TU Munich, Department of Informatics
Chair XXV: Data Science and Engineering
Prof. Dr. Thomas Neumann

Exercise for the *Practical Course: Systems Programming in C++* Summer Term 2021

Michael Freitag, Moritz Sichert ({freitagm,sichert}@in.tum.de)
https://db.in.tum.de/teaching/ss21/c++praktikum

Sheet Nr. 06 (due on 31.05.2021 at 23:59)

Create a fork of the Git repository at `https://gitlab.db.in.tum.de/cpplab21/sheet06`. The repository contains a subdirectory for each exercise on this sheet, into which you should put your answers. **Do not forget to `git push` your solution before the deadline expires.**

Once the deadline for this sheet expires, a signed tag will be created in your fork automatically. Your solutions will be graded based on the state of your repository at this tag. **Do not attempt to modify or remove this tag, as we cannot grade your solution otherwise.**

Exercise 1

(100 points)

An *abstract syntax tree (AST)* is the standard data structure that is used to represent the syntactic structure of source code written in any programming language. In an AST each node represents a syntactic construct, such as an if-then-else statement, an arithmetic expression, or a single literal. The children of an AST node represent the operands of a syntactic construct, for example the left- and right-hand side of a binary addition.

Usually, a suitable polymorphic class hierarchy is required to implement AST nodes, since most nodes can have many different types of nodes as children. In this exercise you will implement an AST for simple arithmetic expressions on `doubles`. Your AST should be able to represent the following operations:

- Unary plus (class name `UnaryPlus`)
- Unary minus (class name `UnaryMinus`)
- Addition (class name `Add`)
- Subtraction (class name `Subtract`)
- Multiplication (class name `Multiply`)
- Division (class name `Divide`)
- Power (class name `Power`)

These operations can either operate on `double` constants, or on named parameters. That is, in addition to the inner nodes listed above your AST implementation should support the following leaf nodes:

- `double` constant (class name `Constant`)
- Parameter (class name `Parameter`)

The AST shown in Figure 1, for example, represents the arithmetic expression `-P0 + 2 * (4 - P1)` where `P0` and `P1` represent parameters.

(a) Use the program scaffold contained in the subdirectory for this exercise. Add your code to the existing files and do not add any new files. Design and implement a polymorphic class hierarchy for the nine AST nodes listed above. All AST nodes should eventually derive from the base class `ASTNode` which is already partly defined in the program scaffold. In

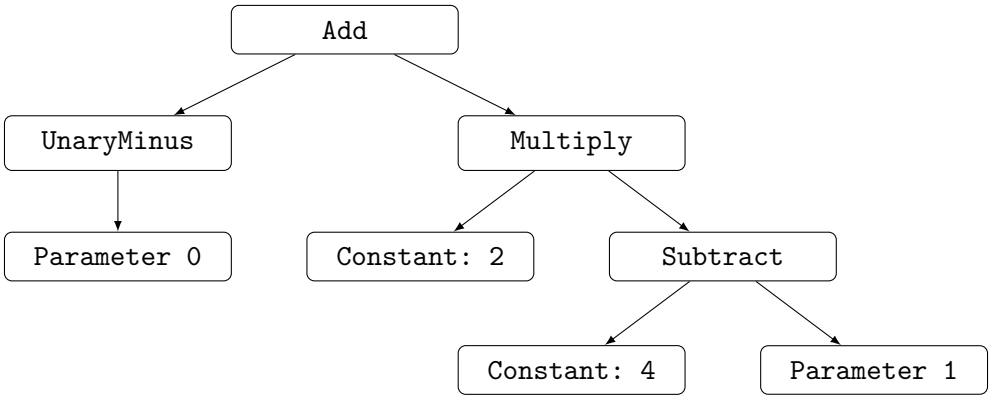

Figure 1: An AST representing the arithmetic expression `-P0 + 2 * (4 - P1)`

addition, you should define classes `BinaryASTNode` and `UnaryASTNode` which serve as base classes for nodes representing binary and unary operations, respectively.

All classes should support the following operations:

- `getType()`: Return the type of the AST node as a member of the enum class `ASTNode::Type` which is already defined in the program scaffold.

In addition, the unary AST nodes should support the following operations:

- Construction with a single child node given as a `unique_ptr`
- `getInput()`: Return a `const` reference to the child node
- `releaseInput()`: Transfer ownership of the child node to the caller

Binary AST nodes should additionally support the following operations:

- Construction with a left and right child node given as `unique_ptr`s
- `getLeft()`: Return a `const` reference to the left child node
- `getRight()`: Return a `const` reference to the right child node
- `releaseLeft()`: Transfer ownership of the left child node to the caller
- `releaseRight()`: Transfer ownership of the right child node to the caller

The `Constant` node should support the following operations:

- Construction with a `double` specifying the value of the constant
- `getValue()`: Return the value of the constant

Finally, the `Parameter` node should support the following operations. Note that parameters are identified by a zero-based index in order to facilitate implementation of subtask (b).

- Construction with an index specifying the index of the parameter
- `getIndex()`: Return the index of the parameter

Take care to minimize code duplication by defining common functionality as far up the class hierarchy as possible. Ensure that all classes can be destroyed properly. Your implementation should contain some minimal documentation.

(b) An AST can be used to actually evaluate the arithmetic expression that it represents. Of course, we need to assign values to all parameters for this to work. Thus, you should first implement an `EvaluationContext` class which supports the following operations.

- `pushParameter(double)`: Push a value for the next parameter. Remember that parameters are identified by zero-based indexes, so we can simply store parameter values in an `std::vector`.
- `getParameter(unsigned)`: Return the value for the specified parameter index

Subsequently, you should add a pure virtual function `evaluate` to the `ASTNode` class. This function should take a `const` reference to an `EvaluationContext` as parameter, and return the computed value of the expression as a `double`.

Implement the `evaluate` function for the AST nodes. The inner nodes should recursively evaluate their children, before computing and returning their own value. For the `Power` node, you can use the `std::pow` function defined in the `<cmath>` header. `Constant` nodes can simply return their stored value, and `Parameter` nodes need to look up their value in the supplied `EvaluationContext`. You can assume that no divisions by zero occur.

For the purposes of this subtask, you can assume that the `EvaluationContext` contains a binding for each required parameter, i.e. you do not need to explicitly check that a value exists for a given parameter.

(c) We can also implement some simple optimizations on our AST. For example, the expression `P0 + 2 + 3` could be simplified to `P0 + 5`. In order to implement such optimizations, add a pure virtual function

```
void optimize(std::unique_ptr<ASTNode>& thisRef)
```

to the `ASTNode` class. This function takes a reference to the `unique_ptr` that owns the current node as parameter (i.e. `thisRef.get() == this`). This is required to implement optimizations where the type of the current node changes, such as simplifying `-(-a)` to `a`. In this case, `optimize` is first called on a `UnaryMinus` node. This node then detects that it is part of the `-(-a)` expression (in particular that it represents the first minus). Finally, it moves the node representing `a` into `thisRef`, effectively changing its own type.

While this pattern is very useful when manipulating recursive data structures such as trees, some care has to be taken. After changing `thisRef`, the current `this` pointer and thus all data members and member functions become invalid. Therefore, the `optimize` function has to ensure that no accesses to the `this` pointer happen after changing `thisRef`.

Furthermore, we cannot directly move one of the inputs of the current node into `thisRef`, such as `thisRef = move(input)`. This would destroy `this` in the move-assignment operator of `unique_ptr` which would in turn also destroy `input`. Instead, we must first move `input` into a temporary variable, before moving it into `thisRef`.

Implement the `optimize` function for the AST nodes. Your implementation should first recursively optimize the children of a node, before attempting to apply the following optimizations, where `E`, `E1`, and `E2` indicate arbitrary subexpressions, `CONST` indicates a constant, and `^` indicates the power operator:

- Subexpressions that only contain constants should be evaluated immediately. You do not have to reorder operands even if this would lead to further optimization opportunities (i.e. `E + 3 + 4` should be optimized to `E + 7`, but `3 + E + 4` does not have to be optimized).
- `+E` should be optimized to `E`
- `-(-E)` should be optimized to `E`
- `E - 0` should be optimized to `E`
- `0 - E` should be optimized to `-E`

- `-(E1 - E2)` should be optimized to `E2 - E1`
- `E1 - (-E2)` should be optimized to `E1 + E2`
- `E + 0` should be optimized to `E`
- `0 + E` should be optimized to `E`
- `(-E1) + E2` should be optimized to `E2 - E1`
- `E1 + (-E2)` should be optimized to `E1 - E2`
- `E * 0` should be optimized to `0`
- `0 * E` should be optimized to `0`
- `E * 1` should be optimized to `E`
- `1 * E` should be optimized to `E`
- `(-E1) * (-E2)` should be optimized to `E1 * E2`
- `E / 1` should be optimized to `E`
- `0 / E` should be optimized to `0`
- `E / CONST` should be optimized to `E * (1 / CONST)`
- `(-E1) / (-E2)` should be optimized to `E1 / E2`
- `E ^ 0` should be optimized to `1`
- `E ^ 1` should be optimized to `E`
- `0 ^ E` should be optimized to `0`
- `1 ^ E` should be optimized to `1`
- `E ^ -1` should be optimized to `1 / E`

(d) Finally, we would also like to print the AST to `std::cout`. While we could simply add another recursive method to the `ASTNode` class, it is generally considered bad practice to tightly couple an algorithm to the object structure on which the algorithm operates. In fact, by the same argumentation we should have decoupled the optimization algorithm from the AST in the previous subtask.

The *visitor pattern* can be used to achieve a looser coupling in such cases. For this, you should define an abstract `ASTVisitor` class which contains one pure virtual `visit` method for each AST node. These methods should each take a `const` reference to the respective AST node as their parameter. Furthermore, you will need to add a pure virtual `accept` method to the `ASTNode` class, and provide implementations in the subclasses. The `accept` method should take a reference to a `ASTVisitor` object and simply invoke the correct `visit` method on this object.

This pattern effectively allows us to add new virtual methods to the AST class hierarchy without actually changing the AST classes themselves. Instead, we can now implement the functionality in a subclass of the `ASTVisitor` class, and rely on *double dispatch* to select the correct `visit` method for a given AST node. When passing this visitor to the `accept` method of an `ASTNode`, the most derived implementation of the `accept` method is selected depending on the actual type of the AST node (first dispatch). Subsequently, this implementation of the `accept` function can then call the correct overload of the `visit` method (second dispatch).

Finally, implement a `PrintVisitor` subclass of the `ASTVisitor` class. The `visit` methods should recursively print a representation of the AST to `std::cout`. Both binary and unary expressions should be surrounded by parentheses.