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.* 22b⟩

  ⟨*Define some useful macros.* 20a⟩

  ⟨*Load the Lispy grammar.* 6c⟩

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

  ⟨*Define the Lispy data structures.* 18c⟩

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. 10b⟩

    return xs;
}
```

Defines:
  `lval_add`, used in chunk 9g.
Uses `lval` 18c.

2c ⟨*lispy.c* 2a⟩+≡

```
lval *lval_pop(lval *xs, int i)
{
    ⟨Extract an element and shift the list. 15d⟩
}
```

Defines:
  `lval_pop`, used in chunks 10h, 14a, and 16b.
Uses `lval` 18c.

3a ⟨*lispy.c* 2a⟩+≡
```
lval *lval_take(lval *xs, int i)
{
    ⟨Pop the list then delete it. 16b⟩
}
```

Defines:
  lval_take, used in chunks 13 and 14.
Uses lval 18c.

Forward declare[4] lval_print, since it's mutually recursive[5] with lval_expr_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 18c and lval_print 3d.

3c ⟨*lispy.c* 2a⟩+≡
```
void lval_expr_print(lval *expr, char open, char close)
{
    ⟨Print an expression. 16e⟩
}
```

Defines:
  lval_expr_print, used in chunks 3c and 17a.
Uses lval 18c.

3d ⟨*lispy.c* 2a⟩+≡
```
void lval_print(lval *val)
{
    ⟨Print a Lispy value. 17a⟩
}
```

Defines:
  lval_print, used in chunks 3 and 16f.
Uses lval 18c.

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

Defines:
  lval_println, used in chunk 16d.
Uses lval 18c and lval_print 3d.

4a ⟨*lispy.c* 2a⟩+≡
```
lval *builtin_list(lval *args)
{
    ⟨Convert an S-expression to a Q-expression. 13c⟩
}
```

Defines:
  `builtin_list`, used in chunk 13b.
Uses `lval` 18c.

4b ⟨*lispy.c* 2a⟩+≡
```
lval *builtin_head(lval *args)
{
    ⟨Pop the list and delete the rest. 13e⟩
}
```

Defines:
  `builtin_head`, used in chunk 13d.
Uses `lval` 18c.

4c ⟨*lispy.c* 2a⟩+≡
```
lval *builtin_op(char *op, lval *args)
{
  ⟨Eval(uate) a built-in operation. 10g⟩
}
```

Defines:
  `builtin_binop`, never used.
Uses `lval` 18c.

4d ⟨*lispy.c* 2a⟩+≡
```
lval *builtin(char *fname, lval *args)
{
    ⟨Evaluate a built-in function or operation. 13b⟩
}
```

Defines:
  `builtin`, used in chunk 15b.
Uses `lval` 18c.

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

4e ⟨*lispy.c* 2a⟩+≡
```
lval *lval_eval(lval* val);
```

Uses `lval` 18c.

5a ⟨*lispy.c* 2a⟩+≡
```
lval* lval_eval_sexpr(lval *args)
{
    ⟨Evaluate an S-expression. 14e⟩
}
```

Uses `lval` 18c.

5b ⟨*lispy.c* 2a⟩+≡
```
lval* lval_eval(lval* val)
{
    ⟨Evaluate an expression. 15c⟩
}
```

Uses `lval` 18c.

5c ⟨*lispy.c* 2a⟩+≡
```
lval *lval_read_num(mpc_ast_t *ast)
{
    ⟨Read a number. 9a⟩
}


lval *lval_read(mpc_ast_t *ast)
{
    ⟨Read a Lispy value. 8e⟩
}
```

Defines:
  `lval_read`, used in chunks 8d and 9g.
Uses `ast` 8d, `lval` 18c, and `mpc_ast_t` 23f.

5d ⟨*lispy.c* 2a⟩+≡
```
int main(int argc, char *argv[])
{
    ⟨Define the language. 6d⟩

    ⟨Print version and exit information. 6a⟩

    ⟨Loop until the input is empty. 17c⟩

    ⟨Undefine and delete the parsers. 7c⟩

    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 v1.1.1");
  puts("Press ctrl-c to exit\n");
```

Uses Lispy 6d.
This code is used in chunk 5d.

Defining the Language

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

6b ⟨*lispy.mpc* 6b⟩≡

```
  number "number" : /[-+]?[0-9]+(\.[0-9]+)?/ ;
  symbol "symbol" : /[a-za-Z_+*%^\/\\=<>!*-]+/ ;
  sexpr           : '(' <symbol> <expr>+ ')' ;
  qexpr           : '{' (<symbol> | <expr>)* '}' ;
  expr            : <number> | <sexpr> | <qexpr> ;
  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 7b.
This code is used in chunk 2a.

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

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

6d ⟨*Define the language.* 6d⟩≡

```
  mpc_parser_t *Number    = mpc_new("number");
  mpc_parser_t *Symbol    = mpc_new("symbol");
  mpc_parser_t *Sexpr     = mpc_new("sexpr");
  mpc_parser_t *Qexpr     = mpc_new("qexpr");
  mpc_parser_t *Expr      = mpc_new("expr");
  mpc_parser_t *Lispy     = mpc_new("lispy");
```

Defines:
Expr, used in chunk 7a.
Lispy, used in chunks 6–8.
Number, used in chunk 7a.
Qexpr, used in chunk 7a.
Sexpr, used in chunk 7a.
Symbol, used in chunk 7a.
Uses mpc_new 23f and mpc_parser_t 23f.
This definition is continued in chunk 7b.
This code is used in chunk 5d.

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

7a ⟨*created parsers* 7a⟩≡
```
Number, Symbol, Sexpr, Qexpr, Expr, Lispy
```
Uses `Expr` 6d, `Lispy` 6d, `Number` 6d, `Qexpr` 6d, `Sexpr` 6d, and `Symbol` 6d.
This code is used in chunk 7.

… we can define the Lispy language.

7b ⟨*Define the language.* 6d⟩+≡
```
mpc_err_t *err = mpca_lang(MPCA_LANG_PREDICTIVE, LISPY_GRAMMAR,
                           ⟨created parsers 7a⟩);

if (err != NULL) {
    puts(LISPY_GRAMMAR);
    mpc_err_print(err);
    mpc_err_delete(err);
    exit(100);
}
```
Uses `LISPY_GRAMMAR` 6c, `mpca_lang` 23f, `mpc_err_delete` 23f, and `mpc_err_print` 23f.

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

7c ⟨*Undefine and delete the parsers.* 7c⟩≡
```
mpc_cleanup(6, ⟨created parsers 7a⟩);
```
Uses `mpc_cleanup` 23f.
This code is used in chunk 5d.

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

7d ⟨*Read a line of user input.* 7d⟩≡
```
char *input = readline("> ");
```
Defines:
`input`, used in chunks 7, 8, and 18b.
Uses `readline` 23e.
This code is used in chunk 17d.

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

7e ⟨`input` *is nonempty* 7e⟩≡
```
input && *input
```
Uses `input` 7d.
This code is used in chunk 18a.

Here, `input` is functionally equivalent to `input ≠ NULL`, and `*input` is functionally equivalent to `input[0] ≠ '\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

8a ⟨*Add* `input` *to the history table.* 8a⟩≡

```
add_history(input);
```

Uses `add_history` 23e and `input` 7d.
This code is used in chunk 18a.

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

8b ⟨*Declare a variable to hold parsing results.* 8b⟩≡

```
mpc_result_t parsed;
```

Defines:
`parsed`, used in chunks 8 and 17b.
Uses `mpc_result_t` 23f.
This code is used in chunk 18a.

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

8c ⟨*the input can be parsed as Lispy code* 8c⟩≡

```
mpc_parse("<stdin>", input, Lispy, &parsed)
```

Uses `Lispy` 6d, `input` 7d, `mpc_parse` 23f, and `parsed` 8b.
This code is used in chunk 18a.

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.

8d ⟨*Eval(uate) the input.* 8d⟩≡

```
mpc_ast_t *ast = parsed.output;

lval *result = lval_eval(lval_read(ast));
```

Defines:
`ast`, used in chunks 5c, 8, 9, and 16d.
Uses `lval` 18c, `lval_read` 5c, `mpc_ast_t` 23f, and `parsed` 8b.
This code is used in chunk 18a.

Describe the evaluation strategy

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

8e ⟨*Read a Lispy value.* 8e⟩≡

```
if (strstr(ast->tag, "number"))
    return lval_read_num(ast);
```

Uses `ast` 8d and `strstr` 23d.
This definition is continued in chunks 9 and 10c.
This code is used in chunk 5c.

Describe this

9a ⟨*Read a number.* 9a⟩≡
```
errno = 0;
double num = strtod(ast->contents, NULL);
return errno != ERANGE ? lval_num(num) : lval_err(LERR_BAD_NUM);
```
Uses ast 8d, lval_err 20c, lval_num 19b, and strtod 23b.
This code is used in chunk 5c.

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

9b ⟨*Read a Lispy value.* 8e⟩+≡
```
if (strstr(ast->tag, "symbol"))
    return lval_sym(ast->contents);
```
Uses ast 8d, lval_sym 20d, and strstr 23d.

Describe this

9c ⟨*Read a Lispy value.* 8e⟩+≡
```
lval *val = NULL;
```
Uses lval 18c.

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

9d ⟨*Read a Lispy value.* 8e⟩+≡
```
if (!strcmp(ast->tag, ">"))
    val = lval_sexpr();
```
Uses ast 8d, lval_sexpr 21a, and strcmp 23d.

If it's tagged as a Q-expression, create an empty list.

9e ⟨*Read a Lispy value.* 8e⟩+≡
```
if (strstr(ast->tag, "qexpr"))
    val = lval_qexpr();
```
Uses ast 8d, lval_qexpr 21b, and strstr 23d.

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

9f ⟨*Read a Lispy value.* 8e⟩+≡
```
if (strstr(ast->tag, "sexpr"))
    val = lval_sexpr();
```
Uses ast 8d, lval_sexpr 21a, and strstr 23d.

Describe this

9g ⟨*Read a Lispy value.* 8e⟩+≡
```
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]->contents, "{")) continue;
    if(!strcmp(ast->children[i]->contents, "}")) continue;
    if(!strcmp(ast->children[i]->tag, "regex")) continue;
    val = lval_add(val, lval_read(ast->children[i]));
}
```
Uses ast 8d, lval_add 2b, lval_read 5c, and strcmp 23d.

10a ⟨*Reallocate the memory used.* 10a⟩≡

```
xs->cell = realloc(xs->cell, sizeof(lval *) * xs->count);
```

Uses `lval` 18c.
This code is used in chunks 10b and 16a.

Describe this, incl. how it's not cons

10b ⟨*Add an element to an S-expression.* 10b⟩≡

```
xs->count++;
⟨Reallocate the memory used. 10a⟩
xs->cell[xs->count - 1] = x;
```

This code is used in chunk 2b.

Finally, return the Lispy value.

10c ⟨*Read a Lispy value.* 8e⟩+≡

```
return val;
```

10d ⟨*For each argument* 10d⟩≡

```
for (int i = 0; i < args->count; i++)
```

This code is used in chunks 10g and 14f.

10e ⟨*the argument is not a number* 10e⟩≡

```
!lval_is_num(args->cell[i])
```

Uses `lval_is_num` 19c.
This code is used in chunk 10g.

10f ⟨*Delete the arguments and return a bad number error.* 10f⟩≡

```
lval_del(args);
return lval_err(LERR_BAD_NUM);
```

Uses `lval_del` 22a and `lval_err` 20c.
This code is used in chunk 10g.

Evaluating built-in operations

Ensure all arguments are numbers.

10g ⟨*Eval(uate) a built-in operation.* 10g⟩≡

```
⟨For each argument 10d⟩ {
    if (⟨the argument is not a number 10e⟩) {
        ⟨Delete the arguments and return a bad number error. 10f⟩
    }
}
```

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

10h ⟨*Pop the first element.* 10h⟩≡

```
lval_pop(args, 0);
```

Uses `lval_pop` 2c.
This code is used in chunks 11 and 15a.

Pop the first element.

11a ⟨*Eval(uate) a built-in operation.* 10g⟩+≡
```
lval *result = ⟨Pop the first element. 10h⟩
```
Uses `lval` 18c.

If the operation is unary subtraction, negate the operand.

11b ⟨*Eval(uate) a built-in operation.* 10g⟩+≡
```
if (!strcmp(op, "-") && !args->count)
    result->num = -result->num;
```
Uses `strcmp` 23d.

11c ⟨*Pop the next element.* 11c⟩≡
```
lval *y = ⟨Pop the first element. 10h⟩
```
Uses `lval` 18c.
This code is used in chunk 11d.

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

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

If the `op` is "+", perform addition.

11e ⟨*Perform a built-in operation.* 11e⟩≡
```
if (!strcmp(op, "+")) {
    result->num += y->num;
}
```
Uses `strcmp` 23d.
This definition is continued in chunks 11 and 12.
This code is used in chunk 11d.

If the `op` is "-", perform subtraction.

11f ⟨*Perform a built-in operation.* 11e⟩+≡
```
else if (!strcmp(op, "-")) {
    result->num -= y->num;
}
```
Uses `strcmp` 23d.

If the `op` is "*", perform multiplication.

11g ⟨*Perform a built-in operation.* 11e⟩+≡
```
else if (!strcmp(op, "*")) {
    result->num *= y->num;
}
```
Uses `strcmp` 23d.

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.* 11e⟩+≡
```
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 `lval_del` 22a, `lval_err` 20c, and `strcmp` 23d.

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

12b ⟨*Perform a built-in operation.* 11e⟩+≡
```
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 `fmod` 23c, `lval_del` 22a, `lval_err` 20c, and `strcmp` 23d.

If the `op` is "^", perform exponentiation.

12c ⟨*Perform a built-in operation.* 11e⟩+≡
```
else if (!strcmp(op, "^")) {
    result->num = pow(result->num, y->num);
}
```
Uses `pow` 23c and `strcmp` 23d.

Otherwise, return a `LERR_BAD_OP` error.

12d ⟨*Perform a built-in operation.* 11e⟩+≡
```
else {
    lval_del(result);
    lval_del(y);
    result = lval_err(LERR_BAD_OP);
    break;
}
```
Uses `lval_del` 22a and `lval_err` 20c.

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

12e ⟨*Perform a built-in operation.* 11e⟩+≡
```
lval_del(y);
```
Uses `lval_del` 22a.

Delete the input expression and return the result.

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

  return result;
```
Uses `lval_del` 22a.

Built-in functions

If the function name is `list`, convert the given S-expression to a Q-expression and return it.

13b ⟨*Evaluate a built-in function or operation.* 13b⟩≡
```
  if (!strcmp("list", fname))
      return builtin_list(args);
```

Uses `builtin_list` 4a and `strcmp` 23d.
This definition is continued in chunks 13 and 14.
This code is used in chunk 4d.

13c ⟨*Convert an S-expression to a Q-expression.* 13c⟩≡
```
  args->type = LVAL_QEXPR;
  return args;
```
Uses `LVAL_QEXPR` 19a.
This code is used in chunk 4a.

If the function name is `"head"`, pop the list and delete the rest.

13d ⟨*Evaluate a built-in function or operation.* 13b⟩+≡
```
  if (!strcmp("head", fname))
      return builtin_head(args);
```
Uses `builtin_head` 4b and `strcmp` 23d.

Ensure there is exactly one argument.

13e ⟨*Pop the list and delete the rest.* 13e⟩≡
```
  LVAL_ASSERT(args, args->count == 1,
      "too many arguments for 'head'");
```
This definition is continued in chunks 13 and 14.
This code is used in chunk 4b.

Ensure the first argument is a Q-expression.

13f ⟨*Pop the list and delete the rest.* 13e⟩+≡
```
  LVAL_ASSERT(args, args->cell[0]->type == LVAL_QEXPR,
      "invalid argument for 'head'");
```
Uses `LVAL_QEXPR` 19a.

Ensure the list passed to `head` is nonempty.

13g ⟨*Pop the list and delete the rest.* 13e⟩+≡
```
  LVAL_ASSERT(args, args->cell[0]->count,
      "cannot get 'head' of the empty list");
```

eTake the first element of the list.

13h ⟨*Pop the list and delete the rest.* 13e⟩+≡
```
  lval *val = lval_take(args, 0);
```
Uses `lval` 18c and `lval_take` 3a.

Delete the rest.

14a ⟨*Pop the list and delete the rest.* 13e⟩+≡
```
while (val->count > 1)
    lval_del(lval_pop(val, 1));
```
Uses `lval_del` 22a and `lval_pop` 2c.

Return the head of the list.

14b ⟨*Pop the list and delete the rest.* 13e⟩+≡
```
return val;
```

If the function name is a built-in operation, perform and return it.

14c ⟨*Evaluate a built-in function or operation.* 13b⟩+≡
```
if (strstr("+-/*^%", fname))
    return builtin_op(fname, args);
```
Uses `strstr` 23d.

Otherwise, free the memory used by `args` and return an error.

14d ⟨*Evaluate a built-in function or operation.* 13b⟩+≡
```
lval_del(args);

return lval_err(LERR_BAD_FUNC);
```
Uses `lval_del` 22a and `lval_err` 20c.

Evaluating (S)-expressions

If the expression is empty, return it;

14e ⟨*Evaluate an S-expression.* 14e⟩≡
```
if (!args->count)
    return args;
```
This definition is continued in chunks 14 and 15.
This code is used in chunk 5a.

14f ⟨*Evaluate an S-expression.* 14e⟩+≡
```
⟨For each argument 10d⟩ {
    args->cell[i] = lval_eval(args->cell[i]);
    if (args->cell[i]->type == LVAL_ERR)
        return lval_take(args, i);
}
```
Uses `LVAL_ERR` 19a 19a and `lval_take` 3a.

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

14g ⟨*Evaluate an S-expression.* 14e⟩+≡
```
if (args->count == 1)
    return lval_take(args, 0);
```
Uses `lval_take` 3a.

15a ⟨*Evaluate an S-expression.* 14e⟩+≡
```
lval *car = ⟨Pop the first element. 10h⟩;
if (car->type != LVAL_SYM) {
    lval_del(car);
    lval_del(args);

    return lval_err(LERR_BAD_SEXPR);
}
```

Uses `LVAL_SYM` 19a, `lval` 18c, `lval_del` 22a, and `lval_err` 20c.

15b ⟨*Evaluate an S-expression.* 14e⟩+≡
```
lval *result = builtin(car->sym, args);
lval_del(car);

return result;
```
Uses `builtin` 4d, `lval` 18c, and `lval_del` 22a.

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

15c ⟨*Evaluate an expression.* 15c⟩≡
```
if (val->type == LVAL_SEXPR)
    return lval_eval_sexpr(val);

return val;
```
Uses `LVAL_SEXPR` 19a.
This code is used in chunk 5b.

Extract the element at index `i`.

15d ⟨*Extract an element and shift the list.* 15d⟩≡
```
lval *elem = xs->cell[i];
```

Uses `lval` 18c.
This definition is continued in chunks 15 and 16a.
This code is used in chunk 2c.

Shift memory after the element at index `i`.

15e ⟨*Extract an element and shift the list.* 15d⟩+≡
```
memmove(&xs->cell[i], &xs->cell[i + 1],
    sizeof(lval *) * (xs->count - i - 1));
```

Uses `lval` 18c.

Decrease the count.

15f ⟨*Extract an element and shift the list.* 15d⟩+≡
```
xs->count--;
```

15g ⟨*Return the extracted element.* 15g⟩≡
```
return elem;
```
This code is used in chunk 16.

Reallocate the memory used and return the extracted element.

16a ⟨*Extract an element and shift the list.* 15d⟩+≡
  ⟨*Reallocate the memory used.* 10a⟩

  ⟨*Return the extracted element.* 15g⟩

Describe this

16b ⟨*Pop the list then delete it.* 16b⟩≡
```
  lval *elem = lval_pop(xs, i);
  lval_del(xs);
```
Uses `lval` 18c, `lval_del` 22a, and `lval_pop` 2c.
This definition is continued in chunk 16c.
This code is used in chunk 3a.

Return the extracted element.

16c ⟨*Pop the list then delete it.* 16b⟩+≡
  ⟨*Return the extracted element.* 15g⟩

P is for Print

Upon success, print the result and delete the AST.

16d ⟨*Print the result and delete the AST.* 16d⟩≡
```
  lval_println(result);

  mpc_ast_delete(ast);
```
Uses `ast` 8d, `lval_println` 3e, and `mpc_ast_delete` 23f.
This code is used in chunk 18a.

Describe this

Print the opening character.

16e ⟨*Print an expression.* 16e⟩≡
```
  putchar(open);
```
This definition is continued in chunk 16.
This code is used in chunk 3c.

Print all but the last element with a trailing space.

16f ⟨*Print an expression.* 16e⟩+≡
```
  for (int i = 0; i < expr->count; i++) {
      lval_print(expr->cell[i]);
      if (i != (expr->count - 1))
          putchar(' ');
  }
```
Uses `lval_print` 3d.

Print the closing character.

16g ⟨*Print an expression.* 16e⟩+≡
```
  putchar(close);
```

17a ⟨*Print a Lispy value.* 17a⟩≡

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

Uses `LVAL_ERR` 19a 19a, `LVAL_NUM` 19a, `LVAL_QEXPR` 19a, `LVAL_SEXPR` 19a, `LVAL_SYM` 19a, `lval_expr_print` 3c, and `printf` 23a.
This code is used in chunk 3d.

Print and delete the error upon failure.

17b ⟨*Print and delete the error.* 17b⟩≡

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

Uses `mpc_err_delete` 23f, `mpc_err_print` 23f, and `parsed` 8b.
This code is used in chunk 18a.

L is for Loop

17c ⟨*Loop until the input is empty.* 17c⟩≡

```
bool nonempty;
do {
  ⟨Read, eval(uate), and print. 17d⟩
} while (nonempty);
```

Defines:
`nonempty`, used in chunk 18a.
Uses `bool` 22c.
This code is used in chunk 5d.

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

17d ⟨*Read, eval(uate), and print.* 17d⟩≡

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

This definition is continued in chunk 18.
This code is used in chunk 17c.

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

18a ⟨*Read, eval(uate), and print.* 17d⟩+≡
```
  if ((nonempty = (⟨input is nonempty 7e⟩))) {
      ⟨Add input to the history table. 8a⟩

      ⟨Declare a variable to hold parsing results. 8b⟩
      if (⟨the input can be parsed as Lispy code 8c⟩) {
          ⟨Eval(uate) the input. 8d⟩
          ⟨Print the result and delete the AST. 16d⟩
      } else {
          ⟨Print and delete the error. 17b⟩
      }
  }
```

Uses `nonempty` 17c.

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

18b ⟨*Read, eval(uate), and print.* 17d⟩+≡
```
  free(input);
```
Uses `free` 23b and `input` 7d.

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

Error Handling

Describe this struct

18c ⟨*Define the Lispy data structures.* 18c⟩≡
```
  typedef struct lval {
      lval_type_t type;
      union {
          double num;
          char *err;
          char *sym;
      };
      int count;
      struct lval **cell;
  } lval;
```

Defines:
`lval`, used in chunks 2–5, 8–11, 13h, 15, 16b, and 19–22.
Uses `lval_type_t` 19a.
This definition is continued in chunks 19–22.
This code is used in chunk 2a.

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

19a ⟨*Define possible lval and error types.* 19a⟩≡
```
typedef enum {
    LVAL_ERR,
    LVAL_NUM,
    LVAL_QEXPR,
    LVAL_SEXPR,
    LVAL_SYM
} lval_type_t;
```

Defines:
  `LVAL_ERR`, used in chunks 14f, 17a, 20c, and 22a.
  `LVAL_NUM`, used in chunks 17a, 19, and 22a.
  `LVAL_QEXPR`, used in chunks 13, 17a, 21b, and 22a.
  `LVAL_SEXPR`, used in chunks 15c, 17a, 21a, and 22a.
  `LVAL_SYM`, used in chunks 15a, 17a, 20d, and 22a.
  `lval_type_t`, used in chunk 18c.
This code is used in chunk 2a.

Define a constructor for numbers.

19b ⟨*Define the Lispy data structures.* 18c⟩+≡
```
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 9a.
Uses `LVAL_NUM` 19a and `lval` 18c.

Define a convenient predicate for numbers.

19c ⟨*Define the Lispy data structures.* 18c⟩+≡
```
bool lval_is_num(lval *val)
{
    return val->type == LVAL_NUM;
}
```

Defines:
  `lval_is_num`, used in chunk 10e.
Uses `LVAL_NUM` 19a, `bool` 22c, and `lval` 18c.

Define a macro for asserting a condition or returning an error.

20a ⟨*Define some useful macros.* 20a⟩≡
```
#define LVAL_ASSERT(args, cond, err) \
    if (!(cond)) { \
        lval_del(args); \
        return lval_err(err); \
    }
```

Uses `lval_del` 22a and `lval_err` 20c.
This definition is continued in chunk 20b.
This code is used in chunk 2a.

20b ⟨*Define some useful macros.* 20a⟩+≡
```
#define LERR_BAD_FUNC "unknown function"
#define LERR_BAD_NUM "invalid number"
#define LERR_BAD_OP "invalid operation"
#define LERR_DIV_ZERO "division by zero"
#define LERR_BAD_SEXPR "invalid S-expression"
```

Define a constructor for errors.

20c ⟨*Define the Lispy data structures.* 18c⟩+≡
```
lval *lval_err(char *err)
{
    lval *val = malloc(sizeof(lval));
    val->type = LVAL_ERR;
    val->err = err;

    return val;
}
```

Defines:
`lval_err`, used in chunks 9a, 10f, 12, 14d, 15a, and 20a.
Uses `LVAL_ERR` 19a 19a and `lval` 18c.

Define a constructor for symbol.

20d ⟨*Define the Lispy data structures.* 18c⟩+≡
```
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 9b.
Uses `LVAL_SYM` 19a and `lval` 18c.

Define a constructor for an S-expression.

21a ⟨*Define the Lispy data structures.* 18c⟩+≡

```
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` 19a and `lval` 18c.

Define a constructor for a Q-expression.

21b ⟨*Define the Lispy data structures.* 18c⟩+≡

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

    return val;
}
```

Defines:
  `lval_qexpr`, used in chunk 9e.
Uses `LVAL_QEXPR` 19a and `lval` 18c.

Define a destructor for `lval*`.

22a ⟨*Define the Lispy data structures.* 18c⟩+≡

```
void lval_del(lval *val)
{
    switch(val->type) {
    case LVAL_ERR:
        free(val->err);
        break;
    case LVAL_NUM:
        break;
    case LVAL_QEXPR:
    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 10f, 12–16, and 20a.
Uses `LVAL_ERR` 19a 19a, `LVAL_NUM` 19a, `LVAL_QEXPR` 19a, `LVAL_SEXPR` 19a,
  `LVAL_SYM` 19a, `free` 23b, and `lval` 18c.

Headers

Describe headers

22b ⟨*Include the necessary headers.* 22b⟩≡
  ⟨*Include the boolean type and values.* 22c⟩
  ⟨*Include the standard I/O functions.* 23a⟩
  ⟨*Include the standard library definitions.* 23b⟩
  ⟨*Include some mathematical definitions.* 23c⟩
  ⟨*Include some string operations.* 23d⟩

  ⟨*Include the line editing functions from libedit.* 23e⟩
  ⟨*Include the micro parser combinator definitions.* 23f⟩
This code is used in chunk 2a.

22c ⟨*Include the boolean type and values.* 22c⟩≡

```
#include <stdbool.h>
```

Defines:
  `bool`, used in chunks 17c and 19c.
This code is used in chunk 22b.

23a ⟨*Include the standard I/O functions.* 23a⟩≡

```
#include <stdio.h>
```

Defines:
`printf`, used in chunk 17a.
This code is used in chunk 22b.

23b ⟨*Include the standard library definitions.* 23b⟩≡

```
#include <stdlib.h>
```

Defines:
`free`, used in chunks 18b and 22a.
`strtod`, used in chunk 9a.
This code is used in chunk 22b.

23c ⟨*Include some mathematical definitions.* 23c⟩≡

```
#include <math.h>
```

Defines:
`fmod`, used in chunk 12b.
`pow`, used in chunk 12c.
This code is used in chunk 22b.

23d ⟨*Include some string operations.* 23d⟩≡

```
#include <string.h>
```

Defines:
`strcmp`, used in chunks 9 and 11–13.
`strstr`, used in chunks 8, 9, and 14c.
This code is used in chunk 22b.

23e ⟨*Include the line editing functions from libedit.* 23e⟩≡

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

Defines:
`add_history`, used in chunk 8a.
`readline`, used in chunks 23e and 7d.
This code is used in chunk 22b.

23f ⟨*Include the micro parser combinator definitions.* 23f⟩≡

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

Defines:
`mpca_lang`, used in chunk 7b.
`mpc_ast_delete`, used in chunk 16d.
`mpc_ast_print`, never used.
`mpc_ast_t`, used in chunks 5c and 8d.
`mpc_cleanup`, used in chunks 23f and 7c.
`mpc_err_delete`, used in chunks 7b and 17b.
`mpc_err_print`, used in chunks 7b and 17b.
`mpc_new`, used in chunk 6d.
`mpc_parse`, used in chunks 23f and 8c.
`mpc_parser_t`, used in chunk 6d.
`mpc_result_t`, used in chunk 8b.
This code is used in chunk 22b.

Full Listings

lispy.mpc:

```
number ”number” : /[-+]?[0-9]+(\.[0-9]+)?/ ;
symbol ”symbol” : /[a-za-Z_+*%^\/\\=<>!*-]+/ ;
sexpr           : '(' <symbol> <expr>+ ')' ;
qexpr           : '{' (<symbol> | <expr>)* '}' ;
expr            : <number> | <sexpr> | <qexpr> ;
lispy           : /^/ <expr>* /$/ ;
```

lispy.c:

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

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


#define LVAL_ASSERT(args, cond, err) \
    if (!(cond)) { \
        lval_del(args); \
        return lval_err(err); \
    }

#define LERR_BAD_FUNC "unknown function"
#define LERR_BAD_NUM "invalid number"
#define LERR_BAD_OP "invalid operation"
#define LERR_DIV_ZERO "division by zero"
#define LERR_BAD_SEXPR "invalid S-expression"


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


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



typedef struct lval {
    lval_type_t type;
    union {
        double num;
        char *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;
}


bool lval_is_num(lval * val)
{
    return val->type == LVAL_NUM;
}


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

    return val;
}


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;
}


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

    return val;
}


lval *lval_qexpr(void)
{
    lval *val = malloc(sizeof(lval));
```

```c
    val->type = LVAL_QEXPR;
    val->count = 0;
    val->cell = NULL;

    return val;
}


void lval_del(lval * val)
{
    switch (val->type) {
    case LVAL_ERR:
        free(val->err);
        break;
    case LVAL_NUM:
        break;
    case LVAL_QEXPR:
    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);
}


lval *lval_add(lval * xs, lval * x)
{
    xs->count++;
    xs->cell = realloc(xs->cell, sizeof(lval *) * xs->count);
    xs->cell[xs->count - 1] = x;

    return xs;
}


lval *lval_pop(lval * xs, int i)
{
    lval *elem = xs->cell[i];

    memmove(&xs->cell[i], &xs->cell[i + 1],
            sizeof(lval *) * (xs->count - i - 1));

    xs->count--;

    xs->cell = realloc(xs->cell, sizeof(lval *) * xs->count);

    return elem;
}


lval *lval_take(lval * xs, int i)
{
    lval *elem = lval_pop(xs, i);
    lval_del(xs);

    return elem;
}


void lval_print(lval * val);


void lval_expr_print(lval * expr, char open, char close)
{
    putchar(open);
    for (int i = 0; i < expr->count; i++) {
        lval_print(expr->cell[i]);
        if (i != (expr->count - 1))
            putchar(' ');
    }
    putchar(close);
}


void lval_print(lval * val)
{
    switch (val->type) {
    case LVAL_ERR:
        printf("Error: %s", val->err);
        break;
    case LVAL_NUM:
        printf("%g", val->num);
        break;
    case LVAL_QEXPR:
        lval_expr_print(val, '{', '}');
        break;
    case LVAL_SEXPR:
        lval_expr_print(val, '(', ')');
        break;
    case LVAL_SYM:
        fputs(val->sym, stdout);
        break;
    }
}


```

```
void lval_println(lval * val)
{
    lval_print(val);
    putchar('\n');
}


lval *builtin_list(lval * args)
{
    args->type = LVAL_QEXPR;
    return args;
}


lval *builtin_head(lval * args)
{
    LVAL_ASSERT(args, args->count == 1, "too many arguments for 'head'");
    LVAL_ASSERT(args, args->cell[0]->type == LVAL_QEXPR,
                "invalid argument for 'head'");
    LVAL_ASSERT(args, args->cell[0]->count,
                "cannot get 'head' of the empty list");
    lval *val = lval_take(args, 0);
    while (val->count > 1)
        lval_del(lval_pop(val, 1));
    return val;
}


lval *builtin_op(char *op, lval * args)
{
    for (int i = 0; i < args->count; i++) {
        if (!lval_is_num(args->cell[i])) {
            lval_del(args);
            return lval_err(LERR_BAD_NUM);
        }
    }

    lval *result = lval_pop(args, 0);

    if (!strcmp(op, "-") && !args->count)
        result->num = -result->num;

    while (args->count > 0) {
        lval *y = lval_pop(args, 0);

        if (!strcmp(op, "+")) {
            result->num += y->num;
        } else if (!strcmp(op, "-")) {
            result->num -= y->num;
        } else if (!strcmp(op, "*")) {
            result->num *= y->num;
        } else if (!strcmp(op, "/")) {
            if (!y->num) {
                lval_del(result);
                lval_del(y);
                result = lval_err(LERR_DIV_ZERO);
                break;
            }
            result->num /= y->num;
        } 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);
        } else if (!strcmp(op, "^")) {
            result->num = pow(result->num, y->num);
        } else {
            lval_del(result);
            lval_del(y);
            result = lval_err(LERR_BAD_OP);
            break;
        }
        lval_del(y);
    }

    lval_del(args);

    return result;
}


lval *builtin(char *fname, lval * args)
{
    if (!strcmp("list", fname))
        return builtin_list(args);

    if (!strcmp("head", fname))
        return builtin_head(args);
    if (strstr("+-/*^%", fname))
        return builtin_op(fname, args);

    lval_del(args);

    return lval_err(LERR_BAD_FUNC);
}

lval *lval_eval(lval * val);


lval *lval_eval_sexpr(lval * args)
{
    if (!args->count)
        return args;
    for (int i = 0; i < args->count; i++) {
        args->cell[i] = lval_eval(args->cell[i]);
        if (args->cell[i]->type == LVAL_ERR)
            return lval_take(args, i);
    }

    if (args->count == 1)
        return lval_take(args, 0);

    lval *car = lval_pop(args, 0);;
    if (car->type != LVAL_SYM) {
        lval_del(car);
        lval_del(args);

        return lval_err(LERR_BAD_SEXPR);
    }

    lval *result = builtin(car->sym, args);
    lval_del(car);

    return result;
}


lval *lval_eval(lval * val)
{
    if (val->type == LVAL_SEXPR)
        return lval_eval_sexpr(val);

    return val;
}


lval *lval_read_num(mpc_ast_t * ast)
{
    errno = 0;
    double num = strtod(ast->contents, NULL);
    return errno != ERANGE ? lval_num(num) : lval_err(LERR_BAD_NUM);
}


lval *lval_read(mpc_ast_t * ast)
{
    if (strstr(ast->tag, "number"))
        return lval_read_num(ast);

    if (strstr(ast->tag, "symbol"))
```

```c
        return lval_sym(ast->contents);

    lval *val = NULL;
    if (!strcmp(ast->tag, ">"))
        val = lval_sexpr();
    if (strstr(ast->tag, "qexpr"))
        val = lval_qexpr();
    if (strstr(ast->tag, "sexpr"))
        val = lval_sexpr();

    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]->contents, "{"))
            continue;
        if (!strcmp(ast->children[i]->contents, "}"))
            continue;
        if (!strcmp(ast->children[i]->tag, "regex"))
            continue;
        val = lval_add(val, lval_read(ast->children[i]));
    }

    return val;
}


int main(int argc, char *argv[])
{
    mpc_parser_t *Number = mpc_new("number");
    mpc_parser_t *Symbol = mpc_new("symbol");
    mpc_parser_t *Sexpr = mpc_new("sexpr");
    mpc_parser_t *Qexpr = mpc_new("qexpr");
    mpc_parser_t *Expr = mpc_new("expr");
    mpc_parser_t *Lispy = mpc_new("lispy");

    mpc_err_t *err = mpca_lang(MPCA_LANG_PREDICTIVE, LISPY_GRAMMAR,
                               Number, Symbol, Sexpr, Qexpr, Expr, Lispy);

    if (err != NULL) {
        puts(LISPY_GRAMMAR);
        mpc_err_print(err);
        mpc_err_delete(err);
        exit(100);
    }

    puts("Lispy v1.1.1");
    puts("Press ctrl-c to exit\n");

    bool nonempty;
    do {
        char *input = readline("> ");
        if ((nonempty = (input && *input))) {
            add_history(input);

            mpc_result_t parsed;
            if (mpc_parse("<stdin>", input, Lispy, &parsed)) {
                mpc_ast_t *ast = parsed.output;

                lval *result = lval_eval(lval_read(ast));
                lval_println(result);

                mpc_ast_delete(ast);
            } else {
                mpc_err_print(parsed.error);
                mpc_err_delete(parsed.error);
            }
        }

        free(input);
    } while (nonempty);

    mpc_cleanup(6, Number, Symbol, Sexpr, Qexpr, Expr, Lispy);

    return 0;
}
```

Chunks

Index

Glossary

AST abstract syntax tree, a tree representation of the abstract syntactic structure of source code. 8, 9, 16

grammar 6, 7

Describe what a grammar is

parser 6

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