Calculator

Christian Bager Bach Houmann

February 25, 2023

Listing 1: calculator.hpp

```
#ifndef INCLUDE_ALGEBRA_HPP
#define INCLUDE_ALGEBRA_HPP

#include <vector>
#include <string>
#include <memory>
#include <algorithm>
#include <stdexcept>

namespace calculator
{
    /** Type to capture the state of entire calculator (one number per variable): */
    using state_t = std::vector<double>;

    /** Forward declarations to get around circular dependencies: */
    class expr_t;

    enum op_t
    {
        plus,
        minus,
        multiply,
        divide,
        assign
    };

    class term_t
    {
    public:
        term_t() = default;
        virtual ~term_t() noexcept = default;
        virtual double operator()(state_t &s) const = 0;
    };

    struct const_t : public term_t
    {
        double value;

    public:
        explicit const_t(double value) : value{value} {}
        double operator()(state_t &) const override { return value; }
    };

    /** Class representing a variable */
    class var_t : public term_t
    {
        size_t id;

        /** only friends are allowed to construct variable instances */
```

```
50          explicit var_t(size_t id) : id{id} {}
51
52      public:
53          [[nodiscard]] size_t get_id() const { return id; }
54
55          /** returns the value of the variable stored in a state */
56          double operator()(state_t &s) const override { return s[id]; }
57
58          /** evaluates an assignment to a given expression and returns the resulting value */
59          double operator()(state_t &, const expr_t &) const;
60
61          friend class symbol_table_t;
62
63          operator expr_t() const;
64
65          var_t(expr_t expr);
66      };
67
68      class assign_t : public term_t
69      {
70          std::shared_ptr<var_t> var;
71          std::shared_ptr<term_t> term;
72          op_t op;
73
74      public:
75          assign_t(std::shared_ptr<var_t> var, std::shared_ptr<term_t> term, op_t op)
76              : var{std::move(var)}, term{std::move(term)}, op{op} {}
77
78          double operator()(state_t &s) const override
79          {
80              double value = (*term)(s);
81              double *var_value = &s[var->get_id()];
82
83              switch (op)
84              {
85              case assign:
86                  *var_value = value;
87                  break;
88              case plus:
89                  *var_value += value;
90                  break;
91              case minus:
92                  *var_value -= value;
93                  break;
94              case multiply:
95                  *var_value *= value;
96                  break;
97              case divide:
98                  if (value == 0)
99                      throw std::runtime_error("division by zero");
100
101                 *var_value /= value;
102                 break;
103             default:
104                 throw std::runtime_error("invalid assignment operator");
105             }
106
107             return *var_value;
108         }
109     };
110
```

```
111     class unary_t : public term_t
112     {
113         std::shared_ptr<term_t> term;
114         op_t op;
115 
116     public:
117         unary_t(std::shared_ptr<term_t> term, op_t op)
118             : term{std::move(term)}, op{op} {}
119 
120         double operator()(state_t &s) const override
121         {
122             switch (op)
123             {
124             case plus:
125                 return +(*term)(s);
126             case minus:
127                 return -(*term)(s);
128             default:
129                 throw std::runtime_error("invalid unary operator");
130             }
131         }
132     };
133 
134     class binary_t : public term_t
135     {
136         std::shared_ptr<term_t> left;
137         std::shared_ptr<term_t> right;
138         op_t op;
139 
140     public:
141         binary_t(std::shared_ptr<term_t> left, std::shared_ptr<term_t> right, op_t op)
142             : left{std::move(left)}, right{std::move(right)}, op{op} {}
143 
144         double operator()(state_t &s) const override
145         {
146             switch (op)
147             {
148             case plus:
149                 return (*left)(s) + (*right)(s);
150             case minus:
151                 return (*left)(s) - (*right)(s);
152             case multiply:
153                 return (*left)(s) * (*right)(s);
154             case divide:
155                 if ((*right)(s) == 0)
156                     throw std::runtime_error("division by zero");
157 
158                 return (*left)(s) / (*right)(s);
159             default:
160                 throw std::runtime_error("invalid binary operator");
161             }
162         }
163     };
164 
165     class symbol_table_t
166     {
167         std::vector<std::string> names;
168         std::vector<double> initial;
169 
170     public:
171         [[nodiscard]] var_t var(std::string name, double init = 0)
```

```
172         {
173             auto res = names.size();
174 
175             names.push_back(std::move(name));
176             initial.push_back(init);
177 
178             return var_t{res};
179         }
180 
181         [[nodiscard]] state_t state() const { return {initial}; }
182     };
183 
184     struct expr_t
185     {
186         std::shared_ptr<term_t> term;
187 
188         explicit expr_t(const var_t &var) : term{std::make_shared<var_t>(var)} {}
189 
190         // Binary constructor
191         expr_t(const expr_t &e1, const expr_t &e2, op_t op)
192             : term{std::make_shared<binary_t>(e1.term, e2.term, op)} {}
193 
194         // Unary constructor
195         expr_t(const expr_t &e, op_t op)
196             : term{std::make_shared<unary_t>(e.term, op)} {}
197 
198         // Const constructor
199         expr_t(double value) : term{std::make_shared<const_t>(value)} {}
200 
201         // Assignment constructor
202         expr_t(const var_t &var, const expr_t &e, op_t op)
203             : term{std::make_shared<assign_t>(std::make_shared<var_t>(var), e.term, op)} {}
204 
205         double operator()(state_t &s) const { return (*term)(s); }
206     };
207 
208     // Converstion operator from var_t to expr_t
209     var_t::operator expr_t() const { return expr_t{*this}; }
210 
211     var_t::var_t(expr_t expr)
212     {
213         throw std::logic_error("assignment destination must be a variable expression");
214     }
215 
216     /** assignment operation */
217     inline double var_t::operator()(state_t &s, const expr_t &e) const { return s[id] = e(s); }
218 
219     /** unary operators: */
220     inline expr_t operator+(const expr_t &e) { return expr_t{e, op_t::plus}; }
221     inline expr_t operator-(const expr_t &e) { return expr_t{e, op_t::minus}; }
222 
223     /** binary operators: */
224     inline expr_t operator+(const expr_t &e1, const expr_t &e2) { return expr_t{e1, e2,
↪op_t::plus}; }
225     inline expr_t operator-(const expr_t &e1, const expr_t &e2) { return expr_t{e1, e2,
↪op_t::minus}; }
226     inline expr_t operator<<=(const var_t &v, const expr_t &e) { return expr_t{v, e,
↪op_t::assign}; }
227     inline expr_t operator*(const expr_t &e1, const expr_t &e2) { return expr_t{e1, e2,
↪op_t::multiply}; }
228     inline expr_t operator/(const expr_t &e1, const expr_t &e2) { return expr_t{e1, e2,
```

```
↪op_t::divide}; }
229     inline expr_t operator+=(const var_t &v, const expr_t &e) { return expr_t{v, e, op_t::plus}; }
230     inline expr_t operator-=(const var_t &v, const expr_t &e) { return expr_t{v, e,   ↙
↪op_t::minus}; }
231 }
232
233 #endif // INCLUDE_ALGEBRA_HPP
```

Listing 2: test_calculator.cpp

```
1  #include "calculator.hpp"
2
3  #define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
4  #include <doctest/doctest.h>
5
6  TEST_CASE("Calculate expressions lazily")
7  {
8      auto sys = calculator::symbol_table_t{};
9      auto a = sys.var("a", 2);
10     auto b = sys.var("b", 3);
11     auto c = sys.var("c");
12     auto state = sys.state();
13     auto os = std::ostringstream();
14
15     SUBCASE("Reading the value of a variable from state")
16     {
17         CHECK(a(state) == 2);
18         CHECK(b(state) == 3);
19         CHECK(c(state) == 0);
20     }
21     SUBCASE("Unary operations")
22     {
23         CHECK((+a)(state) == 2);
24         CHECK((-b)(state) == -3);
25         CHECK((-c)(state) == 0);
26     }
27     SUBCASE("Addition and subtraction")
28     {
29         CHECK((a + b)(state) == 5);
30         CHECK((a - b)(state) == -1);
31         // the state should not have changed:
32         CHECK(a(state) == 2);
33         CHECK(b(state) == 3);
34         CHECK(c(state) == 0);
35     }
36     SUBCASE("Assignment expression evaluation")
37     {
38         CHECK(c(state) == 0);
39         CHECK((c <<= b - a)(state) == 1);
40         CHECK(c(state) == 1);
41         // TODO: implement multiplication
42         // CHECK((c += b - a * c)(state) == 2);
43         // CHECK(c(state) == 2);
44         // CHECK((c += b - a * c)(state) == 1);
45         // CHECK(c(state) == 1);
46         /*
47         TODO: implement other assignments: +=, -=, *=, /=
48         CHECK_THROWS_MESSAGE((c - a += b - c), "assignment destination must be a variable   ↙
↪expression");
49         */
50     }
51     SUBCASE("Parenthesis")
```

```
    {
        CHECK((a - (b - c))(state) == -1);
        CHECK((a - (b - a))(state) == 1);
    }
    // TODO: implement multiplication and division
    SUBCASE("Evaluation of multiplication and division")
    {
        CHECK((a * b)(state) == 6);
        CHECK((a / b)(state) == 2. / 3);
        CHECK_THROWS_MESSAGE((a / c)(state), "division by zero");
    }
    SUBCASE("Mixed addition and multiplication")
    {
        CHECK((a + a * b)(state) == 8);
        CHECK((a - b / a)(state) == 0.5);
    }
    /*
    // TODO: implement support for constant expressions
    SUBCASE("Constant expressions")
    {
        CHECK((7 + a)(state) == 9);
        CHECK((a - 7)(state) == -5);
    }
    SUBCASE("Store expression and evaluate lazily")
    {
        auto expr = (a + b) * c;
        auto c_4 = c <<= 4;
        CHECK(expr(state) == 0);
        CHECK(c_4(state) == 4);
        CHECK(expr(state) == 20);
    }
    */
}
```