Sequence 2.2 – Traversing the tree

P. de Oliveira Castro S. Tardieu

What does a tree contain?

- A *tree* is made of *nodes* and one of them is the *root* of the tree.
- Those nodes may be of different types.
- Every type of node has a specific set of attributes.
- Those attributes may be other trees.

For example, a binary addition can be represented as a `BinaryOperation` node, with an attribute `operation` containing "+", and two subtrees representing the left hand side and the right hand side operands.

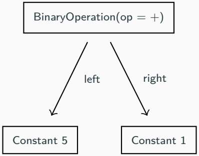

Figure 1: AST for $5 + 1$

More complex expressions

- Any expression can be represented as a tree.

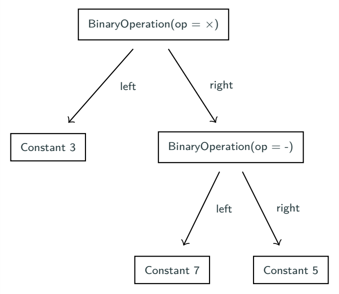

Figure 2: AST for $3\times(7 - 5)$

- Parenthesized expressions are naturally represented using a sub-tree, which expresses the intended grouping.

Evaluating an expression

We can start with simple rules for evaluating an expression:

- A constant evaluates as itself (Constant 3 gives a result of 3).
- A binary operator
  - recursively evaluates its left branch;
  - recursively evaluates its right branch;
  - applies the operator (*e.g.*, $\times$) to the two results obtained above.

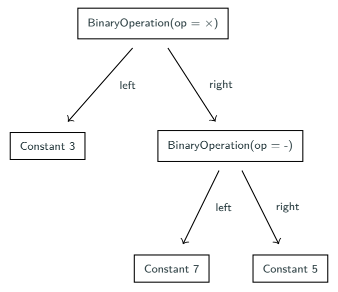

Figure 3: AST for $3\times(7 - 5)$

Example of evaluation

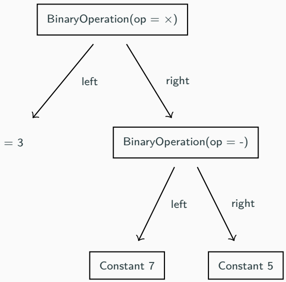

Figure 4: Evaluating the left branch of $\times$

Example of evaluation (ctd)

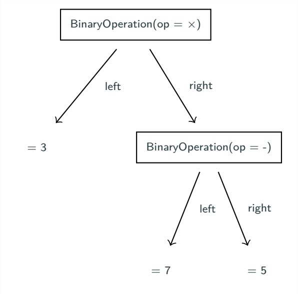

Figure 5: Evaluating the branches of -

Example of evaluation (ctd)

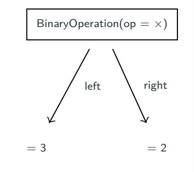

Figure 6: Evaluating -

$= 6$

Figure 7: Evaluating $\times$

Printing an expression

- Let's assume that we now want to print a representation of a binary operation, instead of evaluating it. We can adopt a similar methodology.
- A constant prints its value (Constant 3 prints as 3).
- A binary operation:
  - prints an opening parenthesis "(";
  - prints its left operand by calling this procedure recursively;
  - prints the operator;
  - prints its right operand by calling this procedure recursively;
  - prints a closing parenthesis ")".
- Some parentheses may be superfluous, but they guarantee that the operator priority ($+$ *vs* $\times$ for example) does not need to be considered.

Example of printing

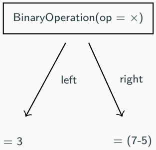

Figure 8: Printing 3 and (7-5)

$= (3\times(7\text{-}5))$

Figure 9: Printing $3\times(7\text{-}5)$

The visitor pattern

- The process for evaluating the value of a tree or for printing it use the same pattern: subtrees are processed recursively using the same methodology.
- It is possible to use C++ dispatching capabilities to use a common pattern (the *visitor* design pattern) to implement this kind of tree traversal.
- A visitor is an object with methods for acting on the various kind of nodes of the visited tree.
- The visitor can maintain internal data in order to perform its job. For example, a pretty-printer may keep track of the current level of indentation it is using to render a tree as source code.

On the tree side

- Every node that can be visited accepts a visitor through a method `accept()`. It then calls the visitor method corresponding to its type through the visitor `visit()` method.

```cpp
class BinaryOperator : public Expression {
  ...
  virtual void accept(Visitor &v) {
    // Call the visitor method named `visit()`
    // taking a BinaryOperator as argument.
    // This works with any object inheriting
    // from the Visitor class.
    v.visit(*this);
  }
  ...
}
```

On the visitor side

- Every node that can be visited accepts a visitor through a method `accept()`. It then calls the visitor method corresponding to its type through the visitor `visit()` method.

```
class PrintingVisitor : public Visitor {
  ...
  void visit(BinaryOperator &o) {
    std::cout << '(';        // Print opening parenthesis
    o.left.accept(*this);   // Print left operand
    std::cout << o.op;       // Print operator
    o.right.accept(*this);  // Print right operand
    std::cout << ')';        // Print closing parenthesis
  }
  ...
}
```

Why the double indirection?

- C++ virtual methods only dispatch on the receiver object (the object on which the method is called). Here the receiver is the tree node.
- By going through the right node using the virtual method `accept`, the right visitor method `visit()` is selected by the compiler from within each `accept()` method.

```
// (inside BinaryOperator's visit) Call the Node accept method
o.left.accept(*this); // Dispatch on Node node kind
...
// (inside Node's accept) Call the Visitor visit method
v.visit(*this); // Dispatch on Visitor kind (Printer or Evaluator)
...
```

Conclusion

- The abstract syntactic tree (AST) representing the program can be easily traversed for various purpose (printing, evaluating, etc.).
- Traversal of the tree can be implemented using the visitor pattern.
- A visitor achieves one goal, and may keep internal data representing the current state of the traversal (such as the current indentation level to use when pretty-printing a subtree).