Build Your Own Lisp

Eric Bailey

May 10, 2018 [1]

[1] Last updated May 13, 2018

Write an abstract

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 v0.6.1");
puts("Press ctrl-c to exit\n");
```
Uses Lispy 2d.
This code is used in chunk 10.

Defining the Language

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

2b ⟨*lispy.mpc* 2b⟩≡
```
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

2c ⟨*Load the Lispy grammar.* 2c⟩≡
```
static const char LISPY_GRAMMAR[] = {
#include "lispy.xxd"
};
```
Defines:
LISPY_GRAMMAR, used in chunk 3b.
This code is used in chunk 10.

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

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

2d ⟨*Define the language.* 2d⟩≡
```
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 3a.
Integer, used in chunk 3a.
Lispy, used in chunks 2–4.
Number, used in chunk 3a.
Operator, used in chunk 3a.
Uses mpc_new 9 and mpc_parser_t 9.
This definition is continued in chunk 3b.
This code is used in chunk 10.

Finally, using the defined grammar and each of the ⟨*created parsers* 3a⟩,

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

... we can define the Lispy language.

3b ⟨*Define the language.* 2d⟩+≡
```
mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
          ⟨created parsers 3a⟩);
```
Uses LISPY_GRAMMAR 2c and mpca_lang 9.

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.

3c ⟨*Undefine and delete the parsers.* 3c⟩≡
```
mpc_cleanup(6, ⟨created parsers 3a⟩);
```
Uses mpc_cleanup 9.
This code is used in chunk 10.

R is for Read

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

acronym

Add a link

3d ⟨*Read a line of user input.* 3d⟩≡
```
char *input = readline("> ");
```
Defines:
input, used in chunks 3, 4, and 7d.
Uses readline 8f.
This code is used in chunk 7b.

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

3e ⟨input *is nonempty* 3e⟩≡
```
input && *input
```
Uses input 3d.
This code is used in chunk 7c.

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.

3f ⟨*Add* input *to the history table.* 3f⟩≡
```
add_history(input);
```
Uses add_history 8f and input 3d.
This code is used in chunk 7c.

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

4a ⟨*Declare a variable to hold parsing results.* 4a⟩≡
```
mpc_result_t parsed;
```
Defines:
`parsed`, used in chunks 4 and 6g.
Uses `mpc_result_t` 9.
This code is used in chunk 7c.

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

4b ⟨*the input can be parsed as Lispy code* 4b⟩≡
```
mpc_parse("<stdin>", input, Lispy, &parsed)
```
Uses `Lispy` 2d, `input` 3d, `mpc_parse` 9, and `parsed` 4a.
This code is used in chunk 7c.

E is for Eval(uate)

Evalute the AST

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.

4c ⟨*Eval(uate) the input.* 4c⟩≡
```
mpc_ast_t *ast = parsed.output;

double result = eval(ast);
```
Defines:
`ast`, used in chunks 4–6 and 10.
`result`, used in chunks 4–6.
Uses `eval` 10, `mpc_ast_t` 9, and `parsed` 4a.
This code is used in chunk 7c.

Describe the evaluation strategy

If the expression is tagged as a number, return it directly.

4d ⟨*Eval(uate) the AST.* 4d⟩≡
```
if (strstr(ast->tag, "number"))
    return atof(ast->contents);
```
Uses `ast` 4c, `atof` 8d, and `strstr` 8e.
This definition is continued in chunks 4–6.
This code is used in chunk 10.

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

4e ⟨*Eval(uate) the AST.* 4d⟩+≡
```
int i = 0;
```

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

5a ⟨*Eval(uate) the AST.* 4d⟩+≡
```
char *op = ast->children[++i]->contents;
```

Defines:
op, used in chunks 5, 6, and 10.
Uses ast 4c.

Evaluate the next child, which is the first operand.

5b ⟨*Eval(uate) the AST.* 4d⟩+≡
```
double result = eval(ast->children[++i]);
```

Uses ast 4c, eval 10, and result 4c.

If the operation is unary subtraction, negate the operand.

5c ⟨*Eval(uate) the AST.* 4d⟩+≡
```
if (!strcmp(op, "-") && ast->children_num == 4)
    return -result;
```

Uses ast 4c, op 5a, result 4c, and strcmp 8e.

While there are more children, i.e.

5d ⟨*there are more operands* 5d⟩≡
```
++i < ast->children_num
```
Uses ast 4c.
This code is used in chunk 6e.

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

5e ⟨*the next child is an expression* 5e⟩≡
```
strstr(ast->children[i]->tag, "expr")
```
Uses ast 4c and strstr 8e.
This code is used in chunk 6e.

... evaluate the next operand.

5f ⟨*Eval(uate) the next operand.* 5f⟩≡
```
result = eval_binop(op, result, eval(ast->children[i]));
```
Uses ast 4c, eval 10, eval_binop 10, op 5a, and result 4c.
This code is used in chunk 6e.

Describe binop evaluation

If the op is "+", perform addition.

5g ⟨*Eval(uate) a binary operation.* 5g⟩≡
```
if (!strcmp(op, "+"))
    return x + y;
```

Uses op 5a and strcmp 8e.
This definition is continued in chunk 6.
This code is used in chunk 10.

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

6a ⟨*Eval(uate) a binary operation.* 5g⟩+≡
```
if (!strcmp(op, "-"))
    return x - y;
```
Uses `op` 5a and `strcmp` 8e.

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

6b ⟨*Eval(uate) a binary operation.* 5g⟩+≡
```
if (!strcmp(op, "*"))
    return x * y;
```
Uses `op` 5a and `strcmp` 8e.

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

6c ⟨*Eval(uate) a binary operation.* 5g⟩+≡
```
if (!strcmp(op, "/"))
    return x / y;
```
Uses `op` 5a and `strcmp` 8e.

Otherwise, return `0`.

Bind an error message or something

6d ⟨*Eval(uate) a binary operation.* 5g⟩+≡
```
return 0;
```
Express the recursive operand evaluation as a `while` loop, and return the result.

6e ⟨*Eval(uate) the AST.* 4d⟩+≡
```
while (⟨there are more operands 5d⟩
       && ⟨the next child is an expression 5e⟩)
    ⟨Eval(uate) the next operand. 5f⟩

return result;
```
Uses `result` 4c.

P is for Print

Upon success, print the result and delete the AST.

acronym

6f ⟨*Print the result and delete the AST.* 6f⟩≡
```
printf("%g\n", result);

mpc_ast_delete(ast);
```
Uses `ast` 4c, `mpc_ast_delete` 9, `printf` 8c, and `result` 4c.
This code is used in chunk 7c.

Print and delete the error upon failure.

6g ⟨*Print and delete the error.* 6g⟩≡
```
mpc_err_print(parsed.error);
mpc_err_delete(parsed.error);
```
Uses `mpc_err_delete` 9, `mpc_err_print` 9, and `parsed` 4a.
This code is used in chunk 7c.

L is for Loop

7a ⟨*Loop until the input is empty.* 7a⟩≡
```
bool nonempty;
do {
  ⟨Read, eval(uate), and print. 7b⟩
} while (nonempty);
```
Defines:
  nonempty, used in chunk 7c.
Uses bool 8b.
This code is used in chunk 10.

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

7b ⟨*Read, eval(uate), and print.* 7b⟩≡
```
⟨Read a line of user input. 3d⟩
```
This definition is continued in chunk 7.
This code is used in chunk 7a.

If, and only if, it's not empty, add it to the history table, **E**val(uate) it, and **P**rint the result.

7c ⟨*Read, eval(uate), and print.* 7b⟩+≡
```
if ((nonempty = (⟨input is nonempty 3e⟩))) {
    ⟨Add input to the history table. 3f⟩

    ⟨Declare a variable to hold parsing results. 4a⟩
    if (⟨the input can be parsed as Lispy code 4b⟩) {
        ⟨Eval(uate) the input. 4c⟩
        ⟨Print the result and delete the AST. 6f⟩
    } else {
        ⟨Print and delete the error. 6g⟩
    }
}
```

Uses nonempty 7a.

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

7d ⟨*Read, eval(uate), and print.* 7b⟩+≡
```
free(input);
```
Uses free 8d and input 3d.

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

Headers

Describe headers

8a ⟨*Include the necessary headers.* 8a⟩≡
⟨*Include the boolean type and values.* 8b⟩
⟨*Include the standard I/O functions.* 8c⟩
⟨*Include the standard library definitions.* 8d⟩
⟨*Include some string operations.* 8e⟩

⟨*Include the line editing functions from libedit.* 8f⟩
⟨*Include the micro parser combinator definitions.* 9⟩
This code is used in chunk 10.

8b ⟨*Include the boolean type and values.* 8b⟩≡
```
#include <stdbool.h>
```
Defines:
`bool`, used in chunk 7a.
This code is used in chunk 8a.

8c ⟨*Include the standard I/O functions.* 8c⟩≡
```
#include <stdio.h>
```
Defines:
`printf`, used in chunk 6f.
This code is used in chunk 8a.

8d ⟨*Include the standard library definitions.* 8d⟩≡
```
#include <stdlib.h>
```
Defines:
`atof`, used in chunk 4d.
`atoi`, never used.
`free`, used in chunk 7d.
This code is used in chunk 8a.

8e ⟨*Include some string operations.* 8e⟩≡
```
#include <string.h>
```
Defines:
`strcmp`, used in chunks 5 and 6.
`strstr`, used in chunks 4d and 5e.
This code is used in chunk 8a.

8f ⟨*Include the line editing functions from libedit.* 8f⟩≡
```
#include <editline/readline.h>
```
Defines:
`add_history`, used in chunk 3f.
`readline`, used in chunks 8f and 3d.
This code is used in chunk 8a.

9 ⟨*Include the micro parser combinator definitions.* 9⟩≡

```
#include <mpc.h>
```

Defines:
`mpca_lang`, used in chunk 3b.
`mpc_ast_delete`, used in chunk 6f.
`mpc_ast_print`, never used.
`mpc_ast_t`, used in chunks 4c and 10.
`mpc_cleanup`, used in chunks 9 and 3c.
`mpc_err_delete`, used in chunk 6g.
`mpc_err_print`, used in chunk 6g.
`mpc_new`, used in chunk 2d.
`mpc_parse`, used in chunks 9 and 4b.
`mpc_parser_t`, used in chunk 2d.
`mpc_result_t`, used in chunk 4a.
This code is used in chunk 8a.

Full Listing

10 ⟨*parsing.c* 10⟩≡

⟨*Include the necessary headers.* 8a⟩

⟨*Load the Lispy grammar.* 2c⟩

```
double eval_binop(char *op, double x, double y)
{
  ⟨Eval(uate) a binary operation. 5g⟩
}

double eval(mpc_ast_t *ast)
{
    ⟨Eval(uate) the AST. 4d⟩
}

int main(int argc, char *argv[])
{
    ⟨Define the language. 2d⟩

    ⟨Print version and exit information. 2a⟩

    ⟨Loop until the input is empty. 7a⟩

    ⟨Undefine and delete the parsers. 3c⟩

    return 0;
}
```

Defines:
  `eval`, used in chunks 4 and 5.
  `eval_binop`, used in chunk 5f.
Uses `ast` 4c, `mpc_ast_t` 9, and `op` 5a.
Root chunk (not used in this document).

parsing.c:

```c
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <editline/readline.h>
#include <mpc.h>


static const char LISPY_GRAMMAR[] = {
#include "lispy.xxd"
};


double eval_binop(char *op, double x, double y)
{
    if (!strcmp(op, "+"))
        return x + y;

    if (!strcmp(op, "-"))
        return x - y;

    if (!strcmp(op, "*"))
        return x * y;

    if (!strcmp(op, "/"))
        return x / y;

    return 0;
}


double eval(mpc_ast_t * ast)
{
    if (strstr(ast->tag, "number"))
        return atof(ast->contents);

    int i = 0;

    char *op = ast->children[++i]->contents;

    double result = eval(ast->children[++i]);

    if (!strcmp(op, "-") && ast->children_num == 4)
        return -result;

    while (++i < ast->children_num
           && strstr(ast->children[i]->tag, "expr"))
        result = eval_binop(op, result, eval(ast->children[i]));

```

```
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 v0.6.1");
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

Index

Add a bibliography

Add a glossary

Todo list