

Build Your Own Lisp

Eric Bailey

*May 10, 2018*¹

¹ Last updated May 12, 2018

```
1  <parsing.c 1>≡
    <Include the necessary headers. 5d>

    <Define the Lispy grammar. 2b>

int main(int argc, char *argv[])
{
    <Define the language. 2c>

    <Print version and exit information. 2a>

    <Loop until the input is empty. 4e>

    <Undefine and delete the parsers. 3b>

    return 0;
}
```

Root chunk (not used in this document).

Contents

<i>Welcome</i>	2
<i>Defining the Language</i>	2
<i>R is for Read</i>	3
<i>E is for Eval(uate)</i>	4
<i>P is for Print</i>	4
<i>L is for Loop</i>	4
<i>Headers</i>	5
<i>Chunks</i>	6
<i>Index</i>	7

Welcome

What good is a REPL without a welcome message? For now, simply print the version and describe how to exit.

acronym

2a `<Print version and exit information. 2a>≡`
`puts("Lispy v0.0.1");`
`puts("Press ctrl-c to exit\n");`

Uses Lispy 2c.

This code is used in chunk 1.

Defining the Language

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

Safely load from a file using
`mpca_lang_contents`.

2b `<Define the Lispy grammar. 2b>≡`
`#define LISPY_GRAMMAR \`
 `" integer : /-?[0-9]+/ ;" " \`
 `" decimal : /-?[0-9]+\.[0-9]+/ ;" " \`
 `" number : <decimal> | <integer> ;" " \`
 `" operator : '+' | '-' | '*' | '/' ;" " \`
 `" expr : <number> | '(' <operator> <expr>+ ')';" " \`
 `" lispy : /^/ <expr>+ /$/ ;" "`

Defines:

`LISPY_GRAMMAR`, used in chunk 3a.

This code is used in chunk 1.

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

2c `<Define the language. 2c>≡`
`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 3a.

`Expr`, used in chunk 3.

`Integer`, used in chunk 3a.

`Lispy`, used in chunks 2–4.

`Number`, used in chunk 3.

`Operator`, used in chunk 3.

Uses `mpc_parser_t` 6b.

This definition is continued in chunk 3a.

This code is used in chunk 1.

Finally, using the defined grammar and each of the created parsers, we can define the Lispy language.

3a *<Define the language. 2c>+≡*
`mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
Integer, Decimal, Number,
Operator, Expr, Lispy);`
Uses `Decimal 2c`, `Expr 2c`, `Integer 2c`, `LISPY_GRAMMAR 2b`, `Lispy 2c`, `Number 2c`,
and `Operator 2c`.

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

3b *<Undefine and delete the parsers. 3b>≡*
`mpc_cleanup(4, Number, Operator, Expr, Lispy);`
Uses `Expr 2c`, `Lispy 2c`, `Number 2c`, `Operator 2c`, and `mpc_cleanup 6b`.
This code is used in chunk 1.

R is for Read

To implement the R in REPL, use `readline` from `editline`.

acronym

Add a link

3c *<Read a line of user input. 3c>≡*
`char *input = readline("> ");`
Defines:
`input`, used in chunks 3–5.
Uses `readline 6a`.
This code is used in chunk 5a.

To check whether user input is nonempty, and thus whether we should continue looping, use the following expression.

3d *<input is nonempty 3d>≡*
`input && *input`
Uses `input 3c`.
This code is used in chunk 5b.

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 `editline` history table.

3e *<Add input to the history table. 3e>≡*
`add_history(input);`
Uses `add_history 6a` and `input 3c`.
This code is used in chunk 5b.

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

3f *<Declare a variable to hold parsing results. 3f>≡*
`mpc_result_t res;`
Uses `mpc_result_t 6b` and `res 4b`.
This code is used in chunk 4b.

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

4a *⟨The input can be parsed as Lispy code. 4a⟩*≡
`mpc_parse("<stdin>", input, Lispy, &res)`

Uses Lispy 2c, input 3c, `mpc_parse` 6b, and `res` 4b.
 This code is used in chunk 4b.

E is for Eval(uate)

Evaluate the AST

4b *⟨Eval(uate) user input and print the result. 4b⟩*≡
⟨Declare a variable to hold parsing results. 3f⟩
`if (⟨The input can be parsed as Lispy code. 4a⟩) {`
 ⟨Print and delete the AST. 4c⟩
`} else {`
 ⟨Print and delete the error. 4d⟩
`}`

Defines:

`res`, used in chunks 4b, 3, and 4.
 This code is used in chunk 5b.

P is for Print

For now, simply print the AST upon success,

acronym

4c *⟨Print and delete the AST. 4c⟩*≡
`mpc_ast_print(res.output);`
`mpc_ast_delete(res.output);`

Uses `mpc_ast_delete` 6b, `mpc_ast_print` 6b, and `res` 4b.
 This code is used in chunk 4b.

or the error upon failure.

4d *⟨Print and delete the error. 4d⟩*≡
`mpc_err_print(res.error);`
`mpc_err_delete(res.error);`

Uses `mpc_err_delete` 6b, `mpc_err_print` 6b, and `res` 4b.
 This code is used in chunk 4b.

L is for Loop

4e *⟨Loop until the input is empty. 4e⟩*≡
`bool nonempty;`
`do {`
 ⟨Read, eval(uate), and print. 5a⟩
`} while (nonempty);`

Defines:

`nonempty`, used in chunk 5b.
 Uses `bool` 5e.
 This code is used in chunk 1.

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

5a *<Read, eval(uate), and print. 5a>≡*
<Read a line of user input. 3c>

This definition is continued in chunk 5.

This code is used in chunk 4e.

If, and only if, it's not empty, add it to the history table, evaluate it, and print the result.

5b *<Read, eval(uate), and print. 5a>+≡*
 if ((nonempty = (<input is nonempty 3d>))) {
 <Add input to the history table. 3e>
 <Eval(uate) user input and print the result. 4b>
 }

Uses *nonempty* 4e.

Dealloc the space pointed to by *input*, making it available for further allocation.

5c *<Read, eval(uate), and print. 5a>+≡*
 free(input);

Uses *free* 5g and *input* 3c.

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

Headers

5d *<Include the necessary headers. 5d>≡*
 <Include the boolean type and values. 5e>
 <Include the standard I/O functions. 5f>
 <Include the standard library definitions. 5g>

 <Include the line editing functions from libedit. 6a>
 <Include the micro parser combinator definitions. 6b>

This code is used in chunk 1.

5e *<Include the boolean type and values. 5e>≡*
 #include <stdbool.h>

Defines:

bool, used in chunk 4e.

This code is used in chunk 5d.

5f *<Include the standard I/O functions. 5f>≡*
 #include <stdio.h>

Defines:

printf, never used.

This code is used in chunk 5d.

5g *<Include the standard library definitions. 5g>≡*
 #include <stdlib.h>

Defines:

free, used in chunk 5c.

This code is used in chunk 5d.

6a *<Include the line editing functions from libedit. 6a>*≡
`#include <editline/readline.h>`

Defines:

`add_history`, used in chunk 3e.
`readline`, used in chunks 6a and 3c.

This code is used in chunk 5d.

6b *<Include the micro parser combinator definitions. 6b>*≡
`#include <mpc.h>`

Defines:

`mpc_ast_delete`, used in chunk 4c.
`mpc_ast_print`, used in chunk 4c.
`mpc_cleanup`, used in chunks 6b and 3b.
`mpc_err_delete`, used in chunk 4d.
`mpc_err_print`, used in chunk 4d.
`mpc_parse`, used in chunks 6b and 4a.
`mpc_parser_t`, used in chunk 2c.
`mpc_result_t`, used in chunk 3f.

This code is used in chunk 5d.

Add a full listing

Chunks

<Add input to the history table. 3e> [3e](#), [5b](#)
<Declare a variable to hold parsing results. 3f> [3f](#), [4b](#)
<Define the Lispy grammar. 2b> [1](#), [2b](#)
<Define the language. 2c> [1](#), [2c](#), [3a](#)
<Eval(uate) user input and print the result. 4b> [4b](#), [5b](#)
<Include the boolean type and values. 5e> [5d](#), [5e](#)
<Include the line editing functions from libedit. 6a> [5d](#), [6a](#)
<Include the micro parser combinator definitions. 6b> [5d](#), [6b](#)
<Include the necessary headers. 5d> [1](#), [5d](#)
<Include the standard I/O functions. 5f> [5d](#), [5f](#)
<Include the standard library definitions. 5g> [5d](#), [5g](#)
<Loop until the input is empty. 4e> [1](#), [4e](#)
<Print and delete the AST. 4c> [4b](#), [4c](#)
<Print and delete the error. 4d> [4b](#), [4d](#)
<Print version and exit information. 2a> [1](#), [2a](#)
<Read a line of user input. 3c> [3c](#), [5a](#)
<Read, eval(uate), and print. 5a> [4e](#), [5a](#), [5b](#), [5c](#)
<The input can be parsed as Lispy code. 4a> [4a](#), [4b](#)
<Undefine and delete the parsers. 3b> [1](#), [3b](#)
<input is nonempty 3d> [3d](#), [5b](#)
<parsing.c 1> [1](#)

Index

Decimal: [2c](#), [3a](#)
Expr: [2c](#), [3a](#), [3b](#)
Integer: [2c](#), [3a](#)
LISPY_GRAMMAR: [2b](#), [3a](#)
Lispy: [2a](#), [2c](#), [3a](#), [3b](#), [4a](#)
Number: [2c](#), [3a](#), [3b](#)
Operator: [2c](#), [3a](#), [3b](#)
add_history: [3e](#), [6a](#)
bool: [4e](#), [5e](#)
free: [5c](#), [5g](#)
input: [3c](#), [3d](#), [3c](#), [3c](#), [3e](#), [4a](#), [3c](#), [5c](#)
mpc_ast_delete: [4c](#), [6b](#)
mpc_ast_print: [4c](#), [6b](#)
mpc_cleanup: [6b](#), [3b](#), [6b](#)
mpc_err_delete: [4d](#), [6b](#)
mpc_err_print: [4d](#), [6b](#)
mpc_parse: [6b](#), [4a](#), [6b](#)
mpc_parser_t: [2c](#), [6b](#)
mpc_result_t: [3f](#), [6b](#)
nonempty: [4e](#), [5b](#)
printf: [5f](#)
readline: [6a](#), [3c](#), [6a](#)
res: [4b](#), [3f](#), [4a](#), [4b](#), [4c](#), [4d](#)

[Add a bibliography](#)

[Add a glossary](#)

Todo list

■ acronym	2
■ Safely load from a file using <code>mpca_lang_contents</code>	2
■ acronym	3
■ Add a link	3
■ Evaluate the AST	4
■ acronym	4
■ Add a full listing	6
■ Add a bibliography	7
■ Add a glossary	7