Lispy: a simple Lisp-like language

Eric Bailey

May 10, 2018 [1]

[1] Current version: `0.8.1`. Last updated May 14, 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 Noweb[3].

[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

Outline

Describe the outline

2a ⟨*lispy.c* 2a⟩≡
```
  ⟨Include the necessary headers. 12b⟩


  ⟨Load the Lispy grammar. 4c⟩


  ⟨Define possible lval and error types. 11a⟩

  ⟨Define the Lispy data structures. 10c⟩
```

This definition is continued in chunks 2 and 3.
Root chunk (not used in this document).

2b ⟨*lispy.c* 2a⟩+≡
```
  void lval_print(lval val)
  {
      ⟨Print a Lispy value. 9a⟩
  }
```

Defines:
  `lval_print`, used in chunk 2c.
Uses `lval` 10c.

2c ⟨*lispy.c* 2a⟩+≡
```
  void lval_println(lval val)
  {
      lval_print(val);
      putchar('\n');
  }
```

Defines:
  `lval_println`, used in chunk 8f.
Uses `lval` 10c and `lval_print` 2b.

3a ⟨*lispy.c* 2a⟩+≡

```
lval eval_binop(char *op, lval x, lval y)
{
  ⟨Eval(uate) a binary operation. 7g⟩
}


lval eval(mpc_ast_t *ast)
{
    ⟨Eval(uate) the AST. 6d⟩
}
```

Defines:
  `eval`, used in chunks 6 and 7.
  `eval_binop`, used in chunk 7f.
Uses `ast` 6c, `lval` 10c, `mpc_ast_t` 13b, and `op` 7a.

3b ⟨*lispy.c* 2a⟩+≡

```
int main(int argc, char *argv[])
{
    ⟨Define the language. 4d⟩

    ⟨Print version and exit information. 4a⟩

    ⟨Loop until the input is empty. 9c⟩

    ⟨Undefine and delete the parsers. 5c⟩

    return 0;
}
```

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.

4a ⟨*Print version and exit information.* 4a⟩≡
```
puts("Lispy v0.8.1");
puts("Press ctrl-c to exit\n");
```
Uses Lispy 4d.
This code is used in chunk 3b.

Defining the Language

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

4b ⟨*lispy.mpc* 4b⟩≡
```
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

4c ⟨*Load the Lispy grammar.* 4c⟩≡
```
static const char LISPY_GRAMMAR[] = {
#include "lispy.xxd"
};
```
Defines:
LISPY_GRAMMAR, used in chunk 5b.
This code is used in chunk 2a.

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

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

4d ⟨*Define the language.* 4d⟩≡
```
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 5a.
Expr, used in chunk 5a.
Integer, used in chunk 5a.
Lispy, used in chunks 4–6.
Number, used in chunk 5a.
Operator, used in chunk 5a.
Uses mpc_new 13b and mpc_parser_t 13b.
This definition is continued in chunk 5b.
This code is used in chunk 3b.

Finally, using the defined *grammar* and each of the ⟨*created parsers* 5a⟩,

5a ⟨*created parsers* 5a⟩≡

```
Integer, Decimal, Number, Operator, Expr, Lispy
```

Uses `Decimal` 4d, `Expr` 4d, `Integer` 4d, `Lispy` 4d, `Number` 4d, and `Operator` 4d.
This code is used in chunk 5.

... we can define the Lispy language.

5b ⟨*Define the language.* 4d⟩+≡

```
mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
          ⟨created parsers 5a⟩);
```

Uses `LISPY_GRAMMAR` 4c and `mpca_lang` 13b.

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.

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

5c ⟨*Undefine and delete the parsers.* 5c⟩≡

```
mpc_cleanup(6, ⟨created parsers 5a⟩);
```

Uses `mpc_cleanup` 13b.
This code is used in chunk 3b.

R is for Read

To implement the R in REPL, use `readline` from `libedit`[5].

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

5d ⟨*Read a line of user input.* 5d⟩≡

```
char *input = readline("> ");
```

Defines:
  `input`, used in chunks 5, 6, and 10b.
Uses `readline` 13a.
This code is used in chunk 9d.

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

5e ⟨`input` *is nonempty* 5e⟩≡

```
input && *input
```

Uses `input` 5d.
This code is used in chunk 10a.

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.

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

5f ⟨*Add* `input` *to the history table.* 5f⟩≡

```
add_history(input);
```

Uses `add_history` 13a and `input` 5d.
This code is used in chunk 10a.

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

6a ⟨*Declare a variable to hold parsing results.* 6a⟩≡
```
mpc_result_t parsed;
```
Defines:
`parsed`, used in chunks 6 and 9b.
Uses `mpc_result_t` 13b.
This code is used in chunk 10a.

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

6b ⟨*the input can be parsed as Lispy code* 6b⟩≡
```
mpc_parse("<stdin>", input, Lispy, &parsed)
```
Uses `Lispy` 4d, `input` 5d, `mpc_parse` 13b, and `parsed` 6a.
This code is used in chunk 10a.

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.

6c ⟨*Eval(uate) the input.* 6c⟩≡
```
mpc_ast_t *ast = parsed.output;

lval result = eval(ast);
```
Defines:
`ast`, used in chunks 3a and 6–8.
`result`, used in chunks 6–8.
Uses `eval` 3a, `lval` 10c, `mpc_ast_t` 13b, and `parsed` 6a.
This code is used in chunk 10a.

Describe the evaluation strategy

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

6d ⟨*Eval(uate) the AST.* 6d⟩≡
```
if (strstr(ast->tag, "number")) {
    errno = 0;
    double x = strtod(ast->contents, NULL);
    return errno != ERANGE ? lval_num(x) : lval_err(LERR_BAD_NUM);
}
```
Uses `LERR_BAD_NUM` 11c, `ast` 6c, `lval_err` 12a, `lval_num` 11b, `strstr` 12f, and `strtod` 12e.
This definition is continued in chunks 6–8.
This code is used in chunk 3a.

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.

6e ⟨*Eval(uate) the AST.* 6d⟩+≡
```
int i = 0;
```

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

7a ⟨*Eval(uate) the AST.* 6d⟩+≡

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

Defines:
`op`, used in chunks 3a, 7, and 8.
Uses `ast` 6c.

Evaluate the next child, which is the first operand.

7b ⟨*Eval(uate) the AST.* 6d⟩+≡

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

Uses `ast` 6c, `eval` 3a, `lval` 10c, and `result` 6c.

If the operation is unary subtraction, negate the operand.

7c ⟨*Eval(uate) the AST.* 6d⟩+≡

```
if (!strcmp(op, "-") && ast->children_num == 4) {
    result.num = -result.num;
    return result;
}
```

Uses `ast` 6c, `op` 7a, `result` 6c, and `strcmp` 12f.

While there are more children, i.e.

7d ⟨*there are more operands* 7d⟩≡

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

Uses `ast` 6c.
This code is used in chunk 8e.

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

7e ⟨*the next child is an expression* 7e⟩≡

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

Uses `ast` 6c and `strstr` 12f.
This code is used in chunk 8e.

... evaluate the next operand.

7f ⟨*Eval(uate) the next operand.* 7f⟩≡

```
result = eval_binop(op, result, eval(ast->children[i]));
```

Uses `ast` 6c, `eval` 3a, `eval_binop` 3a, `op` 7a, and `result` 6c.
This code is used in chunk 8e.

Describe binop evaluation

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

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

```
if (!strcmp(op, "+"))
    return lval_num(x.num + y.num);
```

Uses `lval_num` 11b, `op` 7a, and `strcmp` 12f.
This definition is continued in chunk 8.
This code is used in chunk 3a.

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

8a ⟨*Eval(uate) a binary operation.* 7g⟩+≡
```
if (!strcmp(op, "-"))
    return lval_num(x.num - y.num);
```

Uses `lval_num` 11b, `op` 7a, and `strcmp` 12f.

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

8b ⟨*Eval(uate) a binary operation.* 7g⟩+≡
```
if (!strcmp(op, "*"))
    return lval_num(x.num * y.num);
```

Uses `lval_num` 11b, `op` 7a, and `strcmp` 12f.

If the `op` is `"/"`, perform division, returning the appropriate error when trying to divide by zero.

8c ⟨*Eval(uate) a binary operation.* 7g⟩+≡
```
if (!strcmp(op, "/")) {
    return !y.num
        ? lval_err(LERR_DIV_ZERO)
        : lval_num(x.num / y.num);
}
```

Uses `LERR_DIV_ZERO` 11c, `lval_err` 12a, `lval_num` 11b, `op` 7a, and `strcmp` 12f.

Otherwise, return a `LERR_BAD_OP` error.

8d ⟨*Eval(uate) a binary operation.* 7g⟩+≡
```
return lval_err(LERR_DIV_ZERO);
```
Uses `LERR_DIV_ZERO` 11c and `lval_err` 12a.

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

8e ⟨*Eval(uate) the AST.* 6d⟩+≡
```
while (⟨there are more operands 7d⟩
       && ⟨the next child is an expression 7e⟩)
    ⟨Eval(uate) the next operand. 7f⟩

return result;
```
Uses `result` 6c.

P is for Print

Upon success, print the result and delete the AST.

8f ⟨*Print the result and delete the AST.* 8f⟩≡
```
lval_println(result);

mpc_ast_delete(ast);
```
Uses `ast` 6c, `lval_println` 2c, `mpc_ast_delete` 13b, and `result` 6c.
This code is used in chunk 10a.

9a ⟨*Print a Lispy value.* 9a⟩≡
```
switch (val.type) {
case LVAL_NUM:
    printf("%g", val.num);
    break;

case LVAL_ERR:
    switch (val.err) {
    case LERR_BAD_OP:
        puts("Error: invalid operator");
        break;
    case LERR_BAD_NUM:
        puts("Error: invalid number");
        break;
    case LERR_DIV_ZERO:
        fputs("Error: division by zero", stdout);
        break;
    }
    break;
}
```
Uses `LERR_BAD_NUM` 11c, `LERR_BAD_OP` 11c, `LERR_DIV_ZERO` 11c, `LVAL_ERR` 11a, `LVAL_NUM` 11a, and `printf` 12d.
This code is used in chunk 2b.

Print and delete the error upon failure.

9b ⟨*Print and delete the error.* 9b⟩≡
```
mpc_err_print(parsed.error);
mpc_err_delete(parsed.error);
```
Uses `mpc_err_delete` 13b, `mpc_err_print` 13b, and `parsed` 6a.
This code is used in chunk 10a.

L is for Loop

9c ⟨*Loop until the input is empty.* 9c⟩≡
```
bool nonempty;
do {
  ⟨Read, eval(uate), and print. 9d⟩
} while (nonempty);
```
Defines:
`nonempty`, used in chunk 10a.
Uses `bool` 12c.
This code is used in chunk 3b.

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

9d ⟨*Read, eval(uate), and print.* 9d⟩≡
```
⟨Read a line of user input. 5d⟩
```
This definition is continued in chunk 10.
This code is used in chunk 9c.

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

10a ⟨*Read, eval(uate), and print.* 9d⟩+≡
```
  if ((nonempty = (⟨input is nonempty 5e⟩))) {
      ⟨Add input to the history table. 5f⟩

      ⟨Declare a variable to hold parsing results. 6a⟩
      if (⟨the input can be parsed as Lispy code 6b⟩) {
          ⟨Eval(uate) the input. 6c⟩
          ⟨Print the result and delete the AST. 8f⟩
      } else {
          ⟨Print and delete the error. 9b⟩
      }
  }
```
Uses `nonempty` 9c.

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

10b ⟨*Read, eval(uate), and print.* 9d⟩+≡
```
  free(input);
```
Uses `free` 12e and `input` 5d.

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

Error Handling

Describe this struct

10c ⟨*Define the Lispy data structures.* 10c⟩≡
```
  typedef struct {
      lval_type_t type;
      union {
          double num;
          lval_err_t err;
      };
  } lval;
```
Defines:
`lval`, used in chunks 2, 3a, 6c, 7b, 11b, and 12a.
Uses `lval_err_t` 11c and `lval_type_t` 11a.
This definition is continued in chunks 11b and 12a.
This code is used in chunk 2a.

A Lispy value can be either a number or an error.

11a ⟨*Define possible lval and error types.* 11a⟩≡

```
typedef enum {
    LVAL_NUM,
    LVAL_ERR
} lval_type_t;
```

Defines:
  `LVAL_ERR`, used in chunks 9a and 12a.
  `LVAL_NUM`, used in chunks 9a and 11b.
  `lval_type_t`, used in chunk 10c.
This definition is continued in chunk 11c.
This code is used in chunk 2a.

Define a constructor for numbers.

11b ⟨*Define the Lispy data structures.* 10c⟩+≡

```
lval lval_num(double num)
{
    lval val;
    val.type = LVAL_NUM;
    val.num = num;

    return val;
}
```

Defines:
  `lval_num`, used in chunks 6–8.
Uses `LVAL_NUM` 11a and `lval` 10c.

Possible reasons for error include division by zero, a bad operator, and a bad number.

11c ⟨*Define possible lval and error types.* 11a⟩+≡

```
typedef enum {
    LERR_DIV_ZERO,
    LERR_BAD_OP,
    LERR_BAD_NUM
} lval_err_t;
```

Defines:
  `LERR_BAD_NUM`, used in chunks 6d and 9a.
  `LERR_BAD_OP`, used in chunk 9a.
  `LERR_DIV_ZERO`, used in chunks 8 and 9a.
  `lval_err_t`, used in chunks 10c and 12a.

Define a constructor for errors.

12a ⟨*Define the Lispy data structures.* 10c⟩+≡
```
lval lval_err(lval_err_t err)
{
    lval val;
    val.type = LVAL_ERR;
    val.err = err;

    return val;
}
```
Defines:
`lval_err`, used in chunks 6d and 8.
Uses `LVAL_ERR` 11a, `lval` 10c, and `lval_err_t` 11c.

Headers

Describe headers

12b ⟨*Include the necessary headers.* 12b⟩≡
⟨*Include the boolean type and values.* 12c⟩
⟨*Include the standard I/O functions.* 12d⟩
⟨*Include the standard library definitions.* 12e⟩
⟨*Include some string operations.* 12f⟩

⟨*Include the line editing functions from libedit.* 13a⟩
⟨*Include the micro parser combinator definitions.* 13b⟩

This code is used in chunk 2a.

12c ⟨*Include the boolean type and values.* 12c⟩≡
```
#include <stdbool.h>
```
Defines:
`bool`, used in chunk 9c.
This code is used in chunk 12b.

12d ⟨*Include the standard I/O functions.* 12d⟩≡
```
#include <stdio.h>
```
Defines:
`printf`, used in chunk 9a.
This code is used in chunk 12b.

12e ⟨*Include the standard library definitions.* 12e⟩≡
```
#include <stdlib.h>
```
Defines:
`free`, used in chunk 10b.
`strtod`, used in chunk 6d.
This code is used in chunk 12b.

12f ⟨*Include some string operations.* 12f⟩≡
```
#include <string.h>
```
Defines:
`strcmp`, used in chunks 7 and 8.
`strstr`, used in chunks 6d and 7e.
This code is used in chunk 12b.

13a ⟨*Include the line editing functions from libedit.* 13a⟩≡

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

Defines:
  `add_history`, used in chunk 5f.
  `readline`, used in chunks 13a and 5d.
This code is used in chunk 12b.

13b ⟨*Include the micro parser combinator definitions.* 13b⟩≡

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

Defines:
  `mpca_lang`, used in chunk 5b.
  `mpc_ast_delete`, used in chunk 8f.
  `mpc_ast_print`, never used.
  `mpc_ast_t`, used in chunks 3a and 6c.
  `mpc_cleanup`, used in chunks 13b and 5c.
  `mpc_err_delete`, used in chunk 9b.
  `mpc_err_print`, used in chunk 9b.
  `mpc_new`, used in chunk 4d.
  `mpc_parse`, used in chunks 13b and 6b.
  `mpc_parser_t`, used in chunk 4d.
  `mpc_result_t`, used in chunk 6a.
This code is used in chunk 12b.

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:

```
#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"
};


typedef enum {
    LVAL_NUM,
    LVAL_ERR
} lval_type_t;

typedef enum {
    LERR_DIV_ZERO,
    LERR_BAD_OP,
    LERR_BAD_NUM
} lval_err_t;

typedef struct {
    lval_type_t type;
    union {
        double num;
        lval_err_t err;
    };
} lval;


lval lval_num(double num)
{
    lval val;
    val.type = LVAL_NUM;
    val.num = num;

    return val;
}


lval lval_err(lval_err_t err)
{
    lval val;
    val.type = LVAL_ERR;
    val.err = err;

```

```c
    return val;
}


void lval_print(lval val)
{
    switch (val.type) {
    case LVAL_NUM:
        printf("%g", val.num);
        break;

    case LVAL_ERR:
        switch (val.err) {
        case LERR_BAD_OP:
            puts("Error: invalid operator");
            break;
        case LERR_BAD_NUM:
            puts("Error: invalid number");
            break;
        case LERR_DIV_ZERO:
            fputs("Error: division by zero", stdout);
            break;
        }
        break;
    }
}


void lval_println(lval val)
{
    lval_print(val);
    putchar('\n');
}


lval eval_binop(char *op, lval x, lval y)
{
    if (!strcmp(op, "+"))
        return lval_num(x.num + y.num);

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

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

    if (!strcmp(op, "/")) {
        return !y.num ? lval_err(LERR_DIV_ZERO)
            : lval_num(x.num / y.num);
    }

    return lval_err(LERR_DIV_ZERO);
}


lval eval(mpc_ast_t * ast)
{
    if (strstr(ast->tag, "number")) {
        errno = 0;
        double x = strtod(ast->contents, NULL);
        return errno != ERANGE ? lval_num(x) : lval_err(LERR_BAD_NUM);
    }

    int i = 0;

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

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

    if (!strcmp(op, "-") && ast->children_num == 4) {
        result.num = -result.num;
        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 v0.8.1");
    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;

                lval result = eval(ast);
                lval_println(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. 6, 8

grammar 4, 5

Describe what a grammar is

parser 4

Describe what a parser is

PLT programming language theory, 1

Describe programming language theory

REPL Read-Eval-Print Loop, 4, 5

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