Lispy: a simple Lisp-like language

Eric Bailey

May 10, 2018 [1]

[1] Current version: VERSION. Last updated July 14, 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.* 25b⟩

⟨*Define some useful macros.* 23a⟩

⟨*Load the Lispy grammar.* 7c⟩

⟨*Define possible lval and error types.* 22a⟩

⟨*Define the Lispy data structures.* 21c⟩

This definition is continued in chunks 2–6.
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. 11b⟩

    return xs;
}
```

Defines:
`lval_add`, used in chunks 10g and 16b.
Uses `lval` 21c.

2c ⟨*lispy.c* 2a⟩+≡

```
lval *lval_pop(lval *xs, int i)
{
    ⟨Extract an element and shift the list. 18d⟩
}
```

Defines:
`lval_pop`, used in chunks 11h, 15, 16, and 19b.
Uses `lval` 21c.

3a ⟨*lispy.c* 2a⟩+≡

```
lval *lval_take(lval *xs, int i)
{
    ⟨Pop the list then delete it. 19b⟩
}
```

Defines:
  `lval_take`, used in chunks 14h, 15d, and 17.
Uses `lval` 21c.

3b ⟨*lispy.c* 2a⟩+≡

```
lval *lval_join(lval *xs, lval *ys)
{
    ⟨Add every y in ys to xs. 16b⟩
}
```

Defines:
  `lval_join`, used in chunk 16a.
Uses `lval` 21c.

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

3c ⟨*lispy.c* 2a⟩+≡

```
void lval_print(lval *val);
```

Uses `lval` 21c and `lval_print` 3e.

3d ⟨*lispy.c* 2a⟩+≡

```
void lval_expr_print(lval *expr, char open, char close)
{
    ⟨Print an expression. 19e⟩
}
```

Defines:
  `lval_expr_print`, used in chunks 3d and 20a.
Uses `lval` 21c.

3e ⟨*lispy.c* 2a⟩+≡

```
void lval_print(lval *val)
{
    ⟨Print a Lispy value. 20a⟩
}
```

Defines:
  `lval_print`, used in chunks 3, 4a, and 19f.
Uses `lval` 21c.

4a ⟨*lispy.c* 2a⟩+≡

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

Defines:
  `lval_println`, used in chunk 19d.
Uses `lval` 21c and `lval_print` 3e.

4b ⟨*lispy.c* 2a⟩+≡

```
lval *builtin_list(lval *args)
{
    ⟨Convert an S-expression to a Q-expression. 14c⟩
}
```

Defines:
  `builtin_list`, used in chunk 14b.
Uses `lval` 21c.

4c ⟨*lispy.c* 2a⟩+≡

```
lval *builtin_head(lval *args)
{
    ⟨Pop the list and delete the rest. 14e⟩
}
```

Defines:
  `builtin_head`, used in chunk 14d.
Uses `lval` 21c.

4d ⟨*lispy.c* 2a⟩+≡

```
lval *builtin_tail(lval *args)
{
    ⟨Return the tail of a list. 15d⟩
}
```

Defines:
  `builtin_tail`, used in chunk 15c.
Uses `lval` 21c.

4e ⟨*lispy.c* 2a⟩+≡

```
lval *builtin_join(lval *args)
{
    ⟨Return the concatenation of lists. 15f⟩
}
```

Defines:
  `builtin_join`, used in chunk 15e.
Uses `lval` 21c.

Forward declare `lval_eval`, since it's used by `builtin_eval` and mutually recursive with `lval_eval_sexpr`.

5a ⟨*lispy.c* 2a⟩+≡
```
lval *lval_eval(lval* val);
```

Uses `lval` 21c.

5b ⟨*lispy.c* 2a⟩+≡
```
lval *builtin_eval(lval *args)
{
    ⟨Evaluate a Q-expression. 16d⟩
}
```

Defines:
`builtin_val`, never used.
Uses `lval` 21c.

5c ⟨*lispy.c* 2a⟩+≡
```
lval *builtin_op(char *op, lval *args)
{
  ⟨Eval(uate) a built-in operation. 11g⟩
}
```

Defines:
`builtin_binop`, never used.
Uses `lval` 21c.

5d ⟨*lispy.c* 2a⟩+≡
```
lval *builtin(char *fname, lval *args)
{
    ⟨Evaluate a built-in function or operation. 14b⟩
}
```

Defines:
`builtin`, used in chunk 18b.
Uses `lval` 21c.

5e ⟨*lispy.c* 2a⟩+≡
```
lval* lval_eval_sexpr(lval *args)
{
    ⟨Evaluate an S-expression. 17d⟩
}
```

Uses `lval` 21c.

6a ⟨*lispy.c* 2a⟩+≡

```
lval* lval_eval(lval* val)
{
    ⟨Evaluate an expression. 18c⟩
}
```

Uses `lval` 21c.

6b ⟨*lispy.c* 2a⟩+≡

```
lval *lval_read_num(mpc_ast_t *ast)
{
    ⟨Read a number. 10a⟩
}


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

Defines:
`lval_read`, used in chunks 9d and 10g.
Uses `ast` 9d, `lval` 21c, and `mpc_ast_t` 26f.

6c ⟨*lispy.c* 2a⟩+≡

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

    ⟨Print version and exit information. 7a⟩

    ⟨Loop until the input is empty. 20c⟩

    ⟨Undefine and delete the parsers. 8c⟩

    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.

7a ⟨*Print version and exit information.* 7a⟩≡
```
puts("Lispy v1.4.1");
puts("Press ctrl-c to exit\n");
```
Uses Lispy 7d.
This code is used in chunk 6c.

Defining the Language

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

7b ⟨*lispy.mpc* 7b⟩≡
```
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

7c ⟨*Load the Lispy grammar.* 7c⟩≡
```
static const char LISPY_GRAMMAR[] = {
#include "lispy.xxd"
};
```
Defines:
LISPY_GRAMMAR, used in chunk 8b.
This code is used in chunk 2a.

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

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

7d ⟨*Define the language.* 7d⟩≡
```
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 8a.
Lispy, used in chunks 7–9.
Number, used in chunk 8a.
Qexpr, used in chunk 8a.
Sexpr, used in chunk 8a.
Symbol, used in chunk 8a.
Uses mpc_new 26f and mpc_parser_t 26f.
This definition is continued in chunk 8b.
This code is used in chunk 6c.

Finally, using the defined *grammar* and each of the ⟨*created parsers* 8a⟩,

8a ⟨*created parsers* 8a⟩≡
```
Number, Symbol, Sexpr, Qexpr, Expr, Lispy
```
Uses Expr 7d, Lispy 7d, Number 7d, Qexpr 7d, Sexpr 7d, and Symbol 7d.
This code is used in chunk 8.

... we can define the Lispy language.

8b ⟨*Define the language.* 7d⟩+≡
```
mpc_err_t *err = mpca_lang(MPCA_LANG_PREDICTIVE, LISPY_GRAMMAR,
                           ⟨created parsers 8a⟩);

if (err != NULL) {
    puts(LISPY_GRAMMAR);
    mpc_err_print(err);
    mpc_err_delete(err);
    exit(100);
}
```
Uses LISPY_GRAMMAR 7c, mpca_lang 26f, mpc_err_delete 26f, and mpc_err_print 26f.

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

8c ⟨*Undefine and delete the parsers.* 8c⟩≡
```
mpc_cleanup(6, ⟨created parsers 8a⟩);
```
Uses mpc_cleanup 26f.
This code is used in chunk 6c.

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

8d ⟨*Read a line of user input.* 8d⟩≡
```
char *input = readline("> ");
```
Defines:
input, used in chunks 8, 9, and 21b.
Uses readline 26e.
This code is used in chunk 20d.

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

8e ⟨input *is nonempty* 8e⟩≡
```
input && *input
```
Uses input 8d.
This code is used in chunk 21a.

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

9a ⟨*Add* `input` *to the history table.* 9a⟩≡
```
add_history(input);
```
Uses `add_history` 26e and `input` 8d.
This code is used in chunk 21a.

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

9b ⟨*Declare a variable to hold parsing results.* 9b⟩≡
```
mpc_result_t parsed;
```
Defines:
`parsed`, used in chunks 9 and 20b.
Uses `mpc_result_t` 26f.
This code is used in chunk 21a.

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

9c ⟨*the input can be parsed as Lispy code* 9c⟩≡
```
mpc_parse("<stdin>", input, Lispy, &parsed)
```
Uses `Lispy` 7d, `input` 8d, `mpc_parse` 26f, and `parsed` 9b.
This code is used in chunk 21a.

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.

9d ⟨*Eval(uate) the input.* 9d⟩≡
```
mpc_ast_t *ast = parsed.output;

lval *result = lval_eval(lval_read(ast));
```
Defines:
`ast`, used in chunks 6b, 9, 10, and 19d.
Uses `lval` 21c, `lval_read` 6b, `mpc_ast_t` 26f, and `parsed` 9b.
This code is used in chunk 21a.

Describe the evaluation strategy

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

9e ⟨*Read a Lispy value.* 9e⟩≡
```
if (strstr(ast->tag, "number"))
    return lval_read_num(ast);
```
Uses `ast` 9d and `strstr` 26d.
This definition is continued in chunks 10 and 11c.
This code is used in chunk 6b.

Describe this

10a ⟨*Read a number.* 10a⟩≡
```
errno = 0;
double num = strtod(ast->contents, NULL);
return errno != ERANGE ? lval_num(num) : lval_err(LERR_BAD_NUM);
```
Uses ast 9d, lval_err 23d, lval_num 22b, and strtod 26b.
This code is used in chunk 6b.

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

10b ⟨*Read a Lispy value.* 9e⟩+≡
```
if (strstr(ast->tag, "symbol"))
    return lval_sym(ast->contents);
```
Uses ast 9d, lval_sym 24a, and strstr 26d.

Describe this

10c ⟨*Read a Lispy value.* 9e⟩+≡
```
lval *val = NULL;
```
Uses lval 21c.

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

10d ⟨*Read a Lispy value.* 9e⟩+≡
```
if (!strcmp(ast->tag, ">"))
    val = lval_sexpr();
```
Uses ast 9d, lval_sexpr 24b, and strcmp 26d.

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

10e ⟨*Read a Lispy value.* 9e⟩+≡
```
if (strstr(ast->tag, "qexpr"))
    val = lval_qexpr();
```
Uses ast 9d, lval_qexpr 24c, and strstr 26d.

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

10f ⟨*Read a Lispy value.* 9e⟩+≡
```
if (strstr(ast->tag, "sexpr"))
    val = lval_sexpr();
```
Uses ast 9d, lval_sexpr 24b, and strstr 26d.

Describe this

10g ⟨*Read a Lispy value.* 9e⟩+≡
```
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 9d, lval_add 2b, lval_read 6b, and strcmp 26d.

11a ⟨*Reallocate the memory used.* 11a⟩≡
```
xs->cell = realloc(xs->cell, sizeof(lval *) * xs->count);
```
Uses lval 21c.
This code is used in chunks 11b and 19a.

Describe this, incl. how it's not cons

11b ⟨*Add an element to an S-expression.* 11b⟩≡
```
xs->count++;
⟨Reallocate the memory used. 11a⟩
xs->cell[xs->count - 1] = x;
```
This code is used in chunk 2b.

Finally, return the Lispy value.

11c ⟨*Read a Lispy value.* 9e⟩+≡
```
return val;
```

11d ⟨*For each argument* 11d⟩≡
```
for (int i = 0; i < args->count; i++)
```
This code is used in chunks 11g, 15f, and 17e.

11e ⟨*the argument is not a number* 11e⟩≡
```
!lval_is_num(args->cell[i])
```
Uses lval_is_num 22c.
This code is used in chunk 11g.

11f ⟨*Delete the arguments and return a bad number error.* 11f⟩≡
```
lval_del(args);
return lval_err(LERR_BAD_NUM);
```
Uses lval_del 25a and lval_err 23d.
This code is used in chunk 11g.

Evaluating built-in operations

Ensure all arguments are numbers.

11g ⟨*Eval(uate) a built-in operation.* 11g⟩≡
```
⟨For each argument 11d⟩ {
    if (⟨the argument is not a number 11e⟩) {
        ⟨Delete the arguments and return a bad number error. 11f⟩
    }
}
```
This definition is continued in chunks 12 and 14a.
This code is used in chunk 5c.

11h ⟨*Pop the first element.* 11h⟩≡
```
lval_pop(args, 0);
```
Uses lval_pop 2c.
This code is used in chunks 12 and 18a.

Pop the first element.

12a ⟨*Eval(uate) a built-in operation.* 11g⟩+≡
```
lval *result = ⟨Pop the first element. 11h⟩
```

Uses lval 21c.

If the operation is unary subtraction, negate the operand.

12b ⟨*Eval(uate) a built-in operation.* 11g⟩+≡
```
if (!strcmp(op, "-") && !args->count)
    result->num = -result->num;
```

Uses strcmp 26d.

12c ⟨*Pop the next element.* 12c⟩≡
```
lval *y = ⟨Pop the first element. 11h⟩
```
Uses lval 21c.
This code is used in chunk 12d.

12d ⟨*Eval(uate) a built-in operation.* 11g⟩+≡
```
while (args->count > 0) {
    ⟨Pop the next element. 12c⟩

    ⟨Perform a built-in operation. 12e⟩
}
```

If the op is "+", perform addition.

12e ⟨*Perform a built-in operation.* 12e⟩≡
```
if (!strcmp(op, "+")) {
    result->num += y->num;
}
```
Uses strcmp 26d.
This definition is continued in chunks 12 and 13.
This code is used in chunk 12d.

If the op is "-", perform subtraction.

12f ⟨*Perform a built-in operation.* 12e⟩+≡
```
else if (!strcmp(op, "-")) {
    result->num -= y->num;
}
```
Uses strcmp 26d.

If the op is "*", perform multiplication.

12g ⟨*Perform a built-in operation.* 12e⟩+≡
```
else if (!strcmp(op, "*")) {
    result->num *= y->num;
}
```
Uses strcmp 26d.

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

13a ⟨*Perform a built-in operation.* 12e⟩+≡

```
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` 25a, `lval_err` 23d, and `strcmp` 26d.

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

13b ⟨*Perform a built-in operation.* 12e⟩+≡

```
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` 26c, `lval_del` 25a, `lval_err` 23d, and `strcmp` 26d.

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

13c ⟨*Perform a built-in operation.* 12e⟩+≡

```
else if (!strcmp(op, "^")) {
    result->num = pow(result->num, y->num);
}
```

Uses `pow` 26c and `strcmp` 26d.

Otherwise, return a `LERR_BAD_OP` error.

13d ⟨*Perform a built-in operation.* 12e⟩+≡

```
else {
    lval_del(result);
    lval_del(y);
    result = lval_err(LERR_BAD_OP);
    break;
}
```

Uses `lval_del` 25a and `lval_err` 23d.

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

13e ⟨*Perform a built-in operation.* 12e⟩+≡

```
lval_del(y);
```

Uses `lval_del` 25a.

Delete the input expression and return the result.

14a ⟨*Eval(uate) a built-in operation.* 11g⟩+≡
```
lval_del(args);

return result;
```
Uses lval_del 25a.

Built-in functions

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

14b ⟨*Evaluate a built-in function or operation.* 14b⟩≡
```
if (!strcmp("list", fname))
    return builtin_list(args);
```
Uses builtin_list 4b and strcmp 26d.
This definition is continued in chunks 14–17.
This code is used in chunk 5d.

14c ⟨*Convert an S-expression to a Q-expression.* 14c⟩≡
```
args->type = LVAL_QEXPR;
return args;
```
Uses LVAL_QEXPR 22a.
This code is used in chunk 4b.

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

14d ⟨*Evaluate a built-in function or operation.* 14b⟩+≡
```
if (!strcmp("head", fname))
    return builtin_head(args);
```
Uses builtin_head 4c and strcmp 26d.

Ensure there is exactly one argument.

14e ⟨*Pop the list and delete the rest.* 14e⟩≡
```
LVAL_ASSERT_NUM_ARGS("head", 1, args);
```
This definition is continued in chunks 14 and 15.
This code is used in chunk 4c.

Ensure the first argument is a Q-expression.

14f ⟨*Pop the list and delete the rest.* 14e⟩+≡
```
LVAL_ASSERT(args, args->cell[0]->type == LVAL_QEXPR,
    "invalid argument for 'head'");
```
Uses LVAL_QEXPR 22a.

Ensure the list passed to **head** is nonempty.

14g ⟨*Pop the list and delete the rest.* 14e⟩+≡
```
LVAL_ASSERT(args, args->cell[0]->count,
    "cannot get 'head' of the empty list");
```

Take the first element of the list.

14h ⟨*Pop the list and delete the rest.* 14e⟩+≡
```
lval *val = lval_take(args, 0);
```
Uses lval 21c and lval_take 3a.

Delete the rest.

15a ⟨*Pop the list and delete the rest.* 14e⟩+≡
```
while (val->count > 1)
    lval_del(lval_pop(val, 1));
```
Uses lval_del 25a and lval_pop 2c.

Return the head of the list.

15b ⟨*Pop the list and delete the rest.* 14e⟩+≡
```
return val;
```

If the function name is `tail`, return the given Q-expression with the first element removed.

15c ⟨*Evaluate a built-in function or operation.* 14b⟩+≡
```
if (!strcmp("tail", fname))
    return builtin_tail(args);
```
Uses builtin_tail 4d and strcmp 26d.

Split this up and describe

15d ⟨*Return the tail of a list.* 15d⟩≡
```
LVAL_ASSERT_NUM_ARGS("tail", 1, args);
LVAL_ASSERT(args, args->cell[0]->type == LVAL_QEXPR,
    "invalid argument for 'tail'");
LVAL_ASSERT(args, args->cell[0]->count,
    "cannot get 'tail' of the empty list");

lval *val = lval_take(args, 0);
lval_del(lval_pop(val, 0));

return val;
```
Uses LVAL_QEXPR 22a, lval 21c, lval_del 25a, lval_pop 2c, and lval_take 3a.
This code is used in chunk 4d.

If the function name is `join`, concatenate the given Q-expressions.

15e ⟨*Evaluate a built-in function or operation.* 14b⟩+≡
```
if (!strcmp("join", fname))
    return builtin_join(args);
```
Uses builtin_join 4e and strcmp 26d.

Ensure every argument is a Q-expression.

15f ⟨*Return the concatenation of lists.* 15f⟩≡
```
⟨For each argument 11d⟩ {
    LVAL_ASSERT(args, args->cell[i]->type == LVAL_QEXPR,
                "invalid argument for 'join'");
}
```
Uses LVAL_QEXPR 22a.
This definition is continued in chunk 16a.
This code is used in chunk 4e.

Describe this

16a ⟨*Return the concatenation of lists.* 15f⟩+≡
```
lval *res = lval_pop(args, 0);

while (args->count) {
    res = lval_join(res, lval_pop(args, 0));
}

lval_del(args);

return res;
```
Uses lval 21c, lval_del 25a, lval_join 3b, and lval_pop 2c.

Describe this

16b ⟨*Add every y in ys to xs.* 16b⟩≡
```
while (ys->count) {
    xs = lval_add(xs, lval_pop(ys, 0));
}

lval_del(ys);

return xs;
```
Uses lval_add 2b, lval_del 25a, and lval_pop 2c.
This code is used in chunk 3b.

If the function name is `eval`, convert a given Q-expression to an S-expression, and evaluate it.

16c ⟨*Evaluate a built-in function or operation.* 14b⟩+≡
```
if (!strcmp("eval", fname))
    return builtin_eval(args);
```
Uses strcmp 26d.

Ensure exactly one Q-expression is passed to `eval`.

16d ⟨*Evaluate a Q-expression.* 16d⟩≡
```
LVAL_ASSERT(args, args->count == 1,
            "too many arguments for 'eval'");

LVAL_ASSERT(args, args->cell[0]->type == LVAL_QEXPR,
            "invalid argument for 'eval'");
```
Uses LVAL_QEXPR 22a.
This definition is continued in chunk 17a.
This code is used in chunk 5b.

Convert the Q-expression to an S-expression, by changing it's `type`, then evaluate and return it.

17a ⟨*Evaluate a Q-expression.* 16d⟩+≡
```
lval *expr = lval_take(args, 0);
expr->type = LVAL_SEXPR;

return lval_eval(expr);
```
Uses LVAL_SEXPR 22a, lval 21c, and lval_take 3a.

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

17b ⟨*Evaluate a built-in function or operation.* 14b⟩+≡
```
if (strstr("+-/*^%", fname))
    return builtin_op(fname, args);
```
Uses strstr 26d.

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

17c ⟨*Evaluate a built-in function or operation.* 14b⟩+≡
```
lval_del(args);

return lval_err(LERR_BAD_FUNC);
```
Uses lval_del 25a and lval_err 23d.

Evaluating (S)-expressions

If the expression is empty, return it;

17d ⟨*Evaluate an S-expression.* 17d⟩≡
```
if (!args->count)
    return args;
```
This definition is continued in chunks 17 and 18.
This code is used in chunk 5e.

17e ⟨*Evaluate an S-expression.* 17d⟩+≡
```
⟨For each argument 11d⟩ {
    args->cell[i] = lval_eval(args->cell[i]);
    if (args->cell[i]->type == LVAL_ERR)
        return lval_take(args, i);
}
```
Uses LVAL_ERR 22a 22a and lval_take 3a.

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

17f ⟨*Evaluate an S-expression.* 17d⟩+≡
```
if (args->count == 1)
    return lval_take(args, 0);
```
Uses lval_take 3a.

18a ⟨*Evaluate an S-expression.* 17d⟩+≡
```
lval *car = ⟨Pop the first element. 11h⟩;
if (car->type != LVAL_SYM) {
    lval_del(car);
    lval_del(args);

    return lval_err(LERR_BAD_SEXPR);
}
```
Uses LVAL_SYM 22a, lval 21c, lval_del 25a, and lval_err 23d.

18b ⟨*Evaluate an S-expression.* 17d⟩+≡
```
lval *result = builtin(car->sym, args);
lval_del(car);

return result;
```
Uses builtin 5d, lval 21c, and lval_del 25a.

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

18c ⟨*Evaluate an expression.* 18c⟩≡
```
if (val->type == LVAL_SEXPR)
    return lval_eval_sexpr(val);

return val;
```
Uses LVAL_SEXPR 22a.
This code is used in chunk 6a.

Extract the element at index `i`.

18d ⟨*Extract an element and shift the list.* 18d⟩≡
```
lval *elem = xs->cell[i];
```
Uses lval 21c.
This definition is continued in chunks 18 and 19a.
This code is used in chunk 2c.

Shift memory after the element at index `i`.

18e ⟨*Extract an element and shift the list.* 18d⟩+≡
```
memmove(&xs->cell[i], &xs->cell[i + 1],
    sizeof(lval *) * (xs->count - i - 1));
```
Uses lval 21c.

Decrease the count.

18f ⟨*Extract an element and shift the list.* 18d⟩+≡
```
xs->count--;
```

18g ⟨*Return the extracted element.* 18g⟩≡
```
return elem;
```
This code is used in chunk 19.

Reallocate the memory used and return the extracted element.

19a ⟨*Extract an element and shift the list.* 18d⟩+≡
```
⟨Reallocate the memory used. 11a⟩

⟨Return the extracted element. 18g⟩
```

Describe this

19b ⟨*Pop the list then delete it.* 19b⟩≡
```
lval *elem = lval_pop(xs, i);
lval_del(xs);
```

Uses `lval` 21c, `lval_del` 25a, and `lval_pop` 2c.
This definition is continued in chunk 19c.
This code is used in chunk 3a.

Return the extracted element.

19c ⟨*Pop the list then delete it.* 19b⟩+≡
```
⟨Return the extracted element. 18g⟩
```

P is for Print

Upon success, print the result and delete the AST.

19d ⟨*Print the result and delete the AST.* 19d⟩≡
```
lval_println(result);

mpc_ast_delete(ast);
```

Uses `ast` 9d, `lval_println` 4a, and `mpc_ast_delete` 26f.
This code is used in chunk 21a.

Describe this

Print the opening character.

19e ⟨*Print an expression.* 19e⟩≡
```
putchar(open);
```

This definition is continued in chunk 19.
This code is used in chunk 3d.

Print all but the last element with a trailing space.

19f ⟨*Print an expression.* 19e⟩+≡
```
for (int i = 0; i < expr->count; i++) {
    lval_print(expr->cell[i]);
    if (i != (expr->count - 1))
        putchar(' ');
}
```

Uses `lval_print` 3e.

Print the closing character.

19g ⟨*Print an expression.* 19e⟩+≡
```
putchar(close);
```

20a ⟨*Print a Lispy value.* 20a⟩≡

```
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 22a 22a, LVAL_NUM 22a, LVAL_QEXPR 22a, LVAL_SEXPR 22a, LVAL_SYM 22a, lval_expr_print 3d, and printf 26a.
This code is used in chunk 3e.

Print and delete the error upon failure.

20b ⟨*Print and delete the error.* 20b⟩≡

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

Uses mpc_err_delete 26f, mpc_err_print 26f, and parsed 9b.
This code is used in chunk 21a.

L is for Loop

20c ⟨*Loop until the input is empty.* 20c⟩≡

```
bool nonempty;
do {
  ⟨Read, eval(uate), and print. 20d⟩
} while (nonempty);
```

Defines:
nonempty, used in chunk 21a.
Uses bool 25c.
This code is used in chunk 6c.

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

20d ⟨*Read, eval(uate), and print.* 20d⟩≡

```
⟨Read a line of user input. 8d⟩
```

This definition is continued in chunk 21.
This code is used in chunk 20c.

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

21a ⟨*Read, eval(uate), and print.* 20d⟩+≡
```
if ((nonempty = (⟨input is nonempty 8e⟩))) {
    ⟨Add input to the history table. 9a⟩

    ⟨Declare a variable to hold parsing results. 9b⟩
    if (⟨the input can be parsed as Lispy code 9c⟩) {
        ⟨Eval(uate) the input. 9d⟩
        ⟨Print the result and delete the AST. 19d⟩
    } else {
        ⟨Print and delete the error. 20b⟩
    }
}
```

Uses nonempty 20c.

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

21b ⟨*Read, eval(uate), and print.* 20d⟩+≡
```
free(input);
```
Uses free 26b and input 8d.

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

Error Handling

Describe this struct

21c ⟨*Define the Lispy data structures.* 21c⟩≡
```
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 2–6, 9–12, 14–19, and 22–25.
Uses lval_type_t 22a.
This definition is continued in chunks 22–25.
This code is used in chunk 2a.

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

22a ⟨*Define possible lval and error types.* 22a⟩≡
```
typedef enum {
    LVAL_ERR,
    LVAL_NUM,
    LVAL_QEXPR,
    LVAL_SEXPR,
    LVAL_SYM
} lval_type_t;
```

Defines:
  LVAL_ERR, used in chunks 17e, 20a, 23d, and 25a.
  LVAL_NUM, used in chunks 20a, 22, and 25a.
  LVAL_QEXPR, used in chunks 14–16, 20a, 24c, and 25a.
  LVAL_SEXPR, used in chunks 17a, 18c, 20a, 24b, and 25a.
  LVAL_SYM, used in chunks 18a, 20a, 24a, and 25a.
  lval_type_t, used in chunk 21c.
This code is used in chunk 2a.

Define a constructor for numbers.

22b ⟨*Define the Lispy data structures.* 21c⟩+≡
```
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 10a.
Uses LVAL_NUM 22a and lval 21c.

Define a convenient predicate for numbers.

22c ⟨*Define the Lispy data structures.* 21c⟩+≡
```
bool lval_is_num(lval *val)
{
    return val->type == LVAL_NUM;
}
```

Defines:
  lval_is_num, used in chunk 11e.
Uses LVAL_NUM 22a, bool 25c, and lval 21c.

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

23a ⟨*Define some useful macros.* 23a⟩≡

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

Uses lval_del 25a and lval_err 23d.
This definition is continued in chunk 23.
This code is used in chunk 2a.

23b ⟨*Define some useful macros.* 23a⟩+≡

```
#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"
```

23c ⟨*Define some useful macros.* 23a⟩+≡

```
#define LVAL_ERROR_MSG_NUM_ARGS(fname, prefix) ({ \
    char *_msg = malloc(strlen(prefix) + 17 + strlen(fname) + 2); \
    strcpy(_msg, prefix); \
    strcat(_msg, " arguments for '"); \
    strcat(_msg, fname); \
    strcat(_msg, "'"); \
    _msg; \
})


#define LVAL_ASSERT_NUM_ARGS(fname, num, args) ({ \
    LVAL_ASSERT(args, args->count ≥ num, \
                LVAL_ERROR_MSG_NUM_ARGS(fname, "too few")); \
    LVAL_ASSERT(args, args->count ≤ num, \
                LVAL_ERROR_MSG_NUM_ARGS(fname, "too many")); \
})
```

Define a constructor for errors.

23d ⟨*Define the Lispy data structures.* 21c⟩+≡

```
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 10a, 11f, 13, 17c, 18a, and 23a.
Uses LVAL_ERR 22a 22a and lval 21c.

Define a constructor for symbol.

24a ⟨*Define the Lispy data structures.* 21c⟩+≡

```
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 10b.
Uses LVAL_SYM 22a and lval 21c.

Define a constructor for an S-expression.

24b ⟨*Define the Lispy data structures.* 21c⟩+≡

```
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 10.
Uses LVAL_SEXPR 22a and lval 21c.

Define a constructor for a Q-expression.

24c ⟨*Define the Lispy data structures.* 21c⟩+≡

```
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 10e.
Uses LVAL_QEXPR 22a and lval 21c.

Define a destructor for `lval`*.

25a ⟨*Define the Lispy data structures.* 21c⟩+≡

```
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 11f, 13–19, and 23a.
Uses `LVAL_ERR` 22a 22a, `LVAL_NUM` 22a, `LVAL_QEXPR` 22a, `LVAL_SEXPR` 22a, `LVAL_SYM` 22a, `free` 26b, and `lval` 21c.

Headers

Describe headers

25b ⟨*Include the necessary headers.* 25b⟩≡

```
⟨Include the boolean type and values. 25c⟩
⟨Include the standard I/O functions. 26a⟩
⟨Include the standard library definitions. 26b⟩
⟨Include some mathematical definitions. 26c⟩
⟨Include some string operations. 26d⟩

⟨Include the line editing functions from libedit. 26e⟩
⟨Include the micro parser combinator definitions. 26f⟩
```

This code is used in chunk 2a.

25c ⟨*Include the boolean type and values.* 25c⟩≡

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

Defines:
`bool`, used in chunks 20c and 22c.
This code is used in chunk 25b.

26a ⟨*Include the standard I/O functions.* 26a⟩≡
```
#include <stdio.h>
```
Defines:
`printf`, used in chunk 20a.
This code is used in chunk 25b.

26b ⟨*Include the standard library definitions.* 26b⟩≡
```
#include <stdlib.h>
```
Defines:
`free`, used in chunks 21b and 25a.
`strtod`, used in chunk 10a.
This code is used in chunk 25b.

26c ⟨*Include some mathematical definitions.* 26c⟩≡
```
#include <math.h>
```
Defines:
`fmod`, used in chunk 13b.
`pow`, used in chunk 13c.
This code is used in chunk 25b.

26d ⟨*Include some string operations.* 26d⟩≡
```
#include <string.h>
```
Defines:
`strcmp`, used in chunks 10 and 12–16.
`strstr`, used in chunks 9, 10, and 17b.
This code is used in chunk 25b.

26e ⟨*Include the line editing functions from libedit.* 26e⟩≡
```
#include <editline/readline.h>
```
Defines:
`add_history`, used in chunk 9a.
`readline`, used in chunks 26e and 8d.
This code is used in chunk 25b.

26f ⟨*Include the micro parser combinator definitions.* 26f⟩≡
```
#include <mpc.h>
```
Defines:
`mpca_lang`, used in chunk 8b.
`mpc_ast_delete`, used in chunk 19d.
`mpc_ast_print`, never used.
`mpc_ast_t`, used in chunks 6b and 9d.
`mpc_cleanup`, used in chunks 26f and 8c.
`mpc_err_delete`, used in chunks 8b and 20b.
`mpc_err_print`, used in chunks 8b and 20b.
`mpc_new`, used in chunk 7d.
`mpc_parse`, used in chunks 26f and 9c.
`mpc_parser_t`, used in chunk 7d.
`mpc_result_t`, used in chunk 9b.
This code is used in chunk 25b.

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:

```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"

#define LVAL_ERROR_MSG_NUM_ARGS(fname, prefix) ({ \
    char *_msg = malloc(strlen(prefix) + 17 + strlen(fname) + 2); \
    strcpy(_msg, prefix); \
    strcat(_msg, " arguments for '"); \
    strcat(_msg, fname); \
    strcat(_msg, "'"); \
    _msg; \
})


#define LVAL_ASSERT_NUM_ARGS(fname, num, args) ({ \
    LVAL_ASSERT(args, args->count ≥ num, \
                LVAL_ERROR_MSG_NUM_ARGS(fname, "too few")); \
    LVAL_ASSERT(args, args->count ≤ num, \
                LVAL_ERROR_MSG_NUM_ARGS(fname, "too many")); \
})


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


typedef enum {
    LVAL_ERR,
    LVAL_NUM,
    LVAL_QEXPR,
    LVAL_SEXPR,
```

```
51      LVAL_SYM
52  } lval_type_t;
53
54
55
56  typedef struct lval {
57      lval_type_t type;
58      union {
59          double num;
60          char *err;
61          char *sym;
62      };
63      int count;
64      struct lval **cell;
65  } lval;
66
67
68  lval *lval_num(double num)
69  {
70      lval *val = malloc(sizeof(lval));
71      val->type = LVAL_NUM;
72      val->num = num;
73
74      return val;
75  }
76
77
78  bool lval_is_num(lval * val)
79  {
80      return val->type == LVAL_NUM;
81  }
82
83
84  lval *lval_err(char *err)
85  {
86      lval *val = malloc(sizeof(lval));
87      val->type = LVAL_ERR;
88      val->err = err;
89
90      return val;
91  }
92
93
94  lval *lval_sym(char *s)
95  {
96      lval *val = malloc(sizeof(lval));
97      val->type = LVAL_SYM;
98      val->sym = malloc(strlen(s) + 1);
99      strcpy(val->sym, s);
100
101     return val;
```

```
102 }
103
104
105 lval *lval_sexpr(void)
106 {
107     lval *val = malloc(sizeof(lval));
108     val->type = LVAL_SEXPR;
109     val->count = 0;
110     val->cell = NULL;
111
112     return val;
113 }
114
115
116 lval *lval_qexpr(void)
117 {
118     lval *val = malloc(sizeof(lval));
119     val->type = LVAL_QEXPR;
120     val->count = 0;
121     val->cell = NULL;
122
123     return val;
124 }
125
126
127 void lval_del(lval * val)
128 {
129     switch (val->type) {
130     case LVAL_ERR:
131         free(val->err);
132         break;
133     case LVAL_NUM:
134         break;
135     case LVAL_QEXPR:
136     case LVAL_SEXPR:
137         for (int i = 0; i < val->count; i++)
138             lval_del(val->cell[i]);
139         free(val->cell);
140         break;
141     case LVAL_SYM:
142         free(val->sym);
143         break;
144     }
145
146     free(val);
147 }
148
149
150 lval *lval_add(lval * xs, lval * x)
151 {
152     xs->count++;
```

```
153     xs->cell = realloc(xs->cell, sizeof(lval *) * xs->count);
154     xs->cell[xs->count - 1] = x;
155 
156     return xs;
157 }
158 
159 
160 lval *lval_pop(lval * xs, int i)
161 {
162     lval *elem = xs->cell[i];
163 
164     memmove(&xs->cell[i], &xs->cell[i + 1],
165             sizeof(lval *) * (xs->count - i - 1));
166 
167     xs->count--;
168 
169     xs->cell = realloc(xs->cell, sizeof(lval *) * xs->count);
170 
171     return elem;
172 }
173 
174 
175 lval *lval_take(lval * xs, int i)
176 {
177     lval *elem = lval_pop(xs, i);
178     lval_del(xs);
179 
180     return elem;
181 }
182 
183 
184 lval *lval_join(lval * xs, lval * ys)
185 {
186     while (ys->count) {
187         xs = lval_add(xs, lval_pop(ys, 0));
188     }
189 
190     lval_del(ys);
191 
192     return xs;
193 }
194 
195 
196 void lval_print(lval * val);
197 
198 
199 void lval_expr_print(lval * expr, char open, char close)
200 {
201     putchar(open);
202     for (int i = 0; i < expr->count; i++) {
203         lval_print(expr->cell[i]);
```

```
204         if (i != (expr->count - 1))
205             putchar(' ');
206     }
207     putchar(close);
208 }
209
210
211 void lval_print(lval * val)
212 {
213     switch (val->type) {
214     case LVAL_ERR:
215         printf("Error: %s", val->err);
216         break;
217     case LVAL_NUM:
218         printf("%g", val->num);
219         break;
220     case LVAL_QEXPR:
221         lval_expr_print(val, '{', '}');
222         break;
223     case LVAL_SEXPR:
224         lval_expr_print(val, '(', ')');
225         break;
226     case LVAL_SYM:
227         fputs(val->sym, stdout);
228         break;
229     }
230 }
231
232
233 void lval_println(lval * val)
234 {
235     lval_print(val);
236     putchar('\n');
237 }
238
239
240 lval *builtin_list(lval * args)
241 {
242     args->type = LVAL_QEXPR;
243     return args;
244 }
245
246
247 lval *builtin_head(lval * args)
248 {
249     LVAL_ASSERT_NUM_ARGS("head", 1, args);
250     LVAL_ASSERT(args, args->cell[0]->type == LVAL_QEXPR,
251                 "invalid argument for 'head'");
252     LVAL_ASSERT(args, args->cell[0]->count,
253                 "cannot get 'head' of the empty list");
254     lval *val = lval_take(args, 0);
```

```
255     while (val->count > 1)
256         lval_del(lval_pop(val, 1));
257     return val;
258 }
259
260
261 lval *builtin_tail(lval * args)
262 {
263     LVAL_ASSERT_NUM_ARGS("tail", 1, args);
264     LVAL_ASSERT(args, args->cell[0]->type == LVAL_QEXPR,
265                 "invalid argument for 'tail'");
266     LVAL_ASSERT(args, args->cell[0]->count,
267                 "cannot get 'tail' of the empty list");
268
269     lval *val = lval_take(args, 0);
270     lval_del(lval_pop(val, 0));
271
272     return val;
273 }
274
275
276 lval *builtin_join(lval * args)
277 {
278     for (int i = 0; i < args->count; i++) {
279         LVAL_ASSERT(args, args->cell[i]->type == LVAL_QEXPR,
280                     "invalid argument for 'join'");
281     }
282
283     lval *res = lval_pop(args, 0);
284
285     while (args->count) {
286         res = lval_join(res, lval_pop(args, 0));
287     }
288
289     lval_del(args);
290
291     return res;
292
293 }
294
295 lval *lval_eval(lval * val);
296
297
298 lval *builtin_eval(lval * args)
299 {
300     LVAL_ASSERT(args, args->count == 1, "too many arguments for 'eval'");
301
302     LVAL_ASSERT(args, args->cell[0]->type == LVAL_QEXPR,
303                 "invalid argument for 'eval'");
304
305     lval *expr = lval_take(args, 0);
```

```
306     expr->type = LVAL_SEXPR;
307
308     return lval_eval(expr);
309 }
310
311
312 lval *builtin_op(char *op, lval * args)
313 {
314     for (int i = 0; i < args->count; i++) {
315         if (!lval_is_num(args->cell[i])) {
316             lval_del(args);
317             return lval_err(LERR_BAD_NUM);
318         }
319     }
320
321     lval *result = lval_pop(args, 0);
322
323     if (!strcmp(op, "-") && !args->count)
324         result->num = -result->num;
325
326     while (args->count > 0) {
327         lval *y = lval_pop(args, 0);
328
329         if (!strcmp(op, "+")) {
330             result->num += y->num;
331         } else if (!strcmp(op, "-")) {
332             result->num -= y->num;
333         } else if (!strcmp(op, "*")) {
334             result->num *= y->num;
335         } else if (!strcmp(op, "/")) {
336             if (!y->num) {
337                 lval_del(result);
338                 lval_del(y);
339                 result = lval_err(LERR_DIV_ZERO);
340                 break;
341             }
342             result->num /= y->num;
343         } else if (!strcmp(op, "%")) {
344             if (!y->num) {
345                 lval_del(result);
346                 lval_del(y);
347                 result = lval_err(LERR_DIV_ZERO);
348                 break;
349             }
350             result->num = fmod(result->num, y->num);
351         } else if (!strcmp(op, "^")) {
352             result->num = pow(result->num, y->num);
353         } else {
354             lval_del(result);
355             lval_del(y);
356             result = lval_err(LERR_BAD_OP);
```

```
357             break;
358         }
359         lval_del(y);
360     }
361
362     lval_del(args);
363
364     return result;
365 }
366
367
368 lval *builtin(char *fname, lval * args)
369 {
370     if (!strcmp("list", fname))
371         return builtin_list(args);
372
373     if (!strcmp("head", fname))
374         return builtin_head(args);
375     if (!strcmp("tail", fname))
376         return builtin_tail(args);
377     if (!strcmp("join", fname))
378         return builtin_join(args);
379     if (!strcmp("eval", fname))
380         return builtin_eval(args);
381     if (strstr("+-/*^%", fname))
382         return builtin_op(fname, args);
383
384     lval_del(args);
385
386     return lval_err(LERR_BAD_FUNC);
387 }
388
389 lval *lval_eval_sexpr(lval * args)
390 {
391     if (!args->count)
392         return args;
393     for (int i = 0; i < args->count; i++) {
394         args->cell[i] = lval_eval(args->cell[i]);
395         if (args->cell[i]->type == LVAL_ERR)
396             return lval_take(args, i);
397     }
398
399     if (args->count == 1)
400         return lval_take(args, 0);
401
402     lval *car = lval_pop(args, 0);;
403     if (car->type != LVAL_SYM) {
404         lval_del(car);
405         lval_del(args);
406
407         return lval_err(LERR_BAD_SEXPR);
```

```
408     }
409
410     lval *result = builtin(car->sym, args);
411     lval_del(car);
412
413     return result;
414 }
415
416
417 lval *lval_eval(lval * val)
418 {
419     if (val->type == LVAL_SEXPR)
420         return lval_eval_sexpr(val);
421
422     return val;
423 }
424
425
426 lval *lval_read_num(mpc_ast_t * ast)
427 {
428     errno = 0;
429     double num = strtod(ast->contents, NULL);
430     return errno != ERANGE ? lval_num(num) : lval_err(LERR_BAD_NUM);
431 }
432
433
434 lval *lval_read(mpc_ast_t * ast)
435 {
436     if (strstr(ast->tag, "number"))
437         return lval_read_num(ast);
438
439     if (strstr(ast->tag, "symbol"))
440         return lval_sym(ast->contents);
441
442     lval *val = NULL;
443     if (!strcmp(ast->tag, ">"))
444         val = lval_sexpr();
445     if (strstr(ast->tag, "qexpr"))
446         val = lval_qexpr();
447     if (strstr(ast->tag, "sexpr"))
448         val = lval_sexpr();
449
450     for (int i = 0; i < ast->children_num; i++) {
451         if (!strcmp(ast->children[i]->contents, "("))
452             continue;
453         if (!strcmp(ast->children[i]->contents, ")"))
454             continue;
455         if (!strcmp(ast->children[i]->contents, "{"))
456             continue;
457         if (!strcmp(ast->children[i]->contents, "}"))
458             continue;
```

```
459         if (!strcmp(ast->children[i]->tag, "regex"))
460             continue;
461         val = lval_add(val, lval_read(ast->children[i]));
462     }
463
464     return val;
465 }
466
467
468 int main(int argc, char *argv[])
469 {
470     mpc_parser_t *Number = mpc_new("number");
471     mpc_parser_t *Symbol = mpc_new("symbol");
472     mpc_parser_t *Sexpr = mpc_new("sexpr");
473     mpc_parser_t *Qexpr = mpc_new("qexpr");
474     mpc_parser_t *Expr = mpc_new("expr");
475     mpc_parser_t *Lispy = mpc_new("lispy");
476
477     mpc_err_t *err = mpca_lang(MPCA_LANG_PREDICTIVE, LISPY_GRAMMAR,
478                                Number, Symbol, Sexpr, Qexpr, Expr, Lispy);
479
480     if (err != NULL) {
481         puts(LISPY_GRAMMAR);
482         mpc_err_print(err);
483         mpc_err_delete(err);
484         exit(100);
485     }
486
487     puts("Lispy v1.4.1");
488     puts("Press ctrl-c to exit\n");
489
490     bool nonempty;
491     do {
492         char *input = readline("> ");
493         if ((nonempty = (input && *input))) {
494             add_history(input);
495
496             mpc_result_t parsed;
497             if (mpc_parse("<stdin>", input, Lispy, &parsed)) {
498                 mpc_ast_t *ast = parsed.output;
499
500                 lval *result = lval_eval(lval_read(ast));
501                 lval_println(result);
502
503                 mpc_ast_delete(ast);
504             } else {
505                 mpc_err_print(parsed.error);
506                 mpc_err_delete(parsed.error);
507             }
508         }
509
```

```
510         free(input);
511     } while (nonempty);
512 
513     mpc_cleanup(6, Number, Symbol, Sexpr, Qexpr, Expr, Lispy);
514 
515     return 0;
516 }
```

Chunks

Index

Glossary

AST abstract syntax tree, a tree representation of the abstract syntactic structure of source code. 9, 10, 19

grammar 7, 8

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, 7, 8

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