

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>	7
<i>E is for Eval(uate)</i>	8
<i>Evaluating built-in operations</i>	10
<i>Built-in functions</i>	13
<i>Evaluating (S)-expressions</i>	14
<i>P is for Print</i>	16
<i>L is for Loop</i>	17
<i>Error Handling</i>	18
<i>Headers</i>	22
<i>Full Listings</i>	24
<i>Chunks</i>	34
<i>Index</i>	36
<i>Glossary</i>	38
<i>References</i>	39

Outline

Describe the outline

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

 $\langle \text{Define some useful macros. } 20a \rangle$

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

 $\langle \text{Define possible lval and error types. } 19a \rangle$

 $\langle \text{Define the Lispy data structures. } 18c \rangle$

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)`
`{`
 $\langle \text{Add an element to an S-expression. } 10b \rangle$

`return xs;`
`}`

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

2c $\langle \text{lispy.c } 2a \rangle + \equiv$
`lval *lval_pop(lval *xs, int i)`
`{`
 $\langle \text{Extract an element and shift the list. } 15d \rangle$
`}`

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

```

3a  <lisp.c 2a>+≡
    lval *lval_take(lval *xs, int i)
    {
        <Pop the list then delete it. 16b>
    }

```

Defines:

 lval_take, used in chunks 13 and 14.

Uses lval 18c.

Forward 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 18c and lval_print 3d.

```

3c  <lisp.c 2a>+≡
    void lval_expr_print(lval *expr, char open, char close)
    {
        <Print an expression. 16e>
    }

```

Defines:

 lval_expr_print, used in chunks 3c and 17a.

Uses lval 18c.

```

3d  <lisp.c 2a>+≡
    void lval_print(lval *val)
    {
        <Print a Lispy value. 17a>
    }

```

Defines:

 lval_print, used in chunks 3 and 16f.

Uses lval 18c.

```

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

```

Defines:

 lval_println, used in chunk 16d.

Uses lval 18c and lval_print 3d.

4a $\langle \text{lisp.c } 2a \rangle + \equiv$
`lval *builtin_list(lval *args)`
`{`
 \langle Convert an S-expression to a Q-expression. 13c \rangle
`}`

Defines:

builtin_list, used in chunk 13b.

Uses lval 18c.

4b $\langle \text{lisp.c } 2a \rangle + \equiv$
`lval *builtin_head(lval *args)`
`{`
 \langle Pop the list and delete the rest. 13e \rangle
`}`

Defines:

builtin_head, used in chunk 13d.

Uses lval 18c.

4c $\langle \text{lisp.c } 2a \rangle + \equiv$
`lval *builtin_op(char *op, lval *args)`
`{`
 \langle Eval(uate) a built-in operation. 10g \rangle
`}`

Defines:

builtin_binop, never used.

Uses lval 18c.

4d $\langle \text{lisp.c } 2a \rangle + \equiv$
`lval *builtin(char *fname, lval *args)`
`{`
 \langle Evaluate a built-in function or operation. 13b \rangle
`}`

Defines:

builtin, used in chunk 15b.

Uses lval 18c.

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

4e $\langle \text{lisp.c } 2a \rangle + \equiv$
`lval *lval_eval(lval* val);`

Uses lval 18c.

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

Uses lval 18c.

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

Uses lval 18c.

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

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

Defines:

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

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

 <Print version and exit information. 6a>

 <Loop until the input is empty. 17c>

 <Undefine and delete the parsers. 7c>

 return 0;
 }

Welcome

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

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

Uses Lispy 6d.

This code is used in chunk 5d.

Defining the Language

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

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

Root chunk (not used in this document).

Describe this trick

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

Defines:

LISPY_GRAMMAR, used in chunk 7b.

This code is used in chunk 2a.

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

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

6d `<Define the language. 6d>≡`
`mpc_parser_t *Number = mpc_new("number");`
`mpc_parser_t *Symbol = mpc_new("symbol");`
`mpc_parser_t *Sexpr = mpc_new("sexpr");`
`mpc_parser_t *Qexpr = mpc_new("qexpr");`
`mpc_parser_t *Expr = mpc_new("expr");`
`mpc_parser_t *Lispy = mpc_new("lispy");`

Defines:

Expr, used in chunk 7a.

Lispy, used in chunks 6–8.

Number, used in chunk 7a.

Qexpr, used in chunk 7a.

Sexpr, used in chunk 7a.

Symbol, used in chunk 7a.

Uses `mpc_new` 23f and `mpc_parser_t` 23f.

This definition is continued in chunk 7b.

This code is used in chunk 5d.

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

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

... we can define the Lispy language.

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

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

Uses LISPY_GRAMMAR **6c**, mpc_lang **23f**, mpc_err_delete **23f**, and mpc_err_print **23f**.

Since we're implementing this in C, we need to clean up after ourselves. The `mpc`⁶ library makes this easy, by providing the `mpc_cleanup` function.

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

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.

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

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.

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

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

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

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

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

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

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

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

Defines:

`parsed`, used in chunks 8 and 17b.

Uses `mpc_result_t` 23f.

This code is used in chunk 18a.

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

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

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

E is for Eval(uate)

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

8d *<Eval(uate) the input. 8d>*≡
`mpc_ast_t *ast = parsed.output;`

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

Defines:

`ast`, used in chunks 5c, 8, 9, and 16d.

Uses `lval` 18c, `lval_read` 5c, `mpc_ast_t` 23f, and `parsed` 8b.

This code is used in chunk 18a.

Describe the evaluation strategy

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

8e *<Read a Lispy value. 8e>*≡
`if (strstr(ast->tag, "number"))
 return lval_read_num(ast);`

Uses `ast` 8d and `strstr` 23d.

This definition is continued in chunks 9 and 10c.

This code is used in chunk 5c.

Describe this

9a *<Read a number. 9a>*≡
 errno = 0;
 double num = strtod(ast→contents, NULL);
 return errno ≠ ERANGE ? lval_num(num) : lval_err(LERR_BAD_NUM);

Uses ast 8d, lval_err 20c, lval_num 19b, and strtod 23b.

This code is used in chunk 5c.

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

9b *<Read a Lispy value. 8e>*+≡
 if (strstr(ast→tag, "symbol"))
 return lval_sym(ast→contents);

Uses ast 8d, lval_sym 20d, and strstr 23d.

Describe this

9c *<Read a Lispy value. 8e>*+≡
 lval *val = NULL;

Uses lval 18c.

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

9d *<Read a Lispy value. 8e>*+≡
 if (!strcmp(ast→tag, ">"))
 val = lval_sexpr();

Uses ast 8d, lval_sexpr 21a, and strcmp 23d.

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

9e *<Read a Lispy value. 8e>*+≡
 if (strstr(ast→tag, "qexpr"))
 val = lval_qexpr();

Uses ast 8d, lval_qexpr 21b, and strstr 23d.

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

9f *<Read a Lispy value. 8e>*+≡
 if (strstr(ast→tag, "sexpr"))
 val = lval_sexpr();

Uses ast 8d, lval_sexpr 21a, and strstr 23d.

Describe this

9g *<Read a Lispy value. 8e>*+≡
 for (int i = 0; i < ast→children_num; i++) {
 if (!strcmp(ast→children[i]→contents, "(")) continue;
 if (!strcmp(ast→children[i]→contents, ")")) continue;
 if (!strcmp(ast→children[i]→contents, "{")) continue;
 if (!strcmp(ast→children[i]→contents, "}")") continue;
 if (!strcmp(ast→children[i]→tag, "regex")) continue;
 val = lval_add(val, lval_read(ast→children[i]));
 }

Uses ast 8d, lval_add 2b, lval_read 5c, and strcmp 23d.

10a $\langle \text{Reallocate the memory used. 10a} \rangle \equiv$
`xs->cell = realloc(xs->cell, sizeof(lval *) * xs->count);`

Uses `lval 18c`.

This code is used in chunks 10b and 16a.

Describe this, incl. how it's not
cons

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

This code is used in chunk 2b.

Finally, return the Lispy value.

10c $\langle \text{Read a Lispy value. 8e} \rangle + \equiv$
`return val;`

10d $\langle \text{For each argument 10d} \rangle \equiv$
`for (int i = 0; i < args->count; i++)`

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

10e $\langle \text{the argument is not a number 10e} \rangle \equiv$
`!lval_is_num(args->cell[i])`

Uses `lval_is_num 19c`.

This code is used in chunk 10g.

10f $\langle \text{Delete the arguments and return a bad number error. 10f} \rangle \equiv$
`lval_del(args);`
`return lval_err(LERR_BAD_NUM);`

Uses `lval_del 22a` and `lval_err 20c`.

This code is used in chunk 10g.

Evaluating built-in operations

Ensure all arguments are numbers.

10g $\langle \text{Eval(uate) a built-in operation. 10g} \rangle \equiv$
 $\langle \text{For each argument 10d} \rangle \{$
`if ((the argument is not a number 10e)) {`
 $\quad \langle \text{Delete the arguments and return a bad number error. 10f} \rangle$
`}`
`}`

This definition is continued in chunks 11 and 13a.

This code is used in chunk 4c.

10h $\langle \text{Pop the first element. 10h} \rangle \equiv$
`lval_pop(args, 0);`

Uses `lval_pop 2c`.

This code is used in chunks 11 and 15a.

Pop the first element.

11a $\langle \text{Eval}(\text{uate}) \text{ a built-in operation. 10g} \rangle + \equiv$
`lval *result = $\langle \text{Pop the first element. 10h} \rangle$`

Uses lval 18c.

If the operation is unary subtraction, negate the operand.

11b $\langle \text{Eval}(\text{uate}) \text{ a built-in operation. 10g} \rangle + \equiv$
`if (!strcmp(op, "-") && !args->count)
 result->num = -result->num;`

Uses strcmp 23d.

11c $\langle \text{Pop the next element. 11c} \rangle \equiv$
`lval *y = $\langle \text{Pop the first element. 10h} \rangle$`

Uses lval 18c.

This code is used in chunk 11d.

11d $\langle \text{Eval}(\text{uate}) \text{ a built-in operation. 10g} \rangle + \equiv$
`while (args->count > 0) {
 $\langle \text{Pop the next element. 11c} \rangle$

 $\langle \text{Perform a built-in operation. 11e} \rangle$
}`

If the op is "+", perform addition.

11e $\langle \text{Perform a built-in operation. 11e} \rangle \equiv$
`if (!strcmp(op, "+")) {
 result->num += y->num;
}`

Uses strcmp 23d.

This definition is continued in chunks 11 and 12.

This code is used in chunk 11d.

If the op is "-", perform subtraction.

11f $\langle \text{Perform a built-in operation. 11e} \rangle + \equiv$
`else if (!strcmp(op, "-")) {
 result->num -= y->num;
}`

Uses strcmp 23d.

If the op is "*", perform multiplication.

11g $\langle \text{Perform a built-in operation. 11e} \rangle + \equiv$
`else if (!strcmp(op, "*")) {
 result->num *= y->num;
}`

Uses strcmp 23d.

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

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

Uses `lval_del` 22a, `lval_err` 20c, and `strcmp` 23d.

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

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

Uses `fmod` 23c, `lval_del` 22a, `lval_err` 20c, and `strcmp` 23d.

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

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

Uses `pow` 23c and `strcmp` 23d.

Otherwise, return a `LERR_BAD_OP` error.

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

Uses `lval_del` 22a and `lval_err` 20c.

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

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

Uses `lval_del` 22a.

Delete the input expression and return the result.

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

`return result;`

Uses `lval_del` 22a.

Built-in functions

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

13b $\langle \text{Evaluate a built-in function or operation. 13b} \rangle \equiv$
`if (!strcmp("list", fname))`
`return builtin_list(args);`

Uses `builtin_list` 4a and `strcmp` 23d.

This definition is continued in chunks 13 and 14.

This code is used in chunk 4d.

13c $\langle \text{Convert an S-expression to a Q-expression. 13c} \rangle \equiv$
`args->type = LVAL_QEXPR;`
`return args;`

Uses `LVAL_QEXPR` 19a.

This code is used in chunk 4a.

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

13d $\langle \text{Evaluate a built-in function or operation. 13b} \rangle + \equiv$
`if (!strcmp("head", fname))`
`return builtin_head(args);`

Uses `builtin_head` 4b and `strcmp` 23d.

Ensure there is exactly one argument.

13e $\langle \text{Pop the list and delete the rest. 13e} \rangle \equiv$
`LVAL_ASSERT(args, args->count == 1,`
`"too many arguments for 'head'");`

This definition is continued in chunks 13 and 14.

This code is used in chunk 4b.

Ensure the first argument is a Q-expression.

13f $\langle \text{Pop the list and delete the rest. 13e} \rangle + \equiv$
`LVAL_ASSERT(args, args->cell[0]->type == LVAL_QEXPR,`
`"invalid argument for 'head'");`

Uses `LVAL_QEXPR` 19a.

Ensure the list passed to `head` is nonempty.

13g $\langle \text{Pop the list and delete the rest. 13e} \rangle + \equiv$
`LVAL_ASSERT(args, args->cell[0]->count,`
`"cannot get 'head' of the empty list");`

eTake the first element of the list.

13h $\langle \text{Pop the list and delete the rest. 13e} \rangle + \equiv$
`lval *val = lval_take(args, 0);`

Uses `lval` 18c and `lval_take` 3a.

Delete the rest.

14a $\langle \text{Pop the list and delete the rest. 13e} \rangle + \equiv$
 while (val→count > 1)
 lval_del(lval_pop(val, 1));

Uses lval_del 22a and lval_pop 2c.

Return the head of the list.

14b $\langle \text{Pop the list and delete the rest. 13e} \rangle + \equiv$
 return val;

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

14c $\langle \text{Evaluate a built-in function or operation. 13b} \rangle + \equiv$
 if (strstr("+-/*%", fname))
 return builtin_op(fname, args);

Uses strstr 23d.

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

14d $\langle \text{Evaluate a built-in function or operation. 13b} \rangle + \equiv$
 lval_del(args);

 return lval_err(LERR_BAD_FUNC);

Uses lval_del 22a and lval_err 20c.

Evaluating (S)-expressions

If the expression is empty, return it;

14e $\langle \text{Evaluate an S-expression. 14e} \rangle \equiv$
 if (!args→count)
 return args;

This definition is continued in chunks 14 and 15.

This code is used in chunk 5a.

14f $\langle \text{Evaluate an S-expression. 14e} \rangle + \equiv$
 $\langle \text{For each argument 10d} \rangle \{$
 args→cell[i] = lval_eval(args→cell[i]);
 if (args→cell[i]→type == LVAL_ERR)
 return lval_take(args, i);
 }

Uses LVAL_ERR 19a 19a and lval_take 3a.

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

14g $\langle \text{Evaluate an S-expression. 14e} \rangle + \equiv$
 if (args→count == 1)
 return lval_take(args, 0);

Uses lval_take 3a.

```

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

          return lval_err(LERR_BAD_SEXPR);
      }

```

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

```

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

      return result;

```

Uses builtin 4d, lval 18c, and lval_del 22a.

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

```

15c  <Evaluate an expression. 15c>≡
      if (val->type = LVAL_SEXPR)
          return lval_eval_sexpr(val);

      return val;

```

Uses LVAL_SEXPR 19a.

This code is used in chunk 5b.

Extract the element at index *i*.

```

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

```

Uses lval 18c.

This definition is continued in chunks 15 and 16a.

This code is used in chunk 2c.

Shift memory after the element at index *i*.

```

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

```

Uses lval 18c.

Decrease the count.

```

15f  <Extract an element and shift the list. 15d>+≡
      xs->count--;

```

```

15g  <Return the extracted element. 15g>≡
      return elem;

```

This code is used in chunk 16.

Reallocate the memory used and return the extracted element.

16a \langle Extract an element and shift the list. 15d $\rangle + \equiv$
 \langle Reallocate the memory used. 10a \rangle

\langle Return the extracted element. 15g \rangle

Describe this

16b \langle Pop the list then delete it. 16b $\rangle \equiv$
`lval *elem = lval_pop(xs, i);`
`lval_del(xs);`

Uses `lval` 18c, `lval_del` 22a, and `lval_pop` 2c.

This definition is continued in chunk 16c.

This code is used in chunk 3a.

Return the extracted element.

16c \langle Pop the list then delete it. 16b $\rangle + \equiv$
 \langle Return the extracted element. 15g \rangle

P is for Print

Upon success, print the result and delete the AST.

16d \langle Print the result and delete the AST. 16d $\rangle \equiv$
`lval_println(result);`

`mpc_ast_delete(ast);`

Uses `ast` 8d, `lval_println` 3e, and `mpc_ast_delete` 23f.

This code is used in chunk 18a.

Describe this

Print the opening character.

16e \langle Print an expression. 16e $\rangle \equiv$
`putchar(open);`

This definition is continued in chunk 16.

This code is used in chunk 3c.

Print all but the last element with a trailing space.

16f \langle Print an expression. 16e $\rangle + \equiv$
`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.

16g \langle Print an expression. 16e $\rangle + \equiv$
`putchar(close);`

17a *⟨Print a Lispy value. 17a⟩*≡

```

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

```

Uses LVAL_ERR 19a 19a, LVAL_NUM 19a, LVAL_QEXPR 19a, LVAL_SEXPR 19a, LVAL_SYM 19a, lval_expr_print 3c, and printf 23a.

This code is used in chunk 3d.

Print and delete the error upon failure.

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

```

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

```

Uses mpc_err_delete 23f, mpc_err_print 23f, and parsed 8b.

This code is used in chunk 18a.

L is for Loop

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

```

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

```

Defines:

nonempty, used in chunk 18a.

Uses bool 22c.

This code is used in chunk 5d.

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

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

```

⟨Read a line of user input. 7d⟩

```

This definition is continued in chunk 18.

This code is used in chunk 17c.

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

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

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

Uses **nonempty** 17c.

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

```
18b  <Read, eval(uate), and print. 17d>+≡
      free(input);
```

Uses **free** 23b and **input** 7d.

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

Error Handling

Describe this struct

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

Defines:

lval, used in chunks 2–5, 8–11, 13h, 15, 16b, and 19–22.

Uses **lval_type_t** 19a.

This definition is continued in chunks 19–22.

This code is used in chunk 2a.

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

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

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

Defines:

LVAL_ERR, used in chunks 14f, 17a, 20c, and 22a.

LVAL_NUM, used in chunks 17a, 19, and 22a.

LVAL_QEXPR, used in chunks 13, 17a, 21b, and 22a.

LVAL_SEXPR, used in chunks 15c, 17a, 21a, and 22a.

LVAL_SYM, used in chunks 15a, 17a, 20d, and 22a.

lval_type_t, used in chunk 18c.

This code is used in chunk 2a.

Define a constructor for numbers.

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

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

    return val;
}
```

Defines:

lval_num, used in chunk 9a.

Uses LVAL_NUM 19a and lval 18c.

Define a convenient predicate for numbers.

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

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

Defines:

lval_is_num, used in chunk 10e.

Uses LVAL_NUM 19a, bool 22c, and lval 18c.

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

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

Uses `lval_del` 22a and `lval_err` 20c.

This definition is continued in chunk 20b.

This code is used in chunk 2a.

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

Define a constructor for errors.

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

        return val;
    }
```

Defines:

`lval_err`, used in chunks 9a, 10f, 12, 14d, 15a, and 20a.

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

Define a constructor for symbol.

```
20d <Define the Lispy data structures. 18c>+≡
    lval *lval_sym(char *s)
    {
        lval *val = malloc(sizeof(lval));
        val->type = LVAL_SYM;
        val->sym = malloc(strlen(s) + 1);
        strcpy(val->sym, s);

        return val;
    }
```

Defines:

`lval_sym`, used in chunk 9b.

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

Define a constructor for an S-expression.

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

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

    return val;
}
```

Defines:

`lval_sexpr`, used in chunk 9.

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

Define a constructor for a Q-expression.

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

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

    return val;
}
```

Defines:

`lval_qexpr`, used in chunk 9e.

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

Define a destructor for `lval*`.

22a *<Define the Lispy data structures. 18c>+≡*

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

    free(val);
}
```

Defines:

`lval_del`, used in chunks 10f, 12–16, and 20a.

Uses `LVAL_ERR` 19a 19a, `LVAL_NUM` 19a, `LVAL_QEXPR` 19a, `LVAL_SEXPR` 19a, `LVAL_SYM` 19a, `free` 23b, and `lval` 18c.

Headers

Describe headers

22b *<Include the necessary headers. 22b>≡*

```
<Include the boolean type and values. 22c>
<Include the standard I/O functions. 23a>
<Include the standard library definitions. 23b>
<Include some mathematical definitions. 23c>
<Include some string operations. 23d>

<Include the line editing functions from libedit. 23e>
<Include the micro parser combinator definitions. 23f>
```

This code is used in chunk 2a.

22c *<Include the boolean type and values. 22c>≡*

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

Defines:

`bool`, used in chunks 17c and 19c.

This code is used in chunk 22b.

23a *<Include the standard I/O functions. 23a>*≡
`#include <stdio.h>`

Defines:

`printf`, used in chunk 17a.

This code is used in chunk 22b.

23b *<Include the standard library definitions. 23b>*≡
`#include <stdlib.h>`

Defines:

`free`, used in chunks 18b and 22a.

`strtod`, used in chunk 9a.

This code is used in chunk 22b.

23c *<Include some mathematical definitions. 23c>*≡
`#include <math.h>`

Defines:

`fmod`, used in chunk 12b.

`pow`, used in chunk 12c.

This code is used in chunk 22b.

23d *<Include some string operations. 23d>*≡
`#include <string.h>`

Defines:

`strcmp`, used in chunks 9 and 11–13.

`strstr`, used in chunks 8, 9, and 14c.

This code is used in chunk 22b.

23e *<Include the line editing functions from libedit. 23e>*≡
`#include <editline/readline.h>`

Defines:

`add_history`, used in chunk 8a.

`readline`, used in chunks 23e and 7d.

This code is used in chunk 22b.

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

Defines:

`mpca_lang`, used in chunk 7b.

`mpc_ast_delete`, used in chunk 16d.

`mpc_ast_print`, never used.

`mpc_ast_t`, used in chunks 5c and 8d.

`mpc_cleanup`, used in chunks 23f and 7c.

`mpc_err_delete`, used in chunks 7b and 17b.

`mpc_err_print`, used in chunks 7b and 17b.

`mpc_new`, used in chunk 6d.

`mpc_parse`, used in chunks 23f and 8c.

`mpc_parser_t`, used in chunk 6d.

`mpc_result_t`, used in chunk 8b.

This code is used in chunk 22b.

*Full Listings**lispy.mpc:*

```

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

```


lispy.c:

```

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 void lval_print(lval * val);
168
169
170 void lval_expr_print(lval * expr, char open, char close)
171 {
172     putchar(open);
173     for (int i = 0; i < expr->count; i++) {
174         lval_print(expr->cell[i]);
175         if (i != (expr->count - 1))
176             putchar(' ');
177     }
178     putchar(close);
179 }
180
181
182 void lval_print(lval * val)
183 {
184     switch (val->type) {
185     case LVAL_ERR:
186         printf("Error: %s", val->err);
187         break;
188     case LVAL_NUM:
189         printf("%g", val->num);
190         break;
191     case LVAL_QEXPR:
192         lval_expr_print(val, '{', '}');
193         break;
194     case LVAL_SEXPR:
195         lval_expr_print(val, '(', ')');
196         break;
197     case LVAL_SYM:
198         fputs(val->sym, stdout);
199         break;
200     }
201 }
202
203

```

```

204 void lval_println(lval * val)
205 {
206     lval_print(val);
207     putchar('\n');
208 }
209
210
211 lval *builtin_list(lval * args)
212 {
213     args->type = LVAL_QEXPR;
214     return args;
215 }
216
217
218 lval *builtin_head(lval * args)
219 {
220     LVAL_ASSERT(args, args->count == 1, "too many arguments for 'head'");
221     LVAL_ASSERT(args, args->cell[0]->type == LVAL_QEXPR,
222                 "invalid argument for 'head'");
223     LVAL_ASSERT(args, args->cell[0]->count,
224                 "cannot get 'head' of the empty list");
225     lval *val = lval_take(args, 0);
226     while (val->count > 1)
227         lval_del(lval_pop(val, 1));
228     return val;
229 }
230
231
232 lval *builtin_op(char *op, lval * args)
233 {
234     for (int i = 0; i < args->count; i++) {
235         if (!lval_is_num(args->cell[i])) {
236             lval_del(args);
237             return lval_err(LERR_BAD_NUM);
238         }
239     }
240
241     lval *result = lval_pop(args, 0);
242
243     if (!strcmp(op, "-") && !args->count)
244         result->num = -result->num;
245
246     while (args->count > 0) {
247         lval *y = lval_pop(args, 0);
248
249         if (!strcmp(op, "+")) {
250             result->num += y->num;
251         } else if (!strcmp(op, "-")) {
252             result->num -= y->num;
253         } else if (!strcmp(op, "*")) {
254             result->num *= y->num;

```

```

255     } else if (!strcmp(op, "/")) {
256         if (!y->num) {
257             lval_del(result);
258             lval_del(y);
259             result = lval_err(LERR_DIV_ZERO);
260             break;
261         }
262         result->num /= y->num;
263     } else if (!strcmp(op, "%")) {
264         if (!y->num) {
265             lval_del(result);
266             lval_del(y);
267             result = lval_err(LERR_DIV_ZERO);
268             break;
269         }
270         result->num = fmod(result->num, y->num);
271     } else if (!strcmp(op, "^")) {
272         result->num = pow(result->num, y->num);
273     } else {
274         lval_del(result);
275         lval_del(y);
276         result = lval_err(LERR_BAD_OP);
277         break;
278     }
279     lval_del(y);
280 }
281
282 lval_del(args);
283
284 return result;
285 }
286
287
288 lval *builtin(char *fname, lval * args)
289 {
290     if (!strcmp("list", fname))
291         return builtin_list(args);
292
293     if (!strcmp("head", fname))
294         return builtin_head(args);
295     if (strstr("+-/*%", fname))
296         return builtin_op(fname, args);
297
298     lval_del(args);
299
300     return lval_err(LERR_BAD_FUNC);
301 }
302
303 lval *lval_eval(lval * val);
304
305

```

```

306 lval *lval_eval_sexpr(lval * args)
307 {
308     if (!args->count)
309         return args;
310     for (int i = 0; i < args->count; i++) {
311         args->cell[i] = lval_eval(args->cell[i]);
312         if (args->cell[i]->type == LVAL_ERR)
313             return lval_take(args, i);
314     }
315
316     if (args->count == 1)
317         return lval_take(args, 0);
318
319     lval *car = lval_pop(args, 0);
320     if (car->type != LVAL_SYM) {
321         lval_del(car);
322         lval_del(args);
323
324         return lval_err(LERR_BAD_SEXPR);
325     }
326
327     lval *result = builtin(car->sym, args);
328     lval_del(car);
329
330     return result;
331 }
332
333 lval *lval_eval(lval * val)
334 {
335     if (val->type == LVAL_SEXPR)
336         return lval_eval_sexpr(val);
337
338     return val;
339 }
340
341
342 lval *lval_read_num(mpc_ast_t * ast)
343 {
344     errno = 0;
345     double num = strtod(ast->contents, NULL);
346     return errno != ERANGE ? lval_num(num) : lval_err(LERR_BAD_NUM);
347 }
348
349
350 lval *lval_read(mpc_ast_t * ast)
351 {
352     if (strstr(ast->tag, "number"))
353         return lval_read_num(ast);
354
355     if (strstr(ast->tag, "symbol"))

```

```

357         return lval_sym(ast->contents);
358
359     lval *val = NULL;
360     if (!strcmp(ast->tag, ">"))
361         val = lval_sexpr();
362     if (strstr(ast->tag, "qexpr"))
363         val = lval_qexpr();
364     if (strstr(ast->tag, "sexpr"))
365         val = lval_sexpr();
366
367     for (int i = 0; i < ast->children_num; i++) {
368         if (!strcmp(ast->children[i]->contents, "("))
369             continue;
370         if (!strcmp(ast->children[i]->contents, ")"))
371             continue;
372         if (!strcmp(ast->children[i]->contents, "{"))
373             continue;
374         if (!strcmp(ast->children[i]->contents, "}")
375             continue;
376         if (!strcmp(ast->children[i]->tag, "regex"))
377             continue;
378         val = lval_add(val, lval_read(ast->children[i]));
379     }
380
381     return val;
382 }
383
384
385 int main(int argc, char *argv[])
386 {
387     mpc_parser_t *Number = mpc_new("number");
388     mpc_parser_t *Symbol = mpc_new("symbol");
389     mpc_parser_t *Sexpr = mpc_new("sexpr");
390     mpc_parser_t *Qexpr = mpc_new("qexpr");
391     mpc_parser_t *Expr = mpc_new("expr");
392     mpc_parser_t *Lispy = mpc_new("lispy");
393
394     mpc_err_t *err = mpca_lang(MPCA_LANG_PREDICTIVE, LISPY_GRAMMAR,
395                               Number, Symbol, Sexpr, Qexpr, Expr, Lispy);
396
397     if (err != NULL) {
398         puts(LISPY_GRAMMAR);
399         mpc_err_print(err);
400         mpc_err_delete(err);
401         exit(100);
402     }
403
404     puts("Lispy v1.1.1");
405     puts("Press ctrl-c to exit\n");
406
407     bool nonempty;

```



```

408 do {
409     char *input = readline("> ");
410     if ((nonempty = (input && *input))) {
411         add_history(input);
412
413         mpc_result_t parsed;
414         if (mpc_parse("<stdin>", input, Lispy, &parsed)) {
415             mpc_ast_t *ast = parsed.output;
416
417             lval *result = lval_eval(lval_read(ast));
418             lval_println(result);
419
420             mpc_ast_delete(ast);
421         } else {
422             mpc_err_print(parsed.error);
423             mpc_err_delete(parsed.error);
424         }
425     }
426
427     free(input);
428 } while (nonempty);
429
430 mpc_cleanup(6, Number, Symbol, Sexpr, Qexpr, Expr, Lispy);
431
432 return 0;
433 }

```

Chunks

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

⟨*Reallocate the memory used.* 10a⟩ [10a](#), [10b](#), [16a](#)
 ⟨*Return the extracted element.* 15g⟩ [15g](#), [16a](#), [16c](#)
 ⟨*Undefine and delete the parsers.* 7c⟩ [5d](#), [7c](#)
 ⟨*created parsers* 7a⟩ [7a](#), [7b](#), [7c](#)
 ⟨*input is nonempty* 7e⟩ [7e](#), [18a](#)
 ⟨*lispy.c* 2a⟩ [2a](#), [2b](#), [2c](#), [3a](#), [3b](#), [3c](#), [3d](#), [3e](#), [4a](#), [4b](#), [4c](#), [4d](#), [4e](#), [5a](#), [5b](#), [5c](#),
[5d](#)
 ⟨*lispy.mpc* 6b⟩ [6b](#)
 ⟨*the argument is not a number* 10e⟩ [10e](#), [10g](#)
 ⟨*the input can be parsed as Lispy code* 8c⟩ [8c](#), [18a](#)

Index

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

mpc_ast_delete: [16d](#), [23f](#)
mpc_ast_print: [23f](#)
mpc_ast_t: [5c](#), [8d](#), [23f](#)
mpc_cleanup: [23f](#), [7c](#), [23f](#)
mpc_err_delete: [7b](#), [17b](#), [23f](#)
mpc_err_print: [7b](#), [17b](#), [23f](#)
mpc_new: [6d](#), [23f](#)
mpc_parse: [23f](#), [8c](#), [23f](#)
mpc_parser_t: [6d](#), [23f](#)
mpc_result_t: [8b](#), [23f](#)
nonempty: [17c](#), [18a](#)
parsed: [8b](#), [8b](#), [8c](#), [8d](#), [17b](#)
pow: [12c](#), [23c](#)
printf: [17a](#), [23a](#)
readline: [23e](#), [7d](#), [23e](#)
strcmp: [9d](#), [9g](#), [11b](#), [11e](#), [11f](#), [11g](#), [12a](#), [12b](#), [12c](#), [13b](#), [13d](#), [23d](#)
strstr: [8e](#), [9b](#), [9e](#), [9f](#), [14c](#), [23d](#)
strtod: [9a](#), [23b](#)

Glossary

AST abstract syntax tree, a tree representation of the abstract syntactic structure of source code. [8](#), [9](#), [16](#)

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

Describe what a grammar is

parser [6](#)

Describe what a parser is

PLT programming language theory, [1](#)

Describe programming language theory

REPL Read-Eval-Print Loop, [6](#), [7](#)

Describe what a REPL is

References

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

Todo list

<input type="checkbox"/> Describe the outline	2
<input type="checkbox"/> Describe this trick	6
<input checked="" type="checkbox"/> Describe the evaluation strategy	8
<input type="checkbox"/> Describe this	9
<input type="checkbox"/> Describe this	9
<input type="checkbox"/> Describe this	9
<input type="checkbox"/> Describe this, incl. how it's not cons	10
<input type="checkbox"/> Describe this	16
<input type="checkbox"/> Describe this	16
<input type="checkbox"/> Describe this struct	18
<input type="checkbox"/> Describe headers	22
<input type="checkbox"/> Describe what a grammar is	38
<input type="checkbox"/> Describe what a parser is	38
<input type="checkbox"/> Describe programming language theory	38
<input type="checkbox"/> Describe what a REPL is	38