

Lispy: a simple Lisp-like language

Eric Bailey

*May 10, 2018*¹

¹ Current version: 0.8.1.
Last updated May 14, 2018.

For my own edification, and my eternal love of the LISP family and **PLT**, what follows is an implementation in C of a simple, Lisp-like programming language, based on Build Your Own Lisp [Holden, 2018a]. Since I'm a bit of masochist, this is a **literate program**², written using Noweb³.

² https://en.wikipedia.org/wiki/Literate_programming

³ Norman Ramsey. Noweb – a simple, extensible tool for literate programming. <https://www.cs.tufts.edu/~nr/noweb/>, 2012. Accessed: 2018-05-13

Contents

<i>Outline</i>	2
<i>Welcome</i>	4
<i>Defining the Language</i>	4
<i>R is for Read</i>	5
<i>E is for Eval(uate)</i>	6
<i>P is for Print</i>	8
<i>L is for Loop</i>	9
<i>Error Handling</i>	10
<i>Headers</i>	12
<i>Full Listings</i>	14
<i>Chunks</i>	19
<i>Index</i>	20
<i>Glossary</i>	22
<i>References</i>	23

Outline

Describe the outline

2a $\langle \text{lispy.c } 2a \rangle \equiv$
 $\langle \text{Include the necessary headers. } 12b \rangle$

$\langle \text{Load the Lispy grammar. } 4c \rangle$

$\langle \text{Define possible lval and error types. } 11a \rangle$

$\langle \text{Define the Lispy data structures. } 10c \rangle$

This definition is continued in chunks **2** and **3**.
 Root chunk (not used in this document).

2b $\langle \text{lispy.c } 2a \rangle + \equiv$
`void lval_print(lval val)`
`{`
 $\langle \text{Print a Lispy value. } 9a \rangle$
`}`

Defines:
 `lval_print`, used in chunk **2c**.
 Uses `lval 10c`.

2c $\langle \text{lispy.c } 2a \rangle + \equiv$
`void lval_println(lval val)`
`{`
 `lval_print(val);`
 `putchar('\n');`
`}`

Defines:
 `lval_println`, used in chunk **8f**.
 Uses `lval 10c` and `lval_print 2b`.

3a $\langle \text{lisp.c } 2a \rangle + \equiv$
 lval eval_binop(char *op, lval x, lval y)
 {
 $\langle \text{Eval(uate) a binary operation. } 7g \rangle$
 }

lval eval(mpc_ast_t *ast)
 {
 $\langle \text{Eval(uate) the AST. } 6d \rangle$
 }

Defines:

 eval, used in chunks 6 and 7.

 eval_binop, used in chunk 7f.

Uses ast 6c, lval 10c, mpc_ast_t 13b, and op 7a.

3b $\langle \text{lisp.c } 2a \rangle + \equiv$
 int main(int argc, char *argv[])
 {
 $\langle \text{Define the language. } 4d \rangle$

 $\langle \text{Print version and exit information. } 4a \rangle$

 $\langle \text{Loop until the input is empty. } 9c \rangle$

 $\langle \text{Undefine and delete the parsers. } 5c \rangle$

 return 0;
 }

Welcome

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

```
4a <Print version and exit information. 4a>≡
    puts("Lispy v0.8.1");
    puts("Press ctrl-c to exit\n");
```

Uses Lispy 4d.

This code is used in chunk 3b.

Defining the Language

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

```
4b <lispy.mpc 4b>≡
    integer : /-?[0-9]+/ ;
    decimal : /-?[0-9]+\.[0-9]+/ ;
    number  : <decimal> | <integer> ;
    operator: '+' | '-' | '*' | '/' ;
    expr    : <number> | '(' <operator> <expr>+ ')' ;
    lispy   : /^/ <operator> <expr>+ /$/ ;
```

Root chunk (not used in this document).

Describe this trick

```
4c <Load the Lispy grammar. 4c>≡
    static const char LISPY_GRAMMAR[] = {
        #include "lispy.xxd"
    };
```

Defines:

LISPY_GRAMMAR, used in chunk 5b.

This code is used in chunk 2a.

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

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

```
4d <Define the language. 4d>≡
    mpc_parser_t *Integer = mpc_new("integer");
    mpc_parser_t *Decimal = mpc_new("decimal");
    mpc_parser_t *Number  = mpc_new("number");
    mpc_parser_t *Operator = mpc_new("operator");
    mpc_parser_t *Expr    = mpc_new("expr");
    mpc_parser_t *Lispy   = mpc_new("lispy");
```

Defines:

Decimal, used in chunk 5a.

Expr, used in chunk 5a.

Integer, used in chunk 5a.

Lispy, used in chunks 4–6.

Number, used in chunk 5a.

Operator, used in chunk 5a.

Uses mpc_new 13b and mpc_parser_t 13b.

This definition is continued in chunk 5b.

This code is used in chunk 3b.

Finally, using the defined *grammar* and each of the *created parsers 5a*,

5a *⟨created parsers 5a⟩*≡
 Integer, Decimal, Number, Operator, Expr, Lispy
 Uses Decimal 4d, Expr 4d, Integer 4d, Lispy 4d, Number 4d, and Operator 4d.
 This code is used in chunk 5.

... we can define the Lispy language.

5b *⟨Define the language. 4d⟩*+≡
 mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
⟨created parsers 5a⟩);
 Uses LISPY_GRAMMAR 4c and mpca_lang 13b.

Since we're implementing this in C, we need to clean up after ourselves. The `mpc`⁴ library makes this easy, by providing the `mpc_cleanup` function.

5c *⟨Undefine and delete the parsers. 5c⟩*≡
 mpc_cleanup(6, *⟨created parsers 5a⟩*);
 Uses mpc_cleanup 13b.
 This code is used in chunk 3b.

⁴ 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`⁵.

5d *⟨Read a line of user input. 5d⟩*≡
 char *input = readline("> ");
 Defines:
 input, used in chunks 5, 6, and 10b.
 Uses readline 13a.
 This code is used in chunk 9d.

⁵ 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.

5e *⟨input is nonempty 5e⟩*≡
 input && *input
 Uses input 5d.
 This code is used in chunk 10a.

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`⁶ history table.

5f *⟨Add input to the history table. 5f⟩*≡
 add_history(input);
 Uses add_history 13a and input 5d.
 This code is used in chunk 10a.

⁶ Jess Thrysoee. Editline Library (libedit) – port of netbsd command line editor library. <http://thrysoee.dk/editline/>, 2017. Accessed: 2018-05-13

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

6a $\langle \text{Declare a variable to hold parsing results. 6a} \rangle \equiv$
`mpc_result_t parsed;`

Defines:

parsed, used in chunks 6 and 9b.

Uses `mpc_result_t` 13b.

This code is used in chunk 10a.

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

6b $\langle \text{the input can be parsed as Lispy code 6b} \rangle \equiv$
`mpc_parse("<stdin>", input, Lispy, &parsed)`

Uses Lispy 4d, input 5d, `mpc_parse` 13b, and **parsed** 6a.

This code is used in chunk 10a.

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.

6c $\langle \text{Eval(uate) the input. 6c} \rangle \equiv$
`mpc_ast_t *ast = parsed.output;`

`lval result = eval(ast);`

Defines:

ast, used in chunks 3a and 6–8.

result, used in chunks 6–8.

Uses `eval` 3a, `lval` 10c, `mpc_ast_t` 13b, and **parsed** 6a.

This code is used in chunk 10a.

Describe the evaluation strategy

If the *abstract syntax tree (AST)* is tagged as a number, return it directly.

6d $\langle \text{Eval(uate) the AST. 6d} \rangle \equiv$
`if (strstr(ast->tag, "number")) {
 errno = 0;
 double x = strtod(ast->contents, NULL);
 return errno != ERANGE ? lval_num(x) : lval_err(LERR_BAD_NUM);
}`

Uses `LERR_BAD_NUM` 11c, **ast** 6c, `lval_err` 12a, `lval_num` 11b, `strstr` 12f, and `strtod` 12e.

This definition is continued in chunks 6–8.

This code is used in chunk 3a.

If the **AST** is neither an integer nor a float, then it's an expression. Use the *int* **i** to iterate through the children of the **AST**.

6e $\langle \text{Eval(uate) the AST. 6d} \rangle + \equiv$
`int i = 0;`

In an expression, the operator is always the second child.

7a $\langle \text{Eval}(\text{uate}) \text{ the AST. 6d} \rangle + \equiv$
`char *op = ast->children[++i]->contents;`

Defines:

`op`, used in chunks 3a, 7, and 8.

Uses `ast` 6c.

Evaluate the next child, which is the first operand.

7b $\langle \text{Eval}(\text{uate}) \text{ the AST. 6d} \rangle + \equiv$
`lval result = eval(ast->children[++i]);`

Uses `ast` 6c, `eval` 3a, `lval` 10c, and `result` 6c.

If the operation is unary subtraction, negate the operand.

7c $\langle \text{Eval}(\text{uate}) \text{ the AST. 6d} \rangle + \equiv$
`if (!strcmp(op, "-") && ast->children_num == 4) {`
`result.num = -result.num;`
`return result;`
`}`

Uses `ast` 6c, `op` 7a, `result` 6c, and `strcmp` 12f.

While there are more children, i.e.

7d $\langle \text{there are more operands 7d} \rangle \equiv$
`++i < ast->children_num`

Uses `ast` 6c.

This code is used in chunk 8e.

... and the next child is an expression, i.e.

7e $\langle \text{the next child is an expression 7e} \rangle \equiv$
`strstr(ast->children[i]->tag, "expr")`

Uses `ast` 6c and `strstr` 12f.

This code is used in chunk 8e.

... evaluate the next operand.

7f $\langle \text{Eval}(\text{uate}) \text{ the next operand. 7f} \rangle \equiv$
`result = eval_binop(op, result, eval(ast->children[i]));`

Uses `ast` 6c, `eval` 3a, `eval_binop` 3a, `op` 7a, and `result` 6c.

This code is used in chunk 8e.

Describe binop evaluation

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

7g $\langle \text{Eval}(\text{uate}) \text{ a binary operation. 7g} \rangle \equiv$
`if (!strcmp(op, "+"))`
`return lval_num(x.num + y.num);`

Uses `lval_num` 11b, `op` 7a, and `strcmp` 12f.

This definition is continued in chunk 8.

This code is used in chunk 3a.

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

8a $\langle \text{Eval}(\text{uate}) \text{ a binary operation. } 7g \rangle + \equiv$
 if (!strcmp(op, "-"))
 return lval_num(x.num - y.num);

Uses lval_num 11b, op 7a, and strcmp 12f.

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

8b $\langle \text{Eval}(\text{uate}) \text{ a binary operation. } 7g \rangle + \equiv$
 if (!strcmp(op, "*"))
 return lval_num(x.num * y.num);

Uses lval_num 11b, op 7a, and strcmp 12f.

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

8c $\langle \text{Eval}(\text{uate}) \text{ a binary operation. } 7g \rangle + \equiv$
 if (!strcmp(op, "/")) {
 return !y.num
 ? lval_err(LERR_DIV_ZERO)
 : lval_num(x.num / y.num);
 }

Uses LERR_DIV_ZERO 11c, lval_err 12a, lval_num 11b, op 7a, and strcmp 12f.

Otherwise, return a **LERR_BAD_OP** error.

8d $\langle \text{Eval}(\text{uate}) \text{ a binary operation. } 7g \rangle + \equiv$
 return lval_err(LERR_DIV_ZERO);

Uses LERR_DIV_ZERO 11c and lval_err 12a.

Express the recursive operand evaluation as a **while** loop, and return the result.

8e $\langle \text{Eval}(\text{uate}) \text{ the AST. } 6d \rangle + \equiv$
 while ($\langle \text{there are more operands } 7d \rangle$
 && $\langle \text{the next child is an expression } 7e \rangle$)
 $\langle \text{Eval}(\text{uate}) \text{ the next operand. } 7f \rangle$

 return result;

Uses result 6c.

P is for Print

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

8f $\langle \text{Print the result and delete the AST. } 8f \rangle \equiv$
 lval_println(result);

 mpc_ast_delete(ast);

Uses ast 6c, lval_println 2c, mpc_ast_delete 13b, and result 6c.

This code is used in chunk 10a.

9a *⟨Print a Lispy value. 9a⟩*≡

```

switch (val.type) {
case LVAL_NUM:
    printf("%g", val.num);
    break;

case LVAL_ERR:
    switch (val.err) {
case LERR_BAD_OP:
    puts("Error: invalid operator");
    break;
case LERR_BAD_NUM:
    puts("Error: invalid number");
    break;
case LERR_DIV_ZERO:
    fputs("Error: division by zero", stdout);
    break;
    }
    break;
}

```

Uses LERR_BAD_NUM 11c, LERR_BAD_OP 11c, LERR_DIV_ZERO 11c, LVAL_ERR 11a, LVAL_NUM 11a, and printf 12d.
This code is used in chunk 2b.

Print and delete the error upon failure.

9b *⟨Print and delete the error. 9b⟩*≡

```

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

```

Uses mpc_err_delete 13b, mpc_err_print 13b, and parsed 6a.
This code is used in chunk 10a.

L is for Loop

9c *⟨Loop until the input is empty. 9c⟩*≡

```

bool nonempty;
do {
    ⟨Read, eval(uate), and print. 9d⟩
} while (nonempty);

```

Defines:

nonempty, used in chunk 10a.

Uses bool 12c.

This code is used in chunk 3b.

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

9d *⟨Read, eval(uate), and print. 9d⟩*≡

```

⟨Read a line of user input. 5d⟩

```

This definition is continued in chunk 10.

This code is used in chunk 9c.

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

```
10a  <Read, eval(uate), and print. 9d>+≡
      if ((nonempty = (<input is nonempty 5e>))) {
          <Add input to the history table. 5f>

          <Declare a variable to hold parsing results. 6a>
          if ((the input can be parsed as Lispy code 6b)) {
              <Eval(uate) the input. 6c>
              <Print the result and delete the AST. 8f>
          } else {
              <Print and delete the error. 9b>
          }
      }
```

Uses **nonempty** 9c.

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

```
10b  <Read, eval(uate), and print. 9d>+≡
      free(input);
```

Uses **free** 12e and **input** 5d.

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

Error Handling

Describe this struct

```
10c  <Define the Lispy data structures. 10c>≡
      typedef struct {
          lval_type_t type;
          union {
              double num;
              lval_err_t err;
          };
      } lval;
```

Defines:

lval, used in chunks 2, 3a, 6c, 7b, 11b, and 12a.

Uses **lval_err_t** 11c and **lval_type_t** 11a.

This definition is continued in chunks 11b and 12a.

This code is used in chunk 2a.

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

11a *<Define possible lval and error types. 11a>≡*

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

Defines:

LVAL_ERR, used in chunks 9a and 12a.

LVAL_NUM, used in chunks 9a and 11b.

lval_type_t, used in chunk 10c.

This definition is continued in chunk 11c.

This code is used in chunk 2a.

Define a constructor for numbers.

11b *<Define the Lispy data structures. 10c>+≡*

```
lval lval_num(double num)
{
    lval val;
    val.type = LVAL_NUM;
    val.num = num;

    return val;
}
```

Defines:

lval_num, used in chunks 6–8.

Uses LVAL_NUM 11a and lval 10c.

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

11c *<Define possible lval and error types. 11a>+≡*

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

Defines:

LERR_BAD_NUM, used in chunks 6d and 9a.

LERR_BAD_OP, used in chunk 9a.

LERR_DIV_ZERO, used in chunks 8 and 9a.

lval_err_t, used in chunks 10c and 12a.

Define a constructor for errors.

```
12a  <Define the Lispy data structures. 10c>+≡
      lval lval_err(lval_err_t err)
      {
          lval val;
          val.type = LVAL_ERR;
          val.err = err;

          return val;
      }
```

Defines:

`lval_err`, used in chunks 6d and 8.

Uses `LVAL_ERR` 11a, `lval` 10c, and `lval_err_t` 11c.

Headers

Describe headers

```
12b  <Include the necessary headers. 12b>≡
      <Include the boolean type and values. 12c>
      <Include the standard I/O functions. 12d>
      <Include the standard library definitions. 12e>
      <Include some string operations. 12f>
```

<Include the line editing functions from libedit. 13a>

<Include the micro parser combinator definitions. 13b>

This code is used in chunk 2a.

```
12c  <Include the boolean type and values. 12c>≡
      #include <stdbool.h>
```

Defines:

`bool`, used in chunk 9c.

This code is used in chunk 12b.

```
12d  <Include the standard I/O functions. 12d>≡
      #include <stdio.h>
```

Defines:

`printf`, used in chunk 9a.

This code is used in chunk 12b.

```
12e  <Include the standard library definitions. 12e>≡
      #include <stdlib.h>
```

Defines:

`free`, used in chunk 10b.

`strtod`, used in chunk 6d.

This code is used in chunk 12b.

```
12f  <Include some string operations. 12f>≡
      #include <string.h>
```

Defines:

`strcmp`, used in chunks 7 and 8.

`strstr`, used in chunks 6d and 7e.

This code is used in chunk 12b.

13a *(Include the line editing functions from libedit. 13a)≡*
`#include <editline/readline.h>`

Defines:

`add_history`, used in chunk 5f.

`readline`, used in chunks 13a and 5d.

This code is used in chunk 12b.

13b *(Include the micro parser combinator definitions. 13b)≡*
`#include <mpc.h>`

Defines:

`mpca_lang`, used in chunk 5b.

`mpc_ast_delete`, used in chunk 8f.

`mpc_ast_print`, never used.

`mpc_ast_t`, used in chunks 3a and 6c.

`mpc_cleanup`, used in chunks 13b and 5c.

`mpc_err_delete`, used in chunk 9b.

`mpc_err_print`, used in chunk 9b.

`mpc_new`, used in chunk 4d.

`mpc_parse`, used in chunks 13b and 6b.

`mpc_parser_t`, used in chunk 4d.

`mpc_result_t`, used in chunk 6a.

This code is used in chunk 12b.

Full Listings

lispy.mpc:

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

lispy.c:

```

1  #include <stdbool.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  #include <editline/readline.h>
7  #include <mpc.h>
8
9
10 static const char LISPY_GRAMMAR[] = {
11     #include "lispy.xxd"
12 };
13
14
15 typedef enum {
16     LVAL_NUM,
17     LVAL_ERR
18 } lval_type_t;
19
20 typedef enum {
21     LERR_DIV_ZERO,
22     LERR_BAD_OP,
23     LERR_BAD_NUM
24 } lval_err_t;
25
26 typedef struct {
27     lval_type_t type;
28     union {
29         double num;
30         lval_err_t err;
31     };
32 } lval;
33
34
35 lval lval_num(double num)
36 {
37     lval val;
38     val.type = LVAL_NUM;
39     val.num = num;
40
41     return val;
42 }
43
44
45 lval lval_err(lval_err_t err)
46 {
47     lval val;
48     val.type = LVAL_ERR;
49     val.err = err;
50

```

```

51     return val;
52 }
53
54
55 void lval_print(lval val)
56 {
57     switch (val.type) {
58     case LVAL_NUM:
59         printf("%g", val.num);
60         break;
61
62     case LVAL_ERR:
63         switch (val.err) {
64         case LERR_BAD_OP:
65             puts("Error: invalid operator");
66             break;
67         case LERR_BAD_NUM:
68             puts("Error: invalid number");
69             break;
70         case LERR_DIV_ZERO:
71             fputs("Error: division by zero", stdout);
72             break;
73         }
74         break;
75     }
76 }
77
78
79 void lval_println(lval val)
80 {
81     lval_print(val);
82     putchar('\n');
83 }
84
85
86 lval eval_binop(char *op, lval x, lval y)
87 {
88     if (!strcmp(op, "+"))
89         return lval_num(x.num + y.num);
90
91     if (!strcmp(op, "-"))
92         return lval_num(x.num - y.num);
93
94     if (!strcmp(op, "*"))
95         return lval_num(x.num * y.num);
96
97     if (!strcmp(op, "/")) {
98         return !y.num ? lval_err(LERR_DIV_ZERO)
99             : lval_num(x.num / y.num);
100     }
101 }

```



```

102     return lval_err(LERR_DIV_ZERO);
103 }
104
105
106 lval eval(mpc_ast_t * ast)
107 {
108     if (strstr(ast->tag, "number")) {
109         errno = 0;
110         double x = strtod(ast->contents, NULL);
111         return errno != ERANGE ? lval_num(x) : lval_err(LERR_BAD_NUM);
112     }
113
114     int i = 0;
115
116     char *op = ast->children[++i]->contents;
117
118     lval result = eval(ast->children[++i]);
119
120     if (!strcmp(op, "-") && ast->children_num == 4) {
121         result.num = -result.num;
122         return result;
123     }
124
125     while (++i < ast->children_num
126         && strstr(ast->children[i]->tag, "expr"))
127         result = eval_binop(op, result, eval(ast->children[i]));
128
129     return result;
130 }
131
132
133 int main(int argc, char *argv[])
134 {
135     mpc_parser_t *Integer = mpc_new("integer");
136     mpc_parser_t *Decimal = mpc_new("decimal");
137     mpc_parser_t *Number = mpc_new("number");
138     mpc_parser_t *Operator = mpc_new("operator");
139     mpc_parser_t *Expr = mpc_new("expr");
140     mpc_parser_t *Lispy = mpc_new("lispy");
141
142     mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
143         Integer, Decimal, Number, Operator, Expr, Lispy);
144
145     puts("Lispy v0.8.1");
146     puts("Press ctrl-c to exit\n");
147
148     bool nonempty;
149     do {
150         char *input = readline("> ");
151         if ((nonempty = (input && *input))) {
152             add_history(input);

```

```
153     mpc_result_t parsed;
154     if (mpc_parse("<stdin>", input, Lispy, &parsed)) {
155         mpc_ast_t *ast = parsed.output;
156
157         lval result = eval(ast);
158         lval_println(result);
159
160         mpc_ast_delete(ast);
161     } else {
162         mpc_err_print(parsed.error);
163         mpc_err_delete(parsed.error);
164     }
165 }
166
167     free(input);
168 } while (nonempty);
169
170 mpc_cleanup(6, Integer, Decimal, Number, Operator, Expr, Lispy);
171
172 return 0;
173 }
174 }
```

Chunks

⟨Add input to the history table. 5f⟩ [5f](#), [10a](#)
 ⟨Declare a variable to hold parsing results. 6a⟩ [6a](#), [10a](#)
 ⟨Define possible lval and error types. 11a⟩ [2a](#), [11a](#), [11c](#)
 ⟨Define the Lispy data structures. 10c⟩ [2a](#), [10c](#), [11b](#), [12a](#)
 ⟨Define the language. 4d⟩ [3b](#), [4d](#), [5b](#)
 ⟨Eval(uate) a binary operation. 7g⟩ [3a](#), [7g](#), [8a](#), [8b](#), [8c](#), [8d](#)
 ⟨Eval(uate) the AST. 6d⟩ [3a](#), [6d](#), [6e](#), [7a](#), [7b](#), [7c](#), [8e](#)
 ⟨Eval(uate) the input. 6c⟩ [6c](#), [10a](#)
 ⟨Eval(uate) the next operand. 7f⟩ [7f](#), [8e](#)
 ⟨Include some string operations. 12f⟩ [12b](#), [12f](#)
 ⟨Include the boolean type and values. 12c⟩ [12b](#), [12c](#)
 ⟨Include the line editing functions from libedit. 13a⟩ [12b](#), [13a](#)
 ⟨Include the micro parser combinator definitions. 13b⟩ [12b](#), [13b](#)
 ⟨Include the necessary headers. 12b⟩ [2a](#), [12b](#)
 ⟨Include the standard I/O functions. 12d⟩ [12b](#), [12d](#)
 ⟨Include the standard library definitions. 12e⟩ [12b](#), [12e](#)
 ⟨Load the Lispy grammar. 4c⟩ [2a](#), [4c](#)
 ⟨Loop until the input is empty. 9c⟩ [3b](#), [9c](#)
 ⟨Print a Lispy value. 9a⟩ [2b](#), [9a](#)
 ⟨Print and delete the error. 9b⟩ [9b](#), [10a](#)
 ⟨Print the result and delete the AST. 8f⟩ [8f](#), [10a](#)
 ⟨Print version and exit information. 4a⟩ [3b](#), [4a](#)
 ⟨Read a line of user input. 5d⟩ [5d](#), [9d](#)
 ⟨Read, eval(uate), and print. 9d⟩ [9c](#), [9d](#), [10a](#), [10b](#)
 ⟨Undefine and delete the parsers. 5c⟩ [3b](#), [5c](#)
 ⟨created parsers 5a⟩ [5a](#), [5b](#), [5c](#)
 ⟨input is nonempty 5e⟩ [5e](#), [10a](#)
 ⟨lispy.c 2a⟩ [2a](#), [2b](#), [2c](#), [3a](#), [3b](#)
 ⟨lispy.mpc 4b⟩ [4b](#)
 ⟨the input can be parsed as Lispy code 6b⟩ [6b](#), [10a](#)
 ⟨the next child is an expression 7e⟩ [7e](#), [8e](#)
 ⟨there are more operands 7d⟩ [7d](#), [8e](#)

Index

Decimal: [4d](#), [5a](#)
 Expr: [4d](#), [5a](#)
 Integer: [4d](#), [5a](#)
 LERR_BAD_NUM: [6d](#), [9a](#), [11c](#)
 LERR_BAD_OP: [9a](#), [11c](#)
 LERR_DIV_ZERO: [8c](#), [8d](#), [9a](#), [11c](#)
 LISPY_GRAMMAR: [4c](#), [5b](#)
 LVAL_ERR: [9a](#), [11a](#), [12a](#)
 LVAL_NUM: [9a](#), [11a](#), [11b](#)
 Lispy: [4a](#), [4d](#), [5a](#), [6b](#)
 Number: [4d](#), [5a](#)
 Operator: [4d](#), [5a](#)
 add_history: [5f](#), [13a](#)
 ast: [3a](#), [6c](#), [6d](#), [7a](#), [7b](#), [7c](#), [7d](#), [7e](#), [7f](#), [8f](#)
 bool: [9c](#), [12c](#)
 eval: [3a](#), [6c](#), [7b](#), [7f](#)
 eval_binop: [3a](#), [7f](#)
 free: [10b](#), [12e](#)
 input: [5d](#), [5e](#), [5d](#), [5d](#), [5f](#), [6b](#), [5d](#), [10b](#)
 lval: [2b](#), [2c](#), [3a](#), [6c](#), [7b](#), [10c](#), [11b](#), [12a](#)
 lval_err: [6d](#), [8c](#), [8d](#), [12a](#)
 lval_err_t: [10c](#), [11c](#), [12a](#)
 lval_num: [6d](#), [7g](#), [8a](#), [8b](#), [8c](#), [11b](#)
 lval_print: [2b](#), [2c](#)
 lval_println: [2c](#), [8f](#)
 lval_type_t: [10c](#), [11a](#)
 mpca_lang: [5b](#), [13b](#)
 mpc_ast_delete: [8f](#), [13b](#)
 mpc_ast_print: [13b](#)
 mpc_ast_t: [3a](#), [6c](#), [13b](#)
 mpc_cleanup: [13b](#), [5c](#), [13b](#)
 mpc_err_delete: [9b](#), [13b](#)
 mpc_err_print: [9b](#), [13b](#)
 mpc_new: [4d](#), [13b](#)
 mpc_parse: [13b](#), [6b](#), [13b](#)
 mpc_parser_t: [4d](#), [13b](#)
 mpc_result_t: [6a](#), [13b](#)
 nonempty: [9c](#), [10a](#)
 op: [3a](#), [7a](#), [7c](#), [7f](#), [7a](#), [7g](#), [7a](#), [8a](#), [7a](#), [8b](#), [7a](#), [8c](#)
 parsed: [6a](#), [6a](#), [6b](#), [6c](#), [9b](#)
 printf: [9a](#), [12d](#)
 readline: [13a](#), [5d](#), [13a](#)

result: 6c, 6c, 7b, 7c, 7f, 8e, 8f
strcmp: 7c, 7g, 8a, 8b, 8c, 12f
strstr: 6d, 7e, 12f
strtod: 6d, 12e

Glossary

AST abstract syntax tree, a tree representation of the abstract syntactic structure of source code. 6, 8

grammar 4, 5

Describe what a grammar is

parser 4

Describe what a parser is

PLT programming language theory, 1

Describe programming language theory

REPL Read-Eval-Print Loop, 4, 5

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
Describe this trick	4
Describe the evaluation strategy	6
Describe binop evaluation	7
Describe this struct	10
Describe headers	12
Describe what a grammar is	22
Describe what a parser is	22
Describe programming language theory	22
Describe what a REPL is	22