

Lispy: a simple Lisp-like language

Eric Bailey

*May 10, 2018*¹

¹ Current version: 0.7.0.
Last updated May 13, 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>	3
<i>Defining the Language</i>	3
<i>R is for Read</i>	4
<i>E is for Eval(uate)</i>	5
<i>P is for Print</i>	7
<i>L is for Loop</i>	8
<i>Headers</i>	9
<i>Full Listings</i>	11
<i>Chunks</i>	14
<i>Index</i>	15
<i>Glossary</i>	16
<i>References</i>	17

Outline

Describe the outline

2 `<lisp.c 2>≡`
 <Include the necessary headers. 9a>

<Load the Lispy grammar. 3c>

```
double eval_binop(char *op, double x, double y)
{
    <Eval(uate) a binary operation. 6g>
}
```

```
double eval(mpc_ast_t *ast)
{
    <Eval(uate) the AST. 5d>
}
```

```
int main(int argc, char *argv[])
{
    <Define the language. 3d>

    <Print version and exit information. 3a>

    <Loop until the input is empty. 8a>

    <Undefine and delete the parsers. 4c>

    return 0;
}
```

Defines:

`eval`, used in chunks 5 and 6. `eval_binop`, used in chunk 6f.Uses `ast` 5c, `mpc_ast_t` 10, and `op` 6a.

Root chunk (not used in this document).

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.

```
3a <Print version and exit information. 3a>≡
    puts("Lispy v0.7.0");
    puts("Press ctrl-c to exit\n");
```

Uses Lispy 3d.

This code is used in chunk 2.

Defining the Language

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

```
3b <lispy.mpc 3b>≡
    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

```
3c <Load the Lispy grammar. 3c>≡
    static const char LISPY_GRAMMAR[] = {
        #include "lispy.xxd"
    };
```

Defines:

LISPY_GRAMMAR, used in chunk 4b.

This code is used in chunk 2.

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

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

```
3d <Define the language. 3d>≡
    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 4a.

Expr, used in chunk 4a.

Integer, used in chunk 4a.

Lispy, used in chunks 3–5.

Number, used in chunk 4a.

Operator, used in chunk 4a.

Uses mpc_new 10 and mpc_parser_t 10.

This definition is continued in chunk 4b.

This code is used in chunk 2.

Finally, using the defined *grammar* and each of the *created parsers 4a*,

4a *created parsers 4a* ≡
 Integer, Decimal, Number, Operator, Expr, Lispy
 Uses Decimal 3d, Expr 3d, Integer 3d, Lispy 3d, Number 3d, and Operator 3d.
 This code is used in chunk 4.

... we can define the Lispy language.

4b *Define the language. 3d* + ≡
 mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
 created parsers 4a);
 Uses LISPY_GRAMMAR 3c and mpca_lang 10.

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.

4c *Undefine and delete the parsers. 4c* ≡
 mpc_cleanup(6, *created parsers 4a*);
 Uses mpc_cleanup 10.
 This code is used in chunk 2.

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.

4d *Read a line of user input. 4d* ≡
 char *input = readline("> ");
 Defines:
 input, used in chunks 4, 5, and 8d.
 Uses readline 9f.
 This code is used in chunk 8b.

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.

4e *input is nonempty 4e* ≡
 input && *input
 Uses input 4d.
 This code is used in chunk 8c.

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.

4f *Add input to the history table. 4f* ≡
 add_history(input);
 Uses add_history 9f and input 4d.
 This code is used in chunk 8c.

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.

5a $\langle \text{Declare a variable to hold parsing results. 5a} \rangle \equiv$
`mpc_result_t parsed;`

Defines:

`parsed`, used in chunks 5 and 7g.

Uses `mpc_result_t` 10.

This code is used in chunk 8c.

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

5b $\langle \text{the input can be parsed as Lispy code 5b} \rangle \equiv$
`mpc_parse("<stdin>", input, Lispy, &parsed)`

Uses Lispy 3d, input 4d, `mpc_parse` 10, and `parsed` 5a.

This code is used in chunk 8c.

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.

5c $\langle \text{Eval(uate) the input. 5c} \rangle \equiv$
`mpc_ast_t *ast = parsed.output;`

`double result = eval(ast);`

Defines:

`ast`, used in chunks 2 and 5–7.

`result`, used in chunks 5–7.

Uses `eval` 2, `mpc_ast_t` 10, and `parsed` 5a.

This code is used in chunk 8c.

Describe the evaluation strategy

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

5d $\langle \text{Eval(uate) the AST. 5d} \rangle \equiv$
`if (strstr(ast->tag, "number"))`
`return atof(ast->contents);`

Uses `ast` 5c, `atof` 9d, and `strstr` 9e.

This definition is continued in chunks 5–7.

This code is used in chunk 2.

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*.

5e $\langle \text{Eval(uate) the AST. 5d} \rangle + \equiv$
`int i = 0;`

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

6a $\langle \text{Eval}(\text{uate}) \text{ the AST. 5d} \rangle + \equiv$
`char *op = ast->children[++i]->contents;`

Defines:

`op`, used in chunks 2, 6, and 7.

Uses `ast 5c`.

Evaluate the next child, which is the first operand.

6b $\langle \text{Eval}(\text{uate}) \text{ the AST. 5d} \rangle + \equiv$
`double result = eval(ast->children[++i]);`

Uses `ast 5c`, `eval 2`, and `result 5c`.

If the operation is unary subtraction, negate the operand.

6c $\langle \text{Eval}(\text{uate}) \text{ the AST. 5d} \rangle + \equiv$
`if (!strcmp(op, "-") && ast->children_num == 4)`
`return -result;`

Uses `ast 5c`, `op 6a`, `result 5c`, and `strcmp 9e`.

While there are more children, i.e.

6d $\langle \text{there are more operands 6d} \rangle \equiv$
`++i < ast->children_num`

Uses `ast 5c`.

This code is used in chunk 7e.

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

6e $\langle \text{the next child is an expression 6e} \rangle \equiv$
`strstr(ast->children[i]->tag, "expr")`

Uses `ast 5c` and `strstr 9e`.

This code is used in chunk 7e.

... evaluate the next operand.

6f $\langle \text{Eval}(\text{uate}) \text{ the next operand. 6f} \rangle \equiv$
`result = eval_binop(op, result, eval(ast->children[i]));`

Uses `ast 5c`, `eval 2`, `eval_binop 2`, `op 6a`, and `result 5c`.

This code is used in chunk 7e.

Describe binop evaluation

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

6g $\langle \text{Eval}(\text{uate}) \text{ a binary operation. 6g} \rangle \equiv$
`if (!strcmp(op, "+"))`
`return x + y;`

Uses `op 6a` and `strcmp 9e`.

This definition is continued in chunk 7.

This code is used in chunk 2.

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

```
7a <Eval(uate) a binary operation. 6g>+≡
    if (!strcmp(op, "-"))
        return x - y;
```

Uses `op 6a` and `strcmp 9e`.

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

```
7b <Eval(uate) a binary operation. 6g>+≡
    if (!strcmp(op, "*"))
        return x * y;
```

Uses `op 6a` and `strcmp 9e`.

If the `op` is `"/"`, perform division.

```
7c <Eval(uate) a binary operation. 6g>+≡
    if (!strcmp(op, "/"))
        return x / y;
```

Uses `op 6a` and `strcmp 9e`.

Otherwise, return `0`.

```
7d <Eval(uate) a binary operation. 6g>+≡
    return 0;
```

Bind an error message or something

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

```
7e <Eval(uate) the AST. 5d>+≡
    while ((there are more operands 6d)
           && (the next child is an expression 6e))
        <Eval(uate) the next operand. 6f>
```

return result;

Uses `result 5c`.

P is for Print

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

```
7f <Print the result and delete the AST. 7f>≡
    printf("%g\n", result);
```

`mpc_ast_delete(ast);`

Uses `ast 5c`, `mpc_ast_delete 10`, `printf 9c`, and `result 5c`.
This code is used in chunk `8c`.

Print and delete the error upon failure.

```
7g <Print and delete the error. 7g>≡
    mpc_err_print(parsed.error);
    mpc_err_delete(parsed.error);
```

Uses `mpc_err_delete 10`, `mpc_err_print 10`, and `parsed 5a`.
This code is used in chunk `8c`.

L is for Loop

8a $\langle \text{Loop until the input is empty. 8a} \rangle \equiv$
 bool nonempty;
 do {
 $\langle \text{Read, eval(uate), and print. 8b} \rangle$
 } while (nonempty);

Defines:

 nonempty, used in chunk 8c.

Uses bool 9b.

This code is used in chunk 2.

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

8b $\langle \text{Read, eval(uate), and print. 8b} \rangle \equiv$
 $\langle \text{Read a line of user input. 4d} \rangle$

This definition is continued in chunk 8.

This code is used in chunk 8a.

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

8c $\langle \text{Read, eval(uate), and print. 8b} \rangle + \equiv$
 if ((nonempty = ($\langle \text{input is nonempty 4e} \rangle$))) {
 $\langle \text{Add input to the history table. 4f} \rangle$

 $\langle \text{Declare a variable to hold parsing results. 5a} \rangle$
 if (($\langle \text{the input can be parsed as Lispy code 5b} \rangle$)) {
 $\langle \text{Eval(uate) the input. 5c} \rangle$
 $\langle \text{Print the result and delete the AST. 7f} \rangle$
 } else {
 $\langle \text{Print and delete the error. 7g} \rangle$
 }
 }
 }

Uses nonempty 8a.

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

8d $\langle \text{Read, eval(uate), and print. 8b} \rangle + \equiv$
 free(input);

Uses free 9d and input 4d.

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

Headers

Describe headers

9a *⟨Include the necessary headers. 9a⟩*≡
⟨Include the boolean type and values. 9b⟩
⟨Include the standard I/O functions. 9c⟩
⟨Include the standard library definitions. 9d⟩
⟨Include some string operations. 9e⟩

⟨Include the line editing functions from libedit. 9f⟩
⟨Include the micro parser combinator definitions. 10⟩
 This code is used in chunk **2**.

9b *⟨Include the boolean type and values. 9b⟩*≡
#include <stdbool.h>
 Defines:
bool, used in chunk **8a**.
 This code is used in chunk **9a**.

9c *⟨Include the standard I/O functions. 9c⟩*≡
#include <stdio.h>
 Defines:
printf, used in chunk **7f**.
 This code is used in chunk **9a**.

9d *⟨Include the standard library definitions. 9d⟩*≡
#include <stdlib.h>
 Defines:
atof, used in chunk **5d**.
atoi, never used.
free, used in chunk **8d**.
 This code is used in chunk **9a**.

9e *⟨Include some string operations. 9e⟩*≡
#include <string.h>
 Defines:
strcmp, used in chunks **6** and **7**.
strstr, used in chunks **5d** and **6e**.
 This code is used in chunk **9a**.

9f *⟨Include the line editing functions from libedit. 9f⟩*≡
#include <editline/readline.h>
 Defines:
add_history, used in chunk **4f**.
readline, used in chunks **9f** and **4d**.
 This code is used in chunk **9a**.

10 *(Include the micro parser combinator definitions. 10)*≡
 #include <mpc.h>

Defines:

 mpca_lang, used in chunk 4b.
 mpc_ast_delete, used in chunk 7f.
 mpc_ast_print, never used.
 mpc_ast_t, used in chunks 2 and 5c.
 mpc_cleanup, used in chunks 10 and 4c.
 mpc_err_delete, used in chunk 7g.
 mpc_err_print, used in chunk 7g.
 mpc_new, used in chunk 3d.
 mpc_parse, used in chunks 10 and 5b.
 mpc_parser_t, used in chunk 3d.
 mpc_result_t, used in chunk 5a.

This code is used in chunk 9a.

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 double eval_binop(char *op, double x, double y)
16 {
17     if (!strcmp(op, "+"))
18         return x + y;
19
20     if (!strcmp(op, "-"))
21         return x - y;
22
23     if (!strcmp(op, "*"))
24         return x * y;
25
26     if (!strcmp(op, "/"))
27         return x / y;
28
29     return 0;
30 }
31
32
33 double eval(mpc_ast_t * ast)
34 {
35     if (strstr(ast->tag, "number"))
36         return atof(ast->contents);
37
38     int i = 0;
39
40     char *op = ast->children[++i]->contents;
41
42     double result = eval(ast->children[++i]);
43
44     if (!strcmp(op, "-")) && ast->children_num == 4)
45         return -result;
46
47     while (++i < ast->children_num
48         && strstr(ast->children[i]->tag, "expr"))
49         result = eval_binop(op, result, eval(ast->children[i]));
50

```

```

51     return result;
52 }
53
54
55 int main(int argc, char *argv[])
56 {
57     mpc_parser_t *Integer = mpc_new("integer");
58     mpc_parser_t *Decimal = mpc_new("decimal");
59     mpc_parser_t *Number = mpc_new("number");
60     mpc_parser_t *Operator = mpc_new("operator");
61     mpc_parser_t *Expr = mpc_new("expr");
62     mpc_parser_t *Lispy = mpc_new("lispy");
63
64     mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
65              Integer, Decimal, Number, Operator, Expr, Lispy);
66
67     puts("Lispy v0.7.0");
68     puts("Press ctrl-c to exit\n");
69
70     bool nonempty;
71     do {
72         char *input = readline("> ");
73         if ((nonempty = (input && *input))) {
74             add_history(input);
75
76             mpc_result_t parsed;
77             if (mpc_parse("<stdin>", input, Lispy, &parsed)) {
78                 mpc_ast_t *ast = parsed.output;
79
80                 double result = eval(ast);
81                 printf("%g\n", result);
82
83                 mpc_ast_delete(ast);
84             } else {
85                 mpc_err_print(parsed.error);
86                 mpc_err_delete(parsed.error);
87             }
88         }
89
90         free(input);
91     } while (nonempty);
92
93     mpc_cleanup(6, Integer, Decimal, Number, Operator, Expr, Lispy);
94
95     return 0;
96 }

```

Chunks

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

Index

Decimal: [3d](#), [4a](#)
 Expr: [3d](#), [4a](#)
 Integer: [3d](#), [4a](#)
 LISPY_GRAMMAR: [3c](#), [4b](#)
 Lispy: [3a](#), [3d](#), [4a](#), [5b](#)
 Number: [3d](#), [4a](#)
 Operator: [3d](#), [4a](#)
 add_history: [4f](#), [9f](#)
 ast: [2](#), [5c](#), [5d](#), [6a](#), [6b](#), [6c](#), [6d](#), [6e](#), [6f](#), [7f](#)
 atof: [5d](#), [9d](#)
 atoi: [9d](#)
 bool: [8a](#), [9b](#)
 eval: [2](#), [5c](#), [6b](#), [6f](#)
 eval_binop: [2](#), [6f](#)
 free: [8d](#), [9d](#)
 input: [4d](#), [4e](#), [4d](#), [4d](#), [4f](#), [5b](#), [4d](#), [8d](#)
 mpc_lang: [4b](#), [10](#)
 mpc_ast_delete: [7f](#), [10](#)
 mpc_ast_print: [10](#)
 mpc_ast_t: [2](#), [5c](#), [10](#)
 mpc_cleanup: [10](#), [4c](#), [10](#)
 mpc_err_delete: [7g](#), [10](#)
 mpc_err_print: [7g](#), [10](#)
 mpc_new: [3d](#), [10](#)
 mpc_parse: [10](#), [5b](#), [10](#)
 mpc_parser_t: [3d](#), [10](#)
 mpc_result_t: [5a](#), [10](#)
 nonempty: [8a](#), [8c](#)
 op: [2](#), [6a](#), [6c](#), [6f](#), [6a](#), [6g](#), [6a](#), [7a](#), [6a](#), [7b](#), [6a](#), [7c](#)
 parsed: [5a](#), [5a](#), [5b](#), [5c](#), [7g](#)
 printf: [7f](#), [9c](#)
 readline: [9f](#), [4d](#), [9f](#)
 result: [5c](#), [5c](#), [6b](#), [6c](#), [6f](#), [7e](#), [7f](#)
 strcmp: [6c](#), [6g](#), [7a](#), [7b](#), [7c](#), [9e](#)
 strstr: [5d](#), [6e](#), [9e](#)

Glossary

AST abstract syntax tree, a tree representation of the abstract syntactic structure of source code. [5](#), [7](#)

grammar [3](#), [4](#)

Describe what a grammar is

parser [3](#)

Describe what a parser is

PLT programming language theory, [1](#)

Describe programming language theory











REPL Read-Eval-Print Loop, [3](#), [4](#)

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	3
	Describe the evaluation strategy	5
	Describe binop evaluation	6
	Bind an error message or something	7
	Describe headers	9
	Describe what a grammar is	16
	Describe what a parser is	16
	Describe programming language theory	16
	Describe what a REPL is	16