

Calculator

Alexander Nykjær
Ane Søgaard Jørgensen
Jakob Sønderby Kristensen
Mikael Vind Mikkelsen
Trine Juhl Holmager

March 2, 2022

Listing 1: calculator.hpp

```
1  #ifndef INCLUDE_ALGEBRA_HPP
2  #define INCLUDE_ALGEBRA_HPP
3
4  #include <utility>
5  #include <vector>
6  #include <string>
7  #include <memory>
8  #include <algorithm>
9  #include <stdexcept>
10
11 namespace calculator
12 {
13     /** Type to capture the state of entire calculator (one number per variable): */
14     using state_t = std::vector<double>;
15
16     /** Forward declarations to get around circular dependencies: */
17     struct expr_t;
18     struct const_t;
19     struct unary_t;
20     struct var_t;
21     struct binary_t;
22     struct assign_t;
23
24
25     struct visitor {
26         virtual void visit(const_t&) = 0;
27         virtual void visit(unary_t&) = 0;
28         virtual void visit(var_t&) = 0;
29         virtual void visit(binary_t&) = 0;
30         virtual void visit(assign_t&) = 0;
31
32         virtual ~visitor() noexcept = default;
33     };
34
35     struct term_t {
36         virtual double operator()(state_t&) const = 0;
37         term_t() = default;
38         virtual ~term_t() = default;
39
40         virtual void accept(visitor& v) = 0;
41     };
42
43     struct const_t : public term_t {
```

```

45 private:
46     double value;
47
48 public:
49     double operator()(state_t&) const override {
50         return value;
51     }
52
53     explicit const_t(double val) {
54         value = val;
55     }
56
57     void accept(visitor& v) override {
58         v.visit(*this);
59     }
60 };
61
62 struct unary_t : public term_t {
63 private:
64     std::shared_ptr<term_t> term;
65
66 public:
67     enum op_t { minus, plus } op;
68
69     double operator()(state_t& s) const override {
70         auto& e = *term;
71         switch(op) {
72             case minus:
73                 return -e(s);
74             case plus:
75                 return +e(s);
76         }
77     }
78
79     explicit unary_t(std::shared_ptr<term_t> t, op_t o = op_t::plus) {
80         term = std::move(t);
81         op = o;
82     }
83
84     void accept(visitor& v) override {
85         v.visit(*this);
86     }
87 };
88
89 struct binary_t : public term_t {
90 private:
91     std::shared_ptr<term_t> term1;
92     std::shared_ptr<term_t> term2;
93
94 public:
95     enum op_t { addition, subtraction, multiplication, division } op;
96
97     double operator()(state_t& s) const override {
98         auto& e1 = *term1;
99         auto& e2 = *term2;
100
101         switch(op) {
102             case addition:
103                 return e1(s) + e2(s);
104             case subtraction:
105                 return e1(s) - e2(s);

```

```

106         case multiplication:
107             return e1(s) * e2(s);
108         case division:
109             if (e2(s) == 0) {
110                 throw std::logic_error{"division by zero"};
111             }
112             return e1(s) / e2(s);
113     }
114 }
115
116 binary_t(std::shared_ptr<term_t> t1, std::shared_ptr<term_t> t2, op_t o) {
117     term1 = std::move(t1);
118     term2 = std::move(t2);
119     op = o;
120 }
121
122 void accept(visitor& v) override {
123     v.visit(*this);
124 }
125
126 friend struct assign_t;
127 };
128
129 struct var_t : public term_t {
130 private:
131     size_t id;
132
133 public:
134
135     double operator()(state_t& s) const override {
136         return s[id];
137     }
138     double operator()(state_t& s, const term_t& t) const {
139         return s[id] = t(s);
140     }
141
142     explicit var_t(size_t id) {
143         var_t::id = id;
144     };
145
146     var_t(const var_t& other) {
147         id = other.id;
148     }
149
150     var_t& operator=(const var_t& other) {
151         id = other.id;
152         return *this;
153     }
154
155     void accept(visitor& v) override {
156         v.visit(*this);
157     }
158
159     friend struct assign_t;
160 };
161
162 struct assign_t : public term_t {
163 private:
164     std::shared_ptr<var_t> var;
165     std::shared_ptr<term_t> term;
166

```

```

167 public:
168     enum op_t { assign, plus_ass, minus_ass, mult_ass, div_ass } op;
169
170     double operator()(state_t& s) const override {
171         auto& v = *var;
172         auto& e = *term;
173
174         switch (op) {
175             case assign:
176                 return s[v.id] = e(s);
177             case plus_ass:
178                 return s[v.id] += e(s);
179             case minus_ass:
180                 return s[v.id] -= e(s);
181             case mult_ass:
182                 return s[v.id] *= e(s);
183             case div_ass:
184                 if (e(s) == 0) {
185                     throw std::logic_error{"division by zero"};
186                 }
187                 return s[v.id] /= e(s);
188         }
189     }
190
191     assign_t(std::shared_ptr<var_t> v, std::shared_ptr<term_t> t, op_t o = op_t::assign) {
192         var = std::move(v);
193         term = std::move(t);
194         op = o;
195     }
196
197     void accept(visitor& v) override {
198         v.visit(*this);
199     }
200 };
201
202 struct expr_t {
203     std::shared_ptr<term_t> term;
204     double operator()(state_t& s) {
205         auto& n = *term;
206         return n(s);
207     }
208     expr_t(std::shared_ptr<term_t> t) {
209         term = std::move(t);
210     }
211
212     expr_t(const var_t& t) {
213         term = std::make_shared<var_t>(t);
214     }
215
216     expr_t(double t) {
217         term = std::make_shared<const_t>(t);
218     }
219 };
220
221 class symbol_table_t
222 {
223     std::vector<std::string> names;
224     std::vector<double> initial;
225 public:
226     [[nodiscard]] var_t var(std::string name, double init = 0) {
227         auto res = names.size();

```

```

228         names.push_back(std::move(name));
229         initial.push_back(init);
230         return var_t{res};
231     }
232     [[nodiscard]] state_t state() const { return {initial}; }
233 };
234
235 struct evaluator : public visitor {
236     void visit(const_t& t) override {
237
238     }
239
240     void visit(unary_t& t) override {
241
242     }
243
244     void visit(var_t& t) override {
245
246     }
247
248     void visit(binary_t& t) override {
249
250     }
251
252     void visit(assign_t& t) override {
253
254     }
255 };
256
257 /** assignment operation */
258 inline expr_t operator<=(const var_t& v, const expr_t& e) {
259     assign_t term{std::make_shared<var_t>(v), e.term, assign_t::assign};
260     return expr_t{std::make_shared<assign_t>(term)};
261 }
262 inline expr_t operator+=(const var_t& v, const expr_t& e) {
263     assign_t term{std::make_shared<var_t>(v), e.term, assign_t::plus_ass};
264     return expr_t{std::make_shared<assign_t>(term)};
265 }
266 inline expr_t operator-=(const var_t& v, const expr_t& e) {
267     assign_t term{std::make_shared<var_t>(v), e.term, assign_t::minus_ass};
268     return expr_t{std::make_shared<assign_t>(term)};
269 }
270 inline expr_t operator*=(const var_t& v, const expr_t& e) {
271     assign_t term{std::make_shared<var_t>(v), e.term, assign_t::mult_ass};
272     return expr_t{std::make_shared<assign_t>(term)};
273 }
274 inline expr_t operator/=(const var_t& v, const expr_t& e) {
275     assign_t term{std::make_shared<var_t>(v), e.term, assign_t::div_ass};
276     return expr_t{std::make_shared<assign_t>(term)};
277 }
278
279 /** unary operators: */
280 inline expr_t operator+(const expr_t& e) {
281     unary_t term{e.term, unary_t::plus};
282     return expr_t{std::make_shared<unary_t>(term)};
283 }
284 inline expr_t operator-(const expr_t& e) {
285     unary_t term{e.term, unary_t::minus};
286     return expr_t{std::make_shared<unary_t>(term)};
287 }
288

```

```

289
290 /** binary operators: */
291 inline expr_t operator+(const expr_t& e1, const expr_t& e2) {
292     binary_t term{e1.term, e2.term, binary_t::addition};
293     return expr_t{std::make_shared<binary_t>(term)};
294 }
295 inline expr_t operator-(const expr_t& e1, const expr_t& e2) {
296     binary_t term{e1.term, e2.term, binary_t::subtraction};
297     return expr_t{std::make_shared<binary_t>(term)};
298 }
299 inline expr_t operator*(const expr_t& e1, const expr_t& e2) {
300     binary_t term{e1.term, e2.term, binary_t::multiplication};
301     return expr_t{std::make_shared<binary_t>(term)};
302 }
303 inline expr_t operator/(const expr_t& e1, const expr_t& e2) {
304     binary_t term{e1.term, e2.term, binary_t::division};
305     return expr_t{std::make_shared<binary_t>(term)};
306 }
307 }
308
309 #endif // INCLUDE_ALGEBRA_HPP

```
