```
 1: // $Id: ubigint.h,v 1.11 2016-03-24 19:43:57-07 - - $
 2:
 3: #ifndef __UBIGINT_H__
 4: #define __UBIGINT_H__
 5:
 6: #include <exception>
 7: #include <iostream>
 8: #include <limits>
 9: #include <utility>
10: using namespace std;
11:
12: #include "debug.h"
13: #include "relops.h"
14:
15: class ubigint {
16:    friend ostream& operator<< (ostream&, const ubigint&);
17:    private:
18:       using unumber = unsigned long;
19:       unumber uvalue {};
20:    public:
21:       void multiply_by_2();
22:       void divide_by_2();
23:
24:       ubigint() = default; // Need default ctor as well.
25:       ubigint (unsigned long);
26:       ubigint (const string&);
27:
28:       ubigint operator+ (const ubigint&) const;
29:       ubigint operator- (const ubigint&) const;
30:       ubigint operator* (const ubigint&) const;
31:       ubigint operator/ (const ubigint&) const;
32:       ubigint operator% (const ubigint&) const;
33:
34:       bool operator== (const ubigint&) const;
35:       bool operator<  (const ubigint&) const;
36: };
37:
38: #endif
39:
```

```
 1: // $Id: ubigint.cpp,v 1.13 2016-06-23 16:00:31-07 - - $
 2:
 3: #include <cctype>
 4: #include <cstdlib>
 5: #include <exception>
 6: #include <stack>
 7: #include <stdexcept>
 8: using namespace std;
 9:
10: #include "ubigint.h"
11: #include "debug.h"
12:
13: ubigint::ubigint (unsigned long that): uvalue (that) {
14:    DEBUGF ('~', this << " -> " << uvalue)
15: }
16:
17: ubigint::ubigint (const string& that): uvalue(0) {
18:    DEBUGF ('~', "arg that = \"" << that << "\"");
19:    for (char digit: that) {
20:       if (not isdigit (digit)) {
21:          throw invalid_argument ("ubigint::ubigint(" + that + ")");
22:       }
23:       uvalue = uvalue * 10 + digit - '0';
24:    }
25: }
26:
27: ubigint ubigint::operator+ (const ubigint& that) const {
28:    return ubigint (uvalue + that.uvalue);
29: }
30:
31: ubigint ubigint::operator- (const ubigint& that) const {
32:    if (*this < that) throw domain_error ("ubigint::operator-(a<b)");
33:    return ubigint (uvalue - that.uvalue);
34: }
35:
36: ubigint ubigint::operator* (const ubigint& that) const {
37:    return ubigint (uvalue * that.uvalue);
38: }
39:
40: void ubigint::multiply_by_2() {
41:    uvalue *= 2;
42: }
43:
44: void ubigint::divide_by_2() {
45:    uvalue /= 2;
46: }
47:
```

```
48:
49: struct quo_rem { ubigint quotient; ubigint remainder; };
50: quo_rem udivide (const ubigint& dividend, ubigint divisor) {
51:    // Note: divisor is modified so pass by value (copy).
52:    ubigint zero {0};
53:    if (divisor == zero) throw domain_error ("udivide by zero");
54:    ubigint power_of_2 {1};
55:    ubigint quotient {0};
56:    ubigint remainder {dividend}; // left operand, dividend
57:    while (divisor < remainder) {
58:       divisor.multiply_by_2();
59:       power_of_2.multiply_by_2();
60:    }
61:    while (power_of_2 > zero) {
62:       if (divisor <= remainder) {
63:          remainder = remainder - divisor;
64:          quotient = quotient + power_of_2;
65:       }
66:       divisor.divide_by_2();
67:       power_of_2.divide_by_2();
68:    }
69:    return {.quotient = quotient, .remainder = remainder};
70: }
71:
72: ubigint ubigint::operator/ (const ubigint& that) const {
73:    return udivide (*this, that).quotient;
74: }
75:
76: ubigint ubigint::operator% (const ubigint& that) const {
77:    return udivide (*this, that).remainder;
78: }
79:
80: bool ubigint::operator== (const ubigint& that) const {
81:    return uvalue == that.uvalue;
82: }
83:
84: bool ubigint::operator< (const ubigint& that) const {
85:    return uvalue < that.uvalue;
86: }
87:
88: ostream& operator<< (ostream& out, const ubigint& that) {
89:    return out << "ubigint(" << that.uvalue << ")";
90: }
91:
```

```
 1: // $Id: bigint.h,v 1.29 2016-03-24 19:30:57-07 - - $
 2:
 3: #ifndef __BIGINT_H__
 4: #define __BIGINT_H__
 5:
 6: #include <exception>
 7: #include <iostream>
 8: #include <limits>
 9: #include <utility>
10: using namespace std;
11:
12: #include "debug.h"
13: #include "relops.h"
14: #include "ubigint.h"
15:
16: class bigint {
17:    friend ostream& operator<< (ostream&, const bigint&);
18:    private:
19:       ubigint uvalue;
20:       bool is_negative {false};
21:    public:
22:
23:       bigint() = default; // Needed or will be suppressed.
24:       bigint (long);
25:       bigint (const ubigint&, bool is_negative = false);
26:       explicit bigint (const string&);
27:
28:       bigint operator+() const;
29:       bigint operator-() const;
30:
31:       bigint operator+ (const bigint&) const;
32:       bigint operator- (const bigint&) const;
33:       bigint operator* (const bigint&) const;
34:       bigint operator/ (const bigint&) const;
35:       bigint operator% (const bigint&) const;
36:
37:       bool operator== (const bigint&) const;
38:       bool operator<  (const bigint&) const;
39: };
40:
41: #endif
42:
```

```
 1: // $Id: bigint.cpp,v 1.76 2016-06-14 16:34:24-07 - - $
 2:
 3: #include <cstdlib>
 4: #include <exception>
 5: #include <stack>
 6: #include <stdexcept>
 7: using namespace std;
 8:
 9: #include "bigint.h"
10: #include "debug.h"
11: #include "relops.h"
12:
13: bigint::bigint (long that): uvalue (that), is_negative (that < 0) {
14:    DEBUGF ('~', this << " -> " << uvalue)
15: }
16:
17: bigint::bigint (const ubigint& uvalue, bool is_negative):
18:                 uvalue(uvalue), is_negative(is_negative) {
19: }
20:
21: bigint::bigint (const string& that) {
22:    is_negative = that.size() > 0 and that[0] == '_';
23:    uvalue = ubigint (that.substr (is_negative ? 1 : 0));
24: }
25:
26: bigint bigint::operator+ () const {
27:    return *this;
28: }
29:
30: bigint bigint::operator- () const {
31:    return {uvalue, not is_negative};
32: }
33:
34: bigint bigint::operator+ (const bigint& that) const {
35:    ubigint result = uvalue + that.uvalue;
36:    return result;
37: }
38:
39: bigint bigint::operator- (const bigint& that) const {
40:    ubigint result = uvalue - that.uvalue;
41:    return result;
42: }
43:
44: bigint bigint::operator* (const bigint& that) const {
45:    bigint result = uvalue * that.uvalue;
46:    return result;
47: }
48:
49: bigint bigint::operator/ (const bigint& that) const {
50:    bigint result = uvalue / that.uvalue;
51:    return result;
52: }
53:
54: bigint bigint::operator% (const bigint& that) const {
55:    bigint result = uvalue % that.uvalue;
56:    return result;
57: }
58:
```

```
59: bool bigint::operator== (const bigint& that) const {
60:    return is_negative == that.is_negative and uvalue == that.uvalue;
61: }
62:
63: bool bigint::operator< (const bigint& that) const {
64:    if (is_negative != that.is_negative) return is_negative;
65:    return is_negative ? uvalue > that.uvalue
66:                       : uvalue < that.uvalue;
67: }
68:
69: ostream& operator<< (ostream& out, const bigint& that) {
70:    return out << "bigint(" << (that.is_negative ? "-" : "+")
71:               << "," << that.uvalue << ")";
72: }
73:
```

```
1: // $Id: libfns.h,v 1.2 2015-07-02 16:03:36-07 - - $
2:
3: // Library functions not members of any class.
4:
5: #include "bigint.h"
6:
7: bigint pow (const bigint& base, const bigint& exponent);
8:
```

```
 1: // $Id: libfns.cpp,v 1.4 2015-07-03 14:46:41-07 - - $
 2:
 3: #include "libfns.h"
 4:
 5: //
 6: // This algorithm would be more efficient with operators
 7: // *=, /=2, and is_odd.  But we leave it here.
 8: //
 9:
10: bigint pow (const bigint& base_arg, const bigint& exponent_arg) {
11:    bigint base (base_arg);
12:    bigint exponent (exponent_arg);
13:    static const bigint ZERO (0);
14:    static const bigint ONE (1);
15:    static const bigint TWO (2);
16:    DEBUGF ('^', "base = " << base << ", exponent = " << exponent);
17:    if (base == ZERO) return ZERO;
18:    bigint result = ONE;
19:    if (exponent < ZERO) {
20:       base = ONE / base;
21:       exponent = - exponent;
22:    }
23:    while (exponent > ZERO) {
24:       if (exponent % TWO == ONE) {
25:          result = result * base;
26:          exponent = exponent - 1;
27:       }else {
28:          base = base * base;
29:          exponent = exponent / 2;
30:       }
31:    }
32:    DEBUGF ('^', "result = " << result);
33:    return result;
34: }
35:
```

```
 1: // $Id: scanner.h,v 1.10 2016-06-23 16:00:31-07 - - $
 2:
 3: #ifndef __SCANNER_H__
 4: #define __SCANNER_H__
 5:
 6: #include <iostream>
 7: #include <utility>
 8: using namespace std;
 9:
10: #include "debug.h"
11:
12: enum class tsymbol {SCANEOF, NUMBER, OPERATOR};
13:
14: struct token {
15:    tsymbol symbol;
16:    string lexinfo;
17:    token (tsymbol sym, const string& lex = string()):
18:           symbol(sym), lexinfo(lex){
19:    }
20: };
21:
22: class scanner {
23:    private:
24:       istream& instream;
25:       int nextchar {instream.get()};
26:       bool good() { return nextchar != EOF; }
27:       char get();
28:       string strget() { return {get()}; }
29:    public:
30:       scanner (istream& instream = cin): instream(instream) {
31:          nextchar = instream.get();
32:          DEBUGF ('s', "nextchar = " << nextchar);
33:       }
34:       token scan();
35: };
36:
37: ostream& operator<< (ostream&, tsymbol);
38: ostream& operator<< (ostream&, const token&);
39:
40: #endif
41:
```

```
 1: // $Id: scanner.cpp,v 1.18 2016-06-23 16:00:31-07 - - $
 2:
 3: #include <cassert>
 4: #include <iostream>
 5: #include <locale>
 6: #include <stdexcept>
 7: #include <type_traits>
 8: #include <unordered_map>
 9: using namespace std;
10:
11: #include "scanner.h"
12: #include "debug.h"
13:
14: char scanner::get() {
15:    if (not good()) throw runtime_error ("scanner::get() past EOF");
16:    char currchar = nextchar;
17:    nextchar = instream.get();
18:    return currchar;
19: }
20:
21: token scanner::scan() {
22:    while (good() and isspace (nextchar)) get();
23:    if (not good()) return {tsymbol::SCANEOF};
24:    if (nextchar == '_' or isdigit (nextchar)) {
25:       token result {tsymbol::NUMBER, strget()};
26:       while (good() and isdigit (nextchar)) result.lexinfo += get();
27:       return result;
28:    }
29:    return {tsymbol::OPERATOR, strget()};
30: }
31:
32: ostream& operator<< (ostream& out, tsymbol symbol) {
33:    struct hasher {
34:       auto operator() (tsymbol sym) const {
35:          return static_cast<underlying_type<tsymbol>::type> (sym);
36:       }
37:    };
38:    static const unordered_map<tsymbol,string,hasher> map {
39:       {tsymbol::NUMBER  , "NUMBER"  },
40:       {tsymbol::OPERATOR, "OPERATOR"},
41:       {tsymbol::SCANEOF , "SCANEOF" },
42:    };
43:    return out << map.at(symbol);
44: }
45:
46: ostream& operator<< (ostream& out, const token& token) {
47:    out << "{" << token.symbol << ", \"" << token.lexinfo << "\"}";
48:    return out;
49: }
50:
```

```
 1: // $Id: debug.h,v 1.3 2015-07-01 18:52:26-07 - - $
 2:
 3: #ifndef __DEBUG_H__
 4: #define __DEBUG_H__
 5:
 6: #include <string>
 7: #include <vector>
 8: using namespace std;
 9:
10: //
11: // debug -
12: //    static class for maintaining global debug flags.
13: // setflags -
14: //    Takes a string argument, and sets a flag for each char in the
15: //    string.  As a special case, '@', sets all flags.
16: // getflag -
17: //    Used by the DEBUGF macro to check to see if a flag has been set.
18: //    Not to be called by user code.
19: //
20: class debugflags {
21:    private:
22:       static vector<bool> flags;
23:    public:
24:       static void setflags (const string& optflags);
25:       static bool getflag (char flag);
26:       static void where (char flag, const char* file, int line,
27:                          const char* func);
28: };
29:
30: //
31: // DEBUGF -
32: //    Macro which expands into trace code.  First argument is a
33: //    trace flag char, second argument is output code that can
34: //    be sandwiched between <<.  Beware of operator precedence.
35: //    Example:
36: //       DEBUGF ('u', "foo = " << foo);
37: //    will print two words and a newline if flag 'u' is  on.
38: //    Traces are preceded by filename, line number, and function.
39: //
40: #define DEBUGF(FLAG,CODE) { \
41:            if (debugflags::getflag (FLAG)) { \
42:               debugflags::where (FLAG, __FILE__, __LINE__, __func__); \
43:               cerr << CODE << endl; \
44:            } \
45:         }
46: #define DEBUGS(FLAG,STMT) { \
47:            if (debugflags::getflag (FLAG)) { \
48:               debugflags::where (FLAG, __FILE__, __LINE__, __func__); \
49:               STMT; \
50:            } \
51:         }
52: #endif
53:
```

```
 1: // $Id: debug.cpp,v 1.7 2016-06-14 18:19:17-07 - - $
 2:
 3: #include <climits>
 4: #include <iostream>
 5: #include <vector>
 6: using namespace std;
 7:
 8: #include "debug.h"
 9: #include "util.h"
10:
11: vector<bool> debugflags::flags (UCHAR_MAX + 1, false);
12:
13: void debugflags::setflags (const string& initflags) {
14:    for (const unsigned char flag: initflags) {
15:       if (flag == '@') flags.assign (flags.size(), true);
16:                   else flags[flag] = true;
17:    }
18:    // Note that DEBUGF can trace setflags.
19:    if (getflag ('x')) {
20:       string flag_chars;
21:       for (size_t index = 0; index < flags.size(); ++index) {
22:          if (getflag (index)) flag_chars += static_cast<char> (index);
23:       }
24:       DEBUGF ('x', "debugflags::flags = " << flag_chars);
25:    }
26: }
27:
28: //
29: // getflag -
30: //    Check to see if a certain flag is on.
31: //
32:
33: bool debugflags::getflag (char flag) {
34:    return flags[static_cast<unsigned char> (flag)];
35: }
36:
37: void debugflags::where (char flag, const char* file, int line,
38:                         const char* func) {
39:    note() << "DEBUG(" << flag << ") " << file << "[" << line << "] "
40:           << func << "()" << endl;
41: }
42:
```

```
 1: // $Id: util.h,v 1.1 2016-06-14 18:19:17-07 - - $
 2:
 3: //
 4: // util -
 5: //    A utility class to provide various services
 6: //    not conveniently included in other modules.
 7: //
 8:
 9: #ifndef __UTIL_H__
10: #define __UTIL_H__
11:
12: #include <iomanip>
13: #include <iostream>
14: #include <sstream>
15: #include <stdexcept>
16: #include <vector>
17: using namespace std;
18:
19: #include "debug.h"
20:
21: //
22: // ydc_exn -
23: //    Indicate a problem where processing should be abandoned and
24: //    the main function should take control.
25: //
26:
27: class ydc_exn: public runtime_error {
28:    public:
29:       explicit ydc_exn (const string& what);
30: };
31:
32: //
33: // octal -
34: //    Convert integer to octal string.
35: //
36:
37: template <typename numeric>
38: const string octal (numeric number) {
39:    ostringstream stream;
40:    stream << showbase << oct << number;
41:    return stream.str();
42: }
43:
```

```
44:
45: //
46: // main -
47: //    Keep track of execname and exit status.  Must be initialized
48: //    as the first thing done inside main.  Main should call:
49: //       main::execname (argv[0]);
50: //    before anything else.
51: //
52:
53: class exec {
54:    private:
55:       static string execname_;
56:       static int status_;
57:       static void execname (const string& argv0);
58:       friend int main (int, char**);
59:    public:
60:       static void status (int status);
61:       static const string& execname() {return execname_; }
62:       static int status() {return status_; }
63: };
64:
65: //
66: // complain -
67: //    Used for starting error messages.  Sets the exit status to
68: //    EXIT_FAILURE, writes the program name to cerr, and then
69: //    returns the cerr ostream.  Example:
70: //       complain() << filename << ": some problem" << endl;
71: //
72:
73: ostream& note();
74: ostream& error();
75:
76: #endif
77:
```

```
 1: // $Id: util.cpp,v 1.1 2016-06-14 18:19:17-07 - - $
 2:
 3: #include <cstring>
 4: using namespace std;
 5:
 6: #include "util.h"
 7:
 8: ydc_exn::ydc_exn (const string& what): runtime_error (what) {
 9: }
10:
11: string exec::execname_; // Must be initialized from main().
12: int exec::status_ = EXIT_SUCCESS;
13:
14: void exec::execname (const string& argv0) {
15:    execname_ = basename (argv0.c_str());
16:    cout << boolalpha;
17:    cerr << boolalpha;
18:    DEBUGF ('Y', "execname = " << execname_);
19: }
20:
21: void exec::status (int new_status) {
22:    new_status &= 0xFF;
23:    if (status_ < new_status) status_ = new_status;
24: }
25:
26: ostream& note() {
27:    return cerr << exec::execname() << ": ";
28: }
29:
30: ostream& error() {
31:    exec::status (EXIT_FAILURE);
32:    return note();
33: }
34:
```

```
 1: // $Id: iterstack.h,v 1.13 2014-06-26 17:21:55-07 - - $
 2:
 3: //
 4: // The class std::stack does not provide an iterator, which is
 5: // needed for this class.  So, like std::stack, class iterstack
 6: // is implemented on top of a container.
 7: //
 8: // We use private inheritance because we want to restrict
 9: // operations only to those few that are approved.  All functions
10: // are merely inherited from the container, with only ones needed
11: // being exported as public.
12: //
13: // No implementation file is needed because all functions are
14: // inherited, and the convenience functions that are added are
15: // trivial, and so can be inline.
16: //
17: // Any underlying container which supports the necessary operations
18: // could be used, such as vector, list, or deque.
19: //
20:
21: #ifndef __ITERSTACK_H__
22: #define __ITERSTACK_H__
23:
24: #include <vector>
25: using namespace std;
26:
27: template <typename value_type>
28: class iterstack: private vector<value_type> {
29:    private:
30:       using stack_t = vector<value_type>;
31:       using stack_t::crbegin;
32:       using stack_t::crend;
33:       using stack_t::push_back;
34:       using stack_t::pop_back;
35:       using stack_t::back;
36:       using const_iterator = typename stack_t::const_reverse_iterator;
37:    public:
38:       using stack_t::clear;
39:       using stack_t::empty;
40:       using stack_t::size;
41:       inline const_iterator begin() {return crbegin();}
42:       inline const_iterator end() {return crend();}
43:       inline void push (const value_type& value) {push_back (value);}
44:       inline void pop() {pop_back();}
45:       inline const value_type& top() const {return back();}
46: };
47:
48: #endif
49:
```

```
 1: // $Id: relops.h,v 1.2 2016-06-13 13:47:33-07 - - $
 2:
 3: //
 4: // Assuming that for any given type T, there are operators
 5: // bool operator< (const T&, const T&);
 6: // bool operator== (const T&, const T&);
 7: // as fundamental comparisons for type T, define the other
 8: // six operators in terms of the basic ones.
 9: //
10:
11: #ifndef __REL_OPS_H__
12: #define __REL_OPS_H__
13:
14: template <typename value>
15: inline bool operator!= (const value& left, const value& right) {
16:    return not (left == right);
17: }
18:
19: template <typename value>
20: inline bool operator>  (const value& left, const value& right) {
21:    return right < left;
22: }
23:
24: template <typename value>
25: inline bool operator<= (const value& left, const value& right) {
26:    return not (right < left);
27: }
28:
29: template <typename value>
30: inline bool operator>= (const value& left, const value& right) {
31:    return not (left < right);
32: }
33:
34: #endif
35:
```

```
 1: // $Id: main.cpp,v 1.54 2016-06-14 18:19:17-07 - - $
 2:
 3: #include <cassert>
 4: #include <deque>
 5: #include <iostream>
 6: #include <stdexcept>
 7: #include <unordered_map>
 8: #include <utility>
 9: using namespace std;
10:
11: #include <unistd.h>
12:
13: #include "bigint.h"
14: #include "debug.h"
15: #include "iterstack.h"
16: #include "libfns.h"
17: #include "scanner.h"
18: #include "util.h"
19:
20: using bigint_stack = iterstack<bigint>;
21:
22: void do_arith (bigint_stack& stack, const char oper) {
23:    if (stack.size() < 2) throw ydc_exn ("stack empty");
24:    bigint right = stack.top();
25:    stack.pop();
26:    DEBUGF ('d', "right = " << right);
27:    bigint left = stack.top();
28:    stack.pop();
29:    DEBUGF ('d', "left = " << left);
30:    bigint result;
31:    switch (oper) {
32:       case '+': result = left + right; break;
33:       case '-': result = left - right; break;
34:       case '*': result = left * right; break;
35:       case '/': result = left / right; break;
36:       case '%': result = left % right; break;
37:       case '^': result = pow (left, right); break;
38:       default: throw invalid_argument ("do_arith operator "s + oper);
39:    }
40:    DEBUGF ('d', "result = " << result);
41:    stack.push (result);
42: }
43:
44: void do_clear (bigint_stack& stack, const char) {
45:    DEBUGF ('d', "");
46:    stack.clear();
47: }
48:
```

```
49:
50: void do_dup (bigint_stack& stack, const char) {
51:    bigint top = stack.top();
52:    DEBUGF ('d', top);
53:    stack.push (top);
54: }
55:
56: void do_printall (bigint_stack& stack, const char) {
57:    for (const auto& elem: stack) cout << elem << endl;
58: }
59:
60: void do_print (bigint_stack& stack, const char) {
61:    cout << stack.top() << endl;
62: }
63:
64: void do_debug (bigint_stack& stack, const char) {
65:    (void) stack; // SUPPRESS: warning: unused parameter 'stack'
66:    cout << "Y not implemented" << endl;
67: }
68:
69: class ydc_quit: public exception {};
70: void do_quit (bigint_stack&, const char) {
71:    throw ydc_quit();
72: }
73:
74: using function_t = void (*)(bigint_stack&, const char);
75: using fn_hash = unordered_map<string,function_t>;
76: fn_hash do_functions = {
77:    {"+"s, do_arith},
78:    {"-"s, do_arith},
79:    {"*"s, do_arith},
80:    {"/"s, do_arith},
81:    {"%"s, do_arith},
82:    {"^"s, do_arith},
83:    {"Y"s, do_debug},
84:    {"c"s, do_clear},
85:    {"d"s, do_dup},
86:    {"f"s, do_printall},
87:    {"p"s, do_print},
88:    {"q"s, do_quit},
89: };
90:
```

```
 91:
 92: //
 93: // scan_options
 94: //    Options analysis:  The only option is -Dflags.
 95: //
 96: void scan_options (int argc, char** argv) {
 97:    opterr = 0;
 98:    for (;;) {
 99:       int option = getopt (argc, argv, "@:");
100:       if (option == EOF) break;
101:       switch (option) {
102:          case '@':
103:             debugflags::setflags (optarg);
104:             break;
105:          default:
106:             error() << "-" << static_cast<char> (optopt)
107:                     << ": invalid option" << endl;
108:             break;
109:       }
110:    }
111:    if (optind < argc) {
112:       error() << "operand not permitted" << endl;
113:    }
114: }
115:
```

```
116:
117: //
118: // Main function.
119: //
120: int main (int argc, char** argv) {
121:    exec::execname (argv[0]);
122:    scan_options (argc, argv);
123:    bigint_stack operand_stack;
124:    scanner input;
125:    try {
126:       for (;;) {
127:          try {
128:             token lexeme = input.scan();
129:             switch (lexeme.symbol) {
130:                case tsymbol::SCANEOF:
131:                   throw ydc_quit();
132:                   break;
133:                case tsymbol::NUMBER:
134:                   operand_stack.push (bigint (lexeme.lexinfo));
135:                   break;
136:                case tsymbol::OPERATOR: {
137:                   fn_hash::const_iterator fn
138:                            = do_functions.find (lexeme.lexinfo);
139:                   if (fn == do_functions.end()) {
140:                      throw ydc_exn (octal (lexeme.lexinfo[0])
141:                                     + " is unimplemented");
142:                   }
143:                   fn->second (operand_stack, lexeme.lexinfo.at(0));
144:                   break;
145:                   }
146:                default:
147:                   assert (false);
148:             }
149:          }catch (ydc_exn& exn) {
150:             cout << exn.what() << endl;
151:          }
152:       }
153:    }catch (ydc_quit&) {
154:       // Intentionally left empty.
155:    }
156:    return exec::status();
157: }
158:
```

```
 1: # $Id: Makefile,v 1.20 2016-06-14 18:37:34-07 - - $
 2:
 3: MKFILE      = Makefile
 4: DEPFILE     = ${MKFILE}.dep
 5: NOINCL      = ci clean spotless
 6: NEEDINCL    = ${filter ${NOINCL}, ${MAKECMDGOALS}}
 7: GMAKE       = ${MAKE} --no-print-directory
 8: COMPILECPP  = g++ -std=gnu++14 -g -O0 -Wall -Wextra
 9: MAKEDEPCPP  = g++ -std=gnu++14 -MM
10:
11: MODULES     = ubigint bigint libfns scanner debug util
12: CPPHEADER   = ${MODULES:=.h} iterstack.h relops.h
13: CPPSOURCE   = ${MODULES:=.cpp} main.cpp
14: EXECBIN     = ydc
15: OBJECTS     = ${CPPSOURCE:.cpp=.o}
16: MODULESRC   = ${foreach MOD, ${MODULES}, ${MOD}.h ${MOD}.cpp}
17: OTHERSRC    = ${filter-out ${MODULESRC}, ${CPPHEADER} ${CPPSOURCE}}
18: ALLSOURCES  = ${MODULESRC} ${OTHERSRC} ${MKFILE}
19: LISTING     = Listing.ps
20:
21: all : ${EXECBIN}
22:
23: ${EXECBIN} : ${OBJECTS}
24:         ${COMPILECPP} -o $@ ${OBJECTS}
25:
26: %.o : %.cpp
27:         ${COMPILECPP} -c $<
28:
29: ci : ${ALLSOURCES}
30:         - checksource ${ALLSOURCES}
31:         - cpplint.py.perl ${CPPSOURCE}
32:         cid + ${ALLSOURCES}
33:
34: lis : ${ALLSOURCES}
35:         mkpspdf ${LISTING} ${ALLSOURCES} ${DEPFILE}
36:
37: clean :
38:         - rm ${OBJECTS} ${DEPFILE} core ${EXECBIN}.errs
39:
40: spotless : clean
41:         - rm ${EXECBIN} ${LISTING} ${LISTING:.ps=.pdf}
42:
43: dep : ${CPPSOURCE} ${CPPHEADER}
44:         @ echo "# ${DEPFILE} created `LC_TIME=C date`" >${DEPFILE}
45:         ${MAKEDEPCPP} ${CPPSOURCE} >>${DEPFILE}
46:
47: ${DEPFILE} :
48:         @ touch ${DEPFILE}
49:         ${GMAKE} dep
50:
51: again :
52:         ${GMAKE} spotless dep ci all lis
53:
54: ifeq (${NEEDINCL}, )
55: include ${DEPFILE}
56: endif
57:
```

```
    1: # Makefile.dep created Thu Jun 23 16:00:32 PDT 2016
    2: ubigint.o: ubigint.cpp ubigint.h debug.h relops.h
    3: bigint.o: bigint.cpp bigint.h debug.h relops.h ubigint.h
    4: libfns.o: libfns.cpp libfns.h bigint.h debug.h relops.h ubigint.h
    5: scanner.o: scanner.cpp scanner.h debug.h
    6: debug.o: debug.cpp debug.h util.h
    7: util.o: util.cpp util.h debug.h
    8: main.o: main.cpp bigint.h debug.h relops.h ubigint.h iterstack.h libfns.
h \
    9:  scanner.h util.h
```