

Lispy: a simple Lisp-like language

Eric Bailey

*May 10, 2018*¹

¹ Current version: 0.9.0.
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>	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>	9
<i>L is for Loop</i>	10
<i>Error Handling</i>	11
<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 $\langle \text{lisp.c } 2a \rangle \equiv$
 $\langle \text{Include the necessary headers. } 12c \rangle$

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

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

$\langle \text{Define the Lispy data structures. } 11a \rangle$

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

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

Defines:
 `lval_print`, used in chunk **2c**.
 Uses `lval 11a`.

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

Defines:
 `lval_println`, used in chunk **9b**.
 Uses `lval 11a` 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 11a, mpc_ast_t 13e, and op 7a.

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

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

 $\langle \text{Loop until the input is empty. } 10a \rangle$

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

 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 v0.9.0");`
`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` 13e and `mpc_parser_t` 13e.

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 13e.

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 13e.
 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 10d.
 Uses readline 13d.
 This code is used in chunk 10b.

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 10c.

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 13d and input 5d.
 This code is used in chunk 10c.

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 9d.

Uses `mpc_result_t` 13e.

This code is used in chunk 10c.

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` 13e, and **parsed** 6a.

This code is used in chunk 10c.

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, 6, 7, and 9b.

result, used in chunks 6, 7, and 9.

Uses `eval` 3a, `lval` 11a, `mpc_ast_t` 13e, and **parsed** 6a.

This code is used in chunk 10c.

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` 12a, **ast** 6c, `lval_err` 12b, `lval_num` 11c, `strstr` 13c, and `strtod` 13a.

This definition is continued in chunks 6, 7, and 9a.

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` 11a, 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` 13c.

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 9a.

... 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` 13c.

This code is used in chunk 9a.

... 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 9a.

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` 11c, `op` 7a, and `strcmp` 13c.

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 11c, op 7a, and strcmp 13c.

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 11c, op 7a, and strcmp 13c.

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 12a, lval_err 12b, lval_num 11c, op 7a, and strcmp 13c.

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

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

Uses LERR_DIV_ZERO 12a, fmod 13b, lval_err 12b, lval_num 11c, op 7a, and strcmp 13c.

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

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

Uses lval_num 11c, op 7a, pow 13b, and strcmp 13c.

Otherwise, return a LERR_BAD_OP error.

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

Uses LERR_DIV_ZERO 12a and lval_err 12b.

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

9a $\langle \text{Eval}(\text{uate}) \text{ the AST. 6d} \rangle \equiv$
`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`.

9b $\langle \text{Print the result and delete the AST. 9b} \rangle \equiv$
`lval_println(result);`

`mpc_ast_delete(ast);`

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

This code is used in chunk `10c`.

9c $\langle \text{Print a Lispy value. 9c} \rangle \equiv$
`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 12a`, `LERR_BAD_OP 12a`, `LERR_DIV_ZERO 12a`, `LVAL_ERR 11b`, `LVAL_NUM 11b`, and `printf 12e`.

This code is used in chunk `2b`.

Print and delete the error upon failure.

9d $\langle \text{Print and delete the error. 9d} \rangle \equiv$
`mpc_err_print(parsed.error);`
`mpc_err_delete(parsed.error);`

Uses `mpc_err_delete 13e`, `mpc_err_print 13e`, and `parsed 6a`.

This code is used in chunk `10c`.

L is for Loop

10a $\langle \text{Loop until the input is empty. 10a} \rangle \equiv$
 bool nonempty;
 do {
 $\langle \text{Read, eval(uate), and print. 10b} \rangle$
 } while (nonempty);

Defines:

 nonempty, used in chunk 10c.

Uses bool 12d.

This code is used in chunk 3b.

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

10b $\langle \text{Read, eval(uate), and print. 10b} \rangle \equiv$
 $\langle \text{Read a line of user input. 5d} \rangle$

This definition is continued in chunk 10.

This code is used in chunk 10a.

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

10c $\langle \text{Read, eval(uate), and print. 10b} \rangle + \equiv$
 if ((nonempty = ($\langle \text{input is nonempty 5e} \rangle$))) {
 $\langle \text{Add input to the history table. 5f} \rangle$

 $\langle \text{Declare a variable to hold parsing results. 6a} \rangle$
 if (($\langle \text{the input can be parsed as Lispy code 6b} \rangle$)) {
 $\langle \text{Eval(uate) the input. 6c} \rangle$
 $\langle \text{Print the result and delete the AST. 9b} \rangle$
 } else {
 $\langle \text{Print and delete the error. 9d} \rangle$
 }
 }
 }

Uses nonempty 10a.

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

10d $\langle \text{Read, eval(uate), and print. 10b} \rangle + \equiv$
 free(input);

Uses free 13a and input 5d.

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

Error Handling

Describe this struct

11a \langle Define the Lispy data structures. 11a $\rangle \equiv$

```
typedef struct {
    lval_type_t type;
    union {
        double num;
        lval_err_t err;
    };
} lval;
```

Defines:

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

Uses lval_err_t 12a and lval_type_t 11b.

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

This code is used in chunk 2a.

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

11b \langle Define possible lval and error types. 11b $\rangle \equiv$

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

Defines:

LVAL_ERR, used in chunks 9c and 12b.

LVAL_NUM, used in chunks 9c and 11c.

lval_type_t, used in chunk 11a.

This definition is continued in chunk 12a.

This code is used in chunk 2a.

Define a constructor for numbers.

11c \langle Define the Lispy data structures. 11a $\rangle + \equiv$

```
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 11b and lval 11a.

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

12a *<Define possible lval and error types. 11b>+≡*

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

Defines:

LERR_BAD_NUM, used in chunks 6d and 9c.
 LERR_BAD_OP, used in chunk 9c.
 LERR_DIV_ZERO, used in chunks 8 and 9c.
 lval_err_t, used in chunks 11a and 12b.

Define a constructor for errors.

12b *<Define the Lispy data structures. 11a>+≡*

```
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 11b, lval 11a, and lval_err_t 12a.

Headers

Describe headers

12c *<Include the necessary headers. 12c>≡*

```
<Include the boolean type and values. 12d>
<Include the standard I/O functions. 12e>
<Include the standard library definitions. 13a>
<Include some mathematical definitions. 13b>
<Include some string operations. 13c>

<Include the line editing functions from libedit. 13d>
<Include the micro parser combinator definitions. 13e>
```

This code is used in chunk 2a.

12d *<Include the boolean type and values. 12d>≡*

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

Defines:

bool, used in chunk 10a.

This code is used in chunk 12c.

12e *<Include the standard I/O functions. 12e>≡*

```
#include <stdio.h>
```

Defines:

printf, used in chunk 9c.

This code is used in chunk 12c.

- 13a** *⟨Include the standard library definitions. 13a⟩*≡
`#include <stdlib.h>`
 Defines:
 `free`, used in chunk **10d**.
 `strtod`, used in chunk **6d**.
 This code is used in chunk **12c**.
- 13b** *⟨Include some mathematical definitions. 13b⟩*≡
`#include <math.h>`
 Defines:
 `fmod`, used in chunk **8d**.
 `pow`, used in chunk **8e**.
 This code is used in chunk **12c**.
- 13c** *⟨Include some string operations. 13c⟩*≡
`#include <string.h>`
 Defines:
 `strcmp`, used in chunks **7** and **8**.
 `strstr`, used in chunks **6d** and **7e**.
 This code is used in chunk **12c**.
- 13d** *⟨Include the line editing functions from libedit. 13d⟩*≡
`#include <editline/readline.h>`
 Defines:
 `add_history`, used in chunk **5f**.
 `readline`, used in chunks **13d** and **5d**.
 This code is used in chunk **12c**.
- 13e** *⟨Include the micro parser combinator definitions. 13e⟩*≡
`#include <mpc.h>`
 Defines:
 `mpca_lang`, used in chunk **5b**.
 `mpc_ast_delete`, used in chunk **9b**.
 `mpc_ast_print`, never used.
 `mpc_ast_t`, used in chunks **3a** and **6c**.
 `mpc_cleanup`, used in chunks **13e** and **5c**.
 `mpc_err_delete`, used in chunk **9d**.
 `mpc_err_print`, used in chunk **9d**.
 `mpc_new`, used in chunk **4d**.
 `mpc_parse`, used in chunks **13e** and **6b**.
 `mpc_parser_t`, used in chunk **4d**.
 `mpc_result_t`, used in chunk **6a**.
 This code is used in chunk **12c**.

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 <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_NUM,
18     LVAL_ERR
19 } lval_type_t;
20
21 typedef enum {
22     LERR_DIV_ZERO,
23     LERR_BAD_OP,
24     LERR_BAD_NUM
25 } lval_err_t;
26
27 typedef struct {
28     lval_type_t type;
29     union {
30         double num;
31         lval_err_t err;
32     };
33 } lval;
34
35
36 lval lval_num(double num)
37 {
38     lval val;
39     val.type = LVAL_NUM;
40     val.num = num;
41
42     return val;
43 }
44
45
46 lval lval_err(lval_err_t err)
47 {
48     lval val;
49     val.type = LVAL_ERR;
50     val.err = err;

```

```

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

```



```

102     if (!strcmp(op, "%"))
103         return !y.num ? lval_err(LERR_DIV_ZERO)
104             : lval_num(fmod(x.num, y.num));
105
106     if (!strcmp(op, "^"))
107         return lval_num(pow(x.num, y.num));
108
109     return lval_err(LERR_DIV_ZERO);
110 }
111
112
113 lval eval(mpc_ast_t * ast)
114 {
115     if (strstr(ast->tag, "number")) {
116         errno = 0;
117         double x = strtod(ast->contents, NULL);
118         return errno != ERANGE ? lval_num(x) : lval_err(LERR_BAD_NUM);
119     }
120
121     int i = 0;
122
123     char *op = ast->children[++i]->contents;
124
125     lval result = eval(ast->children[++i]);
126
127     if (!strcmp(op, "-") && ast->children_num == 4) {
128         result.num = -result.num;
129         return result;
130     }
131
132     while (++i < ast->children_num
133         && strstr(ast->children[i]->tag, "expr"))
134         result = eval_binop(op, result, eval(ast->children[i]));
135
136     return result;
137 }
138
139
140 int main(int argc, char *argv[])
141 {
142     mpc_parser_t *Integer = mpc_new("integer");
143     mpc_parser_t *Decimal = mpc_new("decimal");
144     mpc_parser_t *Number = mpc_new("number");
145     mpc_parser_t *Operator = mpc_new("operator");
146     mpc_parser_t *Expr = mpc_new("expr");
147     mpc_parser_t *Lispy = mpc_new("lispy");
148
149     mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
150         Integer, Decimal, Number, Operator, Expr, Lispy);
151
152     puts("Lispy v0.9.0");

```

```

153     puts("Press ctrl-c to exit\n");
154
155     bool nonempty;
156     do {
157         char *input = readline("> ");
158         if ((nonempty = (input && *input))) {
159             add_history(input);
160
161             mpc_result_t parsed;
162             if (mpc_parse("<stdin>", input, Lispy, &parsed)) {
163                 mpc_ast_t *ast = parsed.output;
164
165                 lval result = eval(ast);
166                 lval_println(result);
167
168                 mpc_ast_delete(ast);
169             } else {
170                 mpc_err_print(parsed.error);
171                 mpc_err_delete(parsed.error);
172             }
173         }
174
175         free(input);
176     } while (nonempty);
177
178     mpc_cleanup(6, Integer, Decimal, Number, Operator, Expr, Lispy);
179
180     return 0;
181 }

```

Chunks

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

Index

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

printf: 9c, 12e
readline: 13d, 5d, 13d
result: 6c, 6c, 7b, 7c, 7f, 9a, 9b
strcmp: 7c, 7g, 8a, 8b, 8c, 8d, 8e, 13c
strstr: 6d, 7e, 13c
strtod: 6d, 13a

Glossary

AST abstract syntax tree, a tree representation of the abstract syntactic structure of source code. [6](#), [9](#)

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	11
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