Project information

David Notkin
Autumn Quarter 2008

Finally posted

- Parser: due October 27
  - Must be submitted on time and substantially complete – not graded, but commented upon
  - Late or not substantially complete submissions will be recorded
- Semantic analyzer: due November 10
  - Both parser and semantic analyzer will be graded at this point
- All project information in this slide deck is on the project web pages

Other dates

- Today: office hours only until 2PM
- This Wednesday (10/23): mid-term review
- This Friday (10/25): mid-term
- Tuesday November 4: Election Day
- Friday November 7: no lecture, project focus
- Monday November 10 & Wednesday November 12: guest lecturers

Project B: Extend MiniJava's syntax

- double is a legal (base) type
- A floating-point literal constant is a legal expression
- An or expression (using the || infix operator) is a legal expression
- if statements do not require else clauses
- For loops of the restricted form for (*i* = *expr1*; *expr2*; *i* = *expr3*)*stmt* are allowed, where *expr1*, *expr2*, and *expr3* are arbitrary expressions,*i* is an arbitrary variable (but which has to be the same variable in both the initialization and increment clauses), and *stmt* is an arbitrary statement.
- break statements are allowed.

Arrays

- An array of a base type, e.g., int[], boolean[][][], and in general *type*[] where *type* is an arbitrary base type, is a legal (base) type.
  - Base types are ints, booleans, doubles, and arrays of base types.
  - Only class types are not base types; this restriction is included only because otherwise the language becomes too hard to parse!
- A one-level array allocation is a legal expression, e.g., new int[10], new boolean[20][][], and in general new *type*[*expr*]*dims*
  - where *type* is an arbitrary non-array base type
  - *expr* is an arbitrary expression
  - *dims* is a possibly-empty sequence of []'s.

Arrays con't

- An array dereference, e.g., *a*[*i*], *b*[*i*][*j*][*k*], and in general*expr1*[*expr2*] where *expr1* is an arbitrary atomic expression and *expr2* is an arbitrary expression, is a legal expression.
- An array dereference is also legal on the left-hand side of an assignment statement. (Atomic expressions EXclude unary and binary operator expressions and array allocation expressions.)
- An array length expression, e.g., *a*.length and in general *expr*.length where *expr* is an arbitrary atomic expression
  - length is a reserved word in MiniJava (unlike Java).

Static class variable

- A class variable declaration may be preceded by the static reserved word to declare a static class variable

Precedence/associativity

- You should follow the precedence and associativity rules of regular Java for these extensions.
  - It's OK to use CUP's predecence declarations to achieve this.
  - It's OK to have one shift/reduce conflict in your CUP grammar, for the "dangling else" problem.
    - Add the "-expect 1" option before the minijava.cup argument in the Makefile to build Parser/parser.java if you decide to accept this shift/reduce conflict.
- You should add new AST classes and/or modify existing AST classes so that you can represent the new MiniJava constructs.
  - You should define the appropriate toString operations on these classes so that they can be pretty-printed in a form that is syntactically legal and produces the same AST if it is parsed again.
  - The other operations required of AST nodes, e.g. typechecking, evaluating, and lowering, you should implement by throwing UnimplementedError exceptions.

Project C: MiniJava typechecking

- Extend ResolvedType hierarchy to support the double type
- Extend ResolvedType hierarchy to support the array type constructor, which stores its element type
  - The array type constructor follows structural type equivalence rules
  - MiniJava restricts Java by defining one array type to be a subtype of another array type only when the two array types are equivalent.
- Extend the VarInterface hierarchy to support static class variable declarations

Implement typechecking for new and/or modified AST node classes

- Allow static class variables to be declared, so that they may be legally referenced in variable reads and assignments.
- Allow if statements to omit their else clause.
- Check that a for statement's loop index variable was previously declared to be an int, that its initialization and update expressions return ints, and that its test expression returns a boolean.
- Check that a break statement only appears in the body of a while or for loop. (You may change the interface of the Stmt.typecheck operation to do this.)
- Check that an or (||) expression has boolean operands.
- Allow ints to be assignable to doubles, including in regular assignments, in array assignments, in parameter passing into a method, and in returning from a method.
- Allow the +, -, *, /, <, <=, >=, >, ==, and != operations to also be applied to doubles, and, for binary operations, to mixes of ints and doubles.
- Allow the System.out.println operation to also be applied to a double.

Arrays

- Check that an array new expression has a size subexpression of type int.
- Check that an array length expression has an array subexpression that's an array.
- Check that an array lookup expression has an array subexpression that's an array and an index subexpression that's an int.
- Check that an array assignment statement has an array subexpression that's an array, an index subexpression that's an int, and a right-hand-side expression whose type is assignable to the array's element type.

Design

- What goes in the scanner vs. what goes in the parser?
- How to decide?

Possible answers include…

- Cohesion – why are elements placed together into components?
  - "component" is intentionally pretty vague here, and could include packages, classes, modules, etc.
- Coupling – what are the interconnections and dependences between components (and why)?
- Anticipating change – what are likely changes and how will they be accommodated?
- Simplicity – see Hoare's quotation, next slide
- Conceptual integrity – is there a consistent approach to existing decisions?
- … others?

Hoare sez

- "There are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies, and the other way is to make it so complicated that there are no *obvious* deficiencies. The first method is far more difficult."

Software structure degrades

- There is plenty of evidence that software structure degrades over time
- That is, well-planned and well-designed software systems become increasingly tangled over time
  - Less simple, less clear cohesion, more muddled coupling, harder to change, etc.
- One reason for this is that programmers often change code in a way that is locally sensible but has poor global and long-term consequences
- Reducing the rate of increase in entropy generally demands more global knowledge of the software

MiniJava

- As much as possible, respect the existing design – that is, try to maintain its conceptual integrity
- At least two reasons
  - Chambers, who wrote it originally, is a top-notch designer and programmer
  - You will end up with fewer unexpected interactions and problems

Software testing

- What are possible goals of software testing?

Dijkstra

- "Testing can only be used to show the presence of bugs, not their absence."

What are alternatives to these goals?

- Formal verification of the software
  - Verification vs. validation: Building the system right vs. building the right system [Boehm]
- Inspections, reviews, walkthroughs
- Certifying the process (e.g., ISO9000)
- Certifying the practitioners (e.g., licensing doctors)
- …

A broad-brush of some testing issues

- White-box vs. black-box testing
  - Can see the code, can't see the code
- Functional vs. performance vs. stress vs. acceptance vs. beta vs. … testing
- Structural coverage testing

Some terminology

- A *failure* occurs when a program doesn't satisfy its specification
- A *fault* occurs when a program's internal state is inconsistent with what is expected (this is usually an informal notion)
- A *defect* is the code that leads to a fault (and perhaps a failure)
- An *error* is the mistake the programmer made in creating the defect

A simple problem

- The program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is isosceles, equilateral or scalene.
- Write a set of test cases that would adequately test this program

A study showed…

- 13 kinds of defects were found in actual programs
- Experienced programmers on average write test cases that identify about half of the defects

The lucky thirteen

- Valid scalene
- Valid equilateral
- Valid isosceles
- All permutations that represent valid scalene
- One side is zero
- One side is negative
- All sides are zero
- Three positive integers where two sum to the third
- All permutations of the previous case
- Three positive integers where two sum to less than the third
- All permutations of this
- A non-integer side
- An incorrect number of inputs

Bach adds…

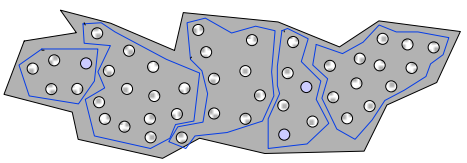
- A GUI that accepts the three inputs
- Asks his students to "try long inputs"
- Interesting lengths
  - 16 digits+: loss of mathematical precision
  - 23+: can't see all of the input
  - 310+: input not understood as a number
  - 1000+: exponentially increasing freeze when navigating to the end of the field by pressing <END>
  - 23,829+: all text in field turns white
  - 2,400,000: reproducible crash
- The programmer was only aware of the first two boundaries

"What stops testers from trying longer inputs?"

- Bach suggests
  - Seduced by what's visible
  - Think they need the specification to tell them the maximum – and if they have one, stop there
  - Satisfied by first boundary
  - Use linear lengthening strategy
  - Think "no one would do that"
  - …

Partition testing

- Basic idea: divide program input space into (quasi-)equivalence classes, selecting at least one test case from each class



Structural coverage testing

- Premise: if significant parts of the program structure are not tested, testing is surely inadequate
- Control flow coverage criteria
  - Statement (node, basic block) coverage
  - Branch (edge) and condition coverage
  - Data flow (syntactic dependency) coverage
  - Others…
- Attempted compromise between the impossible and the inadequate

Statement coverage

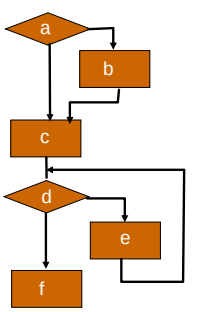
- What's a statement?
  - `max = (x > y) ? x : y;`
  - Using basic blocks can help this issue
- Obviously unsatisfying in trivial cases (such as the second example on the right, from Ghezzi)

```
if x > y then
   max := x
else
   max :=y
endif
```

```
if x < 0 then
   x := -x
endif
z := x;
```


Edge coverage

- Uses control flow graph
  - We'll see these soon!
  - Essentially a flowchart
- Covering all basic blocks (nodes) would not require edge **ac** to be covered
- Edge coverage requires all control flow graph edges to be coverage by at least one test



Condition coverage

- How to handle compound conditions?
  - `if (p != NULL) && (p->left < p->right) …`
- Is this a single conditional in the CFG?
- How do you handle short-circuit conditionals?
  - `andthen, orelse ...`
- Condition coverage treats these as separate conditions and requires tests that handle all combinations

Path coverage

- Edge coverage is in some sense very static
- Edges can be covered without covering actual paths (sequences of edges) that the program may execute
- Note that not all paths in a program are always executable
  - Writing tests for these is hard ☺
  - Not shipping a program until these paths are executed does not provide a competitive advantage ☺
- Loops (or recursion) makes life even harder

Summary

- Software testing – and only parts were covered at the lightest imaginable level – is a complex art
- But you need to be able to wear two hats – that of the developer, and that of the tester – and this is extremely hard
- These ideas may give you some more disciplined way to think about your testing process, informal though it will be