CS 3110

Fall 2008

Lecture 27

Andrew Myers

Logic programming

Pattern matching for Java

Course 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).   i.e., ∀X.∀Y.father(X,Y)⇒parent(X,Y)
parent(X,Y) <= mother(X,Y).
father(bob, alice).           i.e., true⇒father(bob,alice)
# 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: a predicate join(L_1, L_2, L_3) meaning L_1@L_2 = L_3.
- L_1@L_2 = (H_1::T_1)@L_2 = H_1::(T_1@L_2)
- So L_1@L_2 = L_3 if L_3 = H_1::T_3 and T_1@L_2 =T_3

```
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])
```

Reversible computation!

Pattern matching

- Pattern matching is (limited) reversible computation
- Forward:

```
let lst = x::y::rest
```

- Backward:

```
match lst with x::y::rest -> ...
```

- But we can't pattern-match unless we know exactly how the data is represented.
  - Can't pattern-match on an abstract type.
  - This is why OO languages don't have pattern matching.

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)
}
```

i.e., o=List(h,t) ⇔ o.head=h && o.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 Node balance(int color, int value,
                    Tree left, Tree right) {

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

Iteration

- Logic programming has iteration built in.

```
class Node implements IntCollection, Tree {
   Tree left, right; int value; boolean color;

   boolean contains(int x) iterates(x) (
      x < value && left.contains(x) ||
      x = value ||
      x > value && right.contains(x)
   )
}
```

- 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

```
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 = Tree.this.left.iterator();
    state = 1;
    preloadNext();
  }

  public boolean hasNext() {
    return hasNext;
  }

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

```
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 = Tree.this.value;
            return;
          } else {
            hasNext = false;
            return;
          }
      case 2:
        subiterator = Tree.right.iterator();
        state = 3;
        continue loop;
  }
```

Comparing iterators

C# 2.0+

```
class Tree {
  Tree left, right;
  Elem value;
  public IEnumerator<Elem> elements() {
      foreach (Elem e in left.elements()) {
            yield return e;
      }
      yield return value;
      foreach (Elem e in right.elements()) {
            yield return e;
      }
}
```

Both languages have coroutine iterators (CLU)

JMatch

```
public Elem elements() iterates (result) {
    foreach (Elem e = left.elements()) {
        yield e;
    }
    yield value;
    foreach (Elem e = right.elements()) {
        yield e;
    }
}
```

or:

```
Elem elements() iterates (result) (
    result < value && left.contains(result) ||
    result = value ||
    result > value && right.contains(result))
```

Conclusions

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

What was 3110 about?

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

Final exam

- Dec 18, 2-4:30pm, Olin 155
- Open book
- Cumulative

Follow-on courses

- Complexity: CS 3810
- Understanding programming paradigms and language features: CS 4110, CS 6110
- Concurrency: CS 4410
- Language implementation: CS 4120/4121
- Algorithms and algorithm design: CS 4820
- Logic: CS 4860

Credits

- Instructor: Andrew Myers
- TAs: K. Vikram (PS3), Ed McTigue (PS2/6), Rick Ducott (PS4/6), Tanya Gupta (PS5)
- Consultants: Andrew Owens (PS1/5, test harness), Matt Pokrzywa (PS3/5), Dane Wallinga (PS4/5), David Kupiec (PS3), Jerzy Hausknecht (PS6), Matt Paff (PS2/6)

- Don't miss the tournament: Tuesday Dec. 9, Upson B17, 7:30pm.