

```
1  /*****
2      Command line expression evaluator
3
4      Released to the public domain by Steve Hanov
5      steve.hanov@gmail.com
6
7      Compile with:
8
9      Unix:
10         gcc -o calc -lm calc.c
11
12      Windows:
13         cl /out:calc.exe calc.c
14  *****/
15  #include <stdio.h>
16  #include <stdlib.h>
17  #include <assert.h>
18  #include <math.h>
19  #include <setjmp.h>
20  #include <string.h>
21  #include <ctype.h>
22
23  /*
24  expr := var rest_var
25         term rest_expr
26
27  rest_expr := + term rest_expr
28              - term rest_expr
29              (nil)
30
31  term := factor rest_term
32
33  rest_term := * factor rest_term
34              / factor rest_term
35              % factor rest_term
36              <nil>
37
38  factor := - factor
39            num_op
40
41  num_op := num rest_num_op
42            variable rest_num_op
43            ( expr ) rest_num_op
44
45  rest_num_op := ^ num_op rest_num_op
46               <nil>
47
48  rest_var := '=' expr
49             rest_num_op
50
51  */
52
53  /*****
54      Keep variables in a map.
55  *****/
56
57  #define VAR_NAME_SIZE 31
58  typedef struct _MapEntry_t {
59      char name[VAR_NAME_SIZE+1];
60      double value;
61      struct _MapEntry_t* next;
62  } MapEntry_t;
63
64  MapEntry_t* varmap;
```

```
65
66 void
67 map_init(void)
68 {
69     varmap = 0;
70 }
71
72 void
73 map_clear(void)
74 {
75     MapEntry_t* cur = varmap;
76     while( cur ) {
77         MapEntry_t* next = cur->next;
78         free( cur );
79         cur = next;
80     }
81
82     varmap = 0;
83 }
84
85 MapEntry_t*
86 map_find( const char* var )
87 {
88     MapEntry_t* cur = varmap;
89     while( cur ) {
90         if ( strcmp( var, cur->name ) == 0 ) {
91             return cur;
92         }
93         cur = cur->next;
94     }
95
96     return 0;
97 }
98
99 void
100 map_add( const char* var, double value )
101 {
102     MapEntry_t* entry = map_find( var );
103     if ( entry == 0 ) {
104         entry = (MapEntry_t*)malloc( sizeof(MapEntry_t) );
105         strncpy( entry->name, var, VAR_NAME_SIZE + 1 );
106         entry->name[VAR_NAME_SIZE] = 0;
107         entry->next = varmap;
108         varmap = entry;
109     }
110
111     entry->value = value;
112 }
113
114 int
115 map_lookup( const char* var, double* value )
116 {
117     MapEntry_t* entry = map_find( var );
118     if ( entry ) {
119         *value = entry->value;
120         return 1;
121     }
122
123     return 0;
124 }
125
126
127
128 /*****
129     General purpose structure used to represent things returned by the
```

```
130     lexer and values as they are calculated up the parse tree.
131     *****/
132 #define TYPE_CHAR      0
133 #define TYPE_FLOAT     1
134 #define TYPE_EOF       2
135 #define TYPE_ERROR     3
136 #define TYPE_VARIABLE  4
137
138 typedef struct _val_t {
139     int type;
140     union {
141         double fval;
142         char cval;
143         char variable[255];
144     } d;
145 } val_t;
146
147 /*****
148     Print out a value
149     *****/
150 void
151 print_val( val_t* val )
152 {
153     if ( val->type == TYPE_FLOAT ) {
154         printf("%lf\n", val->d.fval );
155     } else if ( val->type == TYPE_CHAR ) {
156         printf("\'%c'\n", val->d.cval);
157     } else if ( val->type == TYPE_VARIABLE ) {
158         printf("Variable \'%s'\n", val->d.variable);
159     } else if ( val->type == TYPE_EOF ) {
160         printf("EOF\n");
161     } else if ( val->type == TYPE_ERROR ) {
162         printf("ERROR\n");
163     } else {
164         printf("Bad val type: %d\n", val->type);
165     }
166 }
167
168 /*****
169     State variables for the lexer
170     *****/
171
172 /* number of command line arguments */
173 int argc;
174
175 /* command line arguments array */
176 char** argv;
177
178 /* array parsed so far. Used for debugging and printing out error messages. */
179 static char buffer[1024];
180
181 /* the token that was most recently scanned by the lexer */
182 val_t next_val;
183
184 /* which argument we are currently scanning */
185 int arg = 0;
186
187 /* the index into argv[arg] that we are currently scanning */
188 int argp = 0;
189
190 /* the position in buffer[] that we are storing characters. */
191 int bpos = 0;
192
193 static int have_next_val = 0;
194
```

```

195 jmp_buf env;
196
197 void
198 reset(int argc, char** argv)
199 {
200     argc = argc;
201     argv = argv;
202     buffer[0] = 0;
203     arg = 0;
204     argp = 0;
205     bpos = 0;
206     have_next_val = 0;
207 }
208
209 /*****
210     Scanner. Scans tokens from the command line arguments.
211     *****/
212 void
213 lex(val_t* val, int next)
214 {
215     char token[25];
216     int tpos = 0;
217     int done = 0;
218     int number = 0;
219     enum {
220         read_start,
221         read_int,
222         read_mantissa,
223         read_hex,
224         read_var
225     } state = read_start;
226
227     if ( next ) {
228         have_next_val = 0;
229         return;
230     } else if ( have_next_val ) {
231         *val = next_val;
232         return;
233     }
234
235     while( !done ) {
236         /* get the next character. Add to buffer. Do not increment the next */
237         /* character to read. */
238         char ch;
239
240         if ( arg == argc ) {
241             val->type = TYPE_EOF;
242             val->d.fval = 0;
243             break;
244         }
245
246         ch = argv[arg][argp];
247         /*printf("argv[%d][%d] = %c (state=%d)\n", */
248         /*    arg, argp, argv[arg][argp], state); */
249
250         switch ( state ) {
251             case read_start:
252                 if ( ch >= '0' && ch <= '9' ) {
253                     state = read_int;
254                     tpos = 0;
255                     token[tpos++] = ch;
256                 } else if ( ch == '+' || ch == '-' ||
257                     ch == '/' || ch == '*' ||
258                     ch == '(' || ch == ')' ||
259                     ch == '%' || ch == '^' ||

```

```
260         ch == '=' )
261     {
262         val->type = TYPE_CHAR;
263         val->d.cval = ch;
264         done = 1;
265     } else if ( ch == ' ' || ch == '\t' || ch == 0 ) {
266
267     } else if ( ch == '.' ) {
268         tpos = 0;
269         token[tpos++] = '0';
270         token[tpos++] = '.';
271         state = read_mantissa;
272     } else if ( isalpha( ch ) ) {
273         state = read_var;
274         tpos = 0;
275         token[tpos++] = ch;
276     } else {
277         buffer[bpos] = 0;
278         printf("Parse error after: %s\n", buffer);
279         longjmp( env, 1 );
280     }
281     break;
282 case read_int:
283     if ( ch >= '0' && ch <= '9' ) {
284         if ( tpos < sizeof(token) ) {
285             token[tpos++] = ch;
286         } else {
287             token[tpos] = 0;
288             printf("Number too long: %s\n", token);
289         }
290     } else if ( ch == 'x' && tpos == 1 ) {
291         state = read_hex;
292     } else if ( ch == '.' ) {
293         if ( tpos < sizeof(token) ) {
294             token[ tpos++ ] = ch;
295         } else {
296             token[tpos] = 0;
297             printf("Number too long: %s\n", token);
298         }
299         state = read_mantissa;
300     } else {
301         token[tpos] = 0;
302         state = read_start;
303         val->type = TYPE_FLOAT;
304         val->d.fval = (double)atoi(token);
305         done = 1;
306         goto done;
307     }
308     break;
309 case read_mantissa:
310     if ( ch >= '0' && ch <= '9' ) {
311         if ( tpos < sizeof(token) ) {
312             token[tpos++] = ch;
313         } else {
314             token[tpos] = 0;
315             printf("Number too long: %s\n", token);
316             longjmp( env, 1 );
317         }
318     } else {
319         token[tpos] = 0;
320         state = read_start;
321         val->type = TYPE_FLOAT;
322         sscanf( token, "%lf", &val->d.fval );
323         done = 1;
324         goto done;
```

```
325     }
326     break;
327 case read_hex:
328     ch = tolower( ch );
329     if ( ch >= '0' && ch <= '9' ) {
330         number <= 4;
331         number += ch - '0';
332     } else if ( ch >= 'a' && ch <= 'f' ) {
333         number <= 4;
334         number += 10 + ch - 'a';
335     } else {
336         token[tpos] = 0;
337         state = read_start;
338         val->type = TYPE_FLOAT;
339         val->d.fval = number;
340         done = 1;
341         goto done;
342     }
343     break;
344 case read_var:
345     if ( ch >= 'a' && ch <= 'z' ||
346         ch >= 'A' && ch <= 'Z' ||
347         ch >= '0' && ch <= '9' ||
348         ch == '_' )
349     {
350         if ( tpos < sizeof(token) ) {
351             token[tpos++] = ch;
352         } else {
353             token[tpos] = 0;
354             printf("Variable too long: %s", token);
355             longjmp( env, 1 );
356         }
357     } else {
358         token[tpos] = 0;
359         state = read_start;
360         val->type = TYPE_VARIABLE;
361         strcpy( val->d.variable, token);
362         done = 1;
363         goto done;
364     }
365 }
366 }
367
368 /* increment the character we are going to read. */
369 if ( ch == 0 ) {
370     argp = 0;
371     argp++;
372 } else {
373     argp++;
374     buffer[bpos++] = ch;
375 }
376
377 }
378
379 done:
380     next_val = *val;
381     have_next_val = 1;
382     /*printf("lex(): "); */
383     /*print_val( val ); */
384     return;
385 }
386
387 /*****
388     If the next token is CH, then consume it and return 1. Otherwise,
389     do not consume it and return 0.
```

```
390  *****/
391  int
392  match_char( char ch )
393  {
394      val_t val;
395      lex(&val, 0);
396
397      if ( val.type == TYPE_CHAR && val.d.cval == ch ) {
398          lex( &val, 1 );
399          return 1;
400      }
401
402      return 0;
403  }
404
405  /*****
406   Return 1 if the next token is the end of file marker.
407   *****/
408  int
409  match_eof()
410  {
411      val_t val;
412      lex(&val, 0);
413
414      if ( val.type == TYPE_EOF ) {
415          return 1;
416      }
417
418      return 0;
419  }
420
421  /*****
422   If the next token is a number, then consume it and return 1. Otherwise,
423   do not consume it and return 0.
424   *****/
425  int
426  match_num( val_t* val )
427  {
428      lex( val, 0 );
429
430      if ( val->type == TYPE_FLOAT ) {
431          lex( val, 1 );
432          return 1;
433      }
434
435      return 0;
436  }
437
438  int
439  match_variable( val_t* val )
440  {
441      lex( val, 0 );
442
443      if ( val->type == TYPE_VARIABLE ) {
444          lex( val, 1 );
445          return 1;
446      }
447
448      return 0;
449  }
450
451  void
452  resolve_variable( val_t* val )
453  {
454      double fval;
```

```
455     if ( val->type != TYPE_VARIABLE ) {
456         printf("Error: value is not a variable.\n");
457         longjmp( env, 1 );
458     }
459
460     if ( !map_lookup( val->d.variable, &fval ) ) {
461         printf("%s not defined.\n", val->d.variable);
462         longjmp( env, 1 );
463     }
464
465     val->type = TYPE_FLOAT;
466     val->d.fval = fval;
467 }
468
469 void parse_term(val_t* val);
470 void parse_expr(val_t* val);
471 void parse_factor( val_t* val );
472 void parse_num_op( val_t* val );
473 void parse_factor( val_t* val );
474 void parse_rest_num_op( val_t* val );
475 void parse_rest_var( val_t* val );
476
477 // #define DEBUG_PRINT 1
478 #ifndef DEBUG_PRINT
479 #define dprintf(A) printf(A)
480 #endif
481
482 int level = 0;
483 void printtab() {
484     int i = 0;
485     for( i = 0; i < level; i++ ) {
486         dprintf("    ");
487     }
488 }
489
490 /*****
491     rest_term := * factor rest_term
492                / factor rest_term
493                % factor rest_term
494                <nil>
495 *****/
496 void
497 parse_rest_term( val_t* val )
498 {
499     printtab();
500     dprintf("parse_rest_term()\n");
501     level++;
502     if ( match_char( '*' ) ) {
503         val_t val2;
504         parse_factor( &val2 );
505         val->d.fval *= val2.d.fval;
506         parse_rest_term( val );
507     } else if ( match_char( '/' ) ) {
508         val_t val2;
509         parse_factor( &val2 );
510         if ( val2.d.fval != 0 ) {
511             val->d.fval /= val2.d.fval;
512         } else {
513             printf("Division by 0\n");
514             longjmp(env, 0);
515         }
516         parse_rest_term( val );
517     } else if ( match_char( '%' ) ) {
518         val_t val2;
519         parse_factor( &val2 );
```



```
520     if ( val2.d.fval != 0 ) {
521         val->d.fval = fmod( val->d.fval, val2.d.fval );
522     } else {
523         printf("Division by 0\n");
524         longjmp(env, 0);
525     }
526     parse_rest_term( val );
527 } else if ( match_eof() ) {
528
529 } else {
530
531 }
532
533 level--;
534 return;
535
536 }
537
538 /*****
539     term := factor rest_term
540 *****/
541 void
542 parse_term( val_t* val )
543 {
544     printtab();
545     dprintf("parse_term()\n");
546     level++;
547
548     parse_factor( val );
549     parse_rest_term( val );
550
551     level--;
552     return;
553 }
554
555 /*****
556     rest_num_op := ^ num_op rest_num_op
557                 <nil>
558 *****/
559 void
560 parse_rest_num_op( val_t* val )
561 {
562     if ( match_char( '^' ) ) {
563         val_t val2;
564         parse_num_op( &val2 );
565         val->d.fval = pow( val->d.fval, val2.d.fval );
566         parse_rest_num_op( val );
567     }
568     return;
569 }
570
571 /*****
572     num_op := num rest_num_op
573             ( expr ) rest_num_op
574 *****/
575 void
576 parse_num_op( val_t* val )
577 {
578     printtab();
579     dprintf("parse_num_op()\n");
580     level++;
581
582     if ( match_num( val ) ) {
583         parse_rest_num_op( val );
584     } else if ( match_variable( val ) ) {
```

```

585     resolve_variable( val );
586     parse_rest_num_op( val );
587 } else if ( match_char( '(' ) ) {
588     parse_expr( val );
589     if ( !match_char( ')' ) ) {
590         buffer[bpos] = 0;
591         printf("Missing bracket: %s\n", buffer);
592         longjmp( env, 1 );
593     }
594     parse_rest_num_op( val );
595 } else {
596     buffer[bpos] = 0;
597     printf("Parse error: %s\n", buffer);
598     longjmp( env, 1 );
599 }
600
601 level--;
602
603 return;
604 }
605
606 /*****
607     factor := - factor
608             num_op
609 *****/
610 void
611 parse_factor( val_t* val )
612 {
613     printtab();
614     dprintf("parse_factor()\n");
615     level++;
616
617     if ( match_char( '-' ) ) {
618         parse_factor( val );
619         val->d.fval = -val->d.fval;
620     } else {
621         parse_num_op( val );
622     }
623
624     level--;
625
626     return;
627 }
628
629 /*****
630     rest_expr := + term rest_expr
631               - term rest_expr
632               (nil)
633 *****/
634 void
635 parse_rest_expr( val_t* val )
636 {
637     printtab();
638     dprintf("parse_rest_expr()\n");
639     level++;
640     if ( match_char( '+' ) ) {
641         val_t val2;
642         parse_term( &val2 );
643         val->d.fval += val2.d.fval;
644         parse_rest_expr( val );
645     } else if ( match_char( '-' ) ) {
646         val_t val2;
647         parse_term( &val2 );
648         val->d.fval -= val2.d.fval;
649         parse_rest_expr( val );

```

```
650     } else if ( match_eof() ) {
651
652     } else {
653
654     }
655
656     level--;
657
658     return;
659 }
660
661 /*****
662     expr := term rest_expr
663     *****/
664 void parse_expr( val_t* val )
665 {
666     printtab();
667     dprintf("parse_expr()\n");
668
669     level++;
670     if ( match_variable( val ) ) {
671         parse_rest_var( val );
672     } else {
673         parse_term( val );
674         parse_rest_expr( val );
675     }
676
677     level--;
678
679     return;
680 }
681
682 /*****
683     rest_var := '=' expr
684               rest_num_op
685     *****/
686 void parse_rest_var( val_t* val )
687 {
688     if ( match_char( '=' ) ) {
689         val_t vexp;
690         parse_expr( &vexp );
691         if ( vexp.type != TYPE_FLOAT ) {
692             printf("Error: Tried to assign non-number to %s.\n", val->d.variable );
693             longjmp( env, 1 );
694         }
695
696         printf("Assigned to %s: ", val->d.variable );
697         map_add( val->d.variable, vexp.d.fval );
698         *val = vexp;
699
700     } else {
701         parse_rest_num_op( val );
702     }
703 }
704
705 int
706 parse( val_t* val )
707 {
708     if ( setjmp( env ) ) {
709         return 0;
710     }
711
712     parse_expr( val );
713     if ( !match_eof() ) {
714         printf("Trailing characters.\n");
```

```
715     longjmp( env, 1 );
716 }
717
718 return 1;
719 }
720
721 /*****
722     Print usage information
723     *****/
724 void
725 usage(void)
726 {
727     printf("Usage: calc [mathematical expression]\n");
728     exit(-1);
729 }
730
731 /*****
732     main
733     *****/
734 int
735 main( int argc, char* argv[] )
736 {
737     val_t val;
738     map_init();
739
740     if ( argc == 1 ) {
741         char cmd[100];
742         char* cmds = cmd;
743         int cmdlen = 0;
744         cmd[0] = 0;
745
746         printf("Use Control-C to quit.\n");
747
748         for( ;; ) {
749             top:
750             // print command line.
751             printf( "\r> %s", cmd );
752
753             cmdlen = strlen(cmd);
754
755             for( ;; ) {
756                 char c = _getch();
757                 if ( c == '\b' ) {
758                     if ( cmdlen > 0 ) {
759                         cmd[--cmdlen] = 0;
760                         printf( "\r> %s \b", cmd );
761                     }
762                 } else if ( c == '\r' ) {
763                     putc('\n', stdout);
764                     break;
765                 } else if ( c == 3 ) {
766                     printf("QUIT\n");
767                     exit(0);
768                 } else if ( cmdlen < sizeof(cmd)-1 ) {
769                     putc(c, stdout);
770                     //printf("%d\n", c);
771                     cmd[cmdlen++] = c;
772                     cmd[cmdlen] = 0;
773                 }
774             }
775         }
776
777         reset( 1, &cmds );
778
779         /* parse the expression. */

```

```
780         if ( parse( &val ) ) {
781             /* print the value. */
782             print_val( &val );
783         } else {
784             printf("Error.\n");
785         }
786     }
787 }
788
789 reset( pargc - 1, pargv + 1 );
790 /* parse the expression. */
791 parse_expr( &val );
792
793 /* print the value. */
794 print_val( &val );
795
796 map_clear();
797
798 return 0;
799 }
800
801
```