UNIVERSITÄT BERN

Exercise 01: Warm-Up – Stack and Syntree

Compiler Construction (Spring 2023)

Handout: Thursday, February 23
Discussion: Thursday, March 02

---

1. Stacks

(a) The following Java code listing shows the interface definition of a generic Stack abstraction comprising the classical operations `push`, `pop`, `isEmpty`, `size` and `top` (aka. `peek`). Your first task is to implement the Stack interface in Java.

```
public interface IStack<E> {
    public boolean isEmpty();
    public int size();
    public void push(E item);
    public E top();
    public void pop();
}
```

(b) Now, let's use our Stack implementation by an algorithm that reads a given String (character by character) and determines whether parentheses ( ), brackets [ ] and braces { } are correctly balanced. For example, `"(hello) (world)"` represents a balanced String, while `"static public void main(String args[]) {"` represents a counterexample. You may use these examples (and others) to test your implementation.

2. Trees

In the sequel, we will work with simple arithmetic expressions which shall be stored in a tree structure as illustrated in Figure 1. Initially, we support three kinds of expressions (`Number`, `Sum` and `Mult`) and one operation (`eval`) that evaluates an expression.

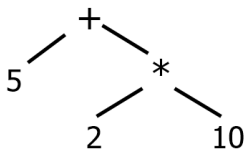

Figure 1: Expression tree

(a) Implement arithmetic expressions in Java using a classical procedural approach as illustrated in Figure 2. The (static) function `eval` traverses the tree data structure in a post-order (i.e, depth-first) fashion. For each visited node, the concrete (semantic) action is determined by the type of the node.

(b) Now, implement arithmetic expressions using a classical object-oriented approach as illustrated in Figure 3. In this implementation variant, each kind of expression knows about its (semantic) action when being asked for an evaluation.

(c) Implement arithmetic expressions using the Visitor design pattern, as illustrated in Figure 4. In this variant, the classes accept a `Visitor`, and we have yet one concrete `EvalVisitor` who knows how to evaluate the different kinds of expressions.

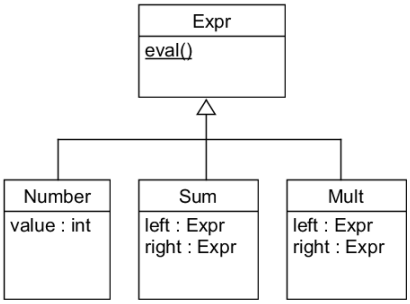

Figure 2: Procedural implementation

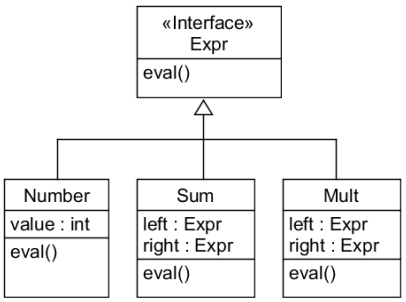

Figure 3: Classical object-oriented implementation

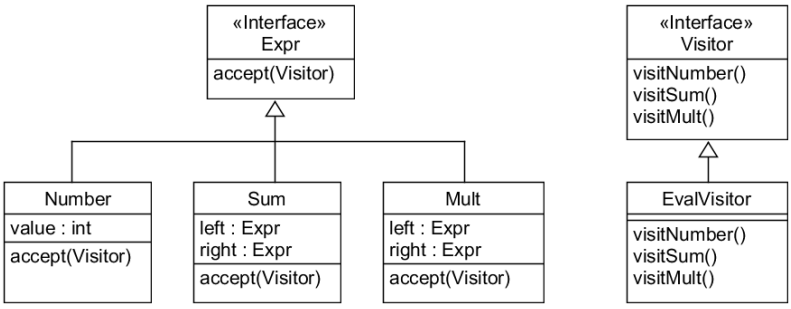

Figure 4: Visitor-based implementation

(d) Write a simple test driver that tests each of the three implementations by setting up and evaluating the sample expression shown in Figure 1. All the three implementation variants should yield the same result.

(e) We now want to extend our arithmetic expressions (i) by adding a new `print` operation that converts an expression tree to a simple String value, and (ii) by adding a new kind of expression supporting subtraction (`Minus`). Your task is to implement these extensions for all the tree implementation variants. Observe which parts of your implementation need to be changed and which ones are stable. Is there a difference among the three variants?