CS 312

Spring 2008

Lecture 28

Logic programming

Wrap-up

Declarative vs. imperative

- Imperative programming: tell computer how to change its state to accomplish a result
- **Declarative** programming: tell computer what you want computed, without specifying state changes
    - Avoids side effects, enables analysis and optimization
    - functional programming: give an expression equal to the desired result
    - **logic programming**: give a logical formula describing what should be true of the result
        - a simple version: database queries

Logic programming in Prolog

Programmer defines boolean-valued **predicates**

- Language figures out all ways to make predicates true.
- Example (syntax modified from Prolog)

```
parent(X,Y) <= father(X,Y).
parent(X,Y) <= mother(X,Y).
father(bob, alice).      i.e., <= true
?- parent(bob, X).
X = alice
?- parent(X, X).
No
sibling(X,Y) <= parent(Z,X), parent(Z,Y).
father(bob, charlie).
?- sibling(alice, X).
X = alice
X = charlie
```

Concatenating lists

- Goal: define predicate join(L1, L2, L3) meaning L1@L2 = L3.
- If T1 @ L2 = T3, then H1::T1@L2 = H1::T3. So:

```
join([], L2, L2).
join(H1::T1, L2, H1::T3) <= join(T1,L2,T3).
?- join([1,2,3], [4,5,6], X)
  X = [1,2,3,4,5,6]
?- join([1,X,3], 4::Y, [1,2,Z,W,5,6])
  X = 2, Y = [5,6], Z = 3, W = 4
?- join([1,X,X], [Y,Y], [X,X,Y,Y,Y])
```

What did we cover?

Goal: better software design and implementation

- New programming paradigms
  - higher-order functions, pattern matching, polymorphism, concurrency, ...
- Specifying functions and data abstractions
- Reasoning about correctness
  - using specifications, logic
- Reasoning about performance
  - asymptotic complexity, recurrences, amortized complexity, locality
- Important data structures and algorithms
  - balanced binary trees, hash tables, splay trees, B-trees, functional impls

Life after 312

- SML is fun and ML variants (SML, OCaml, Haskell) are used in some "real-world" apps.
- Functional *style* is useful in almost any language.
- Most course material is not specific to SML:
  - Specifications, AF, RI, logic and verification
  - Recurrences and complexity analysis
  - Data structures and algorithms
- What if you miss functional programming?

Simulating functions with objects

- First-class functions can be simulated with first-class objects.

  `val f: t->t'= fn(x:t)=>e` is similar to:

  ```
  class Fn {
      t' apply(t x) { return e; }
  }
  Fn f = new Fn();
  ```

  `f(x)` is translated to `f.apply(x)`
- Java nested classes can even mention variables from containing scope.
- C# supports first-class functions directly (delegates)

Pattern matching

- Pattern matching is not supported by object-oriented languages
- Problem: matching type T requires knowing exactly what T is.
  - Doesn't work with abstract types -- conflicts with data abstraction
  - Could not expose pattern matching in SML signatures
- Can we have a pattern-matching mechanism that works with objects and data abstraction?

JMatch: Java + pattern matching

- JMatch supports **predicate methods** with multiple **modes** capturing directions of computation

```
class List {
Object head; List tail;
List(Object h, List t) returns (h, t)
    (head = h & tail = t)
}
```

- **Forward mode**: creates an object. **Backward mode**: pattern matches, binds h and t:

```
switch (lst) {
      case List(1, List(Object x, List rest)):
        return List(x, f(rest))
        }
```

JMatch logic programming

- A limited form of logic programming!

```
List join(List x, List y) returns(x) returns(y) (
      x = List(hx, tx) &
      tr = join(tx, y) &
      result = List(hx, tr)
)

let List(1, List(2, null)) = join(prefix, List y);
... use y here ...
```

Rebalancing a red-black tree in JMatch

```
static RBNode balance(int color, int value,
                      RBTree left, RBTree right) {

  if (color == BLACK) {
     switch (value,left,right) {
        case int z, RBNode(RED,int y,
         RBNode(RED,int x,RBTree a,RBTree b),RBTree c),
         RBTree d:
        case z, RBNode(RED,x,a,RBNode(RED,y,b,c)),d:
        case x, c, RBNode(RED,z,RBNode(RED,y,a,b),d):
        case x, a, RBNode(RED,y,b,RBNode(RED,z,c,d)):
           return RBNode(RED,y,
              RBNode(BLACK,x,a,b),RBNode(BLACK,z,c,d));
     }
  }
  return RBNode(color, value, left, right);
}
```

Iteration

- Logic programming has iteration built in.

```
class RBNode implements IntCollection, Tree {
   RBTree left, right; int value; boolean color;
   boolean contains(int x) iterates(x) (
      x < value && left.contains(value) ||
      x = value ||
      x > value && right.contains(value)
   )
}
```

- Forward mode: usual BST lookup
- Backward mode: in-order tree traversal!

```
foreach (tree.contains(int x) & x < 10) {
   ... use x ...
}
```

The tree iterator in Java

```java
class TreeIterator  implements Iterator {

 Iterator subiterator;
 boolean hasNext;
 Object current;
 int state;
  // states:
  //   1. Iterating through left child.
  //   2. Just yielded current node value
  //   3. Iterating through right child

TreeIterator() {
  subiterator = RBTree.this.left.iterator();
  state = 1;
  preloadNext();
}

public boolean hasNext() {
  return hasNext;
}

public Object next() {
  if (!hasNext) throw new NoSuchElementException();
  Object ret = current;
```

```java
private void preloadNext() {
  loop: while (true) {
    switch (state) {
      case 1:
      case 3:
        hasNext = true;
        if (subiterator.hasNext()) {
          current = subiterator.next();
          return;
        } else {
          if (state == 1) {
            state = 2;
            current = RBTree.this.value;
            return;
          } else {
            hasNext = false;
            return;
          }
        }
      case 2:
        subiterator = RBTree.right.iterator();
        state = 3;
        continue loop;
    }
```

Conclusions

- Object-oriented languages are incorporating many functional programming language features (higher-order functions, polymorphism, lexical scoping...)
- Pattern matching may show up too!

Final exam

- May 13, 7-9:30pm, Phillips 203
- Open book
- Cumulative

Follow-on courses

- Complexity: CS 381
- Understanding programming paradigms and language features: CS 411, CS 611
- Language implementation: CS 412/413
- Algorithms and algorithm design: CS 482
- Logic: CS 486
- Think about participating in 312 (and in other courses) as a course consultant