

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 Noweb³.

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

<i>Outline</i>	2
<i>Welcome</i>	7
<i>Defining the Language</i>	7
<i>R is for Read</i>	8
<i>E is for Eval(uate)</i>	9
<i>Evaluating built-in operations</i>	11
<i>Built-in functions</i>	14
<i>Evaluating (S)-expressions</i>	17
<i>P is for Print</i>	19
<i>L is for Loop</i>	20
<i>Error Handling</i>	21
<i>Headers</i>	25
<i>Full Listings</i>	27
<i>Chunks</i>	38
<i>Index</i>	40
<i>Glossary</i>	42
<i>References</i>	43

Outline

Describe the outline

2a $\langle \text{lispy.c } 2a \rangle \equiv$
 \langle Include the necessary headers. 25b \rangle

\langle Define some useful macros. 23a \rangle

\langle Load the Lispy grammar. 7c \rangle

\langle Define possible lval and error types. 22a \rangle

\langle Define the Lispy data structures. 21c \rangle

This definition is continued in chunks 2–6.
 Root chunk (not used in this document).

2b $\langle \text{lispy.c } 2a \rangle + \equiv$
`lval *lval_add(lval *xs, lval *x)`
`{`
 \langle Add an element to an S-expression. 11b \rangle

 return xs;
`}`

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

2c $\langle \text{lispy.c } 2a \rangle + \equiv$
`lval *lval_pop(lval *xs, int i)`
`{`
 \langle Extract an element and shift the list. 18d \rangle
`}`

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

3a $\langle \text{lispy.c 2a} \rangle + \equiv$
`lval *lval_take(lval *xs, int i)`
`{`
 $\langle \text{Pop the list then delete it. 19b} \rangle$
`}`

Defines:

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

3b $\langle \text{lispy.c 2a} \rangle + \equiv$
`lval *lval_join(lval *xs, lval *ys)`
`{`
 $\langle \text{Add every } y \text{ in } ys \text{ to } xs. 16b \rangle$
`}`

Defines:

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

 Forward declare⁴ `lval_print`, since it's mutually recursive⁵ with `lval_expr_print`.

⁴ https://en.wikipedia.org/wiki/Forward_declaration

⁵ https://en.wikipedia.org/wiki/Mutual_recursion

3c $\langle \text{lispy.c 2a} \rangle + \equiv$
`void lval_print(lval *val);`

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

3d $\langle \text{lispy.c 2a} \rangle + \equiv$
`void lval_expr_print(lval *expr, char open, char close)`
`{`
 $\langle \text{Print an expression. 19e} \rangle$
`}`

Defines:

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

3e $\langle \text{lispy.c 2a} \rangle + \equiv$
`void lval_print(lval *val)`
`{`
 $\langle \text{Print a Lispy value. 20a} \rangle$
`}`

Defines:

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

4a $\langle \text{lisp.c } 2a \rangle + \equiv$

```
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 $\langle \text{lisp.c } 2a \rangle + \equiv$

```
lval *builtin_list(lval *args)
{
     $\langle \text{Convert an S-expression to a Q-expression. } 14c \rangle$ 
}
```

Defines:

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

4c $\langle \text{lisp.c } 2a \rangle + \equiv$

```
lval *builtin_head(lval *args)
{
     $\langle \text{Pop the list and delete the rest. } 14e \rangle$ 
}
```

Defines:

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

4d $\langle \text{lisp.c } 2a \rangle + \equiv$

```
lval *builtin_tail(lval *args)
{
     $\langle \text{Return the tail of a list. } 15d \rangle$ 
}
```

Defines:

builtin_tail, used in chunk 15c.
 Uses lval 21c.

4e $\langle \text{lisp.c } 2a \rangle + \equiv$

```
lval *builtin_join(lval *args)
{
     $\langle \text{Return the concatenation of lists. } 15f \rangle$ 
}
```

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 *⟨lisp.c 2a⟩* $\vdash \equiv$
`lval *lval_eval(lval* val);`

Uses `lval 21c`.

5b *⟨lisp.c 2a⟩* $\vdash \equiv$
`lval *builtin_eval(lval *args)`
`{`
 ⟨Evaluate a Q-expression. 16d⟩
`}`

Defines:

`builtin_val`, never used.

Uses `lval 21c`.

5c *⟨lisp.c 2a⟩* $\vdash \equiv$
`lval *builtin_op(char *op, lval *args)`
`{`
 ⟨Evaluate a built-in operation. 11g⟩
`}`

Defines:

`builtin_binop`, never used.

Uses `lval 21c`.

5d *⟨lisp.c 2a⟩* $\vdash \equiv$
`lval *builtin(char *fname, lval *args)`
`{`
 ⟨Evaluate a built-in function or operation. 14b⟩
`}`

Defines:

`builtin`, used in chunk 18b.

Uses `lval 21c`.

5e *⟨lisp.c 2a⟩* $\vdash \equiv$
`lval* lval_eval_sexpr(lval *args)`
`{`
 ⟨Evaluate an S-expression. 17d⟩
`}`

Uses `lval 21c`.

6a $\langle \text{lispy.c 2a} \rangle + \equiv$

```

lval* lval_eval(lval* val)
{
     $\langle \text{Evaluate an expression. 18c} \rangle$ 
}

```

Uses lval 21c.

6b $\langle \text{lispy.c 2a} \rangle + \equiv$

```

lval *lval_read_num(mpc_ast_t *ast)
{
     $\langle \text{Read a number. 10a} \rangle$ 
}

```

```

lval *lval_read(mpc_ast_t *ast)
{
     $\langle \text{Read a Lispy value. 9e} \rangle$ 
}

```

Defines:

lval_read, used in chunks 9d and 10g.

Uses ast 9d, lval 21c, and mpc_ast_t 26f.

6c $\langle \text{lispy.c 2a} \rangle + \equiv$

```

int main(int argc, char *argv[])
{
     $\langle \text{Define the language. 7d} \rangle$ 

     $\langle \text{Print version and exit information. 7a} \rangle$ 

     $\langle \text{Loop until the input is empty. 20c} \rangle$ 

     $\langle \text{Undefine and delete the parsers. 8c} \rangle$ 

    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 v1.4.0");`
`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 = mpc_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**, mpc_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.

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

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

R is for Read

To implement the R in **REPL**, use `readline` from `libedit`⁷.

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

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

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 != 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

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 $\langle \text{Read a number. 10a} \rangle \equiv$
 errno = 0;
 double num = strtod(ast->contents, NULL);
 return errno \neq ERANGE ? lval_num(num) : lval_err(LERR_BAD_NUM);

Uses ast 9d, lval_err 23c, 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 $\langle \text{Read a Lispy value. 9e} \rangle + \equiv$
 if (strstr(ast->tag, "symbol"))
 return lval_sym(ast->contents);

Uses ast 9d, lval_sym 23d, and strstr 26d.

Describe this

10c $\langle \text{Read a Lispy value. 9e} \rangle + \equiv$
 lval *val = NULL;

Uses lval 21c.

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

10d $\langle \text{Read a Lispy value. 9e} \rangle + \equiv$
 if (!strcmp(ast->tag, ">"))
 val = lval_sexpr();

Uses ast 9d, lval_sexpr 24a, and strcmp 26d.

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

10e $\langle \text{Read a Lispy value. 9e} \rangle + \equiv$
 if (strstr(ast->tag, "qexpr"))
 val = lval_qexpr();

Uses ast 9d, lval_qexpr 24b, and strstr 26d.

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

10f $\langle \text{Read a Lispy value. 9e} \rangle + \equiv$
 if (strstr(ast->tag, "sexpr"))
 val = lval_sexpr();

Uses ast 9d, lval_sexpr 24a, and strstr 26d.

Describe this

10g $\langle \text{Read a Lispy value. 9e} \rangle + \equiv$
 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 $\langle \text{Reallocate the memory used. 11a} \rangle \equiv$
`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 $\langle \text{Add an element to an S-expression. 11b} \rangle \equiv$
`xs->count++;`
 $\langle \text{Reallocate the memory used. 11a} \rangle$
`xs->cell[xs->count - 1] = x;`

This code is used in chunk 2b.

Finally, return the Lispy value.

11c $\langle \text{Read a Lispy value. 9e} \rangle + \equiv$
`return val;`

11d $\langle \text{For each argument 11d} \rangle \equiv$
`for (int i = 0; i < args->count; i++)`

This code is used in chunks 11g, 15f, and 17e.

11e $\langle \text{the argument is not a number 11e} \rangle \equiv$
`!lval_is_num(args->cell[i])`

Uses `lval_is_num 22c`.

This code is used in chunk 11g.

11f $\langle \text{Delete the arguments and return a bad number error. 11f} \rangle \equiv$
`lval_del(args);`
`return lval_err(LERR_BAD_NUM);`

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

This code is used in chunk 11g.

Evaluating built-in operations

Ensure all arguments are numbers.

11g $\langle \text{Eval(uate) a built-in operation. 11g} \rangle \equiv$
 $\langle \text{For each argument 11d} \rangle \{$
 $\quad \text{if } ((\text{the argument is not a number 11e})) \{$
 $\quad \quad \langle \text{Delete the arguments and return a bad number error. 11f} \rangle$
 $\quad \quad \}$
 $\quad \}$
 $\}$

This definition is continued in chunks 12 and 14a.

This code is used in chunk 5c.

11h $\langle \text{Pop the first element. 11h} \rangle \equiv$
`lval_pop(args, 0);`

Uses `lval_pop 2c`.

This code is used in chunks 12 and 18a.

Pop the first element.

12a $\langle \text{Eval}(\text{uate}) \text{ a built-in operation. 11g} \rangle + \equiv$
`lval *result = $\langle \text{Pop the first element. 11h} \rangle$`

Uses lval 21c.

If the operation is unary subtraction, negate the operand.

12b $\langle \text{Eval}(\text{uate}) \text{ a built-in operation. 11g} \rangle + \equiv$
`if (!strcmp(op, "-") && !args->count)
 result->num = -result->num;`

Uses strcmp 26d.

12c $\langle \text{Pop the next element. 12c} \rangle \equiv$
`lval *y = $\langle \text{Pop the first element. 11h} \rangle$`

Uses lval 21c.

This code is used in chunk 12d.

12d $\langle \text{Eval}(\text{uate}) \text{ a built-in operation. 11g} \rangle + \equiv$
`while (args->count > 0) {
 $\langle \text{Pop the next element. 12c} \rangle$

 $\langle \text{Perform a built-in operation. 12e} \rangle$
}`

If the op is "+", perform addition.

12e $\langle \text{Perform a built-in operation. 12e} \rangle \equiv$
`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 $\langle \text{Perform a built-in operation. 12e} \rangle + \equiv$
`else if (!strcmp(op, "-")) {
 result->num -= y->num;
}`

Uses strcmp 26d.

If the op is "*", perform multiplication.

12g $\langle \text{Perform a built-in operation. 12e} \rangle + \equiv$
`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` 23c, 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` 23c, 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` 23c.

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 $\langle \text{Evaluate a built-in operation. 11g} \rangle + \equiv$
`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 $\langle \text{Evaluate a built-in function or operation. 14b} \rangle \equiv$
`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 $\langle \text{Convert an S-expression to a Q-expression. 14c} \rangle \equiv$
`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 $\langle \text{Evaluate a built-in function or operation. 14b} \rangle + \equiv$
`if (!strcmp("head", fname))`
`return builtin_head(args);`

Uses `builtin_head` 4c and `strcmp` 26d.

Ensure there is exactly one argument.

14e $\langle \text{Pop the list and delete the rest. 14e} \rangle \equiv$
`LVAL_ASSERT(args, args->count == 1,`
`"too many arguments for 'head'");`

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 $\langle \text{Pop the list and delete the rest. 14e} \rangle + \equiv$
`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 $\langle \text{Pop the list and delete the rest. 14e} \rangle + \equiv$
`LVAL_ASSERT(args, args->cell[0]->count,`
`"cannot get 'head' of the empty list");`

Take the first element of the list.

14h $\langle \text{Pop the list and delete the rest. 14e} \rangle + \equiv$
`lval *val = lval_take(args, 0);`

Uses `lval` 21c and `lval_take` 3a.

Delete the rest.

15a $\langle \text{Pop the list and delete the rest. 14e} \rangle + \equiv$
`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 $\langle \text{Pop the list and delete the rest. 14e} \rangle + \equiv$
`return val;`

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

15c $\langle \text{Evaluate a built-in function or operation. 14b} \rangle + \equiv$
`if (!strcmp("tail", fname))
 return builtin_tail(args);`

Uses `builtin_tail` 4d and `strcmp` 26d.

Split this up and describe

15d $\langle \text{Return the tail of a list. 15d} \rangle \equiv$
`LVAL_ASSERT(args, args->count == 1,
 "too many arguments for 'tail'");
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 $\langle \text{Evaluate a built-in function or operation. 14b} \rangle + \equiv$
`if (!strcmp("join", fname))
 return builtin_join(args);`

Uses `builtin_join` 4e and `strcmp` 26d.

Ensure every argument is a Q-expression.

15f $\langle \text{Return the concatenation of lists. 15f} \rangle \equiv$
 $\langle \text{For each argument 11d} \rangle \{$
`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 its **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 23c.

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 $\langle \text{Evaluate an S-expression. 17d} \rangle + \equiv$

```

lval *car =  $\langle \text{Pop the first element. 11h} \rangle$ ;
if (car→type  $\neq$  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 23c.

18b $\langle \text{Evaluate an S-expression. 17d} \rangle + \equiv$

```

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 $\langle \text{Evaluate an expression. 18c} \rangle \equiv$

```

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 $\langle \text{Extract an element and shift the list. 18d} \rangle \equiv$

```

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 $\langle \text{Extract an element and shift the list. 18d} \rangle + \equiv$

```

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

```

Uses lval 21c.

Decrease the count.

18f $\langle \text{Extract an element and shift the list. 18d} \rangle + \equiv$

```

xs→count--;

```

18g $\langle \text{Return the extracted element. 18g} \rangle \equiv$

```

return elem;

```

This code is used in chunk 19.

Reallocate the memory used and return the extracted element.

19a $\langle \text{Extract an element and shift the list. 18d} \rangle + \equiv$
 $\langle \text{Reallocate the memory used. 11a} \rangle$

$\langle \text{Return the extracted element. 18g} \rangle$

Describe this

19b $\langle \text{Pop the list then delete it. 19b} \rangle \equiv$
`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 $\langle \text{Pop the list then delete it. 19b} \rangle + \equiv$
 $\langle \text{Return the extracted element. 18g} \rangle$

P is for Print

Upon success, print the result and delete the `AST`.

19d $\langle \text{Print the result and delete the AST. 19d} \rangle \equiv$
`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 $\langle \text{Print an expression. 19e} \rangle \equiv$
`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 $\langle \text{Print an expression. 19e} \rangle + \equiv$

```
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 $\langle \text{Print an expression. 19e} \rangle + \equiv$
`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, `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, **Read** 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, **Eval**(uate) it, and **Print** 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 further 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**, **23c**, and **25a**.

LVAL_NUM, used in chunks **20a**, **22**, and **25a**.

LVAL_QEXPR, used in chunks **14–16**, **20a**, **24b**, and **25a**.

LVAL_SEXPR, used in chunks **17a**, **18c**, **20a**, **24a**, and **25a**.

LVAL_SYM, used in chunks **18a**, **20a**, **23d**, 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` 23c.

This definition is continued in chunk 23b.

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_SEXP "invalid S-expression"
```

Define a constructor for errors.

```
23c <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.

```
23d <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.

24a *<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.

24b *<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:

```

1  #include <stdbool.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <math.h>
5  #include <string.h>
6
7  #include <editline/readline.h>
8  #include <mpc.h>
9
10
11 #define LVAL_ASSERT(args, cond, err) \
12     if (!(cond)) { \
13         lval_del(args); \
14         return lval_err(err); \
15     }
16
17 #define LERR_BAD_FUNC "unknown function"
18 #define LERR_BAD_NUM "invalid number"
19 #define LERR_BAD_OP "invalid operation"
20 #define LERR_DIV_ZERO "division by zero"
21 #define LERR_BAD_SEXPR "invalid S-expression"
22
23
24 static const char LISPY_GRAMMAR[] = {
25     #include "lispy.xxd"
26 };
27
28
29 typedef enum {
30     LVAL_ERR,
31     LVAL_NUM,
32     LVAL_QEXPR,
33     LVAL_SEXPR,
34     LVAL_SYM
35 } lval_type_t;
36
37
38
39 typedef struct lval {
40     lval_type_t type;
41     union {
42         double num;
43         char *err;
44         char *sym;
45     };
46     int count;
47     struct lval **cell;
48 } lval;
49
50

```

```

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

```

```

102     val->type = LVAL_QEXPR;
103     val->count = 0;
104     val->cell = NULL;
105
106     return val;
107 }
108
109
110 void lval_del(lval * val)
111 {
112     switch (val->type) {
113     case LVAL_ERR:
114         free(val->err);
115         break;
116     case LVAL_NUM:
117         break;
118     case LVAL_QEXPR:
119     case LVAL_SEXPR:
120         for (int i = 0; i < val->count; i++)
121             lval_del(val->cell[i]);
122         free(val->cell);
123         break;
124     case LVAL_SYM:
125         free(val->sym);
126         break;
127     }
128
129     free(val);
130 }
131
132
133 lval *lval_add(lval * xs, lval * x)
134 {
135     xs->count++;
136     xs->cell = realloc(xs->cell, sizeof(lval *) * xs->count);
137     xs->cell[xs->count - 1] = x;
138
139     return xs;
140 }
141
142
143 lval *lval_pop(lval * xs, int i)
144 {
145     lval *elem = xs->cell[i];
146
147     memmove(&xs->cell[i], &xs->cell[i + 1],
148             sizeof(lval *) * (xs->count - i - 1));
149
150     xs->count--;
151
152     xs->cell = realloc(xs->cell, sizeof(lval *) * xs->count);

```

```

153
154     return elem;
155 }
156
157
158 lval *lval_take(lval * xs, int i)
159 {
160     lval *elem = lval_pop(xs, i);
161     lval_del(xs);
162
163     return elem;
164 }
165
166
167 lval *lval_join(lval * xs, lval * ys)
168 {
169     while (ys->count) {
170         xs = lval_add(xs, lval_pop(ys, 0));
171     }
172
173     lval_del(ys);
174
175     return xs;
176 }
177
178
179 void lval_print(lval * val);
180
181
182 void lval_expr_print(lval * expr, char open, char close)
183 {
184     putchar(open);
185     for (int i = 0; i < expr->count; i++) {
186         lval_print(expr->cell[i]);
187         if (i != (expr->count - 1))
188             putchar(' ');
189     }
190     putchar(close);
191 }
192
193
194 void lval_print(lval * val)
195 {
196     switch (val->type) {
197     case LVAL_ERR:
198         printf("Error: %s", val->err);
199         break;
200     case LVAL_NUM:
201         printf("%g", val->num);
202         break;
203     case LVAL_QEXPR:

```

```

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

```



```

255     return val;
256 }
257
258
259 lval *builtin_join(lval * args)
260 {
261     for (int i = 0; i < args->count; i++) {
262         LVAL_ASSERT(args, args->cell[i]->type == LVAL_QEXPR,
263             "invalid argument for 'join'");
264     }
265
266     lval *res = lval_pop(args, 0);
267
268     while (args->count) {
269         res = lval_join(res, lval_pop(args, 0));
270     }
271
272     lval_del(args);
273
274     return res;
275 }
276
277 lval *lval_eval(lval * val);
278
279
280
281 lval *builtin_eval(lval * args)
282 {
283     LVAL_ASSERT(args, args->count == 1, "too many arguments for 'eval'");
284
285     LVAL_ASSERT(args, args->cell[0]->type == LVAL_QEXPR,
286         "invalid argument for 'eval'");
287
288     lval *expr = lval_take(args, 0);
289     expr->type = LVAL_SEXPR;
290
291     return lval_eval(expr);
292 }
293
294
295 lval *builtin_op(char *op, lval * args)
296 {
297     for (int i = 0; i < args->count; i++) {
298         if (!lval_is_num(args->cell[i])) {
299             lval_del(args);
300             return lval_err(LERR_BAD_NUM);
301         }
302     }
303
304     lval *result = lval_pop(args, 0);
305

```

```

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

```

```

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

```

```

408
409 lval *lval_read_num(mpc_ast_t * ast)
410 {
411     errno = 0;
412     double num = strtod(ast->contents, NULL);
413     return errno != ERANGE ? lval_num(num) : lval_err(LERR_BAD_NUM);
414 }
415
416
417 lval *lval_read(mpc_ast_t * ast)
418 {
419     if (strstr(ast->tag, "number"))
420         return lval_read_num(ast);
421
422     if (strstr(ast->tag, "symbol"))
423         return lval_sym(ast->contents);
424
425     lval *val = NULL;
426     if (!strcmp(ast->tag, ">"))
427         val = lval_sexpr();
428     if (strstr(ast->tag, "qexpr"))
429         val = lval_qexpr();
430     if (strstr(ast->tag, "sexpr"))
431         val = lval_sexpr();
432
433     for (int i = 0; i < ast->children_num; i++) {
434         if (!strcmp(ast->children[i]->contents, "("))
435             continue;
436         if (!strcmp(ast->children[i]->contents, ")"))
437             continue;
438         if (!strcmp(ast->children[i]->contents, "{"))
439             continue;
440         if (!strcmp(ast->children[i]->contents, "}")
441             continue;
442         if (!strcmp(ast->children[i]->tag, "regex"))
443             continue;
444         val = lval_add(val, lval_read(ast->children[i]));
445     }
446
447     return val;
448 }
449
450
451 int main(int argc, char *argv[])
452 {
453     mpc_parser_t *Number = mpc_new("number");
454     mpc_parser_t *Symbol = mpc_new("symbol");
455     mpc_parser_t *Sexpr = mpc_new("sexpr");
456     mpc_parser_t *Qexpr = mpc_new("qexpr");
457     mpc_parser_t *Expr = mpc_new("expr");
458     mpc_parser_t *Lispy = mpc_new("lispy");

```

```

459
460 mpc_err_t *err = mpca_lang(MPCA_LANG_PREDICTIVE, LISPY_GRAMMAR,
461                             Number, Symbol, Sexpr, Qexpr, Expr, Lispy);
462
463 if (err != NULL) {
464     puts(LISPY_GRAMMAR);
465     mpc_err_print(err);
466     mpc_err_delete(err);
467     exit(100);
468 }
469
470 puts("Lispy v1.4.0");
471 puts("Press ctrl-c to exit\n");
472
473 bool nonempty;
474 do {
475     char *input = readline("> ");
476     if ((nonempty = (input && *input))) {
477         add_history(input);
478
479         mpc_result_t parsed;
480         if (mpc_parse("<stdin>", input, Lispy, &parsed)) {
481             mpc_ast_t *ast = parsed.output;
482
483             lval *result = lval_eval(lval_read(ast));
484             lval_println(result);
485
486             mpc_ast_delete(ast);
487         } else {
488             mpc_err_print(parsed.error);
489             mpc_err_delete(parsed.error);
490         }
491     }
492
493     free(input);
494 } while (nonempty);
495
496 mpc_cleanup(6, Number, Symbol, Sexpr, Qexpr, Expr, Lispy);
497
498 return 0;
499 }

```

Chunks

⟨Add an element to an S-expression. 11b⟩ [2b](#), [11b](#)
 ⟨Add every y in ys to xs. 16b⟩ [3b](#), [16b](#)
 ⟨Add input to the history table. 9a⟩ [9a](#), [21a](#)
 ⟨Convert an S-expression to a Q-expression. 14c⟩ [4b](#), [14c](#)
 ⟨Declare a variable to hold parsing results. 9b⟩ [9b](#), [21a](#)
 ⟨Define possible lval and error types. 22a⟩ [2a](#), [22a](#)
 ⟨Define some useful macros. 23a⟩ [2a](#), [23a](#), [23b](#)
 ⟨Define the Lispy data structures. 21c⟩ [2a](#), [21c](#), [22b](#), [22c](#), [23c](#), [23d](#),
[24a](#), [24b](#), [25a](#)
 ⟨Define the language. 7d⟩ [6c](#), [7d](#), [8b](#)
 ⟨Delete the arguments and return a bad number error. 11f⟩ [11f](#), [11g](#)
 ⟨Evaluate a Q-expression. 16d⟩ [5b](#), [16d](#), [17a](#)
 ⟨Evaluate a built-in function or operation. 14b⟩ [5d](#), [14b](#), [14d](#), [15c](#), [15e](#),
[16c](#), [17b](#), [17c](#)
 ⟨Eval(uate) a built-in operation. 11g⟩ [5c](#), [11g](#), [12a](#), [12b](#), [12d](#), [14a](#)
 ⟨Evaluate an S-expression. 17d⟩ [5e](#), [17d](#), [17e](#), [17f](#), [18a](#), [18b](#)
 ⟨Evaluate an expression. 18c⟩ [6a](#), [18c](#)
 ⟨Eval(uate) the input. 9d⟩ [9d](#), [21a](#)
 ⟨Extract an element and shift the list. 18d⟩ [2c](#), [18d](#), [18e](#), [18f](#), [19a](#)
 ⟨For each argument 11d⟩ [11d](#), [11g](#), [15f](#), [17e](#)
 ⟨Include some mathematical definitions. 26c⟩ [25b](#), [26c](#)
 ⟨Include some string operations. 26d⟩ [25b](#), [26d](#)
 ⟨Include the boolean type and values. 25c⟩ [25b](#), [25c](#)
 ⟨Include the line editing functions from libedit. 26e⟩ [25b](#), [26e](#)
 ⟨Include the micro parser combinator definitions. 26f⟩ [25b](#), [26f](#)
 ⟨Include the necessary headers. 25b⟩ [2a](#), [25b](#)
 ⟨Include the standard I/O functions. 26a⟩ [25b](#), [26a](#)
 ⟨Include the standard library definitions. 26b⟩ [25b](#), [26b](#)
 ⟨Load the Lispy grammar. 7c⟩ [2a](#), [7c](#)
 ⟨Loop until the input is empty. 20c⟩ [6c](#), [20c](#)
 ⟨Perform a built-in operation. 12e⟩ [12d](#), [12e](#), [12f](#), [12g](#), [13a](#), [13b](#), [13c](#),
[13d](#), [13e](#)
 ⟨Pop the first element. 11h⟩ [11h](#), [12a](#), [12c](#), [18a](#)
 ⟨Pop the list and delete the rest. 14e⟩ [4c](#), [14e](#), [14f](#), [14g](#), [14h](#), [15a](#), [15b](#)
 ⟨Pop the list then delete it. 19b⟩ [3a](#), [19b](#), [19c](#)
 ⟨Pop the next element. 12c⟩ [12c](#), [12d](#)
 ⟨Print a Lispy value. 20a⟩ [3e](#), [20a](#)
 ⟨Print an expression. 19e⟩ [3d](#), [19e](#), [19f](#), [19g](#)
 ⟨Print and delete the error. 20b⟩ [20b](#), [21a](#)
 ⟨Print the result and delete the AST. 19d⟩ [19d](#), [21a](#)
 ⟨Print version and exit information. 7a⟩ [6c](#), [7a](#)
 ⟨Read a Lispy value. 9e⟩ [6b](#), [9e](#), [10b](#), [10c](#), [10d](#), [10e](#), [10f](#), [10g](#), [11c](#)

⟨Read a line of user input. 8d⟩ [8d](#), [20d](#)
 ⟨Read a number. 10a⟩ [6b](#), [10a](#)
 ⟨Read, eval(uate), and print. 20d⟩ [20c](#), [20d](#), [21a](#), [21b](#)
 ⟨Reallocate the memory used. 11a⟩ [11a](#), [11b](#), [19a](#)
 ⟨Return the concatenation of lists. 15f⟩ [4e](#), [15f](#), [16a](#)
 ⟨Return the extracted element. 18g⟩ [18g](#), [19a](#), [19c](#)
 ⟨Return the tail of a list. 15d⟩ [4d](#), [15d](#)
 ⟨Undefine and delete the parsers. 8c⟩ [6c](#), [8c](#)
 ⟨created parsers 8a⟩ [8a](#), [8b](#), [8c](#)
 ⟨input is nonempty 8e⟩ [8e](#), [21a](#)
 ⟨lispy.c 2a⟩ [2a](#), [2b](#), [2c](#), [3a](#), [3b](#), [3c](#), [3d](#), [3e](#), [4a](#), [4b](#), [4c](#), [4d](#), [4e](#), [5a](#), [5b](#), [5c](#),
[5d](#), [5e](#), [6a](#), [6b](#), [6c](#)
 ⟨lispy.mpc 7b⟩ [7b](#)
 ⟨the argument is not a number 11e⟩ [11e](#), [11g](#)
 ⟨the input can be parsed as Lispy code 9c⟩ [9c](#), [21a](#)

Index

Expr: [7d](#), [8a](#)
 LISPY_GRAMMAR: [7c](#), [8b](#)
 LVAL_ERR: [17e](#), [20a](#), [22a](#), [22a](#), [23c](#), [25a](#)
 LVAL_NUM: [20a](#), [22a](#), [22b](#), [22c](#), [25a](#)
 LVAL_QEXPR: [14c](#), [14f](#), [15d](#), [15f](#), [16d](#), [20a](#), [22a](#), [24b](#), [25a](#)
 LVAL_SEXPR: [17a](#), [18c](#), [20a](#), [22a](#), [24a](#), [25a](#)
 LVAL_SYM: [18a](#), [20a](#), [22a](#), [23d](#), [25a](#)
 Lispy: [7a](#), [7d](#), [8a](#), [9c](#)
 Number: [7d](#), [8a](#)
 Qexpr: [7d](#), [8a](#)
 Sexpr: [7d](#), [8a](#)
 Symbol: [7d](#), [8a](#)
 add_history: [9a](#), [26e](#)
 ast: [6b](#), [9d](#), [9e](#), [10a](#), [10b](#), [10d](#), [10e](#), [10f](#), [10g](#), [19d](#)
 bool: [20c](#), [22c](#), [25c](#)
 builtin: [5d](#), [18b](#)
 builtin_binop: [5c](#)
 builtin_head: [4c](#), [14d](#)
 builtin_join: [4e](#), [15e](#)
 builtin_list: [4b](#), [14b](#)
 builtin_tail: [4d](#), [15c](#)
 builtin_val: [5b](#)
 fmod: [13b](#), [26c](#)
 free: [21b](#), [25a](#), [26b](#)
 input: [8d](#), [8e](#), [8d](#), [8d](#), [9a](#), [9c](#), [8d](#), [21b](#)
 lval: [2b](#), [2c](#), [3a](#), [3b](#), [3c](#), [3d](#), [3e](#), [4a](#), [4b](#), [4c](#), [4d](#), [4e](#), [5a](#), [5b](#), [5c](#), [5d](#), [5e](#),
[6a](#), [6b](#), [9d](#), [10c](#), [11a](#), [12a](#), [12c](#), [14h](#), [15d](#), [16a](#), [17a](#), [18a](#), [18b](#), [18d](#), [18e](#),
[19b](#), [21c](#), [22b](#), [22c](#), [23c](#), [23d](#), [24a](#), [24b](#), [25a](#)
 lval_add: [2b](#), [10g](#), [16b](#)
 lval_del: [11f](#), [13a](#), [13b](#), [13d](#), [13e](#), [14a](#), [15a](#), [15d](#), [16a](#), [16b](#), [17c](#), [18a](#),
[18b](#), [19b](#), [23a](#), [25a](#)
 lval_err: [10a](#), [11f](#), [13a](#), [13b](#), [13d](#), [17c](#), [18a](#), [23a](#), [23c](#)
 lval_expr_print: [3d](#), [3d](#), [20a](#)
 lval_is_num: [11e](#), [22c](#)
 lval_join: [3b](#), [16a](#)
 lval_num: [10a](#), [22b](#)
 lval_pop: [2c](#), [11h](#), [15a](#), [15d](#), [16a](#), [16b](#), [19b](#)
 lval_print: [3e](#), [3c](#), [3e](#), [4a](#), [19f](#)
 lval_println: [4a](#), [19d](#)
 lval_qexpr: [10e](#), [24b](#)
 lval_read: [6b](#), [9d](#), [10g](#)
 lval_sexpr: [10d](#), [10f](#), [24a](#)

lval_sym: 10b, [23d](#)
lval_take: [3a](#), 14h, 15d, 17a, 17e, 17f
lval_type_t: 21c, [22a](#)
mpca_lang: 8b, [26f](#)
mpc_ast_delete: 19d, [26f](#)
mpc_ast_print: [26f](#)
mpc_ast_t: 6b, 9d, [26f](#)
mpc_cleanup: 26f, 8c, [26f](#)
mpc_err_delete: 8b, 20b, [26f](#)
mpc_err_print: 8b, 20b, [26f](#)
mpc_new: 7d, [26f](#)
mpc_parse: 26f, 9c, [26f](#)
mpc_parser_t: 7d, [26f](#)
mpc_result_t: 9b, [26f](#)
nonempty: [20c](#), 21a
parsed: 9b, [9b](#), 9c, 9d, 20b
pow: 13c, [26c](#)
printf: 20a, [26a](#)
readline: 26e, 8d, [26e](#)
strcmp: 10d, 10g, 12b, 12e, 12f, 12g, 13a, 13b, 13c, 14b, 14d, 15c, 15e,
 [16c](#), [26d](#)
strstr: 9e, 10b, 10e, 10f, 17b, [26d](#)
strtod: 10a, [26b](#)

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

<input type="checkbox"/> Describe the outline	2
<input type="checkbox"/> Describe this trick	7
<input checked="" type="checkbox"/> Describe the evaluation strategy	9
<input type="checkbox"/> Describe this	10
<input type="checkbox"/> Describe this	10
<input type="checkbox"/> Describe this	10
<input type="checkbox"/> Describe this, incl. how it's not cons	11
<input type="checkbox"/> Split this up and describe	15
<input type="checkbox"/> Describe this	16
<input type="checkbox"/> Describe this	16
<input type="checkbox"/> Describe this	19
<input type="checkbox"/> Describe this	19
<input type="checkbox"/> Describe this struct	21
<input type="checkbox"/> Describe headers	25
<input type="checkbox"/> Describe what a grammar is	42
<input type="checkbox"/> Describe what a parser is	42
<input type="checkbox"/> Describe programming language theory	42
<input type="checkbox"/> Describe what a REPL is	42