Calculator

Christian Bager Bach Houmann

February 25, 2023

Listing 1: calculator.hpp

```cpp
#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 */
        explicit var_t(size_t id) : id{id} {}

    public:
        [[nodiscard]] size_t get_id() const { return id; }

        /** returns the value of the variable stored in a state */
        double operator()(state_t &s) const override { return s[id]; }

        /** evaluates an assignment to a given expression and returns the resulting value */
        double operator()(state_t &, const expr_t &) const;

        friend class symbol_table_t;

        operator expr_t() const;

        var_t(expr_t expr);
    };

    class assign_t : public term_t
    {
        std::shared_ptr<var_t> var;
        std::shared_ptr<term_t> term;
        op_t op;

    public:
        assign_t(std::shared_ptr<var_t> var, std::shared_ptr<term_t> term, op_t op)
            : var{std::move(var)}, term{std::move(term)}, op{op} {}

        double operator()(state_t &s) const override
        {
            double value = (*term)(s);
            double *var_value = &s[var->get_id()];

            switch (op)
            {
            case assign:
                *var_value = value;
                break;
            case plus:
                *var_value += value;
                break;
            case minus:
                *var_value -= value;
                break;
            case multiply:
                *var_value *= value;
                break;
            case divide:
                if (value == 0)
                    throw std::runtime_error("division by zero");

                *var_value /= value;
                break;
            default:
                throw std::runtime_error("invalid assignment operator");
            }

            return *var_value;
        }
    };

```

```cpp
class unary_t : public term_t
{
    std::shared_ptr<term_t> term;
    op_t op;

public:
    unary_t(std::shared_ptr<term_t> term, op_t op)
        : term{std::move(term)}, op{op} {}

    double operator()(state_t &s) const override
    {
        switch (op)
        {
        case plus:
            return +(*term)(s);
        case minus:
            return -(*term)(s);
        default:
            throw std::runtime_error("invalid unary operator");
        }
    }
};

class binary_t : public term_t
{
    std::shared_ptr<term_t> left;
    std::shared_ptr<term_t> right;
    op_t op;

public:
    binary_t(std::shared_ptr<term_t> left, std::shared_ptr<term_t> right, op_t op)
        : left{std::move(left)}, right{std::move(right)}, op{op} {}

    double operator()(state_t &s) const override
    {
        switch (op)
        {
        case plus:
            return (*left)(s) + (*right)(s);
        case minus:
            return (*left)(s) - (*right)(s);
        case multiply:
            return (*left)(s) * (*right)(s);
        case divide:
            if ((*right)(s) == 0)
                throw std::runtime_error("division by zero");

            return (*left)(s) / (*right)(s);
        default:
            throw std::runtime_error("invalid binary operator");
        }
    }
};

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 expr_t
    {
        std::shared_ptr<term_t> term;

        explicit expr_t(const var_t &var) : term{std::make_shared<var_t>(var)} {}

        // Binary constructor
        expr_t(const expr_t &e1, const expr_t &e2, op_t op)
            : term{std::make_shared<binary_t>(e1.term, e2.term, op)} {}

        // Unary constructor
        expr_t(const expr_t &e, op_t op)
            : term{std::make_shared<unary_t>(e.term, op)} {}

        // Const constructor
        expr_t(double value) : term{std::make_shared<const_t>(value)} {}

        // Assignment constructor
        expr_t(const var_t &var, const expr_t &e, op_t op)
            : term{std::make_shared<assign_t>(std::make_shared<var_t>(var), e.term, op)} {}

        double operator()(state_t &s) const { return (*term)(s); }
    };

    // Converstion operator from var_t to expr_t
    var_t::operator expr_t() const { return expr_t{*this}; }

    var_t::var_t(expr_t expr)
    {
        throw std::logic_error("assignment destination must be a variable expression");
    }

    /** assignment operation */
    inline double var_t::operator()(state_t &s, const expr_t &e) const { return s[id] = e(s); }

    /** unary operators: */
    inline expr_t operator+(const expr_t &e) { return expr_t{e, op_t::plus}; }
    inline expr_t operator-(const expr_t &e) { return expr_t{e, op_t::minus}; }

    /** binary operators: */
    inline expr_t operator+(const expr_t &e1, const expr_t &e2) { return expr_t{e1, e2,   ↙
↪op_t::plus}; }
    inline expr_t operator-(const expr_t &e1, const expr_t &e2) { return expr_t{e1, e2,   ↙
↪op_t::minus}; }
    inline expr_t operator<<=(const var_t &v, const expr_t &e) { return expr_t{v, e,   ↙
↪op_t::assign}; }
    inline expr_t operator*(const expr_t &e1, const expr_t &e2) { return expr_t{e1, e2,   ↙
↪op_t::multiply}; }
    inline expr_t operator/(const expr_t &e1, const expr_t &e2) { return expr_t{e1, e2,   ↙
op_t::divide}; }
    inline expr_t operator+=(const var_t &v, const expr_t &e) { return expr_t{v, e, op_t::plus}; }
    inline expr_t operator-=(const var_t &v, const expr_t &e) { return expr_t{v, e, op_t::minus}; }
}

#endif // INCLUDE_ALGEBRA_HPP
```

Listing 2: test_calculator.cpp

```
#include "calculator.hpp"

#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include <doctest/doctest.h>

TEST_CASE("Calculate expressions lazily")
{
    auto sys = calculator::symbol_table_t{};
    auto a = sys.var("a", 2);
    auto b = sys.var("b", 3);
    auto c = sys.var("c");
    auto state = sys.state();
    auto os = std::ostringstream();

    SUBCASE("Reading the value of a variable from state")
    {
        CHECK(a(state) == 2);
        CHECK(b(state) == 3);
        CHECK(c(state) == 0);
    }
    SUBCASE("Unary operations")
    {
        CHECK((+a)(state) == 2);
        CHECK((-b)(state) == -3);
        CHECK((-c)(state) == 0);
    }
    SUBCASE("Addition and subtraction")
    {
        CHECK((a + b)(state) == 5);
        CHECK((a - b)(state) == -1);
        // the state should not have changed:
        CHECK(a(state) == 2);
        CHECK(b(state) == 3);
        CHECK(c(state) == 0);
    }
    SUBCASE("Assignment expression evaluation")
    {
        CHECK(c(state) == 0);
        CHECK((c <<= b - a)(state) == 1);
        CHECK(c(state) == 1);
        // TODO: implement multiplication
        // CHECK((c += b - a * c)(state) == 2);
        // CHECK(c(state) == 2);
        // CHECK((c += b - a * c)(state) == 1);
        // CHECK(c(state) == 1);
        /*
        TODO: implement other assignments: +=, -=, *=, /=
        CHECK_THROWS_MESSAGE((c - a += b - c), "assignment destination must be a variable expression");
        */
    }
    SUBCASE("Parenthesis")
```

```cpp
    {
        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);
    }
    */
}
```