Calculator

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

March 2, 2022

Listing 1: calculator.hpp

```cpp
#ifndef INCLUDE_ALGEBRA_HPP
#define INCLUDE_ALGEBRA_HPP

#include <utility>
#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: */
    struct expr_t;
    struct const_t;
    struct unary_t;
    struct var_t;
    struct binary_t;
    struct assign_t;


    struct visitor {
        virtual void visit(const_t&) = 0;
        virtual void visit(unary_t&) = 0;
        virtual void visit(var_t&) = 0;
        virtual void visit(binary_t&) = 0;
        virtual void visit(assign_t&) = 0;

        virtual ~visitor() noexcept = default;
    };

    struct term_t {
        virtual double operator()(state_t&) const = 0;
        term_t() = default;
        virtual ~term_t() = default;

        virtual void accept(visitor& v) = 0;
    };


    struct const_t : public term_t {
```

```
    private:
        double value;

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

        explicit const_t(double val) {
            value = val;
        }

        void accept(visitor& v) override {
            v.visit(*this);
        }
    };

    struct unary_t : public term_t {
    private:
        std::shared_ptr<term_t> term;

    public:
        enum op_t { minus, plus } op;

        double operator()(state_t& s) const override {
            auto& e = *term;
            switch(op) {
                case minus:
                    return -e(s);
                case plus:
                    return +e(s);
            }
        }

        explicit unary_t(std::shared_ptr<term_t> t, op_t o = op_t::plus) {
            term = std::move(t);
            op = o;
        }

        void accept(visitor& v) override {
            v.visit(*this);
        }
    };

    struct binary_t : public term_t {
    private:
        std::shared_ptr<term_t> term1;
        std::shared_ptr<term_t> term2;

    public:
        enum op_t { addition, subtraction, multiplication, division } op;

        double operator()(state_t& s) const override {
            auto& e1 = *term1;
            auto& e2 = *term2;

            switch(op) {
                case addition:
                    return e1(s) + e2(s);
                case subtraction:
                    return e1(s) - e2(s);
            case multiplication:
                return e1(s) * e2(s);
            case division:
                if (e2(s) == 0) {
                    throw std::logic_error{"division by zero"};
                }
                return e1(s) / e2(s);
        }
    }

    binary_t(std::shared_ptr<term_t> t1, std::shared_ptr<term_t> t2, op_t o) {
        term1 = std::move(t1);
        term2 = std::move(t2);
        op = o;
    }

    void accept(visitor& v) override {
        v.visit(*this);
    }

    friend struct assign_t;
};

struct var_t : public term_t {
private:
    size_t id;

public:

    double operator()(state_t& s) const override {
        return s[id];
    }
    double operator()(state_t& s, const term_t& t) const {
        return s[id] = t(s);
    }

    explicit var_t(size_t id) {
        var_t::id = id;
    };

    var_t(const var_t& other) {
        id = other.id;
    }

    var_t& operator=(const var_t& other) {
        id = other.id;
        return *this;
    }

    void accept(visitor& v) override {
        v.visit(*this);
    }

    friend struct assign_t;
};

struct assign_t : public term_t {
private:
    std::shared_ptr<var_t> var;
    std::shared_ptr<term_t> term;

```

```cpp
public:
    enum op_t { assign, plus_ass, minus_ass, mult_ass, div_ass } op;

    double operator()(state_t& s) const override {
        auto& v = *var;
        auto& e = *term;

        switch (op) {
            case assign:
                return s[v.id] = e(s);
            case plus_ass:
                return s[v.id] += e(s);
            case minus_ass:
                return s[v.id] -= e(s);
            case mult_ass:
                return s[v.id] *= e(s);
            case div_ass:
                if (e(s) == 0) {
                    throw std::logic_error{"division by zero"};
                }
                return s[v.id] /= e(s);
        }
    }

    assign_t(std::shared_ptr<var_t> v, std::shared_ptr<term_t> t, op_t o = op_t::assign) {
        var = std::move(v);
        term = std::move(t);
        op = o;
    }

    void accept(visitor& v) override {
        v.visit(*this);
    }
};

struct expr_t {
    std::shared_ptr<term_t> term;
    double operator()(state_t& s) {
        auto& n = *term;
        return n(s);
    }
    expr_t(std::shared_ptr<term_t> t) {
        term = std::move(t);
    }

    expr_t(const var_t& t) {
        term = std::make_shared<var_t>(t);
    }

    expr_t(double t) {
        term = std::make_shared<const_t>(t);
    }
};

class symbol_table_t
{
    std::vector<std::string> names;
    std::vector<double> initial;
public:
    [[nodiscard]] var_t var(std::string name, double init = 0) {
        auto res = names.size();
```

```
            names.push_back(std::move(name));
            initial.push_back(init);
            return var_t{res};
        }
        [[nodiscard]] state_t state() const { return {initial}; }
    };

    struct evaluator : public visitor {
        void visit(const_t& t) override {

        }

        void visit(unary_t& t) override {

        }

        void visit(var_t& t) override {

        }

        void visit(binary_t& t) override {

        }

        void visit(assign_t& t) override {

        }
    };


    /** assignment operation */
    inline expr_t operator<<=(const var_t& v, const expr_t& e) {
        assign_t term{std::make_shared<var_t>(v), e.term, assign_t::assign};
        return expr_t{std::make_shared<assign_t>(term)};
    }
    inline expr_t operator+=(const var_t& v, const expr_t& e) {
        assign_t term{std::make_shared<var_t>(v), e.term, assign_t::plus_ass};
        return expr_t{std::make_shared<assign_t>(term)};
    }
    inline expr_t operator-=(const var_t& v, const expr_t& e) {
        assign_t term{std::make_shared<var_t>(v), e.term, assign_t::minus_ass};
        return expr_t{std::make_shared<assign_t>(term)};
    }
    inline expr_t operator*=(const var_t& v, const expr_t& e) {
        assign_t term{std::make_shared<var_t>(v), e.term, assign_t::mult_ass};
        return expr_t{std::make_shared<assign_t>(term)};
    }
    inline expr_t operator/=(const var_t& v, const expr_t& e) {
        assign_t term{std::make_shared<var_t>(v), e.term, assign_t::div_ass};
        return expr_t{std::make_shared<assign_t>(term)};
    }

    /** unary operators: */
    inline expr_t operator+(const expr_t& e) {
        unary_t term{e.term, unary_t::plus};
        return expr_t{std::make_shared<unary_t>(term)};
    }
    inline expr_t operator-(const expr_t& e) {
        unary_t term{e.term, unary_t::minus};
        return expr_t{std::make_shared<unary_t>(term)};
    }

    /** binary operators: */
    inline expr_t operator+(const expr_t& e1, const expr_t& e2) {
        binary_t term{e1.term, e2.term, binary_t::addition};
        return expr_t{std::make_shared<binary_t>(term)};
    }
    inline expr_t operator-(const expr_t& e1, const expr_t& e2) {
        binary_t term{e1.term, e2.term, binary_t::subtraction};
        return expr_t{std::make_shared<binary_t>(term)};
    }
    inline expr_t operator*(const expr_t& e1, const expr_t& e2) {
        binary_t term{e1.term, e2.term, binary_t::multiplication};
        return expr_t{std::make_shared<binary_t>(term)};
    }
    inline expr_t operator/(const expr_t& e1, const expr_t& e2) {
        binary_t term{e1.term, e2.term, binary_t::division};
        return expr_t{std::make_shared<binary_t>(term)};
    }
}

#endif // INCLUDE_ALGEBRA_HPP
```