Decomposing and Testing Your Project

Principles for Modular Decomposition
Design for Test
Incremental Construction and Testing

Goals of Decomposition

- Incremental development (risk reduction)
- Parallel (team) development
- Evolution
  - (may or may not apply to CS461)

Incremental Development

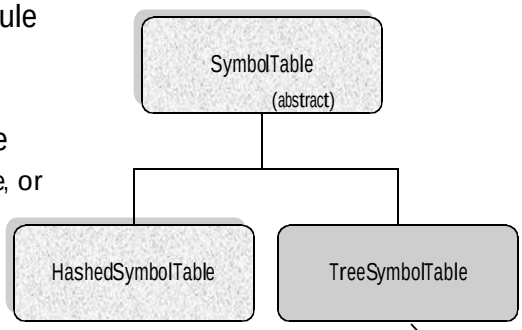
- Must have a well-defined build order
  - With frequent checkpoints — test as you go
  - Chunks of 1 day to 1 week (less for a class project)
- Increments may be by adding *modules* or by adding *features*
  - Adding modules is generally better, but not always practical
  - Example: parts (a) and (b) of assignment

Team Development

- Minimize required communication
  - Simple, precise interfaces, particularly between parts written by different programmers
- Minimize integration difficulties
  - Through precise, complete interface specifications
    - especially interfaces between *people*, or between different *phases* of the project
  - Through independent unit testing
    - code should be tested before it is given to teammates
  - By allowing frequent incremental builds
    - the team should "build and smoke" on a regular basis

Information Hiding

- The purpose of a module is to *hide* a *secret*
  - Hide what may change
- Example: Symbol table
  - Secret: Is it a hash table, or a search tree?
  - Provide a compatible interface
  - Minimize and localize dependence on specific features



The Tree implementation may have an additional "merge" operation; its use should be localized.

Modularizing Type Checking?

- The "syntax separate from interpretation" decomposition has already been forced on you
  - as well as combining type-checking with intermediate code generation
- Parts (a) and (b) of assignment are not natural modules
  - we don't get from (a) to (b) by adding modules, or replacing some entirely
- The procedural breakdown transVar, transExp, etc. is probably as good as you can do

Design for Test

- Build Plan: Testable subsystems
  - Frequent incremental builds with observable behavior
  - Scaffolding as part of the plan and product
  - May be arranged to test riskiest parts first
- Checkable interfaces
  - Particularly between individual developers and teams
  - May involve adding or "moving" interfaces
    - ex: text I/O of critical data structures
    - ex: scriptable abstraction below GUI

Why Unit Test First?

Testability = Controllability + Observability + Partitioning

- Controllability & Observability
  - It is (usually) easier to drive an individual unit through all interesting behaviors, and observe the results, at the unit level
  - Sometimes controllability and observability can be achieved in context; is this the case for the type checker? How?
- Partitioning
  - It is much, much easier to diagnose and fix a fault in the scope of a small unit that *you just wrote*

Automate the Testing

- Driver: Make re-running the tests mechanical
  - At the least: capture your test cases for replay
- Oracle: Make judging the tests mechanical
  - Especially for re-running old tests after each addition to your type checker
  - Approaches:
    - Recording expected outputs; may require special options to the application (observability)
    - Assertions (structural invariants)

Choosing Test Cases

- Devise tests during design and coding, not after
- Devise specification-based tests, then implementation-based tests
  - Example: First tests of "symbol table," then extra tests for "hashed symbol table" (like what?)
  - Every kind of expression, and each case (INCLUDING ERRORS) for evaluation
- Emphasize boundary conditions

Example …

```
ExpTy transExp(Absyn.OpExp e) {  …
  if (oper == Absyn.OpExp.PLUS || …. {    …
    return new ExpTy(null, INT);
  }
  else if (oper == Absyn.OpExp.LT ||… ) {    …
       if (left_type.coerceTo(right_type) || … {
          if (…) {   ;  }
          else {
            String msg = …
            error(e.pos, msg);
          }
   }
    else { …  (etc.)
```

Ideally, you should start with an independent case analysis for testing (why?).

It is essential that each test case include expected outcome.

It is VERY helpful if you can run all tests, and check outcomes, mechanically.

Incremental Build & Test

- Establish a "build plan"
  - An order, and schedule, for adding pieces to your project
    - Ordered so that each addition can be tested before moving to the next
- Increments < 1 week
  - Including regular build-and-smoke points with integration of all parts
- Use version control system and/or process to maintain recoverability
  - Roll back from any disaster