Lispy: a simple Lisp-like language

Eric Bailey

May 10, 2018 [1]

[1] Current version: VERSION. Last updated May 16, 2018.

For my own edification, and my eternal love of the LISP family and PLT, what follows is an implementation in C of a simple, Lisp-like programming language, based on Build Your Own Lisp [Holden, 2018a]. Since I'm a bit of masochist, this is a literate program[2], written using Noweb[3].

[2] https://en.wikipedia.org/wiki/Literate_programming

[3] Norman Ramsey. Noweb – a simple, extensible tool for literate programming. https://www.cs.tufts.edu/~nr/noweb/, 2012. Accessed: 2018-05-13

Contents

Outline

Describe the outline

2a ⟨*lispy.c* 2a⟩≡
⟨*Include the necessary headers.* 21a⟩

⟨*Load the Lispy grammar.* 6c⟩

⟨*Define possible lval and error types.* 18a⟩

⟨*Define the Lispy data structures.* 17c⟩

This definition is continued in chunks 2–5.
Root chunk (not used in this document).

2b ⟨*lispy.c* 2a⟩+≡
```
lval *lval_add(lval *xs, lval *x)
{
    ⟨Add an element to an S-expression. 10c⟩

    return xs;
}
```

Defines:
`lval_add`, used in chunk 10a.
Uses `lval` 17c.

2c ⟨*lispy.c* 2a⟩+≡
```
lval *lval_pop(lval *xs, int i)
{
    ⟨Extract an element and shift the list. 14b⟩
}
```

Defines:
`lval_pop`, used in chunks 11a and 14g.
Uses `lval` 17c.

2d ⟨*lispy.c* 2a⟩+≡
```
lval *lval_take(lval *xs, int i)
{
    ⟨Pop the list then delete it. 14g⟩
}
```

Defines:
`lval_take`, used in chunk 13.
Uses `lval` 17c.

3a ⟨*lispy.c* 2a⟩+≡
```
void lval_print_err(lval *val)
{
    ⟨Print an error. 15f⟩
}
```

Defines:
  `lval_print_err`, used in chunk 16a.
Uses `lval` 17c.

Forward declare[4] `lval_print`, since it's mutually recursive[5] with `lval_sexpr_print`.

[4] https://en.wikipedia.org/wiki/Forward_declaration
[5] https://en.wikipedia.org/wiki/Mutual_recursion

3b ⟨*lispy.c* 2a⟩+≡
```
void lval_print(lval *val);
```

Uses `lval` 17c and `lval_print` 3d.

3c ⟨*lispy.c* 2a⟩+≡
```
void lval_sexpr_print(lval *sexpr, char open, char close)
{
    ⟨Print an S-expression. 15b⟩
}
```

Defines:
  `lval_sexpr_print`, used in chunks 3c and 16a.
Uses `lval` 17c.

3d ⟨*lispy.c* 2a⟩+≡
```
void lval_print(lval *val)
{
    ⟨Print a Lispy value. 16a⟩
}
```

Defines:
  `lval_print`, used in chunks 3 and 15.
Uses `lval` 17c.

3e ⟨*lispy.c* 2a⟩+≡
```
void lval_println(lval *val)
{
    lval_print(val);
    putchar('\n');
}
```

Defines:
  `lval_println`, used in chunk 15a.
Uses `lval` 17c and `lval_print` 3d.

4a ⟨*lispy.c* 2a⟩+≡

```
lval *builtin_op(char *op, lval *args)
{
  ⟨Eval(uate) a built-in operation. 10h⟩
}
```

Defines:
builtin_binop, never used.
Uses lval 17c.

Forward declare `lval_eval`, since it's mutually recursive with `lval_eval_sexpr`.

4b ⟨*lispy.c* 2a⟩+≡

```
lval *lval_eval(lval* val);
```

Uses lval 17c.

4c ⟨*lispy.c* 2a⟩+≡

```
lval* lval_eval_sexpr(lval *args)
{
    ⟨Evaluate an S-expression. 13b⟩
}
```

Uses lval 17c.

4d ⟨*lispy.c* 2a⟩+≡

```
lval* lval_eval(lval* val)
{
    ⟨Evaluate an expression. 14a⟩
}
```

Uses lval 17c.

4e ⟨*lispy.c* 2a⟩+≡

```
lval *lval_read_num(mpc_ast_t *ast){

    ⟨Read a number. 9b⟩
}


lval *lval_read(mpc_ast_t *ast)
{
    ⟨Read a Lispy value. 9a⟩
}
```

Defines:
lval_read, used in chunks 8e and 10a.
Uses ast 8e, lval 17c, and mpc_ast_t 22.

5 ⟨*lispy.c* 2a⟩+≡

```
int main(int argc, char *argv[])
{
    ⟨Define the language. 7a⟩

    ⟨Print version and exit information. 6a⟩

    ⟨Loop until the input is empty. 16c⟩

    ⟨Undefine and delete the parsers. 7d⟩

    return 0;
}
```

Welcome

What good is a *Read-Eval-Print Loop (REPL)* without a welcome message? For now, simply print the version and describe how to exit.

6a ⟨*Print version and exit information.* 6a⟩≡

```
puts("Lispy v0.9.0");
puts("Press ctrl-c to exit\n");
```

Uses Lispy 7a.
This code is used in chunk 5.

Defining the Language

In order to make sense of user input, we need to define a *grammar*.

Support Core Erlang style numbers

6b ⟨*lispy.mpc* 6b⟩≡

```
integer  : /-?[0-9]+/ ;
float    : /-?[0-9]+\.[0-9]+/;
number   : <float> | <integer> ;
symbol   : '+' | '-' | '*' | '/' | '%' | '^' ;
sexpr    : '(' <symbol> <expr>+ ')' ;
expr     : <number> | <sexpr> ;
lispy    : /^/ <expr>* /$/ ;
```

Root chunk (not used in this document).

Describe this trick

6c ⟨*Load the Lispy grammar.* 6c⟩≡

```
static const char LISPY_GRAMMAR[] = {
#include "lispy.xxd"
};
```

Defines:
LISPY_GRAMMAR, used in chunk 7c.
This code is used in chunk 2a.

See: https://stackoverflow.com/a/411000

To implement the *grammar*, we need to create some *parsers*.

7a ⟨*Define the language.* 7a⟩≡
```
mpc_parser_t *Integer  = mpc_new("integer");
mpc_parser_t *Float    = mpc_new("float");
mpc_parser_t *Number   = mpc_new("number");
mpc_parser_t *Symbol   = mpc_new("symbol");
mpc_parser_t *Sexpr    = mpc_new("sexpr");
mpc_parser_t *Expr     = mpc_new("expr");
mpc_parser_t *Lispy    = mpc_new("lispy");
```

Defines:
  Expr, used in chunk 7b.
  Float, used in chunk 7b.
  Integer, used in chunk 7b.
  Lispy, used in chunks 6–8.
  Number, used in chunk 7b.
  Sexpr, used in chunk 7b.
  Symbol, used in chunk 7b.
Uses mpc_new 22 and mpc_parser_t 22.
This definition is continued in chunk 7c.
This code is used in chunk 5.

Finally, using the defined *grammar* and each of the ⟨*created parsers* 7b⟩,

7b ⟨*created parsers* 7b⟩≡
```
Integer, Float, Number, Symbol, Sexpr, Expr, Lispy
```
Uses Expr 7a, Float 7a, Integer 7a, Lispy 7a, Number 7a, Sexpr 7a, and Symbol 7a.
This code is used in chunk 7.

... we can define the Lispy language.

7c ⟨*Define the language.* 7a⟩+≡
```
mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
          ⟨created parsers 7b⟩);
```
Uses LISPY_GRAMMAR 6c and mpca_lang 22.

Since we're implementing this in C, we need to clean up after ourselves. The mpc[6] library makes this easy, by providing the mpc_cleanup function.

[6] Daniel Holden. Micro Parser Combinators. https://github.com/orangeduck/mpc, 2018b. Accessed: 2018-05-13

7d ⟨*Undefine and delete the parsers.* 7d⟩≡
```
mpc_cleanup(9, ⟨created parsers 7b⟩);
```
Uses mpc_cleanup 22.
This code is used in chunk 5.

R is for Read

To implement the R in REPL, use readline from libedit[7].

[7] Jess Thrysoee. Editline Library (libedit) – port of netbsd command line editor library. http://thrysoee.dk/editline/, 2017. Accessed: 2018-05-13

7e ⟨*Read a line of user input.* 7e⟩≡
```
char *input = readline("> ");
```
Defines:
  input, used in chunks 8a, 7, 8, and 17b.
Uses readline 21g.
This code is used in chunk 16d.

To check whether user input is nonempty, and thus whether we should continue looping, use the following expression.

8a ⟨`input` *is nonempty* 8a⟩≡
```
input && *input
```
Uses `input` 7e.
This code is used in chunk 17a.

Here, `input` is functionally equivalent to `input` $\neq$ `NULL`, and `*input` is functionally equivalent to `input[0]` $\neq$ `'\0'`, i.e. `input` is non-null and nonempty, respectively.

So long as `input` is nonempty, add it to the `libedit`[8] history table.

[8] Jess Thrysoee. Editline Library (libedit) – port of netbsd command line editor library. http://thrysoee.dk/editline/, 2017. Accessed: 2018-05-13

8b ⟨*Add* `input` *to the history table.* 8b⟩≡
```
add_history(input);
```
Uses `add_history` 21g and `input` 7e.
This code is used in chunk 17a.

Declare a variable, `parsed`, to hold the results of attempting to parse user input as Lispy code.

8c ⟨*Declare a variable to hold parsing results.* 8c⟩≡
```
mpc_result_t parsed;
```
Defines:
`parsed`, used in chunks 8 and 16b.
Uses `mpc_result_t` 22.
This code is used in chunk 17a.

To attempt said parsing, use `mpc_parse`, the result of which we can branch on to handle success and failure.

8d ⟨*the input can be parsed as Lispy code* 8d⟩≡
```
mpc_parse("<stdin>", input, Lispy, &parsed)
```
Uses `Lispy` 7a, `input` 7e, `mpc_parse` 22, and `parsed` 8c.
This code is used in chunk 17a.

E is for Eval(uate)

Since our terms consist of only numbers and operations thereon, the `result` of evaluating a Lispy expression can be represented as a *`double`*-precision number.

8e ⟨*Eval(uate) the input.* 8e⟩≡
```
mpc_ast_t *ast = parsed.output;

lval *result = lval_eval(lval_read(ast));
```
Defines:
`ast`, used in chunks 4e, 9, 10a, and 15a.
Uses `lval` 17c, `lval_read` 4e, `mpc_ast_t` 22, and `parsed` 8c.
This code is used in chunk 17a.

Describe the evaluation strategy

If the *abstract syntax tree (AST)* is tagged as a number, convert it to a `double`.

9a ⟨*Read a Lispy value.* 9a⟩≡
```
if (strstr(ast->tag, "number"))
    return lval_read_num(ast);
```

Uses `ast` 8e and `strstr` 21f.
This definition is continued in chunks 9 and 10.
This code is used in chunk 4e.

Describe this

9b ⟨*Read a number.* 9b⟩≡
```
errno = 0;
double num = strtod(ast->contents, NULL);
return errno != ERANGE ? lval_num(num) : lval_err(LERR_BAD_NUM);
```
Uses `LERR_BAD_NUM` 19a, `ast` 8e, `lval_err` 19b, `lval_num` 18b, and `strtod` 21d.
This code is used in chunk 4e.

If the AST is tagged as a symbol, convert it to one.

9c ⟨*Read a Lispy value.* 9a⟩+≡
```
if (strstr(ast->tag, "symbol"))
    return lval_sym(ast->contents);
```

Uses `ast` 8e, `lval_sym` 19c, and `strstr` 21f.

Describe this

9d ⟨*Read a symbol.* 9d⟩≡

Root chunk (not used in this document).

Describe this

9e ⟨*Read a Lispy value.* 9a⟩+≡
```
lval *sexpr = NULL;
```
Uses `lval` 17c.

If we're at the root of the AST, create an empty list.

9f ⟨*Read a Lispy value.* 9a⟩+≡
```
if (!strcmp(ast->tag, ">"))
    sexpr = lval_sexpr();
```
Uses `ast` 8e, `lval_sexpr` 20a, and `strcmp` 21f.

Similarly if it's tagged as an S-expression, create an empty list.

9g ⟨*Read a Lispy value.* 9a⟩+≡
```
if (strstr(ast->tag, "sexpr"))
    sexpr = lval_sexpr();
```

Uses `ast` 8e, `lval_sexpr` 20a, and `strstr` 21f.

Describe this

10a ⟨*Read a Lispy value.* 9a⟩+≡
```
for (int i = 0; i < ast->children_num; i++) {
    if(!strcmp(ast->children[i]->contents, "(")) continue;
    if(!strcmp(ast->children[i]->contents, ")")) continue;
    if(!strcmp(ast->children[i]->tag, "regex")) continue;
    sexpr = lval_add(sexpr, lval_read(ast->children[i]));
}
```

Uses ast 8e, lval_add 2b, lval_read 4e, and strcmp 21f.

10b ⟨*Reallocate the memory used.* 10b⟩≡
```
xs->cell = realloc(xs->cell, sizeof(lval *) * xs->count);
```
Uses lval 17c.
This code is used in chunks 10c and 14f.

Describe this, incl. how it's not cons

10c ⟨*Add an element to an S-expression.* 10c⟩≡
```
xs->count++;
⟨Reallocate the memory used. 10b⟩
xs->cell[xs->count - 1] = x;
```
This code is used in chunk 2b.

Finally, return the S-expression.

10d ⟨*Read a Lispy value.* 9a⟩+≡
```
return sexpr;
```

10e ⟨*For each argument* 10e⟩≡
```
for (int i = 0; i < args->count; i++)
```
This code is used in chunks 10h and 13c.

10f ⟨*the argument is not a number* 10f⟩≡
```
!lval_is_num(args->cell[i])
```
Uses lval_is_num 18c.
This code is used in chunk 10h.

10g ⟨*Delete the arguments and return a bad number error.* 10g⟩≡
```
lval_del(args);
return lval_err(LERR_BAD_NUM);
```
Uses LERR_BAD_NUM 19a, lval_del 20b, and lval_err 19b.
This code is used in chunk 10h.

Evaluating built-in operations

Ensure all arguments are numbers.

10h ⟨*Eval(uate) a built-in operation.* 10h⟩≡
```
⟨For each argument 10e⟩ {
    if (⟨the argument is not a number 10f⟩) {
        ⟨Delete the arguments and return a bad number error. 10g⟩
    }
}
```

This definition is continued in chunks 11 and 13a.
This code is used in chunk 4a.

11a ⟨*Pop the first element.* 11a⟩≡
```
lval_pop(args, 0);
```
Uses lval_pop 2c.
This code is used in chunks 11 and 13e.

Pop the first element.

11b ⟨*Eval(uate) a built-in operation.* 10h⟩+≡
```
lval *result = ⟨Pop the first element. 11a⟩
```
Uses lval 17c.

If the operation is unary subtraction, negate the operand.

11c ⟨*Eval(uate) a built-in operation.* 10h⟩+≡
```
if (!strcmp(op, "-") && !args->count)
    result->num = -result->num;
```
Uses strcmp 21f.

11d ⟨*Pop the next element.* 11d⟩≡
```
lval *y = ⟨Pop the first element. 11a⟩
```
Uses lval 17c.
This code is used in chunk 11e.

11e ⟨*Eval(uate) a built-in operation.* 10h⟩+≡
```
while (args->count > 0) {
    ⟨Pop the next element. 11d⟩

    ⟨Perform a built-in operation. 11f⟩
}
```

If the op is "+", perform addition.

11f ⟨*Perform a built-in operation.* 11f⟩≡
```
if (!strcmp(op, "+")) {
    result->num += y->num;
}
```
Uses strcmp 21f.
This definition is continued in chunks 11 and 12.
This code is used in chunk 11e.

If the op is "-", perform subtraction.

11g ⟨*Perform a built-in operation.* 11f⟩+≡
```
else if (!strcmp(op, "-")) {
    result->num -= y->num;
}
```
Uses strcmp 21f.

If the op is "*", perform multiplication.

11h ⟨*Perform a built-in operation.* 11f⟩+≡
```
else if (!strcmp(op, "*")) {
    result->num *= y->num;
}
```
Uses strcmp 21f.

If the op is "/", perform division, returning the appropriate error and cleaning up when trying to divide by zero.

12a ⟨*Perform a built-in operation.* 11f⟩+≡
```
else if (!strcmp(op, "/")) {
    if (!y->num) {
        lval_del(result);
        lval_del(y);
        result = lval_err(LERR_DIV_ZERO);
        break;
    }
    result->num /= y->num;
}
```
Uses LERR_DIV_ZERO 19a, lval_del 20b, lval_err 19b, and strcmp 21f.

If the op is "%", calculate the integer modulo, returning the appropriate error when trying to divide by zero.

12b ⟨*Perform a built-in operation.* 11f⟩+≡
```
else if (!strcmp(op, "%")) {
    if (!y->num) {
        lval_del(result);
        lval_del(y);
        result = lval_err(LERR_DIV_ZERO);
        break;
    }
    result->num = fmod(result->num, y->num);
}
```
Uses LERR_DIV_ZERO 19a, fmod 21e, lval_del 20b, lval_err 19b, and strcmp 21f.

If the opp is "^", perform exponentiation.

12c ⟨*Perform a built-in operation.* 11f⟩+≡
```
else if (!strcmp(op, "^")) {
    result->num = pow(result->num, y->num);
}
```
Uses pow 21e and strcmp 21f.

Otherwise, return a LERR_BAD_OP error.

12d ⟨*Perform a built-in operation.* 11f⟩+≡
```
else {
    lval_del(result);
    lval_del(y);
    result = lval_err(LERR_BAD_OP);
    break;
}
```
Uses LERR_BAD_OP 19a, lval_del 20b, and lval_err 19b.

Delete y, now that we're done with it.

12e ⟨*Perform a built-in operation.* 11f⟩+≡
```
lval_del(y);
```
Uses lval_del 20b.

Delete the input expression and return the result.

13a ⟨*Eval(uate) a built-in operation.* 10h⟩+≡
```
lval_del(args);

return result;
```
Uses lval_del 20b.

Evaluating (S)-expressions

If the expression is empty, return it;

13b ⟨*Evaluate an S-expression.* 13b⟩≡
```
if (!args->count)
    return args;
```
This definition is continued in chunk 13.
This code is used in chunk 4c.

13c ⟨*Evaluate an S-expression.* 13b⟩+≡
```
⟨For each argument 10e⟩ {
    args->cell[i] = lval_eval(args->cell[i]);
    if (args->cell[i]->type == LVAL_ERR)
        return lval_take(args, i);
}
```
Uses LVAL_ERR 18a 18a and lval_take 2d.

If we're dealing with a single expression, return it.

13d ⟨*Evaluate an S-expression.* 13b⟩+≡
```
if (args->count == 1)
    return lval_take(args, 0);
```
Uses lval_take 2d.

13e ⟨*Evaluate an S-expression.* 13b⟩+≡
```
lval *car = ⟨Pop the first element. 11a⟩;
if (car->type ≠ LVAL_SYM) {
    lval_del(car);
    lval_del(args);

    return lval_err(LERR_BAD_SEXPR);
}
```
Uses LVAL_SYM 18a, lval 17c, lval_del 20b, and lval_err 19b.

13f ⟨*Evaluate an S-expression.* 13b⟩+≡
```
lval *result = builtin_op(car->sym, args);
lval_del(car);

return result;
```
Uses lval 17c and lval_del 20b.

If, and only if, an expression is an S-expression, we must evaluate it recursively.

14a ⟨*Evaluate an expression.* 14a⟩≡
```
if (val->type == LVAL_SEXPR)
    return lval_eval_sexpr(val);

return val;
```
Uses LVAL_SEXPR 18a.
This code is used in chunk 4d.

Extract the element at index `i`.

14b ⟨*Extract an element and shift the list.* 14b⟩≡
```
lval *elem = xs->cell[i];
```
Uses lval 17c.
This definition is continued in chunk 14.
This code is used in chunk 2c.

Shift memory after the element at index `i`.

14c ⟨*Extract an element and shift the list.* 14b⟩+≡
```
memmove(&xs->cell[i], &xs->cell[i + 1],
    sizeof(lval *) * (xs->count - i - 1));
```
Uses lval 17c.

Decrease the count.

14d ⟨*Extract an element and shift the list.* 14b⟩+≡
```
xs->count--;
```

14e ⟨*Return the extracted element.* 14e⟩≡
```
return elem;
```
This code is used in chunk 14.

Reallocate the memory used and return the extracted element.

14f ⟨*Extract an element and shift the list.* 14b⟩+≡
```
⟨Reallocate the memory used. 10b⟩

⟨Return the extracted element. 14e⟩
```

Describe this

14g ⟨*Pop the list then delete it.* 14g⟩≡
```
lval *elem = lval_pop(xs, i);
lval_del(xs);
```
Uses lval 17c, lval_del 20b, and lval_pop 2c.
This definition is continued in chunk 14h.
This code is used in chunk 2d.

Return the extracted element.

14h ⟨*Pop the list then delete it.* 14g⟩+≡
```
⟨Return the extracted element. 14e⟩
```

P is for Print

Upon success, print the result and delete the AST.

15a ⟨*Print the result and delete the AST.* 15a⟩≡
```
lval_println(result);

mpc_ast_delete(ast);
```
Uses ast 8e, lval_println 3e, and mpc_ast_delete 22.
This code is used in chunk 17a.

Describe this

Print the opening character.

15b ⟨*Print an S-expression.* 15b⟩≡
```
putchar(open);
```
This definition is continued in chunk 15.
This code is used in chunk 3c.

Print all but the last element with a trailing space.

15c ⟨*Print an S-expression.* 15b⟩+≡
```
int i = 0;
while (i++ < sexpr->count - 1) {
    lval_print(sexpr->cell[i]);
    putchar(' ');
}
```
Uses lval_print 3d.

Print the last element.

15d ⟨*Print an S-expression.* 15b⟩+≡
```
lval_print(sexpr->cell[i]);
```
Uses lval_print 3d.

Print the closing character.

15e ⟨*Print an S-expression.* 15b⟩+≡
```
putchar(close);
```

15f ⟨*Print an error.* 15f⟩≡
```
switch (val->err) {
case LERR_BAD_NUM:
    puts("Error: invalid number");
    break;
case LERR_BAD_OP:
    puts("Error: invalid operator");
    break;
case LERR_BAD_SEXPR:
    puts("Error: S-expression does not start with symbol");
    break;
case LERR_DIV_ZERO:
    puts("Error: division by zero");
    break;
}
```
Uses LERR_BAD_NUM 19a, LERR_BAD_OP 19a, and LERR_DIV_ZERO 19a.
This code is used in chunk 3a.

16a ⟨*Print a Lispy value.* 16a⟩≡

```
switch (val->type) {
case LVAL_ERR:
    lval_print_err(val);
    break;
case LVAL_NUM:
    printf("%g", val->num);
    break;
case LVAL_SEXPR:
    lval_sexpr_print(val, '(', ')');
    break;
case LVAL_SYM:
    fputs(val->sym, stdout);
    break;
}
```

Uses LVAL_ERR 18a 18a, LVAL_NUM 18a, LVAL_SEXPR 18a, LVAL_SYM 18a, lval_print_err 3a, lval_sexpr_print 3c, and printf 21c.
This code is used in chunk 3d.

Print and delete the error upon failure.

16b ⟨*Print and delete the error.* 16b⟩≡

```
mpc_err_print(parsed.error);
mpc_err_delete(parsed.error);
```

Uses mpc_err_delete 22, mpc_err_print 22, and parsed 8c.
This code is used in chunk 17a.

L is for Loop

16c ⟨*Loop until the input is empty.* 16c⟩≡

```
bool nonempty;
do {
  ⟨Read, eval(uate), and print. 16d⟩
} while (nonempty);
```

Defines:
nonempty, used in chunk 17a.
Uses bool 21b.
This code is used in chunk 5.

As previously described, in the body of the loop, **R**ead a line of user input.

16d ⟨*Read, eval(uate), and print.* 16d⟩≡

```
⟨Read a line of user input. 7e⟩
```

This definition is continued in chunk 17.
This code is used in chunk 16c.

If, and only if, it's not empty, add it to the history table, **E**val(uate) it, and **P**rint the result.

17a ⟨*Read, eval(uate), and print.* 16d⟩+≡
```
if ((nonempty = (⟨input is nonempty 8a⟩))) {
    ⟨Add input to the history table. 8b⟩

    ⟨Declare a variable to hold parsing results. 8c⟩
    if (⟨the input can be parsed as Lispy code 8d⟩) {
        ⟨Eval(uate) the input. 8e⟩
        ⟨Print the result and delete the AST. 15a⟩
    } else {
        ⟨Print and delete the error. 16b⟩
    }
}
```

Uses nonempty 16c.

Once we're done, deallocate the space pointed to by `input`, making it available for futher allocation.

17b ⟨*Read, eval(uate), and print.* 16d⟩+≡
```
free(input);
```
Uses free 21d and input 7e.

N.B. This is a no-op when `!input`.

Error Handling

Describe this struct

17c ⟨*Define the Lispy data structures.* 17c⟩≡
```
typedef struct lval {
    lval_type_t type;
    union {
        double num;
        lval_err_t err;
        char* sym;
    };
    int count;
    struct lval **cell;
} lval;
```

Defines:
  lval, used in chunks 2–4, 8–11, 13, 14, and 18–20.
Uses lval_err_t 19a and lval_type_t 18a.
This definition is continued in chunks 18–20.
This code is used in chunk 2a.

A Lispy value can be either a number or an error.

18a ⟨*Define possible lval and error types.* 18a⟩≡
```
typedef enum {
    LVAL_ERR,
    LVAL_NUM,
    LVAL_SEXPR,
    LVAL_SYM
} lval_type_t;
```

Defines:
  LVAL_ERR, used in chunks 13c, 16a, 19b, and 20b.
  LVAL_NUM, used in chunks 16a, 18, and 20b.
  LVAL_SEXPR, used in chunks 14a, 16a, and 20.
  LVAL_SYM, used in chunks 13e, 16a, 19c, and 20b.
  lval_type_t, used in chunk 17c.
This definition is continued in chunk 19a.
This code is used in chunk 2a.

Define a constructor for numbers.

18b ⟨*Define the Lispy data structures.* 17c⟩+≡
```
lval *lval_num(double num)
{
    lval *val = malloc(sizeof(lval));
    val->type = LVAL_NUM;
    val->num = num;

    return val;
}
```

Defines:
  lval_num, used in chunk 9b.
Uses LVAL_NUM 18a and lval 17c.

Define a convenient predicate for numbers.

18c ⟨*Define the Lispy data structures.* 17c⟩+≡
```
bool lval_is_num(lval *val)
{
    return val->type == LVAL_NUM;
}
```

Defines:
  lval_is_num, used in chunk 10f.
Uses LVAL_NUM 18a, bool 21b, and lval 17c.

Possible reasons for error include division by zero, a bad operator, and a bad number.

19a ⟨*Define possible lval and error types.* 18a⟩+≡

```
typedef enum {
    LERR_DIV_ZERO,
    LERR_BAD_OP,
    LERR_BAD_NUM,
    LERR_BAD_SEXPR
} lval_err_t;
```

Defines:
LERR_BAD_NUM, used in chunks 9b, 10g, and 15f.
LERR_BAD_OP, used in chunks 12d and 15f.
LERR_DIV_ZERO, used in chunks 12 and 15f.
lval_err_t, used in chunks 17c and 19b.

Define a constructor for errors.

19b ⟨*Define the Lispy data structures.* 17c⟩+≡

```
lval *lval_err(lval_err_t err)
{
    lval *val = malloc(sizeof(lval));
    val->type = LVAL_ERR;
    val->err = err;

    return val;
}
```

Defines:
lval_err, used in chunks 9b, 10g, 12, and 13e.
Uses LVAL_ERR 18a 18a, lval 17c, and lval_err_t 19a.

Define a constructor for symbol.

19c ⟨*Define the Lispy data structures.* 17c⟩+≡

```
lval *lval_sym(char *s)
{
    lval *val = malloc(sizeof(lval));
    val->type = LVAL_SYM;
    val->sym = malloc(strlen(s) + 1);
    strcpy(val->sym, s);

    return val;
}
```

Defines:
lval_sym, used in chunk 9c.
Uses LVAL_SYM 18a and lval 17c.

Define a constructor for an S-expression.

20a ⟨*Define the Lispy data structures.* 17c⟩+≡

```
lval *lval_sexpr(void)
{
    lval *val = malloc(sizeof(lval));
    val->type = LVAL_SEXPR;
    val->count = 0;
    val->cell = NULL;

    return val;
}
```

Defines:
  lval_sexpr, used in chunk 9.
Uses LVAL_SEXPR 18a and lval 17c.

Define a destructor for `lval`*.

20b ⟨*Define the Lispy data structures.* 17c⟩+≡

```
void lval_del(lval *val)
{
    switch(val->type) {
    case LVAL_ERR:
    case LVAL_NUM:
        break;
    case LVAL_SEXPR:
        for (int i = 0; i < val->count; i++)
            lval_del(val->cell[i]);
        free(val->cell);
        break;
    case LVAL_SYM:
        free(val->sym);
        break;
    }

    free(val);
}
```

Defines:
  lval_del, used in chunks 10g and 12–14.
Uses LVAL_ERR 18a 18a, LVAL_NUM 18a, LVAL_SEXPR 18a, LVAL_SYM 18a, free 21d, and lval 17c.

Headers

Describe headers

21a ⟨*Include the necessary headers.* 21a⟩≡
⟨*Include the boolean type and values.* 21b⟩
⟨*Include the standard I/O functions.* 21c⟩
⟨*Include the standard library definitions.* 21d⟩
⟨*Include some mathematical definitions.* 21e⟩
⟨*Include some string operations.* 21f⟩

⟨*Include the line editing functions from libedit.* 21g⟩
⟨*Include the micro parser combinator definitions.* 22⟩
This code is used in chunk 2a.

21b ⟨*Include the boolean type and values.* 21b⟩≡
```
#include <stdbool.h>
```
Defines:
`bool`, used in chunks 16c and 18c.
This code is used in chunk 21a.

21c ⟨*Include the standard I/O functions.* 21c⟩≡
```
#include <stdio.h>
```
Defines:
`printf`, used in chunk 16a.
This code is used in chunk 21a.

21d ⟨*Include the standard library definitions.* 21d⟩≡
```
#include <stdlib.h>
```
Defines:
`free`, used in chunks 17b and 20b.
`strtod`, used in chunk 9b.
This code is used in chunk 21a.

21e ⟨*Include some mathematical definitions.* 21e⟩≡
```
#include <math.h>
```
Defines:
`fmod`, used in chunk 12b.
`pow`, used in chunk 12c.
This code is used in chunk 21a.

21f ⟨*Include some string operations.* 21f⟩≡
```
#include <string.h>
```
Defines:
`strcmp`, used in chunks 9–12.
`strstr`, used in chunk 9.
This code is used in chunk 21a.

21g ⟨*Include the line editing functions from libedit.* 21g⟩≡
```
#include <editline/readline.h>
```
Defines:
`add_history`, used in chunk 8b.
`readline`, used in chunks 21g and 7e.
This code is used in chunk 21a.

22 ⟨*Include the micro parser combinator definitions.* 22⟩≡

```
#include <mpc.h>
```

Defines:
`mpca_lang`, used in chunk 7c.
`mpc_ast_delete`, used in chunk 15a.
`mpc_ast_print`, never used.
`mpc_ast_t`, used in chunks 4e and 8e.
`mpc_cleanup`, used in chunks 22 and 7d.
`mpc_err_delete`, used in chunk 16b.
`mpc_err_print`, used in chunk 16b.
`mpc_new`, used in chunk 7a.
`mpc_parse`, used in chunks 22 and 8d.
`mpc_parser_t`, used in chunk 7a.
`mpc_result_t`, used in chunk 8c.
This code is used in chunk 21a.

Full Listings

lispy.mpc:

```
integer  : /-?[0-9]+/ ;
float    : /-?[0-9]+\.[0-9]+/;
number   : <float> | <integer> ;
symbol   : '+' | '-' | '*' | '/' | '%' | '^' ;
sexpr    : '(' <symbol> <expr>+ ')' ;
expr     : <number> | <sexpr> ;
lispy    : /^/ <expr>* /$/ ;
```

lispy.c:

```c
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#include <editline/readline.h>
#include <mpc.h>


static const char LISPY_GRAMMAR[] = {
#include "lispy.xxd"
};


typedef enum {
    LVAL_ERR,
    LVAL_NUM,
    LVAL_SEXPR,
    LVAL_SYM
} lval_type_t;


typedef enum {
    LERR_DIV_ZERO,
    LERR_BAD_OP,
    LERR_BAD_NUM,
    LERR_BAD_SEXPR
} lval_err_t;


typedef struct lval {
    lval_type_t type;
    union {
        double num;
        lval_err_t err;
        char *sym;
    };
    int count;
    struct lval **cell;
} lval;


lval *lval_num(double num)
{
    lval *val = malloc(sizeof(lval));
    val->type = LVAL_NUM;
    val->num = num;

    return val;
```

```
51  }
52
53
54  bool lval_is_num(lval * val)
55  {
56      return val->type == LVAL_NUM;
57  }
58
59
60  lval *lval_err(lval_err_t err)
61  {
62      lval *val = malloc(sizeof(lval));
63      val->type = LVAL_ERR;
64      val->err = err;
65
66      return val;
67  }
68
69
70  lval *lval_sym(char *s)
71  {
72      lval *val = malloc(sizeof(lval));
73      val->type = LVAL_SYM;
74      val->sym = malloc(strlen(s) + 1);
75      strcpy(val->sym, s);
76
77      return val;
78  }
79
80
81  lval *lval_sexpr(void)
82  {
83      lval *val = malloc(sizeof(lval));
84      val->type = LVAL_SEXPR;
85      val->count = 0;
86      val->cell = NULL;
87
88      return val;
89  }
90
91
92  void lval_del(lval * val)
93  {
94      switch (val->type) {
95      case LVAL_ERR:
96      case LVAL_NUM:
97          break;
98      case LVAL_SEXPR:
99          for (int i = 0; i < val->count; i++)
100             lval_del(val->cell[i]);
101         free(val->cell);
```

```
102         break;
103     case LVAL_SYM:
104         free(val->sym);
105         break;
106     }
107 
108     free(val);
109 }
110 
111 
112 lval *lval_add(lval * xs, lval * x)
113 {
114     xs->count++;
115     xs->cell = realloc(xs->cell, sizeof(lval *) * xs->count);
116     xs->cell[xs->count - 1] = x;
117 
118     return xs;
119 }
120 
121 
122 lval *lval_pop(lval * xs, int i)
123 {
124     lval *elem = xs->cell[i];
125 
126     memmove(&xs->cell[i], &xs->cell[i + 1],
127             sizeof(lval *) * (xs->count - i - 1));
128 
129     xs->count--;
130 
131     xs->cell = realloc(xs->cell, sizeof(lval *) * xs->count);
132 
133     return elem;
134 }
135 
136 
137 lval *lval_take(lval * xs, int i)
138 {
139     lval *elem = lval_pop(xs, i);
140     lval_del(xs);
141 
142     return elem;
143 }
144 
145 
146 void lval_print_err(lval * val)
147 {
148     switch (val->err) {
149     case LERR_BAD_NUM:
150         puts("Error: invalid number");
151         break;
152     case LERR_BAD_OP:
```

```
153             puts("Error: invalid operator");
154             break;
155         case LERR_BAD_SEXPR:
156             puts("Error: S-expression does not start with symbol");
157             break;
158         case LERR_DIV_ZERO:
159             puts("Error: division by zero");
160             break;
161         }
162     }
163
164
165     void lval_print(lval * val);
166
167
168     void lval_sexpr_print(lval * sexpr, char open, char close)
169     {
170         putchar(open);
171         int i = 0;
172         while (i++ < sexpr->count - 1) {
173             lval_print(sexpr->cell[i]);
174             putchar(' ');
175         }
176         lval_print(sexpr->cell[i]);
177         putchar(close);
178     }
179
180
181     void lval_print(lval * val)
182     {
183         switch (val->type) {
184         case LVAL_ERR:
185             lval_print_err(val);
186             break;
187         case LVAL_NUM:
188             printf("%g", val->num);
189             break;
190         case LVAL_SEXPR:
191             lval_sexpr_print(val, '(', ')');
192             break;
193         case LVAL_SYM:
194             fputs(val->sym, stdout);
195             break;
196         }
197     }
198
199
200     void lval_println(lval * val)
201     {
202         lval_print(val);
203         putchar('\n');
```

```
204 }
205
206
207 lval *builtin_op(char *op, lval * args)
208 {
209     for (int i = 0; i < args->count; i++) {
210         if (!lval_is_num(args->cell[i])) {
211             lval_del(args);
212             return lval_err(LERR_BAD_NUM);
213         }
214     }
215
216     lval *result = lval_pop(args, 0);
217
218     if (!strcmp(op, "-") && !args->count)
219         result->num = -result->num;
220
221     while (args->count > 0) {
222         lval *y = lval_pop(args, 0);
223
224         if (!strcmp(op, "+")) {
225             result->num += y->num;
226         } else if (!strcmp(op, "-")) {
227             result->num -= y->num;
228         } else if (!strcmp(op, "*")) {
229             result->num *= y->num;
230         } else if (!strcmp(op, "/")) {
231             if (!y->num) {
232                 lval_del(result);
233                 lval_del(y);
234                 result = lval_err(LERR_DIV_ZERO);
235                 break;
236             }
237             result->num /= y->num;
238         } else if (!strcmp(op, "%")) {
239             if (!y->num) {
240                 lval_del(result);
241                 lval_del(y);
242                 result = lval_err(LERR_DIV_ZERO);
243                 break;
244             }
245             result->num = fmod(result->num, y->num);
246         } else if (!strcmp(op, "^")) {
247             result->num = pow(result->num, y->num);
248         } else {
249             lval_del(result);
250             lval_del(y);
251             result = lval_err(LERR_BAD_OP);
252             break;
253         }
254         lval_del(y);
```

```
255     }
256
257     lval_del(args);
258
259     return result;
260 }
261
262
263 lval *lval_eval(lval * val);
264
265
266 lval *lval_eval_sexpr(lval * args)
267 {
268     if (!args->count)
269         return args;
270     for (int i = 0; i < args->count; i++) {
271         args->cell[i] = lval_eval(args->cell[i]);
272         if (args->cell[i]->type == LVAL_ERR)
273             return lval_take(args, i);
274     }
275
276     if (args->count == 1)
277         return lval_take(args, 0);
278
279     lval *car = lval_pop(args, 0);;
280     if (car->type != LVAL_SYM) {
281         lval_del(car);
282         lval_del(args);
283
284         return lval_err(LERR_BAD_SEXPR);
285     }
286
287     lval *result = builtin_op(car->sym, args);
288     lval_del(car);
289
290     return result;
291 }
292
293
294 lval *lval_eval(lval * val)
295 {
296     if (val->type == LVAL_SEXPR)
297         return lval_eval_sexpr(val);
298
299     return val;
300 }
301
302
303 lval *lval_read_num(mpc_ast_t * ast)
304 {
305
```

```
306     errno = 0;
307     double num = strtod(ast->contents, NULL);
308     return errno != ERANGE ? lval_num(num) : lval_err(LERR_BAD_NUM);
309 }
310
311
312 lval *lval_read(mpc_ast_t * ast)
313 {
314     if (strstr(ast->tag, "number"))
315         return lval_read_num(ast);
316
317     if (strstr(ast->tag, "symbol"))
318         return lval_sym(ast->contents);
319
320     lval *sexpr = NULL;
321     if (!strcmp(ast->tag, ">"))
322         sexpr = lval_sexpr();
323     if (strstr(ast->tag, "sexpr"))
324         sexpr = lval_sexpr();
325
326     for (int i = 0; i < ast->children_num; i++) {
327         if (!strcmp(ast->children[i]->contents, "("))
328             continue;
329         if (!strcmp(ast->children[i]->contents, ")"))
330             continue;
331         if (!strcmp(ast->children[i]->tag, "regex"))
332             continue;
333         sexpr = lval_add(sexpr, lval_read(ast->children[i]));
334     }
335
336     return sexpr;
337 }
338
339
340 int main(int argc, char *argv[])
341 {
342     mpc_parser_t *Integer = mpc_new("integer");
343     mpc_parser_t *Float = mpc_new("float");
344     mpc_parser_t *Number = mpc_new("number");
345     mpc_parser_t *Symbol = mpc_new("symbol");
346     mpc_parser_t *Sexpr = mpc_new("sexpr");
347     mpc_parser_t *Expr = mpc_new("expr");
348     mpc_parser_t *Lispy = mpc_new("lispy");
349
350     mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
351               Integer, Float, Number, Symbol, Sexpr, Expr, Lispy);
352
353     puts("Lispy v0.9.0");
354     puts("Press ctrl-c to exit\n");
355
356     bool nonempty;
```

```
357     do {
358         char *input = readline("> ");
359         if ((nonempty = (input && *input))) {
360             add_history(input);
361 
362             mpc_result_t parsed;
363             if (mpc_parse("<stdin>", input, Lispy, &parsed)) {
364                 mpc_ast_t *ast = parsed.output;
365 
366                 lval *result = lval_eval(lval_read(ast));
367                 lval_println(result);
368 
369                 mpc_ast_delete(ast);
370             } else {
371                 mpc_err_print(parsed.error);
372                 mpc_err_delete(parsed.error);
373             }
374         }
375 
376         free(input);
377     } while (nonempty);
378 
379     mpc_cleanup(9, Integer, Float, Number, Symbol, Sexpr, Expr, Lispy);
380 
381     return 0;
382 }
```

Chunks

⟨*Undefine and delete the parsers.* 7d⟩ 5, 7d
⟨*created parsers* 7b⟩ 7b, 7c, 7d
⟨`input` *is nonempty* 8a⟩ 8a, 17a
⟨*lispy.c* 2a⟩ 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d, 3e, 4a, 4b, 4c, 4d, 4e, 5
⟨*lispy.mpc* 6b⟩ 6b
⟨*the argument is not a number* 10f⟩ 10f, 10h
⟨*the input can be parsed as Lispy code* 8d⟩ 8d, 17a

Index

Glossary

AST abstract syntax tree, a tree representation of the abstract syntactic structure of source code. 9, 15

grammar 6, 7

Describe what a grammar is

parser 7

Describe what a parser is

PLT programming language theory, 1

Describe programming language theory

REPL Read-Eval-Print Loop, 6, 7

Describe what a REPL is

References

Daniel Holden. Build Your Own Lisp. http://buildyourownlisp.com, 2018a. Accessed: 2018-05-13.

Daniel Holden. Micro Parser Combinators. https://github.com/orangeduck/mpc, 2018b. Accessed: 2018-05-13.

Norman Ramsey. Noweb – a simple, extensible tool for literate programming. https://www.cs.tufts.edu/~nr/noweb/, 2012. Accessed: 2018-05-13.

Jess Thrysoee. Editline Library (libedit) – port of netbsd command line editor library. http://thrysoee.dk/editline/, 2017. Accessed: 2018-05-13.

Todo list