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

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 2d.
  Expr, used in chunk 2d.
  Integer, used in chunk 2d.
  Lispy, used in chunks 2 and 4a.
  Number, used in chunk 2d.
  Operator, used in chunk 2d.
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* 2d⟩,

2d ⟨*created parsers* 2d⟩≡
```
Integer, Decimal, Number, Operator, Expr, Lispy
```
Uses Decimal 2c, Expr 2c, Integer 2c, Lispy 2c, Number 2c, and Operator 2c.
This code is used in chunk 3.

we can define the Lispy language.

3a ⟨*Define the language.* 2c⟩+≡
```
mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
          ⟨created parsers 2d⟩);
```
Uses LISPY_GRAMMAR 2b.

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(6, ⟨created parsers 2d⟩);
```
Uses 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` $\neq$ `NULL`, and `*input` is functionally equivalent to `input[0]` $\neq$ `'\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)

Evalute 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 futher 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

Index

Add a bibliography

Add a glossary

Todo list