 2 it also waits for a keypress. With this addition to your Lisp, you can now step through your code on demand. In addition to tracing, you could also dump the stack. For example, when the user types a `d` for dump followed by RETURN.

14 Adding Error Handling and Exceptions to Lisp

Error handling is practically non-existent in our Lisp at this point. It is an error to take the `car` or `cdr` of a non-pair, to use the value of an undefined symbol, or to try to apply a non-function to arguments. These three error conditions produce the `ERR` value in our Lisp. We could extensively debug our code by tracing (see Section 13) to find all possible bugs and add `err?` checks to our Lisp functions to catch problems (see Appendix E for a definition of `err?`). Analyzing your Lisp code for correctness and debugging your Lisp functions is critical, but adding `err?` calls to Lisp code is cumbersome. Wouldn't it be nice to have a mechanism to catch exceptions?

There are two mechanisms to implement exception handling in our Lisp: in C using `setjmp` or in C++ with `try-catch` if we rename and compile our project in C++.

14.1 C `setjmp`

The `setjmp(jmp_buf jb)` function saves its *calling environment* in `jb` and returns zero. The corresponding `longjmp(jmp_buf jb,int n)` function restores the calling environment `jb` saved by the most recent `setjmp(jb)` and passes `n` to the return value of this `setjmp`. What basically happens is that `longjmp` deletes all active function calls since the last `setjmp` and immediately returns to `setjmp`. This mechanism is somewhat similar to C++ exceptions with `longjmp` acting like `throw` and `setjmp` acting like `try-catch`. As with C++ exceptions, we must be careful to finalize unfinished business before throwing exceptions or add exception handlers to finalize unfinished business, re-throwing the exception when applicable. Unfinished business include open files that must be closed and memory allocations that must freed. Fortunately, we have none of these issues to worry about[18] in our Lisp interpreter.

After including `setjmp` in our interpreter, we define a global calling environment `jmp_buf jb` and a new `err` function to "throw" errors by invoking `longjmp(jb,n)` for nonzero error codes `n`:

```
#include <setjmp.h>
...
jmp_buf jb;
L err(int i) { longjmp(jb,i); }
```

Note that the new `err` function doesn't return a value (you should be able to figure out why it makes no sense to return a value.) After defining a new `err` function, we change the uses of `err` in our implementation to throw three different error codes:

```
L car(L p) { return (T(p)&~(CONS^CLOS)) == CONS ? cell[ord(p)+1] : err(1); }
L cdr(L p) { return (T(p)&~(CONS^CLOS)) == CONS ? cell[ord(p)] : err(1); }
...
L assoc(L v,L e) {
  while (T(e) == CONS && !equ(v,car(car(e)))) e = cdr(e);
  return T(e) == CONS ? cdr(car(e)) : err(2);
}
...
```

[18] Unless you've added file open and close to your Lisp interpreter's primitives. In that case, it helps to keep a table of open file descriptors and close them all when an exception occurs.

```
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(3);
}
```

All other `err` values should be removed from the Lisp interpreter[19]. In `main` we invoke `setjmp(jb)` to initialize `jb` and also to catch errors when `setjmp` returns a nonzero value n from `err(`n`)`:

```
int main() {
  ...
  if ((i = setjmp(jb)) != 0) printf("ERR %d",i);
  while (1) { gc(); printf("\n%u>",sp-hp/8); print(eval(read(),env)); }
}
```

We also moved `gc()` up. After an error is caught by `setjmp` we report it and commence the REPL:

```
930>(car 3)
ERR 1
930>(1 2)
ERR 3
930>'(1 2)
(1 2)
930>
```

Another good idea is to replace `abort()` with `err(4)` to avoid aborting when we ran out of memory.

In addition to our new and practical exception handling mechanism, there are several ways Lisp `catch` and `throw` primitives can be defined. The following is a simplified version that allows you to `(throw` n`)` an error code n that are caught as `(ERR .` n`)` pairs returned by `(catch` *expression*`)` when a `(throw` n`)` is invoked or when an `ERR` n occurs when *expression* is evaluated:

```
L f_catch(L t,L e) {
  L x; int i;
  jmp_buf savedjb;
  memcpy(savedjb,jb,sizeof(jb));
  i = setjmp(jb);
  x = i ? cons(atom("ERR"),i) : eval(car(t),e);
  memcpy(jb,savedjb,sizeof(jb));
  return x;
}
L f_throw(L t,L e) { longjmp(jb,(int)num(car(t))); }
... prim[] = { ... {"catch", f_catch},{"throw", f_throw} ... };
```

where `throw` is a special form that does not evaluate its argument, which must be a constant integer. With some more C code you could pass along the cell index of a cons pair on the stack instead of error codes. That means you could throw lists as errors with more information. However, care must be taken to distinguish internal error codes from explicit exceptions thrown.

[19] You may still want to call `atom("ERR")` in `main` to populate the heap with an `ERR` atom first, because its zero `ord` is implicitly associated with a quiet NaN, or update `num` to throw an error when its argument is NaN.

14.2 Using C++ Exceptions

Compiling our Lisp interpreter in C++ requires almost no change (e.g. renaming `not` to `Not`). With C++ we can use `try-catch` and `throw` in the REPL:

```
L err(int i) { throw(i); }
...
int main() {
  ...
  while (1) {
    printf("\n%u>",sp-hp/8);
    try { print(eval(read(),env)); }
    catch (int i) { printf("ERR %d",i); }
    gc();
  }
}
```

We also change the `car`, `cdr`, `assoc` and `apply` functions as shown in the previous section. Lisp `catch` and `throw` primitives can be defined with the usual C++ `try-catch-throw`.

15 Downsizing Lisp to Single Floating Point Precision

Our Lisp supports double precision floating point values and operations on them without compromise. We can halve the memory use of our Lisp by using single precision floating point instead. This is a small compromise to make to reduce memory consumption if memory comes at a premium or if the machine does not support double floating point precision. Despite downsizing our Lisp, it won't be handicapped to run toy Lisp examples only. In fact, it can handle up to $N = 2^{20}/4 = 262,144$ cells (1,048,576 bytes) to evaluate Lisp.

A single precision NaN allows up to 22 bits to be arbitrarily used to stuff any information we want into a single precision NaN:

$$\mid \; s \; \overbrace{b\,b\,b \mid b\,b\,b\,b \mid b}^{exponent} \; \overbrace{b\; \underbrace{b\,b}_{tag} \mid \underbrace{\cdots \mid b\,b\,b\,b}_{also\ available}}^{fraction} \; \mid$$

All exponent bits must be set for NaN. This time we use the sign bit s together with our *tag* since we need 3 bits for tagging. Therefore, `NIL` has tag `0xfff` with the sign bit set as part of the tag. We also want to maximize the use of the 23 bit fraction. Of these 23 bits the most significant bit must be set for quiet NaN. We use 2 bits for the tag and are left with 20 bits to store an integer "ordinal" value to refer to atoms on the heap, to refer to primitives in the `prim[]` array, and to refer to cons cells and closure cells on the stack.

Only a few minor changes are required to the code to downsize our Lisp to single precision floating point with 4-byte stack cells. We adjust the NaN-boxing tags and reduce the ordinals to 20 bit by changing the NaN-boxing-related functions:

```
#define L float
#define T(x) *(uint32_t*)&x>>20
```

```
...
#define N 1024 /* N should not exceed 262144 = 2^20/4 cells = 1048576 bytes */
I hp=0,sp=N,ATOM=0x7fc,PRIM=0x7fd,CONS=0x7fe,CLOS=0x7ff,NIL=0xfff;
...
L box(I t,I i) { L x; *(uint32_t*)&x = (uint32_t)t<<20|i; return x; }
I ord(L x) { return *(uint32_t*)&x & 0xfffff; }
L num(L n) { return n; }
I equ(L x,L y) { return *(uint32_t*)&x == *(uint32_t*)&y; }
```

The float-addressing stack pointer `sp` should be scaled by 4 instead of 8 to compare to the byte-addressing heap pointer `hp` to check if they meet. Therefore, the `atom` and `cons` functions should be modified to use `sp<<2`:

```
L atom(const char *s) {
  ...
  if (i == hp && (hp += strlen(strcpy(A+i,s))+1) > sp<<2) abort();
  ...
}
L cons(L x,L y) {
  ...
  if (hp > sp<<2) abort();
  ...
}
```

Likewise, the prompt displayed in the REPL is changed to use `sp-hp/4`:

```
int main() {
  ...
  while (1) { printf("\n%u>",sp-hp/4); print(eval(read(),env)); gc(); }
}
```

Furthermore, we tweak the `atomic` and `print` functions to correctly parse and print single precision floats, respectively:

```
L atomic() { L n; int i; return sscanf(buf,"%g%n",&n,&i) > 0 && !buf[i] ? n : atom(buf); }
...
void print(L x) {
  ...
  else printf("%g",x);
}
```

Finally, we may also want to change `f_int` to truncate floats to 32 bit integers:

```
L f_int(L t,L e) { L n = car(evlis(t,e)); return n<1e7 && n>-1e7 ? (int32_t)n: n; }
```

16 Optimizing the interpreter

Understanding Lisp and Lisp evaluation does not require a deep understanding of the optimizations we may want to apply to make our interpreter run faster and use less memory. Optimization is generally nice, but a more critical goal is to optimize tail-calls performed by Lisp functions. Tail-call optimization effectively permits recursion in Lisp without the risk of running out of stack space. To achieve this, we start with optimizing recursion away in our interpreter by replacing recursive calls with loops when possible. Then focus on replacing `eval` with a tail-call optimized iterative version.

16.1 Replacing recursion with loops

The `evlis` C function is the first target we pick for optimization. It calls itself recursively to construct a list of evaluated expressions. The `evlis` function does this by consing the evaluated `car` expression of the list `t` to the `evlis` rest of the list `t` in `(cons(eval(car(t),e),evlis(cdr(t),e)))`. This is inefficient. Worse, the C compiler can't tail-call optimize this C code to avoid the calling overhead and the stack growth associated with it.

Effective C programming with pointers can be exploited to replace recursion in `evlis` by iteration using a pointer `p` that points to the last `cdr` cell of the list we are constructing in `evlis`. To construct the list iteratively, we just need to replace the last `nil` cell of the list that `*p` points to and replace it by a new cons, then update `p` to point to the `nil` of that cons with `p = cell+sp`:

```
L evlis(L t,L e) {
  L s,*p;
  for (s = nil,p = &s; T(t) == CONS; p = cell+sp,t = cdr(t)) *p = cons(eval(car(t),e),nil);
  if (T(t) == ATOM) *p = assoc(t,e);
  return s;
}
```

Instead of `p = cell+sp` which points to the last cell created on the stack that happens to be the `cdr` cell of the last `cons`, we can also use `p = &cell[ord(*p)]` or simply `p = cell+ord(*p)` as an alternative. Initially, `p` points to `s` with `s = nil` as the first step to get the list construction started. Once the list is complete, we return `s` as the result of `evlis`.

Eventually the list `t` runs out in a non-`CONS` value. This is typically a `T(t) == NIL`, but can instead be a symbol when the function application list uses the dot operator as in `(f x . args)` for example. The dotted `args` should be evaluated and appended to the list we construct. This is performed by `if (T(t) != NIL) *p = eval(t,e)` in the optimized version of `evlis`.

Another candidate C function to optimize is `list`, which parses a Lisp list:

```
L list() {
  L t,*p;
  for (t = nil,p = &t; ; *p = cons(parse(),nil),p = cell+sp) {
    if (scan() == ')') return t;
    if (*buf == '.' && !buf[1]) return *p = read(),scan(),t;
  }
}
```

Note that the optimization mirrors the `evlis` optimization by assigning `*p = cons(parse(),nil)` and updating the pointer `p = cell+sp`.

16.2 Tail-call optimization

Our interpreter implements an evaluation algorithm similar to, but not identical to, McCarthy's classic Lisp evaluator. It uses the functions `eval`, `evlis`, `bind` (a.k.a. `pairlis`), `apply`, `reduce`, and `assoc`. The `eval` function lies at the heart of this. It evaluates a function application represented as a list, say `x`. The list has the function stored at the head `car(x)` and the arguments stored in the rest of the list `cdr(x)`. The steps performed by `eval` and the functions it calls are as follows:

1. `eval(L x,L e)` calls `apply(eval(car(x),e),cdr(x),e)` which evaluates the function at the head with `eval(car(x),e)` first to obtain a closure or primitive to apply to the yet-to-be evaluated arguments stored in the rest of the list `cdr(x)`;

2. `apply(L f,L t,L e)` calls `reduce(f,t,e)` when the function `f` passed to it is a closure, where `t` are the unevaluated arguments to `f` and `e` is the current environment with bindings of symbols to values;

3. `reduce(L f,L t,L e)` calls `bind(car(car(f)),evlis(t,e),not(cdr(f)) ? env : cdr(f))` where the list of the closure's variables `car(car(f))` are pair-wise bound to the list of evaluated arguments `evlis(t,e)` starting with the bindings of the lexical scope of the closure `cdr(f)` if not empty or the global environment `env`, then reduces the closure's body `cdr(car(f))` with a call to `eval` as can be seen in the body of the `reduce` function that returns `eval(cdr(car(f)),bind(car(car(f)),evlis(t,e),not(cdr(f)) ? env : cdr(f)))`.

Note that the sequence of events started with `eval` making certain calls that eventually ended with a call to `eval`. Tail-call optimization aims to remove all intermediate calls that lead to the final `eval` call. Then, rather than calling `eval` again, tail-call optimization loops back the originating `eval`. This means that a Lisp function evaluation at the "tail end" do not incur any return stack growth. For example:

```
(define func1
    (lambda (n)
        (func2 (+ n 1))))
```

is tail-call optimized, because the application `(func2 (+ n 1))` is the body of `func1`. By contrast:

```
(define func1
    (lambda (n)
        (+ 1 (func2 n))))
```

is not tail-call optimized, because the result of `func2` is used by the addition operator to increment the value which is returned by `func1`.

Let's break down the following tail-call optimized `eval` implementation:

```
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;
    for (;T(v) == CONS && T(x) == CONS; v = cdr(v),x = cdr(x)) d = pair(car(v),eval(car(x),e),d);
    if (T(v) == CONS) x = eval(x,e);
    for (;T(v) == CONS; v = cdr(v),x = cdr(x)) d = pair(car(v),car(x),d);
    if (T(x) == CONS) x = evlis(x,e);
    else if (T(x) != NIL) x = eval(x,e);
    if (T(v) != NIL) d = pair(v,x,d);
    x = cdr(car(f)); e = d;
  }
}
```

It seems that a lot is going on here, but it is not complicated once we dig into it. The code looks a lot more convoluted than our original simple and elegant `eval`. Important to note is that `eval` loops until `x` is a symbol tagged `ATOM` or a constant, meaning something that is not applicable. By contrast, a `CONS` is a function application that we should perform. To do so, `f = eval(car(x),e)` gives us the closure/primitive `f`. We get the rest of the list of arguments with `x = cdr(x)`.

When `f` is a primitive, we call it with `if (T(f) == PRIM) return prim[ord(f)].f(x,e)` to return its value. Note that `ord(f)` is the index of the primitive into the `prim[]` array with function pointer `f` to call member function `f(x,e)` with unevaluated list of arguments `x` and the current environment `e`.

When `f` is a closure, we obtain its list of variables `v = car(car(f))` and lexical scope of bindings `d = cdr(f)`. If this scope is empty, the function's scope is global (see function `reduce` in Section 5) and we set it accordingly `if (T(d) == NIL) d = env`.

The bindings to the evaluated arguments are performed by the two `for`-loops that essentially combine the functionalities of `evlist` and the former `bind`. The conditions are checked to handle the dot operator in the list of actual arguments and in the list of formal arguments (the variables) of a `lambda` special form. Finally, we assign the body of the closure `x = cdr(car(f))` to replace `x` to be evaluated next in the updated environment `e = d`.

The optimization got rid of `apply`, `bind` and `reduce`, which won't be needed any longer. This optimized implementation performs the same operations as the previous unoptimized implementation, which can be viewed as a reference implementation that implements the Lisp interpreter requirements[20].

Tail-call optimization is only effective with full garbage collection. Our tiny interpreter does not perform garbage collection continuously, but rather waits until returning to the prompt to reclaim memory. This means that evaluated arguments will continue to accumulate in memory and this will eventually exhaust memory. If we can reclaim the list of evaluated arguments before making a tail-call, then we will not run out of memory. Some other small Lisp interpreters use a so-called "ABC" garbage collector, which immediately reclaims the evaluated list of arguments passed to a function when the function returns (or tail-calls). It assumes that Lisp data structures are not cyclic. However, this assumption is not a sufficient requirement for "ABC" garbage collection to be sound. Our Lisp interpreter supports forms like `(lambda args args)` and `(lambda (x . args) args)` that use a single variable `args` to refer to the (partial) list of evaluated arguments passed to the closure. The `args` reference is invalidated after "ABC" garbage collection. Hence, "ABC" garbage collection is not safe. The next sections apply more optimizations to reduce memory usage.

[20]We want our Lisp interpreter to support the dot operator in function application lists and in `lambda` variable lists, such as `(lambda (x . args) args)`, but other Lisp may not support these useful forms.

16.3 Tail-call optimization part deux

Our new tail-call optimized Lisp interpreter works well to optimize "tail end" calls of user-defined functions. But it doesn't work when we use an `if` or a `cond`, which are quite common to limit recursion. For example, the following `foldl` (*fold left*, a.k.a. `reduce`) and `begin` (a.k.a. `progn`) functions are both tail-recursive in the "then-branch" of the `if` special form:

```
(define foldl
    (lambda (f x t)
        (if t
            (foldl f (f (car t) x) (cdr t))
            x)))

(define begin
    (lambda (x . args)
        (if args
            (begin . args)
            x)))
```

In order to tail-call optimize Lisp evaluation through the special forms `if`, `cond`, and `let*`, we mark these three primitives in the `prim[]` array by introducing a new member `short t` flag with the value 1 for tail-calls:

```
struct { const char *s; L (*f)(L,L*); short t; } prim[] = {
{"eval",  f_eval,  1},
{"quote", f_quote, 0},
{"cons",  f_cons,  0},
{"car",   f_car,   0},
{"cdr",   f_cdr,   0},
{"+",     f_add,   0},
{"-",     f_sub,   0},
{"*",     f_mul,   0},
{"/",     f_div,   0},
{"int",   f_int,   0},
{"<",     f_lt,    0},
{"eq?",   f_eq,    0},
{"or",    f_or,    0},
{"and",   f_and,   0},
{"not",   f_not,   0},
{"cond",  f_cond,  1},
{"if",    f_if,    1},
{"let*",  f_leta,  1},
{"lambda",f_lambda,0},
{"define",f_define,0},
{0}};
```

Perhaps surprisingly, we also make `eval` in `prim[]` a tail-call because it just returns its argument to be evaluated next. When the `t` member flag is set, we continue the `eval` function's loop to evaluate the Lisp expression `x` returned by the tail-call enabled primitive:

```
L eval(L x,L e) {
  L f,v,d;
  while (1) {
    ...
```

```
    if (T(f) == PRIM) {
      x = prim[ord(f)].f(x,&e);
      if (prim[ord(f)].t) continue;
      return x;
    }
    ...
  }
}
```

In addition, the `if`, `cond`, and `let*` primitives no longer call `eval` before returning to return the expression to be evaluated instead:

```
L f_cond(L t,L *e) {
  while (T(t) != NIL && not(eval(car(car(t)),*e))) t = cdr(t);
  return car(cdr(car(t)));
}
L f_if(L t,L *e) {
  return car(cdr(not(eval(car(t),*e)) ? cdr(t) : t));
}
L f_leta(L t,L *e) {
  for (;let(t); t = cdr(t)) *e = pair(car(car(t)),eval(car(cdr(car(t))),*e),*e);
  return car(t);
}
```

The `f_leta` primitive shows why we pass `*e` to the primitives instead of `e` by value. This permits the primitive to extend the environment to continue evaluation with an expression `x` that has an extended scope of bindings `e`.

The rest of the primitives remain the same, except that we pass a pointer to the current environment `*e` and for the updated `f_eval` implementation as stated earlier:

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

Appendix C and D show the optimized Lisp interpreter source code.

16.4 Optimizing the Lisp primitives

The `evlis` function constructs a list of evaluated arguments. It is a fundamental Lisp interpreter function. While optimizing our Lisp interpreter we got rid of several Lisp interpreter functions, but not `evlis`, which is called in `eval` to handle the dot operator in lambda variable lists by binding the list of evaluated arguments to a variable. All other calls to `evlis` in our Lisp interpreter are made to evaluate the arguments passed to a primitive. The important point is that `evlis` correctly handles the dot operator in the arguments passed to a primitive. For example, we can define a `sum` function that calls `+` on its list argument `t`:

```
(define sum (lambda (t) (+ . t)))
(sum '(1 2 3))
6
```

Here, the `evlis` call in `f_add` simply passes the list `t = (1 2 3)` back as a return value to be summed in `f_add`'s loop over its arguments. If we were to blindly remove `evlis` from `f_add` to replace it with a call to `eval` for each unevaluated `car(t)` in the list of unevaluated arguments `t`, then we will lose the ability to pass arguments via the dot operator.

Rather than calling `evlis` in a primitive, we implement an iterative approach that calls a function `arg` repeatedly. This function returns the next evaluated argument from the list of arguments `t` passed to the primitive:

```
L arg(short *d,L *t,L x,L e) {
  if (T(*t) == ATOM && !*d) { *t = assoc(*t,e); *d = 1; }
  if (T(*t) != CONS) return x;
  x = *d ? car(*t) : eval(car(*t),e);
  *t = cdr(*t);
  if (T(*t) == ATOM) { *t = assoc(*t,e); *d = 1; }
  return x;
}
```

We pass a pointer `*t` to `arg` to be updated by `arg` to point to the rest of the list of arguments that are not yet evaluated. The `arg` function checks if `t` ends in a symbol, which is the variable after the dot operator. In this case, the value of the variable should be looked up with `assoc` and its list is used as the remaining list of arguments that are already evaluated. When this happens, a flag `*d` is set to prevent double evaluation of dot operator arguments. We also pass a default value `x` to `arg` when no arguments are provided.

The `short` flag `d` is a locally-declared variable of the primitive, which is initially zero and set when the dot operator argument was evaluated. The first set of primitives is modified as follows:

```
L f_eval(L t,L *e) { short d = 0; return arg(&d,&t,err,*e); }
L f_quote(L t,L *_) { return car(t); }
L f_cons(L t,L *e) { short d = 0; L x = arg(&d,&t,err,*e),y = arg(&d,&t,err,*e); return cons(x,y); }
L f_car(L t,L *e) { short d = 0; return car(arg(&d,&t,err,*e)); }
L f_cdr(L t,L *e) { short d = 0; return cdr(arg(&d,&t,err,*e)); }
L f_add(L t,L *e) {
  short d = 0; L n = arg(&d,&t,0,*e);
  while (T(t) == CONS) n += arg(&d,&t,0,*e);
  return num(n);
}
L f_sub(L t,L *e) {
  short d = 0; L n = arg(&d,&t,0,*e);
  while (T(t) == CONS) n -= arg(&d,&t,0,*e);
  return num(n);
}
L f_mul(L t,L *e) {
  short d = 0; L n = arg(&d,&t,1,*e);
  while (T(t) == CONS) n *= arg(&d,&t,1,*e);
  return num(n);
}
L f_div(L t,L *e) {
  short d = 0; L n = arg(&d,&t,1,*e);
  while (T(t) == CONS) n /= arg(&d,&t,1,*e);
  return num(n);
}
L f_int(L t,L *e) { short d = 0; L n = arg(&d,&t,err,*e); return n<1e16 && n>-1e16 ? (long long)n : n; }
L f_lt(L t,L *e) { short d = 0; L x = arg(&d,&t,err,*e),y = arg(&d,&t,err,*e); return x < y ? tru : nil; }
L f_eq(L t,L *e) { short d = 0; L x = arg(&d,&t,err,*e),y = arg(&d,&t,err,*e); return equ(x,y) ? tru : nil; }
L f_not(L t,L *e) { short d = 0; return not(arg(&d,&t,nil,*e)) ? tru : nil; }
```

This optimization is an example of a plethora of possibilities to increase the speed of Lisp evaluation and/or reduce memory usage. In this case we reduced memory usage, but the computational overhead by repeatedly calling `arg` increases. Furthermore, the resulting code becomes rather opaque compared to the original simple and elegant Lisp implementation that we started with.

17 Conclusions

This article demonstrated how a fully-functional Lisp interpreter with 20 Lisp primitives, a REPL and simple garbage collection 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 article, 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 coincidental. I wrote this 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.

18 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<<3) abort();
 return box(ATOM,i);
}
L cons(L x,L y) { cell[--sp] = x; cell[--sp] = y; if (hp > sp<<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 v,L e) { while (T(e) == CONS && !equ(v,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 && !not(cdr(x)); }
L eval(L,L),parse();
L evlis(L t,L e) { return T(t) == CONS ? cons(eval(car(t),e),evlis(cdr(t),e)) : T(t) == ATOM ? assoc(t,e) : nil; }
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(evlis(t,e))) ? tru : nil; }
L f_or(L t,L e) { L x = nil; while (T(t) != NIL && not(x = eval(car(t),e))) t = cdr(t); return x; }
L f_and(L t,L e) { L x = nil; while (T(t) != NIL && !not(x = eval(car(t),e))) t = cdr(t); return x; }
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) { for (;let(t); t = cdr(t)) e = pair(car(car(t)),eval(car(cdr(car(t))),e),e); 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() { int c = getchar(); see = c; if (c == EOF) exit(0); }
I seeing(char c) { return c == ' ' ? see > 0 && see <= c : see == c; }
```

```
char get() { char c = see; look(); return c; }
char scan() {
 int i = 0;
 while (seeing(' ')) look();
 if (seeing('(') || seeing(')') || seeing('\'')) buf[i++] = get();
 else do buf[i++] = get(); while (i < 39 && !seeing('(') && !seeing(')') && !seeing(' '));
 return buf[i] = 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; int i; return sscanf(buf,"%lg%n",&n,&i) > 0 && !buf[i] ? n : atom(buf); }
L parse() { return *buf == '(' ? list() : *buf == '\'' ? quote() : atomic(); }
void print(L);
void printlist(L t) {
 for (putchar('('); ; putchar(' ')) {
  print(car(t));
  if (not(t = cdr(t))) break;
  if (T(t) != CONS) { printf(" . "); print(t); break; }
 }
 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() {
 int 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/8); print(eval(Read(),env)); gc(); }
}
```

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

Lisp for the PC-G850 in 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<<3) abort();
 return box(ATOM,i);
}
L cons(L x,L y) { cell[--sp] = x; cell[--sp] = y; if (hp > sp<<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 v,L e) { while (T(e) == CONS && !equ(v,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 && !not(cdr(x)); }
L eval(L,L),parse();
L evlis(L t,L e) { return T(t) == CONS ? cons(eval(car(t),e),evlis(cdr(t),e)) : T(t) == ATOM ? assoc(t,e) : nil; }
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(evlis(t,e))) ? tru : nil; }
L f_or(L t,L e) { L x = nil; while (T(t) != NIL && not(x = eval(car(t),e))) t = cdr(t); return x; }
L f_and(L t,L e) { L x = nil; while (T(t) != NIL && !not(x = eval(car(t),e))) t = cdr(t); return x; }
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) { for (;let(t); t = cdr(t)) e = pair(car(car(t)),eval(car(cdr(car(t))),e),e); 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() { int c = getchar(); see = c; if (c == -1) exit(0); }
I seeing(char c) { return c == ' ' ? see > 0 && see <= c : see == c; }
char get() { char c = see; look(); return c; }
char scan() {
 int i = 0;
```

```
 while (seeing(' ')) look();
 if (seeing('(') || seeing(')') || seeing('\'')) buf[i++] = get();
 else do buf[i++] = get(); while (i < 39 && !seeing('(') && !seeing(')') && !seeing(' '));
 return buf[i] = 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; int 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) {
 for (putchar('('); ; putchar(' ')) {
  print(car(t));
  if (not(t = cdr(t))) break;
  if (T(t) != CONS) { printf(" . "); print(t); break; }
 }
 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() {
 int 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/8); print(eval(read(),env)); gc(); }
}
```

C Optimized Lisp Interpreter with NaN boxing

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

```
#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<<3) abort();
 return box(ATOM,i);
}
L cons(L x,L y) { cell[--sp] = x; cell[--sp] = y; if (hp > sp<<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 v,L e) { while (T(e) == CONS && !equ(v,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,*p;
 for (s = nil,p = &s; T(t) == CONS; p = cell+sp,t = cdr(t)) *p = cons(eval(car(t),e),nil);
 if (T(t) == ATOM) *p = assoc(t,e);
 return s;
}
L f_eval(L t,L *e) { return car(evlis(t,*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(evlis(t,*e))) ? tru : nil; }
L f_or(L t,L *e) { L x = nil; while (T(t) != NIL && not(x = eval(car(t),*e))) t = cdr(t); return x; }
L f_and(L t,L *e) { L x = nil; while (T(t) != NIL && !not(x = eval(car(t),*e))) t = cdr(t); return x; }
L f_cond(L t,L *e) { while (T(t) != NIL && not(eval(car(car(t)),*e))) t = cdr(t); return car(cdr(car(t))); }
L f_if(L t,L *e) { return car(cdr(not(eval(car(t),*e)) ? cdr(t) : t)); }
L f_leta(L t,L *e) { for (;let(t); t = cdr(t)) *e = pair(car(car(t)),eval(car(cdr(car(t))),*e),*e); return car(t); }
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*); short t; } prim[] = {
{"eval",  f_eval,  1},{"quote", f_quote, 0},{"cons",f_cons,0},{"car", f_car, 0},{"cdr",f_cdr,0},{"+",   f_add, 0},
{"-",     f_sub,   0},{"*",     f_mul,   0},{"/",   f_div, 0},{"int", f_int, 0},{"<",  f_lt, 0},{"eq?", f_eq,  0},
{"or",    f_or,    0},{"and",   f_and,   0},{"not", f_not, 0},{"cond",f_cond,1},{"if", f_if, 1},{"let*",f_leta,1},
{"lambda",f_lambda,0},{"define",f_define,0},{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) {
   x = prim[ord(f)].f(x,&e);
   if (prim[ord(f)].t) continue;
   return x;
  }
  if (T(f) != CLOS) return err;
  v = car(car(f)); d = cdr(f);
  if (T(d) == NIL) d = env;
  for (;T(v) == CONS && T(x) == CONS; v = cdr(v),x = cdr(x)) d = pair(car(v),eval(car(x),e),d);
  if (T(v) == CONS) x = eval(x,e);
  for (;T(v) == CONS; v = cdr(v),x = cdr(x)) d = pair(car(v),car(x),d);
  if (T(x) == CONS) x = evlis(x,e);
  else if (T(x) != NIL) x = eval(x,e);
  if (T(v) != NIL) d = pair(v,x,d);
  x = cdr(car(f)); e = d;
 }
}
char buf[40],see = ' ';
void look() { int c = getchar(); see = c; if (c == EOF) exit(0); }
I seeing(char c) { return c == ' ' ? see > 0 && see <= c : see == c; }
char get() { char c = see; look(); return c; }
char scan() {
 int i = 0;
 while (seeing(' ')) look();
 if (seeing('(') || seeing(')') || seeing('\'')) buf[i++] = get();
 else do buf[i++] = get(); while (i < 39 && !seeing('(') && !seeing(')') && !seeing(' '));
 return buf[i] = 0,*buf;
}
L Read() { return scan(),parse(); }
L list() {
 L t,*p;
 for (t = nil,p = &t; ; *p = cons(parse(),nil),p = cell+sp) {
  if (scan() == ')') return t;
  if (*buf == '.' && !buf[1]) return *p = Read(),scan(),t;
 }
}
L parse() {
 L n; int i;
 if (*buf == '(') return list();
 if (*buf == '\'') return cons(atom("quote"),cons(Read(),nil));
 return sscanf(buf,"%lg%n",&n,&i) > 0 && !buf[i] ? n : atom(buf);
}
void print(L);
void printlist(L t) {
 for (putchar('('); ; putchar(' ')) {
  print(car(t));
  if (not(t = cdr(t))) break;
  if (T(t) != CONS) { printf(" . "); print(t); break; }
 }
 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() {
 int 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/8); print(eval(Read(),env)); gc(); }
}
```

D Optimized Lisp Interpreter with BCD boxing

The following version of the Lisp interpreter for the PC-G850 is tail-call 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<<3) abort();
 return box(ATOM,i);
}
L cons(L x,L y) { cell[--sp] = x; cell[--sp] = y; if (hp > sp<<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 closure(L v,L x,L e) { return box(CLOS,ord(pair(v,x,e == env ? nil : e))); }
L assoc(L v,L e) { while (T e == CONS && v != 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,*p;
 for (s = nil,p = &s; T t == CONS; p = cell+sp,t = cdr(t)) *p = cons(eval(car(t),e),nil);
 if (T t == ATOM) *p = assoc(t,e);
 return s;
}
L f_eval(L t,L *e) { return car(evlis(t,*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 ? (unsigned 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),car(t) == car(cdr(t)) ? tru : nil; }
L f_not(L t,L *e) { return not(car(evlis(t,*e))) ? tru : nil; }
L f_or(L t,L *e) { L x = nil; while (T t != NIL && not(x = eval(car(t),*e))) t = cdr(t); return x; }
L f_and(L t,L *e) { L x = nil; while (T t != NIL && !not(x = eval(car(t),*e))) t = cdr(t); return x; }
L f_cond(L t,L *e) { while (T t != NIL && not(eval(car(car(t)),*e))) t = cdr(t); return car(cdr(car(t))); }
L f_if(L t,L *e) { return car(cdr(not(eval(car(t),*e)) ? cdr(t) : t)); }
L f_leta(L t,L *e) { for (;let(t); t = cdr(t)) *e = pair(car(car(t)),eval(car(cdr(car(t))),*e),*e); return car(t); }
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*); short t; } prim[] = {
{"eval",  f_eval,  1},{"quote", f_quote, 0},{"cons",f_cons,0},{"car", f_car, 0},{"cdr",f_cdr,0},{"+",   f_add, 0},
{"-",     f_sub,   0},{"*",     f_mul,   0},{"/",   f_div, 0},{"int", f_int, 0},{"<",  f_lt, 0},{"eq?", f_eq,  0},
{"or",    f_or,    0},{"and",   f_and,   0},{"not", f_not, 0},{"cond",f_cond,1},{"if", f_if, 1},{"let*",f_leta,1},
{"lambda",f_lambda,0},{"define",f_define,0},{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) {
   x = prim[ord(f)].f(x,&e);
   if (prim[ord(f)].t) continue;
   return x;
  }
  if (T f != CLOS) return err;
  v = car(car(f)); d = cdr(f);
  if (T d == NIL) d = env;
  for (;T v == CONS && T x == CONS; v = cdr(v),x = cdr(x)) d = pair(car(v),eval(car(x),e),d);
  if (T v == CONS) x = eval(x,e);
  for (;T v == CONS; v = cdr(v),x = cdr(x)) d = pair(car(v),car(x),d);
  if (T x == CONS) x = evlis(x,e);
  else if (T x != NIL) x = eval(x,e);
  if (T v != NIL) d = pair(v,x,d);
  x = cdr(car(f)); e = d;
 }
}
char buf[40],see = ' ';
void look() { int c = getchar(); see = c; if (c == -1) exit(0); }
I seeing(char c) { return c == ' ' ? see > 0 && see <= c : see == c; }
char get() { char c = see; look(); return c; }
char scan() {
 int i = 0;
 while (seeing(' ')) look();
 if (seeing('(') || seeing(')') || seeing('\'')) buf[i++] = get();
 else do buf[i++] = get(); while (i < 39 && !seeing('(') && !seeing(')') && !seeing(' '));
 return buf[i] = 0,*buf;
}
L read() { return scan(),parse(); }
L list() {
 L t,*p;
 for (t = nil,p = &t; ; *p = cons(parse(),nil),p = cell+sp) {
  if (scan() == ')') return t;
  if (*buf == '.' && !buf[1]) return *p = read(),scan(),t;
 }
}
L parse() {
 L n; int i;
 if (*buf == '(') return list();
 if (*buf == '\'') return cons(atom("quote"),cons(read(),nil));
 i = strlen(buf);
 return isdigit(buf[*buf == '-']) && sscanf(buf,"%lg%n",&n,&i) > 0 && !buf[i] ? n : atom(buf);
}
void print(L);
void printlist(L t) {
 for (putchar('('); ; putchar(' ')) {
  print(car(t));
  if (not(t = cdr(t))) break;
  if (T(t) != CONS) { printf(" . "); print(t); break; }
 }
 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() {
 int 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/8); 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
            x
            (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 (lambda (x) (car (cdr x))))
(define caddr (lambda (x) (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)
- `symbol?` tests if the value is an atom excluding the empty list.
- `atom?` tests if the value is an atom including the empty list, but beware that some Lisp implementation also return `#t` for numbers.

- `list` returns a list of the values of the arguments. For example, `(list 1 2 (+ 1 2))` gives `(1 2 3)`.
- `begin` (called `progn` in some other Lisp) 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. This variation of `begin` does not work without arguments passed to `begin`. Add `(define progn (lambda args (if args (begin args) ())))` to define `progn` with optional arguments.

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 append1
    (lambda (s t)
        (if s
            (cons (car s) (append1 (cdr s) t))
            t)))
(define append
```

```
    (lambda (t . args)
        (if args
            (append1 t (append . args))
            t)))
(define rev1
    (lambda (r t)
        (if t
            (rev1 (cons (car t) r) (cdr t))
            r)))
(define reverse (lambda (t) (rev1 () t)))
(define nthcdr
    (lambda (t n)
        (if (eq? n 0)
            t
            (nthcdr (cdr t) (- n 1)))))
(define nth (lambda (t n) (car (nthcdr t n))))
(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
            (and
                (f (car t))
                (all? f (cdr t)))
            #t)))
(define any?
    (lambda (f t)
        (if t
            (or
                (f (car t))
                (any? f (cdr t)))
            ())))
(define mapcar
    (lambda (f t)
        (if t
            (cons (f (car t)) (mapcar f (cdr t)))
            ())))
(define map
    (lambda (f . args)
        (if (any? null? args)
            ()
            (let*
                (x (mapcar car args))
                (t (mapcar cdr args))
                (cons (f . x) (map f . t))))))
(define zip (lambda args (map list . args)))
(define seq
    (lambda (n m)
        (if (< n m)
            (cons n (seq (+ n 1) m))
            ())))
(define seqby
    (lambda (n m k)
        (if (< 0 (* k (- m n)))
            (cons n (seqby (+ n k) m k))
            ())))
(define range
    (lambda (n m . args)
        (if args
            (seqby n m (car args))
            (seq n m))))
```

Explanation:

- `length` returns the length of a list.
- `append` returns the concatenation of multiple lists.
- `reverse` reverses a list (tail recursive)
- `nth` returns the n'th element of a list
- `nthcdr` skips n elements of a list to return the n'th cdr

- `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.
- `mapcar` applies `f` to the elements of a list `t`.
- `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.
- `seq` generates a list of successive values `n` up to but not including `m`
- `seqby` generates a list of values from `n` to but not including `m` and stepping by `k`
- `range` generates a list of values from `n` to but not including `m` with an optional step value.

E.4 Higher-Order Functions

The following functions 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 in turn returns a self-applying (recursive) function. 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
  ```

E.5 Revealing Closures

The following function reveals the arguments and body of the closure of a lambda:

```
(define reveal (lambda (f) (cons 'lambda (cons (car (car f)) (cons (cdr (car f)) ())))))
```

Explanation:

- `reveal` takes a closure and "unparses" it as a lambda. For example `(reveal reveal)` shows `(lambda (f) (cons (quote lambda) (cons (car (car f)) (cons (cdr (car f)) ()))))`.