

# *Lispy: a simple Lisp-like language*

*Eric Bailey*

*May 10, 2018*<sup>1</sup>

<sup>1</sup> 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**<sup>2</sup>, written using Noweb<sup>3</sup>.

<sup>2</sup> [https://en.wikipedia.org/wiki/Literate\\_programming](https://en.wikipedia.org/wiki/Literate_programming)

<sup>3</sup> 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>	7
<i>E is for Eval(uate)</i>	8
<i>Evaluating built-in operations</i>	10
<i>Evaluating (S)-expressions</i>	13
<i>P is for Print</i>	15
<i>L is for Loop</i>	16
<i>Error Handling</i>	17
<i>Headers</i>	21
<i>Full Listings</i>	23
<i>Chunks</i>	32
<i>Index</i>	34
<i>Glossary</i>	36
<i>References</i>	37

## Outline

Describe the outline

2a  $\langle \text{lispy.c } 2a \rangle \equiv$   
 $\langle \text{Include the necessary headers. } 21a \rangle$

$\langle \text{Load the Lispy grammar. } 6c \rangle$

$\langle \text{Define possible lval and error types. } 18a \rangle$

$\langle \text{Define the Lispy data structures. } 17c \rangle$

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)`  
`{`  
 $\langle \text{Add an element to an S-expression. } 10c \rangle$   
`return xs;`  
`}`

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

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

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

2d  $\langle \text{lispy.c } 2a \rangle + \equiv$   
`lval *lval_take(lval *xs, int i)`  
`{`  
 $\langle \text{Pop the list then delete it. } 14g \rangle$   
`}`

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

```

3a  <lisp.c 2a>+≡
    void lval_print_err(lval *val)
    {
        <Print an error. 15e>
    }

```

Defines:

lval\_print\_err, used in chunk 16a.

Uses lval 17c.

Forward declare<sup>4</sup> lval\_print, since it's mutually recursive<sup>5</sup> with lval\_expr\_print.

<sup>4</sup> [https://en.wikipedia.org/wiki/Forward\\_declaration](https://en.wikipedia.org/wiki/Forward_declaration)

<sup>5</sup> [https://en.wikipedia.org/wiki/Mutual\\_recursion](https://en.wikipedia.org/wiki/Mutual_recursion)

```

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

```

Uses lval 17c and lval\_print 3d.

```

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

```

Defines:

lval\_expr\_print, used in chunks 3c and 16a.

Uses lval 17c.

```

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

```

Defines:

lval\_print, used in chunks 3 and 15c.

Uses lval 17c.

```

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

```

Defines:

lval\_println, used in chunk 15a.

Uses lval 17c and lval\_print 3d.

4a *<lisp.c 2a>+≡*  
 lval \*builtin\_op(char \*op, lval \*args)  
 {  
   *<Eval(uate) a built-in operation. 10h>*  
 }

Defines:

  builtin\_binop, never used.

Uses lval 17c.

Forward declare lval\_eval, since it's mutually recursive with  
 lval\_eval\_sexpr.

4b *<lisp.c 2a>+≡*  
 lval \*lval\_eval(lval\* val);

Uses lval 17c.

4c *<lisp.c 2a>+≡*  
 lval\* lval\_eval\_sexpr(lval \*args)  
 {  
   *<Evaluate an S-expression. 13b>*  
 }

Uses lval 17c.

4d *<lisp.c 2a>+≡*  
 lval\* lval\_eval(lval\* val)  
 {  
   *<Evaluate an expression. 14a>*  
 }

Uses lval 17c.

4e *<lisp.c 2a>+≡*  
 lval \*lval\_read\_num(mpc\_ast\_t \*ast)  
 {  
   *<Read a number. 9b>*  
 }

lval \*lval\_read(mpc\_ast\_t \*ast)  
 {  
   *<Read a Lispy value. 9a>*  
 }

Defines:

  lval\_read, used in chunks 8e and 10a.

Uses ast 8e, lval 17c, and mpc\_ast\_t 22.

```
5  <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. 16c>

        <Undefine and delete the parsers. 7d>

        return 0;
    }
```

## Welcome

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

6a `<Print version and exit information. 6a>≡`  

```
puts("Lispy v1.0.1");
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*.

6b `<lispy.mpc 6b>≡`  

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

Root chunk (not used in this document).

Support Core Erlang style numbers

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.

Describe this trick

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

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

7a *<Define the language. 7a>*≡  
 mpc\_parser\_t \*Integer = mpc\_new("integer");  
 mpc\_parser\_t \*Float = mpc\_new("float");  
 mpc\_parser\_t \*Number = mpc\_new("number");  
 mpc\_parser\_t \*Symbol = mpc\_new("symbol");  
 mpc\_parser\_t \*Sexpr = mpc\_new("sexpr");  
 mpc\_parser\_t \*Expr = mpc\_new("expr");  
 mpc\_parser\_t \*Lispy = mpc\_new("lispy");

Defines:

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

Uses mpc\_new 22 and mpc\_parser\_t 22.

This definition is continued in chunk 7c.

This code is used in chunk 5.

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

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

... we can define the Lispy language.

7c *<Define the language. 7a>*+≡  
 mpca\_lang(MPCA\_LANG\_DEFAULT, LISPY\_GRAMMAR,  
   *<created parsers 7b>*);

Uses LISPY\_GRAMMAR 6c and mpca\_lang 22.

Since we're implementing this in C, we need to clean up after ourselves. The mpc<sup>6</sup> library makes this easy, by providing the mpc\_cleanup function.

7d *<Undefine and delete the parsers. 7d>*≡  
 mpc\_cleanup(7, *<created parsers 7b>*);

Uses mpc\_cleanup 22.

This code is used in chunk 5.

## *R is for Read*

To implement the R in *REPL*, use *readline* from *libedit*<sup>7</sup>.

7e *<Read a line of user input. 7e>*≡  
 char \*input = readline("> ");

Defines:

input, used in chunks 8a, 7, 8, and 17b.

Uses readline 21g.

This code is used in chunk 16d.

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

<sup>7</sup> 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.

8a  $\langle \text{input is nonempty 8a} \rangle \equiv$   
`input && *input`

Uses `input 7e`.

This code is used in chunk 17a.

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

So long as `input` is nonempty, add it to the `libedit`<sup>8</sup> history table.

8b  $\langle \text{Add input to the history table. 8b} \rangle \equiv$   
`add_history(input);`

Uses `add_history 21g` and `input 7e`.

This code is used in chunk 17a.

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

8c  $\langle \text{Declare a variable to hold parsing results. 8c} \rangle \equiv$   
`mpc_result_t parsed;`

Defines:

`parsed`, used in chunks 8 and 16b.

Uses `mpc_result_t 22`.

This code is used in chunk 17a.

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

8d  $\langle \text{the input can be parsed as Lispy code 8d} \rangle \equiv$   
`mpc_parse("<stdin>", input, Lispy, &parsed)`

Uses `Lispy 7a`, `input 7e`, `mpc_parse 22`, and `parsed 8c`.

This code is used in chunk 17a.

*E is for Eval(uate)*

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

8e  $\langle \text{Eval(uate) the input. 8e} \rangle \equiv$   
`mpc_ast_t *ast = parsed.output;`

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

Defines:

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

Uses `lval 17c`, `lval_read 4e`, `mpc_ast_t 22`, and `parsed 8c`.

This code is used in chunk 17a.

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



Describe the evaluation strategy

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

9a  $\langle \text{Read a Lispy value. 9a} \rangle \equiv$   
 if (strstr(ast→tag, "number"))  
     return lval\_read\_num(ast);

Uses ast 8e and strstr 21f.

This definition is continued in chunks 9 and 10.

This code is used in chunk 4e.

Describe this

9b  $\langle \text{Read a number. 9b} \rangle \equiv$   
 errno = 0;  
 double num = strtod(ast→contents, NULL);  
 return errno  $\neq$  ERANGE ? lval\_num(num) : lval\_err(LERR\_BAD\_NUM);

Uses LERR\_BAD\_NUM 19a, ast 8e, lval\_err 19b, lval\_num 18b, and strtod 21d.

This code is used in chunk 4e.

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

9c  $\langle \text{Read a Lispy value. 9a} \rangle + \equiv$   
 if (strstr(ast→tag, "symbol"))  
     return lval\_sym(ast→contents);

Uses ast 8e, lval\_sym 19c, and strstr 21f.

Describe this

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

Describe this

9e  $\langle \text{Read a Lispy value. 9a} \rangle + \equiv$   
 lval \*sexpr = NULL;  
 Uses lval 17c.

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

9f  $\langle \text{Read a Lispy value. 9a} \rangle + \equiv$   
 if (!strcmp(ast→tag, ">"))  
     sexpr = lval\_sexpr();

Uses ast 8e, lval\_sexpr 20a, and strcmp 21f.

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

9g  $\langle \text{Read a Lispy value. 9a} \rangle + \equiv$   
 if (strstr(ast→tag, "sexpr"))  
     sexpr = lval\_sexpr();

Uses ast 8e, lval\_sexpr 20a, and strstr 21f.

Describe this

10a  $\langle \text{Read a Lispy value. 9a} \rangle + \equiv$

```

    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]->tag, "regex")) continue;
        sexpr = lval_add(sexpr, lval_read(ast->children[i]));
    }

```

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

10b  $\langle \text{Reallocate the memory used. 10b} \rangle \equiv$

```

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

```

Uses `lval` 17c.

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

Describe this, incl. how it's not cons

10c  $\langle \text{Add an element to an S-expression. 10c} \rangle \equiv$

```

    xs->count++;
     $\langle \text{Reallocate the memory used. 10b} \rangle$ 
    xs->cell[xs->count - 1] = x;

```

This code is used in chunk 2b.

Finally, return the S-expression.

10d  $\langle \text{Read a Lispy value. 9a} \rangle + \equiv$

```

    return sexpr;

```

10e  $\langle \text{For each argument 10e} \rangle \equiv$

```

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

```

This code is used in chunks 10h and 13c.

10f  $\langle \text{the argument is not a number 10f} \rangle \equiv$

```

    !lval_is_num(args->cell[i])

```

Uses `lval_is_num` 18c.

This code is used in chunk 10h.

10g  $\langle \text{Delete the arguments and return a bad number error. 10g} \rangle \equiv$

```

    lval_del(args);
    return lval_err(LERR_BAD_NUM);

```

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

This code is used in chunk 10h.

### Evaluating built-in operations

Ensure all arguments are numbers.

10h  $\langle \text{Eval(uate) a built-in operation. 10h} \rangle \equiv$

```

     $\langle \text{For each argument 10e} \rangle \{$ 
        if ( $\langle \text{the argument is not a number 10f} \rangle$ ) {
             $\langle \text{Delete the arguments and return a bad number error. 10g} \rangle$ 
        }
    }

```

This definition is continued in chunks 11 and 13a.

This code is used in chunk 4a.

11a *⟨Pop the first element. 11a⟩*≡  
 lval\_pop(args, 0);

Uses lval\_pop 2c.

This code is used in chunks 11 and 13e.

Pop the first element.

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

Uses lval 17c.

If the operation is unary subtraction, negate the operand.

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

Uses strcmp 21f.

11d *⟨Pop the next element. 11d⟩*≡  
 lval \*y = *⟨Pop the first element. 11a⟩*

Uses lval 17c.

This code is used in chunk 11e.

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

If the op is "+", perform addition.

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

Uses strcmp 21f.

This definition is continued in chunks 11 and 12.

This code is used in chunk 11e.

If the op is "-", perform subtraction.

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

Uses strcmp 21f.

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

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

Uses strcmp 21f.

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

```
12a  <Perform a built-in operation. 11f>+≡
      else if (!strcmp(op, "/")) {
          if (!y->num) {
              lval_del(result);
              lval_del(y);
              result = lval_err(LERR_DIV_ZERO);
              break;
          }
          result->num /= y->num;
      }
```

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

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

```
12b  <Perform a built-in operation. 11f>+≡
      else if (!strcmp(op, "%")) {
          if (!y->num) {
              lval_del(result);
              lval_del(y);
              result = lval_err(LERR_DIV_ZERO);
              break;
          }
          result->num = fmod(result->num, y->num);
      }
```

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

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

```
12c  <Perform a built-in operation. 11f>+≡
      else if (!strcmp(op, "")) {
          result->num = pow(result->num, y->num);
      }
```

Uses `pow` 21e and `strcmp` 21f.

Otherwise, return a `LERR_BAD_OP` error.

```
12d  <Perform a built-in operation. 11f>+≡
      else {
          lval_del(result);
          lval_del(y);
          result = lval_err(LERR_BAD_OP);
          break;
      }
```

Uses `LERR_BAD_OP` 19a, `lval_del` 20b, and `lval_err` 19b.

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

```
12e  <Perform a built-in operation. 11f>+≡
      lval_del(y);
```

Uses `lval_del` 20b.

Delete the input expression and return the result.

13a  $\langle \text{Eval(uate) a built-in operation. 10h} \rangle + \equiv$   
`lval_del(args);`

`return result;`

Uses `lval_del` 20b.

### *Evaluating (S)-expressions*

If the expression is empty, return it;

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

This definition is continued in chunk 13.

This code is used in chunk 4c.

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

Uses `LVAL_ERR` 18a 18a and `lval_take` 2d.

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

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

Uses `lval_take` 2d.

13e  $\langle \text{Evaluate an S-expression. 13b} \rangle + \equiv$   
`lval *car =  $\langle \text{Pop the first element. 11a} \rangle$ ;`  
`if (car->type  $\neq$  LVAL_SYM) {`  
`lval_del(car);`  
`lval_del(args);`  
  
`return lval_err(LERR_BAD_SEXP);`  
`}`

Uses `LVAL_SYM` 18a, `lval` 17c, `lval_del` 20b, and `lval_err` 19b.

13f  $\langle \text{Evaluate an S-expression. 13b} \rangle + \equiv$   
`lval *result = builtin_op(car->sym, args);`  
`lval_del(car);`

`return result;`

Uses `lval` 17c and `lval_del` 20b.

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

14a  $\langle$ Evaluate an expression. 14a $\rangle \equiv$   
     if (val $\rightarrow$ type == LVAL\_SEXPR)  
         return lval\_eval\_sexpr(val);

    return val;

Uses LVAL\_SEXPR 18a.

This code is used in chunk 4d.

    Extract the element at index i.

14b  $\langle$ Extract an element and shift the list. 14b $\rangle \equiv$   
     lval \*elem = xs $\rightarrow$ cell[i];

Uses lval 17c.

This definition is continued in chunk 14.

This code is used in chunk 2c.

    Shift memory after the element at index i.

14c  $\langle$ Extract an element and shift the list. 14b $\rangle + \equiv$   
     memmove(&xs $\rightarrow$ cell[i], &xs $\rightarrow$ cell[i + 1],  
         sizeof(lval \*) \* (xs $\rightarrow$ count - i - 1));

Uses lval 17c.

    Decrease the count.

14d  $\langle$ Extract an element and shift the list. 14b $\rangle + \equiv$   
     xs $\rightarrow$ count-;

14e  $\langle$ Return the extracted element. 14e $\rangle \equiv$   
     return elem;

This code is used in chunk 14.

    Reallocate the memory used and return the extracted element.

14f  $\langle$ Extract an element and shift the list. 14b $\rangle + \equiv$   
      $\langle$ Reallocate the memory used. 10b $\rangle$   
  
      $\langle$ Return the extracted element. 14e $\rangle$

Describe this

14g  $\langle$ Pop the list then delete it. 14g $\rangle \equiv$   
     lval \*elem = lval\_pop(xs, i);  
     lval\_del(xs);

Uses lval 17c, lval\_del 20b, and lval\_pop 2c.

This definition is continued in chunk 14h.

This code is used in chunk 2d.

    Return the extracted element.

14h  $\langle$ Pop the list then delete it. 14g $\rangle + \equiv$   
      $\langle$ Return the extracted element. 14e $\rangle$

*P is for Print*

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

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

`mpc_ast_delete(ast);`

Uses `ast` 8e, `lval_println` 3e, and `mpc_ast_delete` 22.

This code is used in chunk 17a.

Describe this

Print the opening character.

15b *<Print an expression. 15b>*≡  
`putchar(open);`

This definition is continued in chunk 15.

This code is used in chunk 3c.

Print all but the last element with a trailing space.

15c *<Print an expression. 15b>*+≡  
`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.

15d *<Print an expression. 15b>*+≡  
`putchar(close);`

15e *<Print an error. 15e>*≡  
`switch (val->err) {`  
`case LERR_BAD_NUM:`  
`puts("Error: invalid number");`  
`break;`  
`case LERR_BAD_OP:`  
`puts("Error: invalid operator");`  
`break;`  
`case LERR_BAD_SEXPR:`  
`puts("Error: S-expression does not start with symbol");`  
`break;`  
`case LERR_DIV_ZERO:`  
`puts("Error: division by zero");`  
`break;`  
`}`

Uses `LERR_BAD_NUM` 19a, `LERR_BAD_OP` 19a, and `LERR_DIV_ZERO` 19a.

This code is used in chunk 3a.

16a *⟨Print a Lispy value. 16a⟩*≡

```

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

```

Uses `LVAL_ERR` 18a 18a, `LVAL_NUM` 18a, `LVAL_SEXPR` 18a, `LVAL_SYM` 18a, `lval_expr_print` 3c, `lval_print_err` 3a, and `printf` 21c.

This code is used in chunk 3d.

Print and delete the error upon failure.

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

```

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

```

Uses `mpc_err_delete` 22, `mpc_err_print` 22, and `parsed` 8c.

This code is used in chunk 17a.

## *L is for Loop*

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

```

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

```

Defines:

`nonempty`, used in chunk 17a.

Uses `bool` 21b.

This code is used in chunk 5.

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

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

```

⟨Read a line of user input. 7e⟩

```

This definition is continued in chunk 17.

This code is used in chunk 16c.



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

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

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

Uses **nonempty** 16c.

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

```
17b  <Read, eval(uate), and print. 16d>+≡
      free(input);
```

Uses **free** 21d and **input** 7e.

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

## Error Handling

Describe this struct

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

Defines:

**lval**, used in chunks 2–4, 8–11, 13, 14, and 18–20.

Uses **lval\_err\_t** 19a and **lval\_type\_t** 18a.

This definition is continued in chunks 18–20.

This code is used in chunk 2a.

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

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

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

Defines:

LVAL\_ERR, used in chunks 13c, 16a, 19b, and 20b.

LVAL\_NUM, used in chunks 16a, 18, and 20b.

LVAL\_SEXPR, used in chunks 14a, 16a, and 20.

LVAL\_SYM, used in chunks 13e, 16a, 19c, and 20b.

lval\_type\_t, used in chunk 17c.

This definition is continued in chunk 19a.

This code is used in chunk 2a.

Define a constructor for numbers.

18b *<Define the Lispy data structures. 17c>*+≡

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

    return val;
}
```

Defines:

lval\_num, used in chunk 9b.

Uses LVAL\_NUM 18a and lval 17c.

Define a convenient predicate for numbers.

18c *<Define the Lispy data structures. 17c>*+≡

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

Defines:

lval\_is\_num, used in chunk 10f.

Uses LVAL\_NUM 18a, bool 21b, and lval 17c.

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

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

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

Defines:

LERR\_BAD\_NUM, used in chunks 9b, 10g, and 15e.

LERR\_BAD\_OP, used in chunks 12d and 15e.

LERR\_DIV\_ZERO, used in chunks 12 and 15e.

lval\_err\_t, used in chunks 17c and 19b.

Define a constructor for errors.

19b *<Define the Lispy data structures. 17c>+≡*

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

    return val;
}
```

Defines:

lval\_err, used in chunks 9b, 10g, 12, and 13e.

Uses LVAL\_ERR 18a 18a, lval 17c, and lval\_err\_t 19a.

Define a constructor for symbol.

19c *<Define the Lispy data structures. 17c>+≡*

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

    return val;
}
```

Defines:

lval\_sym, used in chunk 9c.

Uses LVAL\_SYM 18a and lval 17c.

Define a constructor for an S-expression.

20a *(Define the Lispy data structures. 17c)+≡*

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

    return val;
}
```

Defines:

`lval_sexpr`, used in chunk 9.

Uses `LVAL_SEXPR` 18a and `lval` 17c.

Define a destructor for `lval*`.

20b *(Define the Lispy data structures. 17c)+≡*

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

    free(val);
}
```

Defines:

`lval_del`, used in chunks 10g and 12–14.

Uses `LVAL_ERR` 18a 18a, `LVAL_NUM` 18a, `LVAL_SEXPR` 18a, `LVAL_SYM` 18a, `free` 21d, and `lval` 17c.

*Headers*

## Describe headers

- 21a** *<Include the necessary headers. 21a>*≡  
*<Include the boolean type and values. 21b>*  
*<Include the standard I/O functions. 21c>*  
*<Include the standard library definitions. 21d>*  
*<Include some mathematical definitions. 21e>*  
*<Include some string operations. 21f>*  
  
*<Include the line editing functions from libedit. 21g>*  
*<Include the micro parser combinator definitions. 22>*  
 This code is used in chunk **2a**.
- 21b** *<Include the boolean type and values. 21b>*≡  
`#include <stdbool.h>`  
 Defines:  
`bool`, used in chunks **16c** and **18c**.  
 This code is used in chunk **21a**.
- 21c** *<Include the standard I/O functions. 21c>*≡  
`#include <stdio.h>`  
 Defines:  
`printf`, used in chunk **16a**.  
 This code is used in chunk **21a**.
- 21d** *<Include the standard library definitions. 21d>*≡  
`#include <stdlib.h>`  
 Defines:  
`free`, used in chunks **17b** and **20b**.  
`strtod`, used in chunk **9b**.  
 This code is used in chunk **21a**.
- 21e** *<Include some mathematical definitions. 21e>*≡  
`#include <math.h>`  
 Defines:  
`fmod`, used in chunk **12b**.  
`pow`, used in chunk **12c**.  
 This code is used in chunk **21a**.
- 21f** *<Include some string operations. 21f>*≡  
`#include <string.h>`  
 Defines:  
`strcmp`, used in chunks **9–12**.  
`strstr`, used in chunk **9**.  
 This code is used in chunk **21a**.
- 21g** *<Include the line editing functions from libedit. 21g>*≡  
`#include <editline/readline.h>`  
 Defines:  
`add_history`, used in chunk **8b**.  
`readline`, used in chunks **21g** and **7e**.  
 This code is used in chunk **21a**.

22 *(Include the micro parser combinator definitions. 22)*≡  
`#include <mpc.h>`

Defines:

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

This code is used in chunk 21a.

*Full Listings*

lispy.mpc:

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

## lispy.c:

```

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_SEXPR,
20     LVAL_SYM
21 } lval_type_t;
22
23
24 typedef enum {
25     LERR_DIV_ZERO,
26     LERR_BAD_OP,
27     LERR_BAD_NUM,
28     LERR_BAD_SEXPR
29 } lval_err_t;
30
31
32 typedef struct lval {
33     lval_type_t type;
34     union {
35         double num;
36         lval_err_t err;
37         char *sym;
38     };
39     int count;
40     struct lval **cell;
41 } lval;
42
43
44 lval *lval_num(double num)
45 {
46     lval *val = malloc(sizeof(lval));
47     val->type = LVAL_NUM;
48     val->num = num;
49
50     return val;

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

306     return errno != ERANGE ? lval_num(num) : lval_err(LERR_BAD_NUM);
307 }
308
309
310 lval *lval_read(mpc_ast_t * ast)
311 {
312     if (strstr(ast->tag, "number"))
313         return lval_read_num(ast);
314
315     if (strstr(ast->tag, "symbol"))
316         return lval_sym(ast->contents);
317
318     lval *sexpr = NULL;
319     if (!strcmp(ast->tag, ">"))
320         sexpr = lval_sexpr();
321     if (strstr(ast->tag, "sexpr"))
322         sexpr = lval_sexpr();
323
324     for (int i = 0; i < ast->children_num; i++) {
325         if (!strcmp(ast->children[i]->contents, "("))
326             continue;
327         if (!strcmp(ast->children[i]->contents, ")"))
328             continue;
329         if (!strcmp(ast->children[i]->tag, "regex"))
330             continue;
331         sexpr = lval_add(sexpr, lval_read(ast->children[i]));
332     }
333
334     return sexpr;
335 }
336
337
338 int main(int argc, char *argv[])
339 {
340     mpc_parser_t *Integer = mpc_new("integer");
341     mpc_parser_t *Float = mpc_new("float");
342     mpc_parser_t *Number = mpc_new("number");
343     mpc_parser_t *Symbol = mpc_new("symbol");
344     mpc_parser_t *Sexpr = mpc_new("sexpr");
345     mpc_parser_t *Expr = mpc_new("expr");
346     mpc_parser_t *Lispy = mpc_new("lispy");
347
348     mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
349              Integer, Float, Number, Symbol, Sexpr, Expr, Lispy);
350
351     puts("Lispy v1.0.1");
352     puts("Press ctrl-c to exit\n");
353
354     bool nonempty;
355     do {
356         char *input = readline("> ");

```

```

357     if ((nonempty = (input && *input))) {
358         add_history(input);
359
360         mpc_result_t parsed;
361         if (mpc_parse("<stdin>", input, Lispy, &parsed)) {
362             mpc_ast_t *ast = parsed.output;
363
364             lval *result = lval_eval(lval_read(ast));
365             lval_println(result);
366
367             mpc_ast_delete(ast);
368         } else {
369             mpc_err_print(parsed.error);
370             mpc_err_delete(parsed.error);
371         }
372     }
373
374     free(input);
375 } while (nonempty);
376
377 mpc_cleanup(7, Integer, Float, Number, Symbol, Sexpr, Expr, Lispy);
378
379 return 0;
380 }

```

*Chunks*

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



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

*Index*

Expr: [7a](#), [7b](#)  
 Float: [7a](#), [7b](#)  
 Integer: [7a](#), [7b](#)  
 LERR\_BAD\_NUM: [9b](#), [10g](#), [15e](#), [19a](#)  
 LERR\_BAD\_OP: [12d](#), [15e](#), [19a](#)  
 LERR\_DIV\_ZERO: [12a](#), [12b](#), [15e](#), [19a](#)  
 LISPY\_GRAMMAR: [6c](#), [7c](#)  
 LVAL\_ERR: [13c](#), [16a](#), [18a](#), [18a](#), [19b](#), [20b](#)  
 LVAL\_NUM: [16a](#), [18a](#), [18b](#), [18c](#), [20b](#)  
 LVAL\_SEXPR: [14a](#), [16a](#), [18a](#), [20a](#), [20b](#)  
 LVAL\_SYM: [13e](#), [16a](#), [18a](#), [19c](#), [20b](#)  
 Lispy: [6a](#), [7a](#), [7b](#), [8d](#)  
 Number: [7a](#), [7b](#)  
 Sexpr: [7a](#), [7b](#)  
 Symbol: [7a](#), [7b](#)  
 add\_history: [8b](#), [21g](#)  
 ast: [4e](#), [8e](#), [9a](#), [9b](#), [9c](#), [9f](#), [9g](#), [10a](#), [15a](#)  
 bool: [16c](#), [18c](#), [21b](#)  
 builtin\_binop: [4a](#)  
 fmod: [12b](#), [21e](#)  
 free: [17b](#), [20b](#), [21d](#)  
 input: [7e](#), [8a](#), [7e](#), [7e](#), [8b](#), [8d](#), [7e](#), [17b](#)  
 lval: [2b](#), [2c](#), [2d](#), [3a](#), [3b](#), [3c](#), [3d](#), [3e](#), [4a](#), [4b](#), [4c](#), [4d](#), [4e](#), [8e](#), [9e](#), [10b](#),  
     [11b](#), [11d](#), [13e](#), [13f](#), [14b](#), [14c](#), [14g](#), [17c](#), [18b](#), [18c](#), [19b](#), [19c](#), [20a](#), [20b](#)  
 lval\_add: [2b](#), [10a](#)  
 lval\_del: [10g](#), [12a](#), [12b](#), [12d](#), [12e](#), [13a](#), [13e](#), [13f](#), [14g](#), [20b](#)  
 lval\_err: [9b](#), [10g](#), [12a](#), [12b](#), [12d](#), [13e](#), [19b](#)  
 lval\_err\_t: [17c](#), [19a](#), [19b](#)  
 lval\_expr\_print: [3c](#), [3c](#), [16a](#)  
 lval\_is\_num: [10f](#), [18c](#)  
 lval\_num: [9b](#), [18b](#)  
 lval\_pop: [2c](#), [11a](#), [14g](#)  
 lval\_print: [3d](#), [3b](#), [3d](#), [3e](#), [15c](#)  
 lval\_print\_err: [3a](#), [16a](#)  
 lval\_println: [3e](#), [15a](#)  
 lval\_read: [4e](#), [8e](#), [10a](#)  
 lval\_sexpr: [9f](#), [9g](#), [20a](#)  
 lval\_sym: [9c](#), [19c](#)  
 lval\_take: [2d](#), [13c](#), [13d](#)  
 lval\_type\_t: [17c](#), [18a](#)  
 mpca\_lang: [7c](#), [22](#)  
 mpc\_ast\_delete: [15a](#), [22](#)

mpc\_ast\_print: [22](#)  
mpc\_ast\_t: [4e](#), [8e](#), [22](#)  
mpc\_cleanup: [22](#), [7d](#), [22](#)  
mpc\_err\_delete: [16b](#), [22](#)  
mpc\_err\_print: [16b](#), [22](#)  
mpc\_new: [7a](#), [22](#)  
mpc\_parse: [22](#), [8d](#), [22](#)  
mpc\_parser\_t: [7a](#), [22](#)  
mpc\_result\_t: [8c](#), [22](#)  
nonempty: [16c](#), [17a](#)  
parsed: [8c](#), [8c](#), [8d](#), [8e](#), [16b](#)  
pow: [12c](#), [21e](#)  
printf: [16a](#), [21c](#)  
readline: [21g](#), [7e](#), [21g](#)  
strcmp: [9f](#), [10a](#), [11c](#), [11f](#), [11g](#), [11h](#), [12a](#), [12b](#), [12c](#), [21f](#)  
strstr: [9a](#), [9c](#), [9g](#), [21f](#)  
strtod: [9b](#), [21d](#)

## *Glossary*

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

*grammar* [6](#), [7](#)  Describe what a grammar is

*parser* [7](#)  Describe what a parser is

*PLT* programming language theory, [1](#)  Describe programming language theory

*REPL* Read-Eval-Print Loop, [6](#), [7](#)  Describe what a REPL is

*References*

- Daniel Holden. Build Your Own Lisp. <http://buildyourownlisp.com>, 2018a. Accessed: 2018-05-13.
- Daniel Holden. Micro Parser Combinators. <https://github.com/orangeduck/mpc>, 2018b. Accessed: 2018-05-13.
- Norman Ramsey. Noweb – a simple, extensible tool for literate programming. <https://www.cs.tufts.edu/~nr/noweb/>, 2012. Accessed: 2018-05-13.
- Jess Thrysoee. Editline Library (libedit) – port of netbsd command line editor library. <http://thrysoee.dk/editline/>, 2017. Accessed: 2018-05-13.

*Todo list*

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 . . . . .	14
Describe this . . . . .	15
Describe this struct . . . . .	17
Describe headers . . . . .	21
Describe what a grammar is . . . . .	36
Describe what a parser is . . . . .	36
Describe programming language theory . . . . .	36
Describe what a REPL is . . . . .	36