

Lispy: a simple Lisp-like language

Eric Bailey

*May 10, 2018*¹

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

For my own edification, and my eternal love of the LISP family and **PLT**, what follows is an implementation in C of a simple, Lisp-like programming language, based on Build Your Own Lisp [Holden, 2018a]. Since I'm a bit of masochist, this is a **literate program**², written using 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>	6
<i>Defining the Language</i>	6
<i>R is for Read</i>	8
<i>E is for Eval(uate)</i>	9
<i>Evaluating built-in operations</i>	11
<i>Evaluating (S)-expressions</i>	13
<i>P is for Print</i>	15
<i>L is for Loop</i>	17
<i>Error Handling</i>	18
<i>Headers</i>	22
<i>Full Listings</i>	24
<i>Chunks</i>	33
<i>Index</i>	35
<i>Glossary</i>	37
<i>References</i>	38

Outline

Describe the outline

2a $\langle \text{lispy.c } 2a \rangle \equiv$
 \langle Include the necessary headers. 22a \rangle

\langle Load the Lispy grammar. 6c \rangle

\langle Define possible lval and error types. 18b \rangle

\langle Define the Lispy data structures. 18a \rangle

This definition is continued in chunks 2–5.
 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. 10f \rangle

 return xs;
`}`

Defines:
 lval_add, used in chunk 10d.
 Uses lval 18a.

2c $\langle \text{lispy.c } 2a \rangle + \equiv$
`lval *lval_pop(lval *xs, int i)`
`{`
 \langle Extract an element and shift the list. 14d \rangle
`}`

Defines:
 lval_pop, used in chunks 11d and 15b.
 Uses lval 18a.

2d $\langle \text{lispy.c } 2a \rangle + \equiv$
`lval *lval_take(lval *xs, int i)`
`{`
 \langle Pop the list then delete it. 15b \rangle
`}`

Defines:
 lval_take, used in chunk 13.
 Uses lval 18a.

3a *<lisp.c 2a>+≡*
 void lval_print_err(lval *val)
 {
 <Print an error. 16a>
 }

Defines:

lval_print_err, used in chunk 16b.

Uses lval 18a.

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

3b *<lisp.c 2a>+≡*
 void lval_print(lval *val);

Uses lval 18a and lval_print 3d.

3c *<lisp.c 2a>+≡*
 void lval_expr_print(lval *expr, char open, char close)
 {
 <Print an expression. 15e>
 }

Defines:

lval_expr_print, used in chunks 3c and 16b.

Uses lval 18a.

3d *<lisp.c 2a>+≡*
 void lval_print(lval *val)
 {
 <Print a Lispy value. 16b>
 }

Defines:

lval_print, used in chunks 3 and 15f.

Uses lval 18a.

3e *<lisp.c 2a>+≡*
 void lval_println(lval *val)
 {
 lval_print(val);
 putchar('\n');
 }

Defines:

lval_println, used in chunk 15d.

Uses lval 18a and lval_print 3d.

4a *<lispy.c 2a>+≡*
`lval *builtin_op(char *op, lval *args)`
`{`
 <Eval(uate) a built-in operation. 11c>
`}`

Defines:

builtin_binop, never used.

Uses **lval 18a**.

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

4b *<lispy.c 2a>+≡*
`lval *lval_eval(lval* val);`

Uses **lval 18a**.

4c *<lispy.c 2a>+≡*
`lval* lval_eval_sexpr(lval *args)`
`{`
 <Evaluate an S-expression. 13e>
`}`

Uses **lval 18a**.

4d *<lispy.c 2a>+≡*
`lval* lval_eval(lval* val)`
`{`
 <Evaluate an expression. 14c>
`}`

Uses **lval 18a**.

4e *<lispy.c 2a>+≡*
`lval *lval_read_num(mpc_ast_t *ast)`
`{`
 <Read a number. 9c>
`}`

`lval *lval_read(mpc_ast_t *ast)`
`{`
 <Read a Lispy value. 9b>
`}`

Defines:

lval_read, used in chunks 9a and 10d.

Uses **ast 9a**, **lval 18a**, and **mpc_ast_t 23**.

```
5  <lisp.c 2a>+≡
    int main(int argc, char *argv[])
    {
        <Define the language. 7a>

        <Print version and exit information. 6a>

        <Loop until the input is empty. 17a>

        <Undefine and delete the parsers. 7d>

        return 0;
    }
```

Welcome

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

6a `<Print version and exit information. 6a>≡`
`puts("Lispy v1.1.1");`
`puts("Press ctrl-c to exit\n");`

Uses Lispy 7a.

This code is used in chunk 5.

Defining the Language

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

Support Core Erlang style numbers

6b `<lispy.mpc 6b>≡`
`integer : /-?[0-9]+/ ;`
`float : /-?[0-9]+\.[0-9]+/;`
`number : <float> | <integer> ;`
`symbol : '+' | '-' | '*' | '/' | '%' | '^' ;`
`sexpr : '(' <symbol> <expr>+ ')' ;`
`qexpr : '{' <symbol>? <expr>+ '}' ;`
`expr : <number> | <sexpr> | <qexpr> ;`
`lispy : /^/ <expr>* /$/ ;`

Root chunk (not used in this document).

Describe this trick

6c `<Load the Lispy grammar. 6c>≡`
`static const char LISPY_GRAMMAR[] = {`
`#include "lispy.xxd"`
`};`

Defines:

LISPY_GRAMMAR, used in chunk 7c.

This code is used in chunk 2a.

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

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

7a *<Define the language. 7a>*≡

```

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

```

Defines:

Expr, used in chunk 7b.
 Float, used in chunk 7b.
 Integer, used in chunk 7b.
 Lispy, used in chunks 6–8.
 Number, used in chunk 7b.
 Qexpr, used in chunk 7b.
 Sexpr, used in chunk 7b.
 Symbol, used in chunk 7b.

Uses `mpc_new` 23 and `mpc_parser_t` 23.

This definition is continued in chunk 7c.

This code is used in chunk 5.

Finally, using the defined *grammar* and each of the *<created parsers 7b>*,

7b *<created parsers 7b>*≡

```

Integer, Float, Number, Symbol, Sexpr, Qexpr, Expr, Lispy

```

 Uses Expr 7a, Float 7a, Integer 7a, Lispy 7a, Number 7a, Qexpr 7a, Sexpr 7a,
 and Symbol 7a.
 This code is used in chunk 7.

... we can define the Lispy language.

7c *<Define the language. 7a>*+≡

```

mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
          <created parsers 7b>);

```

Uses LISPY_GRAMMAR 6c and `mpca_lang` 23.

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.

7d *<Undefine and delete the parsers. 7d>*≡

```

mpc_cleanup(8, <created parsers 7b>);

```

Uses `mpc_cleanup` 23.

This code is used in chunk 5.

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.

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

8a *<Read a line of user input. 8a>*≡
`char *input = readline("> ");`

Defines:

`input`, used in chunks 8 and 17d.

Uses **readline** 22g.

This code is used in chunk 17b.

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

8b *<input is nonempty 8b>*≡
`input && *input`

Uses `input` 8a.

This code is used in chunk 17c.

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

8c *<Add input to the history table. 8c>*≡
`add_history(input);`

Uses **add_history** 22g and `input` 8a.

This code is used in chunk 17c.

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

8d *<Declare a variable to hold parsing results. 8d>*≡
`mpc_result_t parsed;`

Defines:

`parsed`, used in chunks 8, 9a, and 16c.

Uses **mpc_result_t** 23.

This code is used in chunk 17c.

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

8e *<the input can be parsed as Lispy code 8e>*≡
`mpc_parse("<stdin>", input, Lispy, &parsed)`

Uses **Lispy** 7a, `input` 8a, **mpc_parse** 23, and `parsed` 8d.

This code is used in chunk 17c.

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.

9a $\langle \text{Eval}(\text{uate}) \text{ the input. } 9a \rangle \equiv$
`mpc_ast_t *ast = parsed.output;`

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

Defines:

`ast`, used in chunks 4e, 9, 10, and 15d.

Uses `lval` 18a, `lval_read` 4e, `mpc_ast_t` 23, and `parsed` 8d.

This code is used in chunk 17c.

Describe the evaluation strategy

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

9b $\langle \text{Read a Lispy value. } 9b \rangle \equiv$
`if (strstr(ast->tag, "number"))`
`return lval_read_num(ast);`

Uses `ast` 9a and `strstr` 22f.

This definition is continued in chunks 9 and 10.

This code is used in chunk 4e.

Describe this

9c $\langle \text{Read a number. } 9c \rangle \equiv$
`errno = 0;`
`double num = strtod(ast->contents, NULL);`
`return errno != ERANGE ? lval_num(num) : lval_err(LERR_BAD_NUM);`

Uses `LERR_BAD_NUM` 19c, `ast` 9a, `lval_err` 20a, `lval_num` 19a, and `strtod` 22d.

This code is used in chunk 4e.

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

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

Uses `ast` 9a, `lval_sym` 20b, and `strstr` 22f.

Describe this

9e $\langle \text{Read a symbol. } 9e \rangle \equiv$
 Root chunk (not used in this document).

Describe this

9f $\langle \text{Read a Lispy value. } 9b \rangle + \equiv$
`lval *val = NULL;`

Uses `lval` 18a.

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

10a *⟨Read a Lispy value. 9b⟩* +=
 if (!strcmp(ast→tag, ">"))
 val = lval_sexpr();
 Uses ast 9a, lval_sexpr 20c, and strcmp 22f.

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

10b *⟨Read a Lispy value. 9b⟩* +=
 if (strstr(ast→tag, "qexpr"))
 val = lval_qexpr();
 Uses ast 9a, lval_qexpr 21a, and strstr 22f.

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

10c *⟨Read a Lispy value. 9b⟩* +=
 if (strstr(ast→tag, "sexpr"))
 val = lval_sexpr();
 Uses ast 9a, lval_sexpr 20c, and strstr 22f.

Describe this

10d *⟨Read a Lispy value. 9b⟩* +=
 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 9a, lval_add 2b, lval_read 4e, and strcmp 22f.

10e *⟨Reallocate the memory used. 10e⟩* =
 xs→cell = realloc(xs→cell, sizeof(lval *) * xs→count);
 Uses lval 18a.
 This code is used in chunks 10f and 15a.

Describe this, incl. how it's not cons

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

Finally, return the Lispy value.

10g *⟨Read a Lispy value. 9b⟩* +=
 return val;

10h *⟨For each argument 10h⟩* =
 for (int i = 0; i < args→count; i++)
 This code is used in chunks 11c and 13f.

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

Uses `lval_is_num` 19b.

This code is used in chunk 11c.

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

Uses `LERR_BAD_NUM` 19c, `lval_del` 21b, and `lval_err` 20a.

This code is used in chunk 11c.

Evaluating built-in operations

Ensure all arguments are numbers.

11c \langle Eval(uate) a built-in operation. 11c $\rangle \equiv$
 \langle For each argument 10h \rangle {
 if (\langle the argument is not a number 11a \rangle) {
 \langle Delete the arguments and return a bad number error. 11b \rangle
 }
}

This definition is continued in chunks 11 and 13d.

This code is used in chunk 4a.

11d \langle Pop the first element. 11d $\rangle \equiv$
`lval_pop(args, 0);`

Uses `lval_pop` 2c.

This code is used in chunks 11 and 14a.

Pop the first element.

11e \langle Eval(uate) a built-in operation. 11c $\rangle + \equiv$
`lval *result = \langle Pop the first element. 11d \rangle`

Uses `lval` 18a.

If the operation is unary subtraction, negate the operand.

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

Uses `strcmp` 22f.

11g \langle Pop the next element. 11g $\rangle \equiv$
`lval *y = \langle Pop the first element. 11d \rangle`

Uses `lval` 18a.

This code is used in chunk 11h.

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

 \langle Perform a built-in operation. 12a \rangle
`}`

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

```
12a <Perform a built-in operation. 12a>≡
    if (!strcmp(op, "+")) {
        result->num += y->num;
    }
```

Uses `strcmp` 22f.

This definition is continued in chunks 12 and 13.

This code is used in chunk 11h.

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

```
12b <Perform a built-in operation. 12a>+≡
    else if (!strcmp(op, "-")) {
        result->num -= y->num;
    }
```

Uses `strcmp` 22f.

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

```
12c <Perform a built-in operation. 12a>+≡
    else if (!strcmp(op, "*")) {
        result->num *= y->num;
    }
```

Uses `strcmp` 22f.

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

```
12d <Perform a built-in operation. 12a>+≡
    else if (!strcmp(op, "/")) {
        if (!y->num) {
            lval_del(result);
            lval_del(y);
            result = lval_err(LERR_DIV_ZERO);
            break;
        }
        result->num /= y->num;
    }
```

Uses `LERR_DIV_ZERO` 19c, `lval_del` 21b, `lval_err` 20a, and `strcmp` 22f.

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

```
12e <Perform a built-in operation. 12a>+≡
    else if (!strcmp(op, "%")) {
        if (!y->num) {
            lval_del(result);
            lval_del(y);
            result = lval_err(LERR_DIV_ZERO);
            break;
        }
        result->num = fmod(result->num, y->num);
    }
```

Uses `LERR_DIV_ZERO` 19c, `fmod` 22e, `lval_del` 21b, `lval_err` 20a, and `strcmp` 22f.

If the **opp** is **"^"**, perform exponentiation.

13a $\langle \text{Perform a built-in operation. 12a} \rangle + \equiv$
 else if (!strcmp(op, "^")) {
 result->num = pow(result->num, y->num);
 }

Uses **pow 22e** and **strcmp 22f**.

Otherwise, return a **LERR_BAD_OP** error.

13b $\langle \text{Perform a built-in operation. 12a} \rangle + \equiv$
 else {
 lval_del(result);
 lval_del(y);
 result = lval_err(LERR_BAD_OP);
 break;
 }

Uses **LERR_BAD_OP 19c**, **lval_del 21b**, and **lval_err 20a**.

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

13c $\langle \text{Perform a built-in operation. 12a} \rangle + \equiv$
 lval_del(y);

Uses **lval_del 21b**.

Delete the input expression and return the result.

13d $\langle \text{Eval(uate) a built-in operation. 11c} \rangle + \equiv$
 lval_del(args);

return result;

Uses **lval_del 21b**.

Evaluating (S)-expressions

If the expression is empty, return it;

13e $\langle \text{Evaluate an S-expression. 13e} \rangle \equiv$
 if (!args->count)
 return args;

This definition is continued in chunks **13** and **14**.

This code is used in chunk **4c**.

13f $\langle \text{Evaluate an S-expression. 13e} \rangle + \equiv$
 $\langle \text{For each argument 10h} \rangle \{$
 args->cell[i] = lval_eval(args->cell[i]);
 if (args->cell[i]->type == LVAL_ERR)
 return lval_take(args, i);
 }

Uses **LVAL_ERR 18b 18b** and **lval_take 2d**.

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

13g $\langle \text{Evaluate an S-expression. 13e} \rangle + \equiv$
 if (args->count == 1)
 return lval_take(args, 0);

Uses **lval_take 2d**.

```

14a  <Evaluate an S-expression. 13e>+≡
      lval *car = <Pop the first element. 11d>;
      if (car->type ≠ LVAL_SYM) {
          lval_del(car);
          lval_del(args);

          return lval_err(LERR_BAD_SEXPR);
      }

```

Uses LVAL_SYM 18b, lval 18a, lval_del 21b, and lval_err 20a.

```

14b  <Evaluate an S-expression. 13e>+≡
      lval *result = builtin_op(car->sym, args);
      lval_del(car);

      return result;

```

Uses lval 18a and lval_del 21b.

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

```

14c  <Evaluate an expression. 14c>≡
      if (val->type = LVAL_SEXPR)
          return lval_eval_sexpr(val);

      return val;

```

Uses LVAL_SEXPR 18b.

This code is used in chunk 4d.

Extract the element at index *i*.

```

14d  <Extract an element and shift the list. 14d>≡
      lval *elem = xs->cell[i];

```

Uses lval 18a.

This definition is continued in chunks 14 and 15a.

This code is used in chunk 2c.

Shift memory after the element at index *i*.

```

14e  <Extract an element and shift the list. 14d>+≡
      memmove(&xs->cell[i], &xs->cell[i + 1],
              sizeof(lval *) * (xs->count - i - 1));

```

Uses lval 18a.

Decrease the count.

```

14f  <Extract an element and shift the list. 14d>+≡
      xs->count--;

```

```

14g  <Return the extracted element. 14g>≡
      return elem;

```

This code is used in chunk 15.

Reallocate the memory used and return the extracted element.

15a \langle Extract an element and shift the list. 14d $\rangle + \equiv$
 \langle Reallocate the memory used. 10e \rangle

\langle Return the extracted element. 14g \rangle

Describe this

15b \langle Pop the list then delete it. 15b $\rangle \equiv$
`lval *elem = lval_pop(xs, i);`
`lval_del(xs);`

Uses lval 18a, lval_del 21b, and lval_pop 2c.

This definition is continued in chunk 15c.

This code is used in chunk 2d.

Return the extracted element.

15c \langle Pop the list then delete it. 15b $\rangle + \equiv$
 \langle Return the extracted element. 14g \rangle

P is for Print

Upon success, print the result and delete the AST.

15d \langle Print the result and delete the AST. 15d $\rangle \equiv$
`lval_println(result);`

`mpc_ast_delete(ast);`

Uses ast 9a, lval_println 3e, and mpc_ast_delete 23.

This code is used in chunk 17c.

Describe this

Print the opening character.

15e \langle Print an expression. 15e $\rangle \equiv$
`putchar(open);`

This definition is continued in chunk 15.

This code is used in chunk 3c.

Print all but the last element with a trailing space.

15f \langle Print an expression. 15e $\rangle + \equiv$
`for (int i = 0; i < expr->count; i++) {`
`lval_print(expr->cell[i]);`
`if (i != (expr->count - 1))`
`putchar(' ');`
`}`

Uses lval_print 3d.

Print the closing character.

15g \langle Print an expression. 15e $\rangle + \equiv$
`putchar(close);`

16a *<Print an error. 16a>*≡

```

switch (val→err) {
case LERR_BAD_NUM:
    puts("Error: invalid number");
    break;
case LERR_BAD_OP:
    puts("Error: invalid operator");
    break;
case LERR_BAD_SEXPR:
    puts("Error: S-expression does not start with symbol");
    break;
case LERR_DIV_ZERO:
    puts("Error: division by zero");
    break;
}

```

Uses LERR_BAD_NUM 19c, LERR_BAD_OP 19c, and LERR_DIV_ZERO 19c.
This code is used in chunk 3a.

16b *<Print a Lispy value. 16b>*≡

```

switch (val→type) {
case LVAL_ERR:
    lval_print_err(val);
    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 18b 18b, LVAL_NUM 18b, LVAL_QEXPR 18b, LVAL_SEXPR 18b,
LVAL_SYM 18b, lval_expr_print 3c, lval_print_err 3a, and printf 22c.
This code is used in chunk 3d.

Print and delete the error upon failure.

16c *<Print and delete the error. 16c>*≡

```

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

```

Uses mpc_err_delete 23, mpc_err_print 23, and parsed 8d.
This code is used in chunk 17c.

L is for Loop

17a $\langle \text{Loop until the input is empty. 17a} \rangle \equiv$
 bool nonempty;
 do {
 $\langle \text{Read, eval(uate), and print. 17b} \rangle$
 } while (nonempty);

Defines:

 nonempty, used in chunk 17c.

Uses bool 22b.

This code is used in chunk 5.

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

17b $\langle \text{Read, eval(uate), and print. 17b} \rangle \equiv$
 $\langle \text{Read a line of user input. 8a} \rangle$

This definition is continued in chunk 17.

This code is used in chunk 17a.

If, and only if, it's not empty, add it to the history table, **Eval**(uate) it, and **Print** the result.

17c $\langle \text{Read, eval(uate), and print. 17b} \rangle + \equiv$
 if ((nonempty = ($\langle \text{input is nonempty 8b} \rangle$))) {
 $\langle \text{Add input to the history table. 8c} \rangle$

 $\langle \text{Declare a variable to hold parsing results. 8d} \rangle$
 if (($\langle \text{the input can be parsed as Lispy code 8e} \rangle$)) {
 $\langle \text{Eval(uate) the input. 9a} \rangle$
 $\langle \text{Print the result and delete the AST. 15d} \rangle$
 } else {
 $\langle \text{Print and delete the error. 16c} \rangle$
 }
 }

Uses nonempty 17a.

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

17d $\langle \text{Read, eval(uate), and print. 17b} \rangle + \equiv$
 free(input);

Uses free 22d and input 8a.

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

Error Handling

Describe this struct

18a *<Define the Lispy data structures. 18a>*≡

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

Defines:

lval, used in chunks 2–4, 9–11, 14, 15b, and 19–21.

Uses lval_err_t 19c and lval_type_t 18b.

This definition is continued in chunks 19–21.

This code is used in chunk 2a.

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

18b *<Define possible lval and error types. 18b>*≡

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

Defines:

LVAL_ERR, used in chunks 13f, 16b, 20a, and 21b.

LVAL_NUM, used in chunks 16b, 19, and 21b.

LVAL_QEXPR, used in chunks 16b and 21.

LVAL_SEXPR, used in chunks 14c, 16b, 20c, and 21b.

LVAL_SYM, used in chunks 14a, 16b, 20b, and 21b.

lval_type_t, used in chunk 18a.

This definition is continued in chunk 19c.

This code is used in chunk 2a.

Define a constructor for numbers.

19a *<Define the Lispy data structures. 18a>+≡*

```
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 9c.

Uses `LVAL_NUM` 18b and `lval` 18a.

Define a convenient predicate for numbers.

19b *<Define the Lispy data structures. 18a>+≡*

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

Defines:

`lval_is_num`, used in chunk 11a.

Uses `LVAL_NUM` 18b, `bool` 22b, and `lval` 18a.

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

19c *<Define possible lval and error types. 18b>+≡*

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

Defines:

`LERR_BAD_NUM`, used in chunks 9c, 11b, and 16a.

`LERR_BAD_OP`, used in chunks 13b and 16a.

`LERR_DIV_ZERO`, used in chunks 12 and 16a.

`lval_err_t`, used in chunks 18a and 20a.

Define a constructor for errors.

20a *<Define the Lispy data structures. 18a>+≡*

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

    return val;
}
```

Defines:

`lval_err`, used in chunks 9c and 11–14.

Uses `LVAL_ERR` 18b 18b, `lval` 18a, and `lval_err_t` 19c.

Define a constructor for symbol.

20b *<Define the Lispy data structures. 18a>+≡*

```
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 9d.

Uses `LVAL_SYM` 18b and `lval` 18a.

Define a constructor for an S-expression.

20c *<Define the Lispy data structures. 18a>+≡*

```
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` 18b and `lval` 18a.

Define a constructor for a Q-expression.

21a *<Define the Lispy data structures. 18a>+≡*

```
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 10b.

Uses `LVAL_QEXPR` 18b and `lval` 18a.

Define a destructor for `lval*`.

21b *<Define the Lispy data structures. 18a>+≡*

```
void lval_del(lval *val)
{
    switch(val->type) {
        case LVAL_ERR:
        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 11–15.

Uses `LVAL_ERR` 18b 18b, `LVAL_NUM` 18b, `LVAL_QEXPR` 18b, `LVAL_SEXPR` 18b, `LVAL_SYM` 18b, `free` 22d, and `lval` 18a.

Headers

Describe headers

- 22a** *<Include the necessary headers. 22a>*≡
<Include the boolean type and values. 22b>
<Include the standard I/O functions. 22c>
<Include the standard library definitions. 22d>
<Include some mathematical definitions. 22e>
<Include some string operations. 22f>

<Include the line editing functions from libedit. 22g>
<Include the micro parser combinator definitions. 23>
 This code is used in chunk **2a**.
- 22b** *<Include the boolean type and values. 22b>*≡
`#include <stdbool.h>`
 Defines:
`bool`, used in chunks **17a** and **19b**.
 This code is used in chunk **22a**.
- 22c** *<Include the standard I/O functions. 22c>*≡
`#include <stdio.h>`
 Defines:
`printf`, used in chunk **16b**.
 This code is used in chunk **22a**.
- 22d** *<Include the standard library definitions. 22d>*≡
`#include <stdlib.h>`
 Defines:
`free`, used in chunks **17d** and **21b**.
`strtod`, used in chunk **9c**.
 This code is used in chunk **22a**.
- 22e** *<Include some mathematical definitions. 22e>*≡
`#include <math.h>`
 Defines:
`fmod`, used in chunk **12e**.
`pow`, used in chunk **13a**.
 This code is used in chunk **22a**.
- 22f** *<Include some string operations. 22f>*≡
`#include <string.h>`
 Defines:
`strcmp`, used in chunks **10–13**.
`strstr`, used in chunks **9** and **10**.
 This code is used in chunk **22a**.
- 22g** *<Include the line editing functions from libedit. 22g>*≡
`#include <editline/readline.h>`
 Defines:
`add_history`, used in chunk **8c**.
`readline`, used in chunks **22g** and **8a**.
 This code is used in chunk **22a**.

23 *(Include the micro parser combinator definitions. 23)*≡
 #include <mpc.h>

Defines:

 mpca_lang, used in chunk 7c.
 mpc_ast_delete, used in chunk 15d.
 mpc_ast_print, never used.
 mpc_ast_t, used in chunks 4e and 9a.
 mpc_cleanup, used in chunks 23 and 7d.
 mpc_err_delete, used in chunk 16c.
 mpc_err_print, used in chunk 16c.
 mpc_new, used in chunk 7a.
 mpc_parse, used in chunks 23 and 8e.
 mpc_parser_t, used in chunk 7a.
 mpc_result_t, used in chunk 8d.

This code is used in chunk 22a.

Full Listings

lispy.mpc:

```
integer : /-?[0-9]+/ ;  
float   : /-?[0-9]+\.[0-9]+/ ;  
number  : <float> | <integer> ;  
symbol  : '+' | '-' | '*' | '/' | '%' | '^' ;  
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 static const char LISPY_GRAMMAR[] = {
12     #include "lispy.xxd"
13 };
14
15
16 typedef enum {
17     LVAL_ERR,
18     LVAL_NUM,
19     LVAL_QEXPR,
20     LVAL_SEXPR,
21     LVAL_SYM
22 } lval_type_t;
23
24
25 typedef enum {
26     LERR_DIV_ZERO,
27     LERR_BAD_OP,
28     LERR_BAD_NUM,
29     LERR_BAD_SEXPR
30 } lval_err_t;
31
32
33 typedef struct lval {
34     lval_type_t type;
35     union {
36         double num;
37         lval_err_t err;
38         char *sym;
39     };
40     int count;
41     struct lval **cell;
42 } lval;
43
44
45 lval *lval_num(double num)
46 {
47     lval *val = malloc(sizeof(lval));
48     val->type = LVAL_NUM;
49     val->num = num;
50

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

306 }
307
308
309 lval *lval_eval(lval * val)
310 {
311     if (val->type == LVAL_SEXPR)
312         return lval_eval_sexpr(val);
313
314     return val;
315 }
316
317
318 lval *lval_read_num(mpc_ast_t * ast)
319 {
320     errno = 0;
321     double num = strtod(ast->contents, NULL);
322     return errno != ERANGE ? lval_num(num) : lval_err(LERR_BAD_NUM);
323 }
324
325
326 lval *lval_read(mpc_ast_t * ast)
327 {
328     if (strstr(ast->tag, "number"))
329         return lval_read_num(ast);
330
331     if (strstr(ast->tag, "symbol"))
332         return lval_sym(ast->contents);
333
334     lval *val = NULL;
335     if (!strcmp(ast->tag, ">"))
336         val = lval_sexpr();
337     if (strstr(ast->tag, "qexpr"))
338         val = lval_qexpr();
339     if (strstr(ast->tag, "sexpr"))
340         val = lval_sexpr();
341
342     for (int i = 0; i < ast->children_num; i++) {
343         if (!strcmp(ast->children[i]->contents, "("))
344             continue;
345         if (!strcmp(ast->children[i]->contents, ")"))
346             continue;
347         if (!strcmp(ast->children[i]->contents, "{"))
348             continue;
349         if (!strcmp(ast->children[i]->contents, "}")
350             continue;
351         if (!strcmp(ast->children[i]->tag, "regex"))
352             continue;
353         val = lval_add(val, lval_read(ast->children[i]));
354     }
355
356     return val;

```

```

357 }
358
359
360 int main(int argc, char *argv[])
361 {
362     mpc_parser_t *Integer = mpc_new("integer");
363     mpc_parser_t *Float = mpc_new("float");
364     mpc_parser_t *Number = mpc_new("number");
365     mpc_parser_t *Symbol = mpc_new("symbol");
366     mpc_parser_t *Sexpr = mpc_new("sexpr");
367     mpc_parser_t *Qexpr = mpc_new("qexpr");
368     mpc_parser_t *Expr = mpc_new("expr");
369     mpc_parser_t *Lispy = mpc_new("lispy");
370
371     mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
372              Integer, Float, Number, Symbol, Sexpr, Qexpr, Expr, Lispy);
373
374     puts("Lispy v1.1.1");
375     puts("Press ctrl-c to exit\n");
376
377     bool nonempty;
378     do {
379         char *input = readline("> ");
380         if ((nonempty = (input && *input))) {
381             add_history(input);
382
383             mpc_result_t parsed;
384             if (mpc_parse("<stdin>", input, Lispy, &parsed)) {
385                 mpc_ast_t *ast = parsed.output;
386
387                 lval *result = lval_eval(lval_read(ast));
388                 lval_println(result);
389
390                 mpc_ast_delete(ast);
391             } else {
392                 mpc_err_print(parsed.error);
393                 mpc_err_delete(parsed.error);
394             }
395         }
396
397         free(input);
398     } while (nonempty);
399
400     mpc_cleanup(8, Integer, Float, Number, Symbol, Sexpr, Qexpr, Expr,
401               Lispy);
402
403     return 0;
404 }

```


Chunks

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

⟨*Undefine and delete the parsers.* 7d⟩ 5, [7d](#)
 ⟨*created parsers* 7b) [7b](#), [7c](#), [7d](#)
 ⟨*input is nonempty* 8b) [8b](#), [17c](#)
 ⟨*lispy.c* 2a) [2a](#), [2b](#), [2c](#), [2d](#), [3a](#), [3b](#), [3c](#), [3d](#), [3e](#), [4a](#), [4b](#), [4c](#), [4d](#), [4e](#), [5](#)
 ⟨*lispy.mpc* 6b) [6b](#)
 ⟨*the argument is not a number* 11a) [11a](#), [11c](#)
 ⟨*the input can be parsed as Lispy code* 8e) [8e](#), [17c](#)

Index

Expr: [7a](#), [7b](#)
 Float: [7a](#), [7b](#)
 Integer: [7a](#), [7b](#)
 LERR_BAD_NUM: [9c](#), [11b](#), [16a](#), [19c](#)
 LERR_BAD_OP: [13b](#), [16a](#), [19c](#)
 LERR_DIV_ZERO: [12d](#), [12e](#), [16a](#), [19c](#)
 LISPY_GRAMMAR: [6c](#), [7c](#)
 LVAL_ERR: [13f](#), [16b](#), [18b](#), [18b](#), [20a](#), [21b](#)
 LVAL_NUM: [16b](#), [18b](#), [19a](#), [19b](#), [21b](#)
 LVAL_QEXPR: [16b](#), [18b](#), [21a](#), [21b](#)
 LVAL_SEXPR: [14c](#), [16b](#), [18b](#), [20c](#), [21b](#)
 LVAL_SYM: [14a](#), [16b](#), [18b](#), [20b](#), [21b](#)
 Lispy: [6a](#), [7a](#), [7b](#), [8e](#)
 Number: [7a](#), [7b](#)
 Qexpr: [7a](#), [7b](#)
 Sexpr: [7a](#), [7b](#)
 Symbol: [7a](#), [7b](#)
 add_history: [8c](#), [22g](#)
 ast: [4e](#), [9a](#), [9b](#), [9c](#), [9d](#), [10a](#), [10b](#), [10c](#), [10d](#), [15d](#)
 bool: [17a](#), [19b](#), [22b](#)
 builtin_binop: [4a](#)
 fmod: [12e](#), [22e](#)
 free: [17d](#), [21b](#), [22d](#)
 input: [8a](#), [8b](#), [8a](#), [8a](#), [8c](#), [8e](#), [8a](#), [17d](#)
 lval: [2b](#), [2c](#), [2d](#), [3a](#), [3b](#), [3c](#), [3d](#), [3e](#), [4a](#), [4b](#), [4c](#), [4d](#), [4e](#), [9a](#), [9f](#), [10e](#), [11e](#),
 [11g](#), [14a](#), [14b](#), [14d](#), [14e](#), [15b](#), [18a](#), [19a](#), [19b](#), [20a](#), [20b](#), [20c](#), [21a](#), [21b](#)
 lval_add: [2b](#), [10d](#)
 lval_del: [11b](#), [12d](#), [12e](#), [13b](#), [13c](#), [13d](#), [14a](#), [14b](#), [15b](#), [21b](#)
 lval_err: [9c](#), [11b](#), [12d](#), [12e](#), [13b](#), [14a](#), [20a](#)
 lval_err_t: [18a](#), [19c](#), [20a](#)
 lval_expr_print: [3c](#), [3c](#), [16b](#)
 lval_is_num: [11a](#), [19b](#)
 lval_num: [9c](#), [19a](#)
 lval_pop: [2c](#), [11d](#), [15b](#)
 lval_print: [3d](#), [3b](#), [3d](#), [3e](#), [15f](#)
 lval_print_err: [3a](#), [16b](#)
 lval_println: [3e](#), [15d](#)
 lval_qexpr: [10b](#), [21a](#)
 lval_read: [4e](#), [9a](#), [10d](#)
 lval_sexpr: [10a](#), [10c](#), [20c](#)
 lval_sym: [9d](#), [20b](#)
 lval_take: [2d](#), [13f](#), [13g](#)

lval_type_t: [18a](#), [18b](#)
mpca_lang: [7c](#), [23](#)
mpc_ast_delete: [15d](#), [23](#)
mpc_ast_print: [23](#)
mpc_ast_t: [4e](#), [9a](#), [23](#)
mpc_cleanup: [23](#), [7d](#), [23](#)
mpc_err_delete: [16c](#), [23](#)
mpc_err_print: [16c](#), [23](#)
mpc_new: [7a](#), [23](#)
mpc_parse: [23](#), [8e](#), [23](#)
mpc_parser_t: [7a](#), [23](#)
mpc_result_t: [8d](#), [23](#)
nonempty: [17a](#), [17c](#)
parsed: [8d](#), [8d](#), [8e](#), [9a](#), [16c](#)
pow: [13a](#), [22e](#)
printf: [16b](#), [22c](#)
readline: [22g](#), [8a](#), [22g](#)
strcmp: [10a](#), [10d](#), [11f](#), [12a](#), [12b](#), [12c](#), [12d](#), [12e](#), [13a](#), [22f](#)
strstr: [9b](#), [9d](#), [10b](#), [10c](#), [22f](#)
strtod: [9c](#), [22d](#)

Glossary

AST abstract syntax tree, a tree representation of the abstract syntactic structure of source code. [9](#), [10](#), [15](#)

grammar [6](#), [7](#)

Describe what a grammar is

parser [7](#)

Describe what a parser is

PLT programming language theory, [1](#)

Describe programming language theory

REPL Read-Eval-Print Loop, [6](#), [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

Describe the outline	2
Support Core Erlang style numbers	6
Describe this trick	6
Describe the evaluation strategy	9
Describe this	9
Describe this	9
Describe this	9
Describe this	10
Describe this, incl. how it's not cons	10
Describe this	15
Describe this	15
Describe this struct	18
Describe headers	22
Describe what a grammar is	37
Describe what a parser is	37
Describe programming language theory	37
Describe what a REPL is	37