COMP 442/6421 Compiler Design

Instructor: Dr. Joey Paquet paquet@cse.concordia.ca
TA: Zachary Lapointe zachary.lapointe@mail.Concordia.ca

LAB 5 – PROCESSING THE ABSTRACT SYNTAX TREE: THE VISITOR PATTERN

Abstract Syntax Tree

- A rich structure which represents the meaning of the program
  - It is the primary artifact used for semantic analysis (A3) and code generation (A4)
  - Persistent: can be traversed any number of times
- It represents a program which obeys lexical and grammatical specifications
  - Semantic specifications are next

Abstract Syntax Tree

- We want to be able to traverse the syntax tree in multiple phases, performing different sets of actions on the nodes each time

- Action will depend on both the type of node, and the phase we are in.
  - **Double dispatch**: Polymorphism, depending on the runtime type of two objects
  - In OOP languages, this can be achieved with the *visitor* pattern

The Visitor Pattern – Why

- Primarily used here to cleanly achieve **double dispatch**, in static OO languages
  - Action taken depends on which phase we are in (type checking, code generation, etc.) and which node we are in (type declaration, assignment, etc.)

- Pros
  - Clean and organized: double dispatch is possible without the pattern, but messy and error prone
  - Pattern centralizes behaviour by phase (each phase, or visitor, gets its own class)
  - Adding new phases does not require modifying existing phases
  - Code for actions not inside the nodes of AST
  - Versatile pattern, high potential to customize when implementing

- Cons
  - Somewhat complex to first understand
  - Requires a fair amount of boilerplate code
  - Kind of a *hack*

The Visitor Pattern – How

- Two functions:
  - **`Visit(ConcreteNodeType)`**
    - Uses polymorphism (visitor class) and overloading (concrete node type)
  - **`Accept(AbstractVisitor)`**
    - Makes the above overloading polymorphic.
    - Boilerplate code in every node type

- Demo

The Visitor Pattern – Variations

- Traversal
  - We want the visitor to visit an entire tree
    - Implement directly in visitor
    - Iterator
- Parent class function type
  - Abstract
  - Empty implementation
    - Pitfall
- How many visitors?
  - Adding new visitors is easy
  - A3 requires at least 2 (semantic analysis in 2 passes)
  - Edge cases might warrant their own visitor (pre-processing, propagating information through AST, etc.)

AST traversal

- Traversal
  - It may be worth differentiating *pre, in and post* order visits to nodes
    - Maintaining symbol table scope during traversal
    - Code generation for control structures
    - *in* may not be required
  - Euler Tour
- Traversal implementation
  - Built into Visitor
  - Separate Iterator connects visitor and AST nodes
- Euler tour algorithm
  - Using DFS

Euler Tour – Using DFS

Starting at root . . .

```
pre-visit action
for each child
    recurse into child
    in-order-visit action
post-visit action
```