Object-Oriented Programming

CSCI-UA 0470-001
Class 4
Instructor: Randy Shepherd

Project Intro

Project Intro

- Translator from a subset of Java to a subset of C++
- Primary tool in this effort is a tool called Xtc, it has many features, you need to learn how to use them.
- Also, a supporting toolchain with Sbt, Junit, Git, etc.
- We will look closer at Xtc and the toolchain in next week

Source Language

- The source language is the language of programs serving as inputs to the translator; it is a restricted version of Java.

- Java *without* nested classes, anonymous classes, interfaces, enums, annotations, generics, the enhanced for loop, varargs, boxing/unboxing, abstract classes, synchronized methods and statements, strictfp, transient, volatile, lambdas, etc.

- Whats left?
  - Primarily interested in modeling basic translation and dynamic dispatch at first
  - Therefore omit method overloading but not method overriding for the first version
  - Other basic language features, static methods, packages, etc.

- You will be provided with test inputs to test your translator that exercise the features of source language.

- You can assume that source programs are free of compile-time errors.

Source Language

- Java includes an extensive set of platform libraries, with the facilities in the java.lang package

- For our translator, we only support..
  - the hashCode(), equals(), and toString() methods in java.lang.Object (C++ code provided)

  - printing of numbers and strings through out.print() and out.println() in java.lang.System.

- Furthermore, we do not support the dynamic loading of Java classes, but rather translate all Java classes of the source program into one C++ program.

Target Language

- The target language is the language of outputs of the translator; it is a restricted version of C++

- C++ without virtual methods, inheritance, templates, lambda abstractions, auto, decltype, etc

- Whats left?

  - basic classes, exceptions and namespaces

  - “C with classes”

  - For the first version of the translator, you can ignore memory management

Translator Language

- Java 1.7
- You should make extensive use of the OO structures available in Java. (Abstract Classes, Interfaces, etc.)
- You should leverage good OOP design principles. (Encapsulation, Polymorphism, Inheritance)
- You should leverage design patterns discussed in class. (Chain of Responsibility, Decorator, etc.)
- You should make use of all the libraries available to you. (Xtc, Junit, Logback, etc.)
- You should use good software engineering practices. (TDD, Pair Programming, Code Reviews, etc.)

How do we start?

- CRC cards exercise, at the end of class today.
- We also need some testing strategies:
  - Java programs that are test inputs and use translator.
  - Compile and run C++ output; then compare.
  - Write unit tests for small pieces of the translator
- We need to decompose the project into subproblems!

Phased Approach

- The translator will have several ‘phases’.
- Incremental steps towards goal.
- Allows for “divide and conquer” approach in your team.
- Phases can be worked on independently, to an extent.
- These are our “subproblems”.
- Additional phases will be added in the second half of semester.
- Before we can talk about what the phases are we need to understand an important concept….

Source Representation

- Compilers first create a ‘parse tree’ aka ‘concrete syntax tree’.
- A tree that represents the actual source code (expressions, statements, etc.).
- Contains the ‘tokens’ of the language, tied closely to the ‘grammar’ of the language.
- For us, what is important for translation are the *semantic constructs* of the language. Not the syntax itself.
- Concrete Syntax / Parse trees not right choice for our translator….

Abstract Syntax Tree

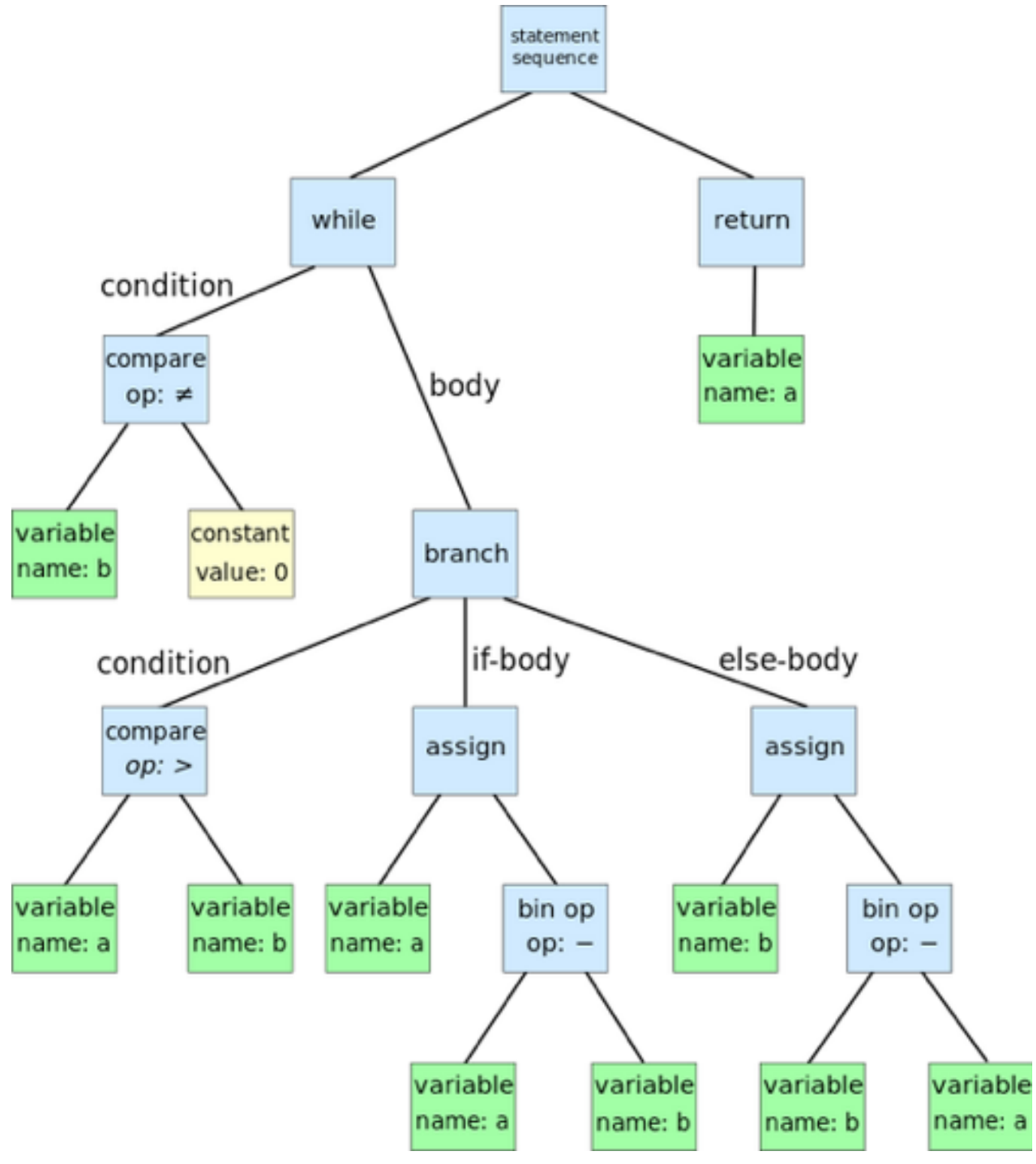
- An abstract syntax tree (AST) is a tree representation of abstract syntactic structure of a program.

- Each *node* of the tree denotes a construct occurring in the source code.

- The syntax is "abstract" in not representing every detail appearing in the real syntax.

Example

```
while b ≠ 0
    if a > b
        a := a – b
    else
        b := b – a
return a
```



Why Abstract Syntax Tree?

- An AST can be edited and enhanced with information on each node. Such editing is impossible with the source code of a program, since it would imply changing it.

- An AST usually contains extra information about the program, due to the *consecutive phases of analysis.*

- Traversing an AST is done many times during translation. Making that process convenient is important, so it often makes sense to use the Visitor Pattern.

- We'll cover the Visitor Pattern in this class.

Abstract Syntax Tree Types

- **Statically-typed AST**
  - You define types that map to nodes of the AST.
  - Figuring out how to map a source node type to a target node type is the translation approach.
  - You can have all phases in one class.
  - Data kinds are fixed with this approach but there will be more phases in the second half. Trickier to evolve elegantly.
- **Dynamically-typed AST**
  - All nodes of the same type (xtc.lang.GNode)
  - Trade-off: Flexibility for safety. Normally I would vote for safety, but we need to move quickly.
  - Enables a class per phase, much easier to add a new phase in the second half.
- *Xtc supports both, but we will write our translator with the dynamically-typed AST.*

Lets take a look at a Java Ast in Xtc…

Project Phases

Phase 1
Load all sources as Java AST

- Input: Source Java File
- Steps:
    - Load source file, generate Java AST.
    - Load dependency classes that are in scope, eg. other classes in package and imports
    - Generate Java AST for dependencies
- Output: Set of ASTs representing all Java classes to be translated.
- (Xtc and JavaFiveImportParser do most of the work for this phase)

Phase 2
Generate AST for inheritance hierarchy

- Input: Set of output nodes from phase 1
- Steps:
    - Traverse set of nodes from step 1
        - For each node, build an ast that represents the data layout and vtable
        - layout that will be in the C++ header file.
- Output: A set of nodes representing the vtable and data layout
- (You will design the AST schema before writing the code for this.)

Phase 3
Write C++ header with inheritance hierarchy

- Input: Set of output nodes from phase 2

- Steps:
    - Traverse set of output nodes from step 2

    - For each node, print into the C++ header file that contains all the class definitions.

- Output: No output. However, the output.h file is on disk and complete at this point.

Phase 4
Mutate/Decorate Java Ast to C++ Ast

- Input: Set of output nodes from phase 1
- Steps:
    - Traverse set of nodes from step.
        - For each node, mutate it and translate to a node representing C++ code.
        - You will need to mutate and generate additional nodes.
- Output: Set of nodes representing the C++ implementation code to be printed

Phase 5
Write C++ implementation files

- Input: Set of output nodes from phase 4
- Steps:
  - Traverse set of nodes from phase 4
  - For each node, print into the C++ implementation file that contains all the class definitions.
- Output: No output. However, the output.cc file is on disk and complete at this point.