

1. REPL. Uses editline to read lines and concatenate them into a rope which **LossLess** parses and evaluates.

```
#include <histedit.h>
#include <stdio.h>
#include "lossless.h"
  ⟨Function declarations 2⟩
  ⟨Global variables 3⟩
```

2. ⟨Function declarations 2⟩ ≡
 char **prompt*(**EditLine** *);

This code is used in section 1.

3. Arbitrary history size limit can be made dynamic later.

```
#define HISTORY_SIZE 1000
  ⟨Global variables 3⟩ ≡
    EditLine *E;
    History *H;
```

This code is used in section 1.

```
#define INITIALISE "(do\n\n"
"(define!_(root-environment)_list?_(lambda_())\n"
"_____ (define!_(current-environment)_list?_(lambda_(OBJECT)\n"
"_____ (if_(null?_OBJECT)\n"
"_____ #t\n"
"_____ (if_(pair?_OBJECT)\n"
"_____ (-list?_(cdr_OBJECT))\n"
"_____ #f))))\n"
"_____ -list?))\n"
"\n\n"
"(define!_(root-environment)_and_(lambda_())\n"
"_____ (define!_(current-environment)_and_(vov_( (ARGS_vov\
/args)_ (ENV_vov/env))\n"
"_____ (arity!_ARGS_list?)\n"
"_____ (if_(null?_ARGS)\n"
"_____ #t\n"
"_____ (do_____ (define!_(current-environme\
nt)_ARG_(eval_(car_ARGS)_ENV))\n"
"_____ (if_(if_(null?_(cdr_ARGS))\n"
"_____ ARG\n"
"_____ (eval_(cons\
_-and_(cdr_ARGS))_ENV))\n"
"_____ #f))))))\n"
"_____ -and))\n"
"\n"
"(define!_(root-environment)_or_(lambda_())\n"
"_____ (define!_(current-environment)_or_(vov_( (ARGS_vov/a\
rgs)_ (ENV_vov/env))\n"
"_____ (arity!_ARGS_list?)\n"
"_____ (if_(null?_ARGS)\n"
"_____ #f\n"
"_____ (do_____ (define!_(current-environme\
nt)_ARG_(eval_(car_ARGS)_ENV))\n"
"_____ (if_(if_(if_(if_(cdr_ARGS)_ENV))))\n"
"_____ or))\n"
"\n"
"(define!_(root-environment)_not_(lambda_(OBJECT)_ (if_(OBJECT_#f_#t)))\n"
"\n"
"(define!_(root-environment)_so_(lambda_(OBJECT)_ (if_(OBJECT_#t_#f)))\n"
"\n"
"(define!_(root-environment)_arity!_(lambda_ARGS\n"
"_____ #f))\n"
"\n"
"(define!_(root-environment)_maybe_(lambda_(PREDICATE)\n"
"_____ (lambda_(OBJECT)\n"
"_____ (or_(null?_OBJECT)_ (PREDICATE_OBJECT))))\n"
"\n\n")
```

```

int main(int argc, char **argv)
{
    HistEvent event;
    int length;
    const char *line;
    bool pending = false, valid;
    cellx;

```

```

sigjmp_buf cleanup;
Verror reason = LERR_NONE;
⟨ Initialise editline 5 ⟩
mem_init(); /* Initialise LossLess. */
if (failure_p(reason = sigsetjmp(cleanup, 1))) goto die;
lprint("Initialising...\n%s\n", INITIALISE);
stack_push(NIL, &cleanup);
SS(0, x = rope_new_buffer(false, false, INITIALISE, sizeof (INITIALISE) - 1, &cleanup));
SS(0, x = lex_rope(x, &cleanup));
valid = true;
SS(0, x = parse(x, &valid, &cleanup));
assert(valid);
evaluate_program(SO(0), &cleanup);
ACC = VOID;
stack_push(NIL, &cleanup);
do {⟨ Read and dispatch a line 6 ⟩} while (length);
if (H ≠ Λ) history_end(H);
el_end(E);
}

```

5. ⟨ Initialise editline 5 ⟩ ≡
- ```

E = el_init(argv[0], stdin, stdout, stderr);
el_set(E, EL_PROMPT, &prompt);
el_set(E, EL_EDITOR, "emacs"); /* TODO: use environment. */
H = history_init();
if ((H = history_init()) ≡ Λ) fprintf(stderr, "WARNING: could not initialise history\n");
else {
 history(H, &event, H_SETSIZE, HISTORY_SIZE);
 el_set(E, EL_HIST, history, H);
}

```

This code is used in section 4.

```

6. ⟨ Read and dispatch a line 6 ⟩ ≡
 line = el_gets(E, &length);
 if (failure_p(reason = sigsetjmp(cleanup, 1))) {
 switch (reason) {
 case LERR_UNCLOSED_OPEN: pending = true;
 break;
 default: die: fprintf(stderr, "FATAL_%u:_%s.\n", reason, Error[reason].message);
 abort();
 }
 }
 if (length > 0) {
 if (H ≠ Λ) history(H, &event, H_ENTER, line);
 SS(0, x = rope_new_buffer(false, false, line, length, &cleanup)); if (pending) {
 serial(lapi_User_Register(UNDEFINED), SERIAL_DETAIL, 12, NIL, Λ, &cleanup);
 }
 SS(0, x = cons(SO(0), NIL, &cleanup));
 SS(0, x = cons(lapi_User_Register(UNDEFINED), x, &cleanup));
 SS(0, x = cons(symbol_new_const("rope/append"), x, &cleanup));
 evaluate_program(SO(0), &cleanup);
 SS(0, x = ACC);
 #if 0
 cons("rope/append", USERREG, x
 ... SS(0, x = rope_append(lapi_User_Register(UNDEFINED), x, &cleanup));
 #endif
 } lapi_User_Register(x);
 x = lex_rope(x, &cleanup);
 SS(0, x);
 valid = true;
 x = parse(SO(0), &valid, &cleanup);
 if (valid) {
 ACC = VOID;
 evaluate_program(x, &cleanup);
 lapi_User_Register(NIL);
 pending = false;
 }
 else if (pair_p(lcdr(x)) ∧ fix_value(lcar(lcar(lcdr(x)))) ≡ LERR_SYNTAX ∧
 pair_p(lcdr(lcdr(x))) ∧ fix_value(lcar(lcar(lcdr(lcdr(x))))) ≡
 LERR_UNCLOSED_OPEN ∧ null_p(lcdr(lcdr(lcdr(x))))) {
 pending = true;
 }
 else {
 SS(0, x = lcdr(x));
 while (pair_p(x)) {
 printf("%d%s==", fix_value(lcar(lcar(x))), Error[fix_value(lcar(lcar(x)))].message);
 serial(lcar(x), SERIAL_DETAIL, 12, NIL, Λ, &cleanup);
 printf("\n");
 SS(0, x = lcdr(x));
 }
 printf("\n");
 lapi_User_Register(NIL);
 pending = false;
 }
 printf("DONE");
 serial(Accumulator, SERIAL_DETAIL, 12, NIL, Λ, &cleanup);
 printf("\n"); }
 else printf("\n");

```

This code is used in section 4.

```
7. char *prompt(EditLine *eLunused)
{
 return "OK_";
}
```

**8. Index.**

*abort*: 6.  
*ACC*: 4, 6.  
*Accumulator*: 6.  
*argc*: 4.  
*argv*: 4, 5.  
*assert*: 4.  
*cell*: 4.  
*cleanup*: 4, 6.  
*cons*: 6.  
*die*: 4, 6.  
**EditLine**: 2, 3, 7.  
*EL\_EDITOR*: 5.  
*el\_end*: 4.  
*el\_gets*: 6.  
*EL\_HIST*: 5.  
*el\_init*: 5.  
*EL\_PROMPT*: 5.  
*el\_set*: 5.  
*evaluate\_program*: 4, 6.  
*event*: 4, 5, 6.  
*failure\_p*: 4, 6.  
*false*: 4, 6.  
*fix\_value*: 6.  
*fprintf*: 5, 6.  
*H\_ENTER*: 6.  
*H\_SETSIZE*: 5.  
**HistEvent**: 4.  
*history*: 5, 6.  
**History**: 3.  
*history\_end*: 4.  
*history\_init*: 5.  
*HISTORY\_SIZE*: 3, 5.  
*Ierror*: 6.  
*INITIALISE*: 4.  
*lapi\_User\_Register*: 6.  
*lcar*: 6.  
*lcdr*: 6.  
*length*: 4, 6.  
*LERR\_NONE*: 4.  
*LERR\_SYNTAX*: 6.  
*LERR\_UNCLOSED\_OPEN*: 6.  
*lex\_rop*: 4, 6.  
*line*: 4, 6.  
*lprint*: 4.  
*Lunused*: 7.  
*main*: 4.  
*mem\_init*: 4.  
*message*: 6.  
*NIL*: 4, 6.  
*null\_p*: 6.  
*pair\_p*: 6.  
*parse*: 4, 6.  
*pending*: 4, 6.  
*printf*: 6.  
*prompt*: 2, 5, 7.  
*reason*: 4, 6.  
*rope\_append*: 6.  
*rope\_new\_buffer*: 4, 6.  
*serial*: 6.  
*SERIAL\_DETAIL*: 6.  
*sigjmp\_buf*: 4.  
*sigsetjmp*: 4, 6.  
*SO*: 4, 6.  
*SS*: 4, 6.  
*stack\_push*: 4.  
*stderr*: 5, 6.  
*stdin*: 5.  
*stdout*: 5.  
*symbol\_new\_const*: 6.  
*true*: 4, 6.  
*UNDEFINED*: 6.  
*USERREG*: 6.  
*valid*: 4, 6.  
*Verror*: 4.  
*VOID*: 4, 6.

- ⟨Function declarations [2](#)⟩ Used in section [1](#).
- ⟨Global variables [3](#)⟩ Used in section [1](#).
- ⟨Initialise editline [5](#)⟩ Used in section [4](#).
- ⟨Read and dispatch a line [6](#)⟩ Used in section [4](#).

# REPL

|                    | Section           | Page |
|--------------------|-------------------|------|
| <b>REPL</b> .....  | <a href="#">1</a> | 1    |
| <b>Index</b> ..... | <a href="#">8</a> | 6    |