Quiz 1 Review

Bryan, Esha, Sunila, Vlad

© 2013 6.005 Staff
State Machine

- A set of states
- Rules for transition between states
State Machine

Does the string have an even number of a’s?
State Machine

• Formally, a state machine consists of
  • A set of states, $\text{State} = \{\ldots\}$
  • A set of initial states, $\text{Init} = \{\ldots\}$
  • A set of events, $\text{Event} = \{\ldots\}$
  • A transition relation $\text{trans} \subseteq \text{State} \times \text{Event} \times \text{State}$
  • A set of traces (derived from $\text{trans}$ and $\text{Init}$)
State Machine Example: Stack

- A set of states, State = {EMPTY, NONEMPTY}
- A set of initial states, Init = {EMPTY}
- A set of events, Event = {push, pop, size}
- A transition relation trans ⊆ State × Event × State
  \{ (EMPTY size EMPTY), (EMPTY push NONEMPTY), (NONEMPTY pop EMPTY), (NONEMPTY push NONEMPTY), (NONEMPTY pop NONEMPTY), (NONEMPTY size NONEMPTY) \}
- A set of traces
Grammars

- Grammars define a set of sentences that are considered valid sequences of symbols/tokens (aka “terminals”)

Structure:

- A grammar is a set of productions
- Each production defines a non-terminal
- A non-terminal is a variable that stands for a set of sub-sentences
Grammars

Backus-Naur Form (BNF)

• non-terminal ::= expression of terminals, non-terminals, and operators
• Non-terminals are capitalized, terminals are lower-case

Basic Operators:

• Sequence: A ::= B C
  ○ i.e. A is a B followed by C
• Iteration: A ::= B*
  ○ i.e. A is zero or more B’s
• Choice: A ::= B | C
  i.e. A is either B or C
Grammars

Backus-Naur Form (BNF)

- non-terminal ::= expression of terminals, non-terminals, and operators
- Non-terminals are capitalized, terminals are lower-case

Additional Operators:

- Grouping: A ::= (B C)*
  - A is zero or more B C pairs
- 1+ iteration: A ::= B+
  - Same as A ::= BB*
- Character classes
  - A ::= [abc] is the same as A ::= a | b | c
- A ::= [^b] is the same as A ::= a | c | d | e |...
Grammars

Example:

```
Markdown ::= ( Normal | Italic ) *
Italic ::= _ Text _
Normal ::= Text
Text ::= [^ _ ]*
```

Building blocks:

- Terminals: _, strings of text without _
- Non-terminals: Markdown, Italic, Normal, Text
Regular Grammars

Regular Grammars:

• Can be reduced to a single production by substituting every non-terminal (except the root) with its right hand side

Markdown ::= ([^_]* | _[^_]*_ ) *

This is called a regular expression.
Lexing & Parsing

Lexer:
- Transforms a stream of characters into higher-level symbols (tokens)

Parser:
- Interprets the stream of higher-level symbols
- Responsible for understanding the relationships between tokens as defined in the grammar
Recursive-Descent Parsing

1. Treat the parser as a state machine that incrementally consumes tokens from the lexer.
2. For a sequence of tokens, look at the beginning of the token stream to determine which case of the production applies.
3. Given the case, parse each of the terminals & non-terminals in order:
   - Terminals parsed by consuming the token
   - Parsing a non-terminal: make a recursive call to the associated function
Testing: Strategies

• What’s being tested?
  - Unit testing: individual module (method, class, interface)
  - Subsystem testing: entire subsystems
  - Integration, System, Acceptance testing: whole system

• How are inputs chosen?
  - Systematic
    - Partitioning large input space into a few representatives
      - Identify sets of input with the same behavior
      - Try one input from each set
    - Boundary testing: zero, min/max, empty string
Testing: Strategies

• How good is the test suite?
  o Coverage: how much of the specification or code is exercised by tests? all-statements, branches, paths?
• When is testing done
  o Test-first programming: tests are written first, before the code
  o Regression testing: a new test is added for every discovered bug, and tests are run after every change to the code
• Essential characteristics of test
  o Modularity: no dependence of test driver on internals of unit being tested
  Automation: must be able to run (and check results)
Black box Vs. Glass box

- Black box testing
  - Choosing test data only from spec, without looking at implementation
- Glass box (white box) testing
  - Choosing test data with knowledge of implementation
    - if implementation selects different algorithms depending on the input, should choose inputs that exercise all the algorithms
  - Good tests should be modular -- depending only on the spec, not on the implementation
Java: Inheritance/Dynamic Dispatch

Shape:

Is a

Polygon:
- int centerX()
- int centerY()
- int area()
- int circumference()
- int numSides()
- int sideLength()

Is a

Ellipse:
- int centerX()
- int centerY()
- int area()
- int circumference()
- int lengthX()
- int lengthY()
Power of Inheritance

- Can share common code
- Can extend functionality without cluttering the original
- Can override a method in the superclass in the subclass
  - Have an area method in Shape as well as Polygon and Ellipse
- When invoking the method, one in the Object’s class, NOT in the class of the type of the variable pointing to the Object, will be called
  - Dynamic Dispatch
  - Invoking area() will call the area in either Polygon or Ellipse depending on the type of the object
Overloading

- Can have multiple methods with the same name within a class
  - Differ in argument type and/or return type

- At compile time, the type signature of the method invocation is matched to select the method to call
  - Will upcast the arguments as needed to find a matching type signature
Java: Generics

• Parameterized types
  
```java
public interface List<E>{
  void add(E x);
  E get();
  Iterator<E> iterator();
}
```

```java
public interface Iterator<E>{
  E next();
  boolean hasNext();
}
```

• It can store any type of data in the List or can contain any elements of any type E

• Can be instantiated with an arbitrary type
  - List<Integer>, List<String>, List<Foo>
Java: Generics

List ls = new ArrayList(); // List should contain strings
ls.add("Hello");
ls.add(new Integer(5));
String hello = (String) ls.get(0);

List<String> ls = new ArrayList<String>();
ls.add("Hello");
ls.add(new Integer(5)); // This is a compile time error
String hello = ls.get(0);

• Benefits
  o Comments are replaced by statically checked type parameters
  o Compiler enforces type safety
  o No need for messy typecasts
Java: Exceptions

• Unchecked
  o Used for unexpected/catastrophic failures
  o E.g NullPointerException: “Thrown when an application attempts to use null in a case where an object is required.” [Java Platform Docs]
  o Compiler *does not* check that methods declare these
  o Compiler *does not* check that callers declare or catch unchecked exceptions

• Checked
  o Used for special results
  o Compiler requires method signature state that may throw checked exception
  o Caller must either also declare it throws that exception, or handle it
Specifications

• Why Specifications?

• Preconditions ("requires") -> Assumptions made by implementer -> @param

• Postconditions ("effects") -> What the implementer of the method promises to do -> @return

• Frame condition ("modifies") -> Identifies which objects may be modified -> @param
Judging Specifications

Declarative Specification

```java
static int findA (int [] a, int val) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] == val) return i;
    }
    return a.length;
}
```

Operational Specification

• requires: val occurs in a
• returns: iterates from the beginning of the list until it finds the first instance of val, and returns its index

• requires: val occurs in a
• returns: the index of the first occurrence of val in a
Judging Specifications

• A good specification should be **coherent**
  – Avoid long argument lists, deeply-nested if statements, random boolean flags, doing too many things, unnecessary side effects

• Return values should be **informative**
  – Ambiguous return values are more (scarily) common than you might think
  – Good design considers information users of your method need/want

• Specifications should be **strong enough**
  – Avoid ambiguities in what mutations occur, especially in exceptional cases
  – Checked exceptions don’t save you from partial mutations

• Specifications should be **weak enough**
  – Do not guarantee things that may not happen
  – Think ahead to exceptional cases, especially ones that are likely to occur (e.g. the path is wrong or no permission to read/write)

• Specifications should **take advantage of abstraction**
  – Use abstract types where possible
  – Increases utility of the method for users
  – Increases implementation freedom for implementers
Class Diagrams

Simple way to express class hierarchy relationships
Abstraction

- Modularity
- Encapsulation
- Separation of Concerns
- Information Hiding

- **Abstract Data Types**: Instead of thinking about how a type stores values, programmers only need to think about operations on the type
Classifying Types

• Mutable
  – Includes operations that alter the object such that other subsequent operations on the same object give different results
  – E.g. StringBuilder

• Immutable
  – Object is not designed to change once created
  – Often, new objects created instead of changing existing one
  – E.g. String
Classifying Operations on Types

• **Constructors** - Create new objects of a type (New ArrayList<String>(); for List)

• **Producers** - Create new objects from old objects of the same type (concat() in String)

• **Mutators** - Alter the object (add() in List)

• **Observers** - Take as input an object of the abstract type and return a different kind of object (size() in List or toString() in Object)
Classifying Operations on Types

• For an abstract type $T$ and other types $t$

\[\text{Name: Input Param(s) Type} \rightarrow \text{Output Param Type}\]
Constructor: $t^* \rightarrow T$
Producers: $T^+, t^* \rightarrow T$
Mutators: $T^+, t^* \rightarrow \text{void}$
Observers: $T^+, t^* \rightarrow t$

• Useful classification, but not perfect
  – In particular, some operations may be mutator +observer or mutator+producer
Designing Abstract Data Types

1. Choosing operations

2. Choosing representation

3. Choosing abstract value space for the specification

4. Choosing the representation value space for the implementation

5. Defining RI and AF (Deciding what representation values to use & how to interpret them)

• Rules of thumb
  – Better to have **few, simple operations, well-defined purpose**, which is **adequate** or can be combined into more complex operations that clients likely to want

  – If a type is generic it should contain **no domain-specific operations**. If a type is domain-specific, it should not have too general operations.
2. Choosing a Representation

- Actual data structures used internally to support operations on an abstract data type
  - In practice, the collection of fields of the Java object, where each field is a Java type (including of the same type as the ADT object)

- The use of an abstract type should be independent of its representation
  - Changes in representation should not impact any client code
  - Requires all operations to have fully-specified preconditions, post conditions, and frame conditions

- Representation exposure (bad)
  - Code outside of the class can modify the internal representation
3.4. Representation Values & Abstract Values

• Representation values
  – Values of the actual entities used to implement the ADT
  – Usually more than one object
  – To simplify, we will think about it as just a mathematical value

• Abstract values
  – Values that the ADT is designed to support
  – Do not actually exist but are how we want clients to view the elements of the ADT
5. Representation Invariants

• Invariant – property of a program or object that is always true

• How to establish invariants?
  – Must ensure the invariant is true when the object is constructed
  – All other operations on the object must preserve invariant
  – Also want to avoid representation exposure
5b. Rep Invariant and Abstraction Function

- Abstraction Function & Representation Invariant relate ADT and its representation

- **AF**: \( R \rightarrow A \)
- **RI**: \( R \rightarrow \text{boolean} \)
Interfaces

• Interfaces allow formalization of ADTs by specifying the set of operations needed
Recursive Data Types

• Recursive data types are ADTs that may reference instances of themselves

• Abstract syntax tree: an important recursive data type used represent parsing languages & performing operations over the parsed tree
Equality

- Implementation of equals() must satisfy the 3 properties
  
  - Reflexive: `a.equals(a)` for all non-null references a
  
  - Symmetric: `a.equals(b) ⇒ b.equals(a)`
  
  - Transitive: `a.equals(b) ∧ b.equals(c) ⇒ a.equals(c)`
Observational equality vs. behavioral equality

• Two expressions are equal when they cannot be distinguished by observation
• Check out String’s equal()

• If 2 objects are equal at one instance of time, they will stay equal at subsequent calls to equal().
• Check out StringBuilder’s equal()
## Two Views of Equality

<table>
<thead>
<tr>
<th></th>
<th>Referential Equality</th>
<th>Object Equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td><code>==</code></td>
<td><code>Equals()</code></td>
</tr>
<tr>
<td>Objective C</td>
<td><code>==</code></td>
<td><code>isEqual:</code></td>
</tr>
<tr>
<td>C#</td>
<td><code>==</code></td>
<td><code>Equals()</code></td>
</tr>
<tr>
<td>Python</td>
<td><code>is</code></td>
<td><code>==</code></td>
</tr>
<tr>
<td>Javascript</td>
<td><code>==</code></td>
<td><code>n/a</code></td>
</tr>
</tbody>
</table>
Design Patterns: Datatype Example

- Medieval persons, as in Game of Thrones
- *Person* interface; specifies *getName* and *
  getNameAndTitle*
- Classes that implement *Person*: *Peasant*, *Lord*, and *King*
  
  - *Lord* and *King* have "vassals," who are subordinate to them
Design Patterns: Datatype Usage

• Person vlad = new Peasant("Vlad");
• Person windy = new Peasant("Windy");
• george.getName(); // "Windy"
• Person lordStark = new Lord("Eddard Stark", windy, vlad);
• lordStark.getNameAndTitle(); // "Lord Eddard Stark"
• Person kingBob = new King("Robert Baratheon", lordStark);
• kingBob.getNameAndTitle(); // "King Robert Baratheon, Protector of the Realm"
Interpreter Pattern: subordinates

• Want to get array of names of subordinates of a Person
• Do not know at compile time if the Person is a Peasant, Lord, or King
• Implement a getSubNames method for each class, which uses getSubNames recursively!
• Nitty-gritty: method signature in interface; implement for each variant
Interpreter Usage Example

• `george.getSubNames();` // {"George"}
• `lordStark.getSubNames();` // calls `getSubNames` recursively on each vassal; {"Eddard Stark", "Windy", "Vlad"}
• `kingBob.getSubNames();` // calls `getSubNames` on `lordStark`, which recurses further. `result?`
Visitor Pattern

• Want to perform additional operations: e.g. get array of subordinates with official titles
• Don't want to or can't modify *Person* too many times or add methods to each variant
• Remove the algorithm from the data structure!
• Method with *if instanceof* statements? *No!* But good starting point.
Visitor Pattern, cont.

- Create Visitor classes, like `SubordinateArray` or `SubordinateTitleArray` to do our work: object as function
- Visit methods in each Visitor instance: where the computation takes place
  - Interpreter pattern: within each variant
Visitor Pattern, cont.

- Accept method of Person and of each variant: "calling" the appropriate handler, following the appropriate "if statement"
  - Interpreter pattern: calling `getSubNames` automatically calls appropriate handler due to dynamic dispatch
- Visitor interface: ensure that the Visitor is generic (has appropriate "if statements")
Visitor Pattern, visualized

The concrete types of the Element and Visitor objects have been "recovered". Perform the work appropriate for their pair of types.

(image credit: http://sourcemaking.com/design_patterns/visitor)
Growing a Visitor

1. Create Visitor interface, either nested or standalone; visit method for each variant

2. Create accept method for each variant which accepts a visitor and lets it "visit" the current variant

3. Create specific Visitor class that implements the Visitor interface; create visit methods that implement computation
git

• git config --global user.name "Ben Bitdiddle"
• git config --global user.email "benbitdiddle@mit.edu"
• git init ps7 #create new repository in ps7 subdirectory of current directory
• git add SATOracle.java
• git commit -m "implemented and added oracle for boolean satisfiability"
git, cont.

• git status #do we have any files we need to add? untracked changes?

• git log #get list of commits by various authors

• git diff #compare local changes with most recent commit

• git checkout SATOracle.java #get the freshest copy of SATOracle.java

• git checkout <hex string> SATOracle.java #get some older commit of SATOracle.java
git, on a remote computer

- git clone ssh://benbitdiddle@dialup.mit.edu:/mit/user/b/e/benbitdiddle/ps7
  #clone the remote git repository
- git pull  #pull changes from remote repository
- git push  #push local commits to remote repository
- git reset SATOracle2.java  #remove this uncommitted file from version control
- git rm SATOracle.java  #remove this file from version control; be sure to commit and push
git, preventing catastrophe

- git pull origin master #when things go wrong
- git annotate SATOracle.java #who did what?

- Branches