Lispy: a simple Lisp-like language

Eric Bailey

May 10, 2018 [1]

[1] Current version: `VERSION`. Last updated May 16, 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. 22a⟩


⟨Load the Lispy grammar. 6c⟩


⟨Define possible lval and error types. 18b⟩

⟨Define the Lispy data structures. 18a⟩
```

This definition is continued in chunks 2–5.
Root chunk (not used in this document).

2b ⟨*lispy.c* 2a⟩+≡
```
lval *lval_add(lval *xs, lval *x)
{
    ⟨Add an element to an S-expression. 10f⟩

    return xs;
}
```

Defines:
`lval_add`, used in chunk 10d.
Uses `lval` 18a.

2c ⟨*lispy.c* 2a⟩+≡
```
lval *lval_pop(lval *xs, int i)
{
    ⟨Extract an element and shift the list. 14d⟩
}
```

Defines:
`lval_pop`, used in chunks 11d and 15b.
Uses `lval` 18a.

2d ⟨*lispy.c* 2a⟩+≡
```
lval *lval_take(lval *xs, int i)
{
    ⟨Pop the list then delete it. 15b⟩
}
```

Defines:
`lval_take`, used in chunk 13.
Uses `lval` 18a.

3a ⟨*lispy.c* 2a⟩+≡
```
void lval_print_err(lval *val)
{
    ⟨Print an error. 16a⟩
}
```

Defines:
  `lval_print_err`, used in chunk 16b.
Uses `lval` 18a.

Forward declare[4] `lval_print`, since it's mutually recursive[5] with `lval_expr_print`.

[4] https://en.wikipedia.org/wiki/Forward_declaration
[5] https://en.wikipedia.org/wiki/Mutual_recursion

3b ⟨*lispy.c* 2a⟩+≡
```
void lval_print(lval *val);
```

Uses `lval` 18a and `lval_print` 3d.

3c ⟨*lispy.c* 2a⟩+≡
```
void lval_expr_print(lval *expr, char open, char close)
{
    ⟨Print an expression. 15e⟩
}
```

Defines:
  `lval_expr_print`, used in chunks 3c and 16b.
Uses `lval` 18a.

3d ⟨*lispy.c* 2a⟩+≡
```
void lval_print(lval *val)
{
    ⟨Print a Lispy value. 16b⟩
}
```

Defines:
  `lval_print`, used in chunks 3 and 15f.
Uses `lval` 18a.

3e ⟨*lispy.c* 2a⟩+≡
```
void lval_println(lval *val)
{
    lval_print(val);
    putchar('\n');
}
```

Defines:
  `lval_println`, used in chunk 15d.
Uses `lval` 18a and `lval_print` 3d.

4a ⟨*lispy.c* 2a⟩+≡
```
lval *builtin_op(char *op, lval *args)
{
  ⟨Eval(uate) a built-in operation. 11c⟩
}
```

Defines:
  `builtin_binop`, never used.
Uses `lval` 18a.

Forward declare `lval_eval`, since it's mutually recursive with `lval_eval_sexpr`.

4b ⟨*lispy.c* 2a⟩+≡
```
lval *lval_eval(lval* val);
```

Uses `lval` 18a.

4c ⟨*lispy.c* 2a⟩+≡
```
lval* lval_eval_sexpr(lval *args)
{
    ⟨Evaluate an S-expression. 13e⟩
}
```

Uses `lval` 18a.

4d ⟨*lispy.c* 2a⟩+≡
```
lval* lval_eval(lval* val)
{
    ⟨Evaluate an expression. 14c⟩
}
```

Uses `lval` 18a.

4e ⟨*lispy.c* 2a⟩+≡
```
lval *lval_read_num(mpc_ast_t *ast)
{
    ⟨Read a number. 9c⟩
}


lval *lval_read(mpc_ast_t *ast)
{
    ⟨Read a Lispy value. 9b⟩
}
```

Defines:
  `lval_read`, used in chunks 9a and 10d.
Uses `ast` 9a, `lval` 18a, and `mpc_ast_t` 23.

5 ⟨*lispy.c* 2a⟩+≡

```
int main(int argc, char *argv[])
{
    ⟨Define the language. 7a⟩

    ⟨Print version and exit information. 6a⟩

    ⟨Loop until the input is empty. 17a⟩

    ⟨Undefine and delete the parsers. 7d⟩

    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.

6a ⟨*Print version and exit information.* 6a⟩≡
```
puts("Lispy v1.1.1");
puts("Press ctrl-c to exit\n");
```
Uses `Lispy` 7a.
This code is used in chunk 5.

Defining the Language

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

Support Core Erlang style numbers

6b ⟨*lispy.mpc* 6b⟩≡
```
integer  : /-?[0-9]+/ ;
float    : /-?[0-9]+\.[0-9]+/;
number   : <float> | <integer> ;
symbol   : '+' | '-' | '*' | '/' | '%' | '^' ;
sexpr    : '(' <symbol> <expr>+ ')' ;
qexpr    : '{' <symbol>? <expr>+ '}' ;
expr     : <number> | <sexpr> | <qexpr> ;
lispy    : /^/ <expr>* /$/ ;
```
Root chunk (not used in this document).

Describe this trick

6c ⟨*Load the Lispy grammar.* 6c⟩≡
```
static const char LISPY_GRAMMAR[] = {
#include "lispy.xxd"
};
```
Defines:
`LISPY_GRAMMAR`, used in chunk 7c.
This code is used in chunk 2a.

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

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

7a ⟨*Define the language.* 7a⟩≡
```
mpc_parser_t *Integer  = mpc_new("integer");
mpc_parser_t *Float    = mpc_new("float");
mpc_parser_t *Number   = mpc_new("number");
mpc_parser_t *Symbol   = mpc_new("symbol");
mpc_parser_t *Sexpr    = mpc_new("sexpr");
mpc_parser_t *Qexpr    = mpc_new("qexpr");
mpc_parser_t *Expr     = mpc_new("expr");
mpc_parser_t *Lispy    = mpc_new("lispy");
```

Defines:
Expr, used in chunk 7b.
Float, used in chunk 7b.
Integer, used in chunk 7b.
Lispy, used in chunks 6–8.
Number, used in chunk 7b.
Qexpr, used in chunk 7b.
Sexpr, used in chunk 7b.
Symbol, used in chunk 7b.
Uses mpc_new 23 and mpc_parser_t 23.
This definition is continued in chunk 7c.
This code is used in chunk 5.

Finally, using the defined *grammar* and each of the ⟨*created parsers* 7b⟩,

7b ⟨*created parsers* 7b⟩≡
```
Integer, Float, Number, Symbol, Sexpr, Qexpr, Expr, Lispy
```
Uses Expr 7a, Float 7a, Integer 7a, Lispy 7a, Number 7a, Qexpr 7a, Sexpr 7a, and Symbol 7a.
This code is used in chunk 7.

... we can define the Lispy language.

7c ⟨*Define the language.* 7a⟩+≡
```
mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
          ⟨created parsers 7b⟩);
```
Uses LISPY_GRAMMAR 6c and mpca_lang 23.

Since we're implementing this in C, we need to clean up after ourselves. The `mpc`[6] library makes this easy, by providing the `mpc_cleanup` function.

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

7d ⟨*Undefine and delete the parsers.* 7d⟩≡
```
mpc_cleanup(8, ⟨created parsers 7b⟩);
```
Uses mpc_cleanup 23.
This code is used in chunk 5.

R is for Read

To implement the R in REPL, use `readline` from `libedit`[7].

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

8a ⟨*Read a line of user input.* 8a⟩≡

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

Defines:
  `input`, used in chunks 8 and 17d.
Uses `readline` 22g.
This code is used in chunk 17b.

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

8b ⟨`input` *is nonempty* 8b⟩≡

```
input && *input
```

Uses `input` 8a.
This code is used in chunk 17c.

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 `libedit`[8] history table.

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

8c ⟨*Add* `input` *to the history table.* 8c⟩≡

```
add_history(input);
```

Uses `add_history` 22g and `input` 8a.
This code is used in chunk 17c.

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

8d ⟨*Declare a variable to hold parsing results.* 8d⟩≡

```
mpc_result_t parsed;
```

Defines:
  `parsed`, used in chunks 8, 9a, and 16c.
Uses `mpc_result_t` 23.
This code is used in chunk 17c.

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

8e ⟨*the input can be parsed as Lispy code* 8e⟩≡

```
mpc_parse("<stdin>", input, Lispy, &parsed)
```

Uses `Lispy` 7a, `input` 8a, `mpc_parse` 23, and `parsed` 8d.
This code is used in chunk 17c.

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.

9a ⟨*Eval(uate) the input.* 9a⟩≡

```
mpc_ast_t *ast = parsed.output;

lval *result = lval_eval(lval_read(ast));
```

Defines:
`ast`, used in chunks 4e, 9, 10, and 15d.
Uses `lval` 18a, `lval_read` 4e, `mpc_ast_t` 23, and `parsed` 8d.
This code is used in chunk 17c.

Describe the evaluation strategy

If the *abstract syntax tree (AST)* is tagged as a number, convert it to a `double`.

9b ⟨*Read a Lispy value.* 9b⟩≡

```
if (strstr(ast->tag, "number"))
    return lval_read_num(ast);
```

Uses `ast` 9a and `strstr` 22f.
This definition is continued in chunks 9 and 10.
This code is used in chunk 4e.

Describe this

9c ⟨*Read a number.* 9c⟩≡

```
errno = 0;
double num = strtod(ast->contents, NULL);
return errno ≠ ERANGE ? lval_num(num) : lval_err(LERR_BAD_NUM);
```

Uses `LERR_BAD_NUM` 19c, `ast` 9a, `lval_err` 20a, `lval_num` 19a, and `strtod` 22d.
This code is used in chunk 4e.

If the AST is tagged as a symbol, convert it to one.

9d ⟨*Read a Lispy value.* 9b⟩+≡

```
if (strstr(ast->tag, "symbol"))
    return lval_sym(ast->contents);
```

Uses `ast` 9a, `lval_sym` 20b, and `strstr` 22f.

Describe this

9e ⟨*Read a symbol.* 9e⟩≡
Root chunk (not used in this document).

Describe this

9f ⟨*Read a Lispy value.* 9b⟩+≡

```
lval *val = NULL;
```

Uses `lval` 18a.

If we're at the root of the AST, create an empty list.

10a ⟨*Read a Lispy value.* 9b⟩+≡
```
if (!strcmp(ast->tag, ">"))
    val = lval_sexpr();
```
Uses `ast` 9a, `lval_sexpr` 20c, and `strcmp` 22f.

If it's tagged as a Q-expression, create an empty list.

10b ⟨*Read a Lispy value.* 9b⟩+≡
```
if (strstr(ast->tag, "qexpr"))
    val = lval_qexpr();
```
Uses `ast` 9a, `lval_qexpr` 21a, and `strstr` 22f.

Similarly if it's tagged as an S-expression, create an empty list.

10c ⟨*Read a Lispy value.* 9b⟩+≡
```
if (strstr(ast->tag, "sexpr"))
    val = lval_sexpr();
```
Uses `ast` 9a, `lval_sexpr` 20c, and `strstr` 22f.

Describe this

10d ⟨*Read a Lispy value.* 9b⟩+≡
```
for (int i = 0; i < ast->children_num; i++) {
    if(!strcmp(ast->children[i]->contents, "(")) continue;
    if(!strcmp(ast->children[i]->contents, ")")) continue;
    if(!strcmp(ast->children[i]->contents, "{")) continue;
    if(!strcmp(ast->children[i]->contents, "}")) continue;
    if(!strcmp(ast->children[i]->tag, "regex")) continue;
    val = lval_add(val, lval_read(ast->children[i]));
}
```
Uses `ast` 9a, `lval_add` 2b, `lval_read` 4e, and `strcmp` 22f.

10e ⟨*Reallocate the memory used.* 10e⟩≡
```
xs->cell = realloc(xs->cell, sizeof(lval *) * xs->count);
```
Uses `lval` 18a.
This code is used in chunks 10f and 15a.

Describe this, incl. how it's not cons

10f ⟨*Add an element to an S-expression.* 10f⟩≡
```
xs->count++;
⟨Reallocate the memory used. 10e⟩
xs->cell[xs->count - 1] = x;
```
This code is used in chunk 2b.

Finally, return the Lispy value.

10g ⟨*Read a Lispy value.* 9b⟩+≡
```
return val;
```

10h ⟨*For each argument* 10h⟩≡
```
for (int i = 0; i < args->count; i++)
```
This code is used in chunks 11c and 13f.

11a ⟨*the argument is not a number* 11a⟩≡
```
!lval_is_num(args->cell[i])
```
Uses `lval_is_num` 19b.
This code is used in chunk 11c.

11b ⟨*Delete the arguments and return a bad number error.* 11b⟩≡
```
lval_del(args);
return lval_err(LERR_BAD_NUM);
```
Uses `LERR_BAD_NUM` 19c, `lval_del` 21b, and `lval_err` 20a.
This code is used in chunk 11c.

Evaluating built-in operations

Ensure all arguments are numbers.

11c ⟨*Eval(uate) a built-in operation.* 11c⟩≡
```
⟨For each argument 10h⟩ {
    if (⟨the argument is not a number 11a⟩) {
        ⟨Delete the arguments and return a bad number error. 11b⟩
    }
}
```

This definition is continued in chunks 11 and 13d.
This code is used in chunk 4a.

11d ⟨*Pop the first element.* 11d⟩≡
```
lval_pop(args, 0);
```
Uses `lval_pop` 2c.
This code is used in chunks 11 and 14a.

Pop the first element.

11e ⟨*Eval(uate) a built-in operation.* 11c⟩+≡
```
lval *result = ⟨Pop the first element. 11d⟩
```

Uses `lval` 18a.

If the operation is unary subtraction, negate the operand.

11f ⟨*Eval(uate) a built-in operation.* 11c⟩+≡
```
if (!strcmp(op, "-") && !args->count)
    result->num = -result->num;
```

Uses `strcmp` 22f.

11g ⟨*Pop the next element.* 11g⟩≡
```
lval *y = ⟨Pop the first element. 11d⟩
```
Uses `lval` 18a.
This code is used in chunk 11h.

11h ⟨*Eval(uate) a built-in operation.* 11c⟩+≡
```
while (args->count > 0) {
    ⟨Pop the next element. 11g⟩

    ⟨Perform a built-in operation. 12a⟩
}
```

If the op is "+", perform addition.

12a ⟨*Perform a built-in operation.* 12a⟩≡
```
if (!strcmp(op, "+")) {
    result->num += y->num;
}
```
Uses strcmp 22f.
This definition is continued in chunks 12 and 13.
This code is used in chunk 11h.

If the op is "-", perform subtraction.

12b ⟨*Perform a built-in operation.* 12a⟩+≡
```
else if (!strcmp(op, "-")) {
    result->num -= y->num;
}
```
Uses strcmp 22f.

If the op is "*", perform multiplication.

12c ⟨*Perform a built-in operation.* 12a⟩+≡
```
else if (!strcmp(op, "*")) {
    result->num *= y->num;
}
```
Uses strcmp 22f.

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

12d ⟨*Perform a built-in operation.* 12a⟩+≡
```
else if (!strcmp(op, "/")) {
    if (!y->num) {
        lval_del(result);
        lval_del(y);
        result = lval_err(LERR_DIV_ZERO);
        break;
    }
    result->num /= y->num;
}
```
Uses LERR_DIV_ZERO 19c, lval_del 21b, lval_err 20a, and strcmp 22f.

If the op is "%", calculate the integer modulo, returning the appropriate error when trying to divide by zero.

12e ⟨*Perform a built-in operation.* 12a⟩+≡
```
else if (!strcmp(op, "%")) {
    if (!y->num) {
        lval_del(result);
        lval_del(y);
        result = lval_err(LERR_DIV_ZERO);
        break;
    }
    result->num = fmod(result->num, y->num);
}
```
Uses LERR_DIV_ZERO 19c, fmod 22e, lval_del 21b, lval_err 20a, and strcmp 22f.

If the `opp` is `"^"`, perform exponentiation.

13a ⟨*Perform a built-in operation.* 12a⟩+≡
```
else if (!strcmp(op, "^")) {
    result->num = pow(result->num, y->num);
}
```
Uses `pow` 22e and `strcmp` 22f.

Otherwise, return a `LERR_BAD_OP` error.

13b ⟨*Perform a built-in operation.* 12a⟩+≡
```
else {
    lval_del(result);
    lval_del(y);
    result = lval_err(LERR_BAD_OP);
    break;
}
```
Uses `LERR_BAD_OP` 19c, `lval_del` 21b, and `lval_err` 20a.

Delete `y`, now that we're done with it.

13c ⟨*Perform a built-in operation.* 12a⟩+≡
```
lval_del(y);
```
Uses `lval_del` 21b.

Delete the input expression and return the result.

13d ⟨*Eval(uate) a built-in operation.* 11c⟩+≡
```
lval_del(args);

return result;
```
Uses `lval_del` 21b.

Evaluating (S)-expressions

If the expression is empty, return it;

13e ⟨*Evaluate an S-expression.* 13e⟩≡
```
if (!args->count)
    return args;
```
This definition is continued in chunks 13 and 14.
This code is used in chunk 4c.

13f ⟨*Evaluate an S-expression.* 13e⟩+≡
```
⟨For each argument 10h⟩ {
    args->cell[i] = lval_eval(args->cell[i]);
    if (args->cell[i]->type == LVAL_ERR)
        return lval_take(args, i);
}
```

Uses `LVAL_ERR` 18b 18b and `lval_take` 2d.

If we're dealing with a single expression, return it.

13g ⟨*Evaluate an S-expression.* 13e⟩+≡
```
if (args->count == 1)
    return lval_take(args, 0);
```

Uses `lval_take` 2d.

14a ⟨*Evaluate an S-expression.* 13e⟩+≡
```
lval *car = ⟨Pop the first element. 11d⟩;
if (car->type != LVAL_SYM) {
    lval_del(car);
    lval_del(args);

    return lval_err(LERR_BAD_SEXPR);
}
```
Uses `LVAL_SYM` 18b, `lval` 18a, `lval_del` 21b, and `lval_err` 20a.

14b ⟨*Evaluate an S-expression.* 13e⟩+≡
```
lval *result = builtin_op(car->sym, args);
lval_del(car);

return result;
```
Uses `lval` 18a and `lval_del` 21b.

If, and only if, an expression is an S-expression, we must evaluate it recursively.

14c ⟨*Evaluate an expression.* 14c⟩≡
```
if (val->type == LVAL_SEXPR)
    return lval_eval_sexpr(val);

return val;
```
Uses `LVAL_SEXPR` 18b.
This code is used in chunk 4d.

Extract the element at index `i`.

14d ⟨*Extract an element and shift the list.* 14d⟩≡
```
lval *elem = xs->cell[i];
```
Uses `lval` 18a.
This definition is continued in chunks 14 and 15a.
This code is used in chunk 2c.

Shift memory after the element at index `i`.

14e ⟨*Extract an element and shift the list.* 14d⟩+≡
```
memmove(&xs->cell[i], &xs->cell[i + 1],
    sizeof(lval *) * (xs->count - i - 1));
```
Uses `lval` 18a.

Decrease the count.

14f ⟨*Extract an element and shift the list.* 14d⟩+≡
```
xs->count-;
```

14g ⟨*Return the extracted element.* 14g⟩≡
```
return elem;
```
This code is used in chunk 15.

Reallocate the memory used and return the extracted element.

15a ⟨*Extract an element and shift the list.* 14d⟩+≡

```
  ⟨Reallocate the memory used. 10e⟩

  ⟨Return the extracted element. 14g⟩
```

Describe this

15b ⟨*Pop the list then delete it.* 15b⟩≡

```
  lval *elem = lval_pop(xs, i);
  lval_del(xs);
```

Uses `lval` 18a, `lval_del` 21b, and `lval_pop` 2c.
This definition is continued in chunk 15c.
This code is used in chunk 2d.

Return the extracted element.

15c ⟨*Pop the list then delete it.* 15b⟩+≡

```
  ⟨Return the extracted element. 14g⟩
```

P is for Print

Upon success, print the result and delete the AST.

15d ⟨*Print the result and delete the AST.* 15d⟩≡

```
  lval_println(result);

  mpc_ast_delete(ast);
```

Uses `ast` 9a, `lval_println` 3e, and `mpc_ast_delete` 23.
This code is used in chunk 17c.

Describe this

Print the opening character.

15e ⟨*Print an expression.* 15e⟩≡

```
  putchar(open);
```

This definition is continued in chunk 15.
This code is used in chunk 3c.

Print all but the last element with a trailing space.

15f ⟨*Print an expression.* 15e⟩+≡

```
  for (int i = 0; i < expr->count; i++) {
      lval_print(expr->cell[i]);
      if (i != (expr->count - 1))
          putchar(' ');
  }
```

Uses `lval_print` 3d.

Print the closing character.

15g ⟨*Print an expression.* 15e⟩+≡

```
  putchar(close);
```

16a ⟨*Print an error.* 16a⟩≡

```
switch (val->err) {
case LERR_BAD_NUM:
    puts("Error: invalid number");
    break;
case LERR_BAD_OP:
    puts("Error: invalid operator");
    break;
case LERR_BAD_SEXPR:
    puts("Error: S-expression does not start with symbol");
    break;
case LERR_DIV_ZERO:
    puts("Error: division by zero");
    break;
}
```

Uses `LERR_BAD_NUM` 19c, `LERR_BAD_OP` 19c, and `LERR_DIV_ZERO` 19c.
This code is used in chunk 3a.

16b ⟨*Print a Lispy value.* 16b⟩≡

```
switch (val->type) {
case LVAL_ERR:
    lval_print_err(val);
    break;
case LVAL_NUM:
    printf("%g", val->num);
    break;
case LVAL_QEXPR:
    lval_expr_print(val, '{', '}');
    break;
case LVAL_SEXPR:
    lval_expr_print(val, '(', ')');
    break;
case LVAL_SYM:
    fputs(val->sym, stdout);
    break;
}
```

Uses `LVAL_ERR` 18b 18b, `LVAL_NUM` 18b, `LVAL_QEXPR` 18b, `LVAL_SEXPR` 18b,
`LVAL_SYM` 18b, `lval_expr_print` 3c, `lval_print_err` 3a, and `printf` 22c.
This code is used in chunk 3d.

Print and delete the error upon failure.

16c ⟨*Print and delete the error.* 16c⟩≡

```
mpc_err_print(parsed.error);
mpc_err_delete(parsed.error);
```

Uses `mpc_err_delete` 23, `mpc_err_print` 23, and `parsed` 8d.
This code is used in chunk 17c.

L is for Loop

17a ⟨*Loop until the input is empty.* 17a⟩≡
```
bool nonempty;
do {
  ⟨Read, eval(uate), and print. 17b⟩
} while (nonempty);
```
Defines:
`nonempty`, used in chunk 17c.
Uses `bool` 22b.
This code is used in chunk 5.

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

17b ⟨*Read, eval(uate), and print.* 17b⟩≡
```
⟨Read a line of user input. 8a⟩
```
This definition is continued in chunk 17.
This code is used in chunk 17a.

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

17c ⟨*Read, eval(uate), and print.* 17b⟩+≡
```
if ((nonempty = (⟨input is nonempty 8b⟩))) {
    ⟨Add input to the history table. 8c⟩

    ⟨Declare a variable to hold parsing results. 8d⟩
    if (⟨the input can be parsed as Lispy code 8e⟩) {
        ⟨Eval(uate) the input. 9a⟩
        ⟨Print the result and delete the AST. 15d⟩
    } else {
        ⟨Print and delete the error. 16c⟩
    }
}
```

Uses `nonempty` 17a.

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

17d ⟨*Read, eval(uate), and print.* 17b⟩+≡
```
free(input);
```
Uses `free` 22d and `input` 8a.

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

Error Handling

Describe this struct

18a ⟨*Define the Lispy data structures.* 18a⟩≡
```
typedef struct lval {
    lval_type_t type;
    union {
        double num;
        lval_err_t err;
        char* sym;
    };
    int count;
    struct lval **cell;
} lval;
```

Defines:
`lval`, used in chunks 2–4, 9–11, 14, 15b, and 19–21.
Uses `lval_err_t` 19c and `lval_type_t` 18b.
This definition is continued in chunks 19–21.
This code is used in chunk 2a.

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

18b ⟨*Define possible lval and error types.* 18b⟩≡
```
typedef enum {
    LVAL_ERR,
    LVAL_NUM,
    LVAL_QEXPR,
    LVAL_SEXPR,
    LVAL_SYM
} lval_type_t;
```

Defines:
`LVAL_ERR`, used in chunks 13f, 16b, 20a, and 21b.
`LVAL_NUM`, used in chunks 16b, 19, and 21b.
`LVAL_QEXPR`, used in chunks 16b and 21.
`LVAL_SEXPR`, used in chunks 14c, 16b, 20c, and 21b.
`LVAL_SYM`, used in chunks 14a, 16b, 20b, and 21b.
`lval_type_t`, used in chunk 18a.
This definition is continued in chunk 19c.
This code is used in chunk 2a.

Define a constructor for numbers.

19a ⟨*Define the Lispy data structures.* 18a⟩+≡
```
lval *lval_num(double num)
{
    lval *val = malloc(sizeof(lval));
    val->type = LVAL_NUM;
    val->num = num;

    return val;
}
```

Defines:
  `lval_num`, used in chunk 9c.
Uses `LVAL_NUM` 18b and `lval` 18a.

Define a convenient predicate for numbers.

19b ⟨*Define the Lispy data structures.* 18a⟩+≡
```
bool lval_is_num(lval *val)
{
    return val->type == LVAL_NUM;
}
```

Defines:
  `lval_is_num`, used in chunk 11a.
Uses `LVAL_NUM` 18b, `bool` 22b, and `lval` 18a.

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

19c ⟨*Define possible lval and error types.* 18b⟩+≡
```
typedef enum {
    LERR_DIV_ZERO,
    LERR_BAD_OP,
    LERR_BAD_NUM,
    LERR_BAD_SEXPR
} lval_err_t;
```

Defines:
  `LERR_BAD_NUM`, used in chunks 9c, 11b, and 16a.
  `LERR_BAD_OP`, used in chunks 13b and 16a.
  `LERR_DIV_ZERO`, used in chunks 12 and 16a.
  `lval_err_t`, used in chunks 18a and 20a.

Define a constructor for errors.

20a ⟨*Define the Lispy data structures.* 18a⟩+≡
```
lval *lval_err(lval_err_t err)
{
    lval *val = malloc(sizeof(lval));
    val->type = LVAL_ERR;
    val->err = err;

    return val;
}
```

Defines:
lval_err, used in chunks 9c and 11–14.
Uses LVAL_ERR 18b 18b, lval 18a, and lval_err_t 19c.

Define a constructor for symbol.

20b ⟨*Define the Lispy data structures.* 18a⟩+≡
```
lval *lval_sym(char *s)
{
    lval *val = malloc(sizeof(lval));
    val->type = LVAL_SYM;
    val->sym = malloc(strlen(s) + 1);
    strcpy(val->sym, s);

    return val;
}
```

Defines:
lval_sym, used in chunk 9d.
Uses LVAL_SYM 18b and lval 18a.

Define a constructor for an S-expression.

20c ⟨*Define the Lispy data structures.* 18a⟩+≡
```
lval *lval_sexpr(void)
{
    lval *val = malloc(sizeof(lval));
    val->type = LVAL_SEXPR;
    val->count = 0;
    val->cell = NULL;

    return val;
}
```

Defines:
lval_sexpr, used in chunk 10.
Uses LVAL_SEXPR 18b and lval 18a.

Define a constructor for a Q-expression.

21a ⟨*Define the Lispy data structures.* 18a⟩+≡

```
lval *lval_qexpr(void)
{
    lval *val = malloc(sizeof(lval));
    val->type = LVAL_QEXPR;
    val->count = 0;
    val->cell = NULL;

    return val;
}
```

Defines:
  `lval_qexpr`, used in chunk 10b.
Uses `LVAL_QEXPR` 18b and `lval` 18a.

Define a destructor for `lval*`.

21b ⟨*Define the Lispy data structures.* 18a⟩+≡

```
void lval_del(lval *val)
{
    switch(val->type) {
    case LVAL_ERR:
    case LVAL_NUM:
        break;
    case LVAL_QEXPR:
    case LVAL_SEXPR:
        for (int i = 0; i < val->count; i++)
            lval_del(val->cell[i]);
        free(val->cell);
        break;
    case LVAL_SYM:
        free(val->sym);
        break;
    }

    free(val);
}
```

Defines:
  `lval_del`, used in chunks 11–15.
Uses `LVAL_ERR` 18b 18b, `LVAL_NUM` 18b, `LVAL_QEXPR` 18b, `LVAL_SEXPR` 18b,
  `LVAL_SYM` 18b, `free` 22d, and `lval` 18a.

Headers

Describe headers

22a ⟨*Include the necessary headers.* 22a⟩≡
  ⟨*Include the boolean type and values.* 22b⟩
  ⟨*Include the standard I/O functions.* 22c⟩
  ⟨*Include the standard library definitions.* 22d⟩
  ⟨*Include some mathematical definitions.* 22e⟩
  ⟨*Include some string operations.* 22f⟩

  ⟨*Include the line editing functions from libedit.* 22g⟩
  ⟨*Include the micro parser combinator definitions.* 23⟩

This code is used in chunk 2a.

22b ⟨*Include the boolean type and values.* 22b⟩≡
```
#include <stdbool.h>
```
Defines:
  `bool`, used in chunks 17a and 19b.

This code is used in chunk 22a.

22c ⟨*Include the standard I/O functions.* 22c⟩≡
```
#include <stdio.h>
```
Defines:
  `printf`, used in chunk 16b.

This code is used in chunk 22a.

22d ⟨*Include the standard library definitions.* 22d⟩≡
```
#include <stdlib.h>
```
Defines:
  `free`, used in chunks 17d and 21b.
  `strtod`, used in chunk 9c.

This code is used in chunk 22a.

22e ⟨*Include some mathematical definitions.* 22e⟩≡
```
#include <math.h>
```
Defines:
  `fmod`, used in chunk 12e.
  `pow`, used in chunk 13a.

This code is used in chunk 22a.

22f ⟨*Include some string operations.* 22f⟩≡
```
#include <string.h>
```
Defines:
  `strcmp`, used in chunks 10–13.
  `strstr`, used in chunks 9 and 10.

This code is used in chunk 22a.

22g ⟨*Include the line editing functions from libedit.* 22g⟩≡
```
#include <editline/readline.h>
```
Defines:
  `add_history`, used in chunk 8c.
  `readline`, used in chunks 22g and 8a.

This code is used in chunk 22a.

23 ⟨*Include the micro parser combinator definitions.* 23⟩≡

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

Defines:
`mpca_lang`, used in chunk 7c.
`mpc_ast_delete`, used in chunk 15d.
`mpc_ast_print`, never used.
`mpc_ast_t`, used in chunks 4e and 9a.
`mpc_cleanup`, used in chunks 23 and 7d.
`mpc_err_delete`, used in chunk 16c.
`mpc_err_print`, used in chunk 16c.
`mpc_new`, used in chunk 7a.
`mpc_parse`, used in chunks 23 and 8e.
`mpc_parser_t`, used in chunk 7a.
`mpc_result_t`, used in chunk 8d.

This code is used in chunk 22a.

Full Listings

lispy.mpc:

```
integer   : /-?[0-9]+/ ;
float     : /-?[0-9]+\.[0-9]+/;
number    : <float> | <integer> ;
symbol    : '+' | '-' | '*' | '/' | '%' | '^' ;
sexpr     : '(' <symbol> <expr>+ ')' ;
qexpr     : '{' <symbol>? <expr>+ '}' ;
expr      : <number> | <sexpr> | <qexpr> ;
lispy     : /^/ <expr>* /$/ ;
```

lispy.c:

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

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


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


typedef enum {
    LVAL_ERR,
    LVAL_NUM,
    LVAL_QEXPR,
    LVAL_SEXPR,
    LVAL_SYM
} lval_type_t;


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


typedef struct lval {
    lval_type_t type;
    union {
        double num;
        lval_err_t err;
        char *sym;
    };
    int count;
    struct lval **cell;
} lval;


lval *lval_num(double num)
{
    lval *val = malloc(sizeof(lval));
    val->type = LVAL_NUM;
    val->num = num;

    return val;
}


bool lval_is_num(lval * val)
{
    return val->type == LVAL_NUM;
}


lval *lval_err(lval_err_t err)
{
    lval *val = malloc(sizeof(lval));
    val->type = LVAL_ERR;
    val->err = err;

    return val;
}


lval *lval_sym(char *s)
{
    lval *val = malloc(sizeof(lval));
    val->type = LVAL_SYM;
    val->sym = malloc(strlen(s) + 1);
    strcpy(val->sym, s);

    return val;
}


lval *lval_sexpr(void)
{
    lval *val = malloc(sizeof(lval));
    val->type = LVAL_SEXPR;
    val->count = 0;
    val->cell = NULL;

    return val;
}


lval *lval_qexpr(void)
{
    lval *val = malloc(sizeof(lval));
    val->type = LVAL_QEXPR;
    val->count = 0;
    val->cell = NULL;

    return val;
}
```

```c


void lval_del(lval * val)
{
    switch (val->type) {
    case LVAL_ERR:
    case LVAL_NUM:
        break;
    case LVAL_QEXPR:
    case LVAL_SEXPR:
        for (int i = 0; i < val->count; i++)
            lval_del(val->cell[i]);
        free(val->cell);
        break;
    case LVAL_SYM:
        free(val->sym);
        break;
    }

    free(val);
}


lval *lval_add(lval * xs, lval * x)
{
    xs->count++;
    xs->cell = realloc(xs->cell, sizeof(lval *) * xs->count);
    xs->cell[xs->count - 1] = x;

    return xs;
}


lval *lval_pop(lval * xs, int i)
{
    lval *elem = xs->cell[i];

    memmove(&xs->cell[i], &xs->cell[i + 1],
            sizeof(lval *) * (xs->count - i - 1));

    xs->count--;

    xs->cell = realloc(xs->cell, sizeof(lval *) * xs->count);

    return elem;
}


lval *lval_take(lval * xs, int i)
{
    lval *elem = lval_pop(xs, i);
    lval_del(xs);

    return elem;
}


void lval_print_err(lval * val)
{
    switch (val->err) {
    case LERR_BAD_NUM:
        puts("Error: invalid number");
        break;
    case LERR_BAD_OP:
        puts("Error: invalid operator");
        break;
    case LERR_BAD_SEXPR:
        puts("Error: S-expression does not start with symbol");
        break;
    case LERR_DIV_ZERO:
        puts("Error: division by zero");
        break;
    }
}


void lval_print(lval * val);


void lval_expr_print(lval * expr, char open, char close)
{
    putchar(open);
    for (int i = 0; i < expr->count; i++) {
        lval_print(expr->cell[i]);
        if (i != (expr->count - 1))
            putchar(' ');
    }
    putchar(close);
}


void lval_print(lval * val)
{
    switch (val->type) {
    case LVAL_ERR:
        lval_print_err(val);
        break;
    case LVAL_NUM:
        printf("%g", val->num);
        break;
    case LVAL_QEXPR:
        lval_expr_print(val, '{', '}');
        break;
    case LVAL_SEXPR:
        lval_expr_print(val, '(', ')');
        break;
    case LVAL_SYM:
        fputs(val->sym, stdout);
        break;
    }
}


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


lval *builtin_op(char *op, lval * args)
{
    for (int i = 0; i < args->count; i++) {
        if (!lval_is_num(args->cell[i])) {
            lval_del(args);
            return lval_err(LERR_BAD_NUM);
        }
    }

    lval *result = lval_pop(args, 0);

    if (!strcmp(op, "-") && !args->count)
        result->num = -result->num;

    while (args->count > 0) {
        lval *y = lval_pop(args, 0);

        if (!strcmp(op, "+")) {
            result->num += y->num;
        } else if (!strcmp(op, "-")) {
            result->num -= y->num;
        } else if (!strcmp(op, "*")) {
            result->num *= y->num;
        } else if (!strcmp(op, "/")) {
            if (!y->num) {
                lval_del(result);
                lval_del(y);
                result = lval_err(LERR_DIV_ZERO);
                break;
            }
            result->num /= y->num;
        } else if (!strcmp(op, "%")) {
            if (!y->num) {
                lval_del(result);
                lval_del(y);
                result = lval_err(LERR_DIV_ZERO);
                break;
            }
            result->num = fmod(result->num, y->num);
        } else if (!strcmp(op, "^")) {
            result->num = pow(result->num, y->num);
        } else {
            lval_del(result);
            lval_del(y);
            result = lval_err(LERR_BAD_OP);
            break;
        }
        lval_del(y);
    }

    lval_del(args);

    return result;
}


lval *lval_eval(lval * val);


lval *lval_eval_sexpr(lval * args)
{
    if (!args->count)
        return args;
    for (int i = 0; i < args->count; i++) {
        args->cell[i] = lval_eval(args->cell[i]);
        if (args->cell[i]->type == LVAL_ERR)
            return lval_take(args, i);
    }

    if (args->count == 1)
        return lval_take(args, 0);

    lval *car = lval_pop(args, 0);;
    if (car->type != LVAL_SYM) {
        lval_del(car);
        lval_del(args);

        return lval_err(LERR_BAD_SEXPR);
    }

    lval *result = builtin_op(car->sym, args);
    lval_del(car);

    return result;
```

```c
}


lval *lval_eval(lval * val)
{
    if (val->type == LVAL_SEXPR)
        return lval_eval_sexpr(val);

    return val;
}


lval *lval_read_num(mpc_ast_t * ast)
{
    errno = 0;
    double num = strtod(ast->contents, NULL);
    return errno != ERANGE ? lval_num(num) : lval_err(LERR_BAD_NUM);
}


lval *lval_read(mpc_ast_t * ast)
{
    if (strstr(ast->tag, "number"))
        return lval_read_num(ast);

    if (strstr(ast->tag, "symbol"))
        return lval_sym(ast->contents);

    lval *val = NULL;
    if (!strcmp(ast->tag, ">"))
        val = lval_sexpr();
    if (strstr(ast->tag, "qexpr"))
        val = lval_qexpr();
    if (strstr(ast->tag, "sexpr"))
        val = lval_sexpr();

    for (int i = 0; i < ast->children_num; i++) {
        if (!strcmp(ast->children[i]->contents, "("))
            continue;
        if (!strcmp(ast->children[i]->contents, ")"))
            continue;
        if (!strcmp(ast->children[i]->contents, "{"))
            continue;
        if (!strcmp(ast->children[i]->contents, "}"))
            continue;
        if (!strcmp(ast->children[i]->tag, "regex"))
            continue;
        val = lval_add(val, lval_read(ast->children[i]));
    }

    return val;
```

```
}


int main(int argc, char *argv[])
{
    mpc_parser_t *Integer = mpc_new("integer");
    mpc_parser_t *Float = mpc_new("float");
    mpc_parser_t *Number = mpc_new("number");
    mpc_parser_t *Symbol = mpc_new("symbol");
    mpc_parser_t *Sexpr = mpc_new("sexpr");
    mpc_parser_t *Qexpr = mpc_new("qexpr");
    mpc_parser_t *Expr = mpc_new("expr");
    mpc_parser_t *Lispy = mpc_new("lispy");

    mpca_lang(MPCA_LANG_DEFAULT, LISPY_GRAMMAR,
              Integer, Float, Number, Symbol, Sexpr, Qexpr, Expr, Lispy);

    puts("Lispy v1.1.1");
    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 = lval_eval(lval_read(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(8, Integer, Float, Number, Symbol, Sexpr, Qexpr, Expr,
                Lispy);

    return 0;
}
```

Chunks

Index

Glossary

AST abstract syntax tree, a tree representation of the abstract syntactic structure of source code. 9, 10, 15

grammar 6, 7

Describe what a grammar is

parser 7

Describe what a parser is

PLT programming language theory, 1

Describe programming language theory

REPL Read-Eval-Print Loop, 6, 8

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