Lispy: a simple Lisp-like language

Eric Bailey

May 10, 2018 [1]

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 Noweb[3].

Contents

[1] Current version: `0.7.0`. Last updated May 13, 2018.

[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

Outline

Describe the outline

```
2     ⟨lispy.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.

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.

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.

[4] Daniel Holden. Micro Parser Combinators. `https://github.com/orangeduck/mpc`, 2018b. Accessed: 2018-05-13

[5] Jess Thrysoee. Editline Library (libedit) – port of netbsd command line editor library. `http://thrysoee.dk/editline/`, 2017. Accessed: 2018-05-13

[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 ⟨*Declare a variable to hold parsing results.* 5a⟩≡

```
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 ⟨*the input can be parsed as Lispy code* 5b⟩≡

```
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 ⟨*Eval(uate) the input.* 5c⟩≡

```
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 ⟨*Eval(uate) the AST.* 5d⟩≡

```
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 interate through the children of the AST.

5e ⟨*Eval(uate) the AST.* 5d⟩+≡

```
int i = 0;
```

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

6a ⟨*Eval(uate) the AST.* 5d⟩+≡

```
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 ⟨*Eval(uate) the AST.* 5d⟩+≡

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

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

If the operation is unary subtraction, negate the operand.

6c ⟨*Eval(uate) the AST.* 5d⟩+≡

```
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 ⟨*there are more operands* 6d⟩≡

```
++i < ast->children_num
```

Uses `ast` 5c.
This code is used in chunk 7e.

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

6e ⟨*the next child is an expression* 6e⟩≡

```
strstr(ast->children[i]->tag, "expr")
```

Uses `ast` 5c and `strstr` 9e.
This code is used in chunk 7e.

... evaluate the next operand.

6f ⟨*Eval(uate) the next operand.* 6f⟩≡

```
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 ⟨*Eval(uate) a binary operation.* 6g⟩≡

```
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`.

Bind an error message or something

7d ⟨*Eval(uate) a binary operation.* 6g⟩+≡

```
return 0;
```

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 ⟨*Loop until the input is empty.* 8a⟩≡

```
bool nonempty;
do {
  ⟨Read, eval(uate), and print. 8b⟩
} 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, **R**ead a line of user input.

8b ⟨*Read, eval(uate), and print.* 8b⟩≡

```
⟨Read a line of user input. 4d⟩
```

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, **E**val(uate) it, and **P**rint the result.

8c ⟨*Read, eval(uate), and print.* 8b⟩+≡

```
if ((nonempty = (⟨input is nonempty 4e⟩))) {
    ⟨Add input to the history table. 4f⟩

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

Uses `nonempty` 8a.

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

8d ⟨*Read, eval(uate), and print.* 8b⟩+≡

```
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:

```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]));

    return result;
}


int main(int argc, char *argv[])
{
    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");

    mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
              Integer, Decimal, Number, Operator, Expr, Lispy);

    puts("Lispy v0.7.0");
    puts("Press ctrl-c to exit\n");

    bool nonempty;
    do {
        char *input = readline("> ");
        if ((nonempty = (input && *input))) {
            add_history(input);

            mpc_result_t parsed;
            if (mpc_parse("<stdin>", input, Lispy, &parsed)) {
                mpc_ast_t *ast = parsed.output;

                double result = eval(ast);
                printf("%g\n", result);

                mpc_ast_delete(ast);
            } else {
                mpc_err_print(parsed.error);
                mpc_err_delete(parsed.error);
            }
        }

        free(input);
    } while (nonempty);

    mpc_cleanup(6, Integer, Decimal, Number, Operator, Expr, Lispy);

    return 0;
}
```

Chunks

Index

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