The opposite of object-oriented programming

A brief overview of functional programming

What is an object?

- Objects have state
- Objects have identity
- Objects respond to requests
- We can view objects with different degrees of abstraction

Why are objects useful?

- They model reality...
- ...even if that reality doesn t exist

The trouble with objects

- They are potentially unruly
  - side effects
  - irrevocable actions
- Example: modeling a stock portfolio
  - easy to make economic conditions influence stock process and portfolio value
  - much harder to rerun the model under different assumptions about inflation rate

Why is it hard to rerun the model?

- We don t know everything that might have changed
- Therefore, we can t capture what changed and reset it to its old value

A random example

- The C library has two functions for random numbers

```
n = rand();
srand(k);
```

- These functions rely on the random seed as a side effect
- Functional style:

```
val (n, new_s) = rand(s)
```

Functional programming

- Computation should be expressed in terms of functions
- The result of a function depends only on its arguments
- Functions are first-class values
- Variables, if they exist at all, are second-class

Varieties of functional languages

- Languages that are purely functional
  - Example: Miranda, Haskell
  - Lazy evaluation is typical
  - Still have performance problems
- Hybrid functional languages
  - Example: Scheme, ML
  - Eager evaluation is typical
  - Surprisingly fast implementations exist

Getting by without variables

- You can bind a name to a value
  - similar to a function argument
  - you can t change the value, but you can rebind the same name later
- Iteration becomes difficult
  - you can use recursion instead
  - compiler sometimes translates recursion back into iteration

Example in ML

- Function to compute factorials

```
fun fact(0) = 1
  | fact(n) = n * fact(n-1)
```

- Alternative implementation

```
fun kfact(0, k) = k
  | kfact(n, k) =
        kfact(n-1, k*n)
fun fact(n) = kfact(n, 1)
```

- Why the difference?

Tail recursion optimization

- If a function `f` calls a function `g` and immediately returns the result of `g`, the generated code can destroy `f`'s local variables and return address before calling `g`
- In effect, the call becomes a jump
- The optimization applies to all tail calls, not just recursive ones

Tail recursion example

- In

```
fun fg(x) = f(g(x))
```

the call to `f` is a tail call, but the call to `g` is not

Value bindings

- Saying

  `val` *name* `=` *exp*

  binds *name* to the value of *exp*.
- It may look like it creates a variable, but it doesn t really:

```
val x = 42
fun f() = x
val x = 24
val y = f()      (* y = 42 *)
```

Local bindings

- It is often useful to compute temporary values that are used only to help compute another value.
- The syntax:

```
let  declaration1
     declaration2 ...
  in expr
end
```

Example of local bindings

```
fun hypot(x, y) =
    let val xsq = x * x
        val ysq = y * y
     in Math.sqrt(xsq + ysq)
    end
```

Another example of local bindings

```
fun fact(n) =
    let fun f(0, k) = k
          | f(x, k) =
                  f(x-1, x*k)
     in f(n, 1)
    end
```

Control abstraction

- Just as object-oriented languages make data abstraction easier, functional languages make control abstraction easier
- In other words, they offer easy ways of defining specialized control structures

An example of control abstraction

- ML does not have a `for` statement
- Suppose we wanted to define one; what would it do?
  - Set a variable to an initial value
  - Do a computation that depends on that value
  - Set the variable to a new value
  - Repeat until a condition is satisfied

Repeating a computation

- We can represent any computation by a function
- But what does it mean to repeat it?
  - A function always gives the same result if we give it the same arguments
  - Therefore we must give it different arguments for repetition to have meaning

An obvious, but bad approach

- Suppose we had a function called `for` to which we could give
  - a starting value for a variable
  - a way to compute the variable's next value from the current one
  - a way to test if we're done, and
  - a function to evaluate
- That's not enough, for a subtle reason

Implementing the bad approach

```
fun for(start, step, test, f) =
  if test(start)
  then (f(start);
    for(step(start), step, test, f))
  else ()
```

- This function actually compiles; what's wrong with it?

What's the problem?

- The result of a function is the outcome of the work that the function did
- Therefore, it rarely makes sense to throw away the result of a function call, either
- Instead, we should think of `f` as a way of transforming a state into another state

What must we do?

- Pass each result of `f` on to the next call
- Take an initial argument for `f` as our argument
- Yield the final result of `f` as our result

The revised function

```
fun for(start, step, test, f, x) =
 if test(start)
 then for(step(start), step, test,
          f, f(start, x))
 else x
```

- Example of use:

```
fun incr(n) = n+1
fun less100(n) = n<100
fun plus(m, n) = m+n
val x = for(0, incr, less100, plus, 0)
            (* x = 4950 *)
```

Nameless functions

- It is a nuisance to have to define functions such as `less100` merely to use them as arguments to other functions
- Therefore, we can write (`fn` *x*=>*y*) to mean "the function that takes argument *x* and yields the result *y* "
- Often called *lambda expressions*

Using lambda expressions

- We can rewrite our call to `for` as

```
for(0, fn x=>x+1, fn n=>n<100,
    fn (x, y)=>x+y, 0)
```

- We can do better, because
  - Instead of `fn (x, y)=>x+y`, we can write `op+`
  - We can write an auxiliary function for comparison

An auxiliary comparison function

- We want a function that checks if an argument is less than *n*
- We would like to fix *n* once and do the comparison repeatedly
- Here is how to do it:

```
fun less(n) =
    let fun f(x) = x<n
     in f
    end
```

Using the function

```
val less100 = less(100)
less100(42)     (* true *)
less100(123)    (* false *)
```

- An alternative definition:

```
fun less(n) = fn x => x<n
```

- So we can rewrite our call to `for`:

```
for(0, fn x => x + 1, less 100,
    op+, 0)
```

We have barely scratched the surface

- Functions that return functions are amazingly powerful
- Functional languages typically support interesting data structures as well
- On the other hand, functional programming requires a different state of mind
- It probably deserves more popularity

A last example

```
datatype expr = INT of int
  | UNARY of string * expr
  | BINARY of string * expr * expr
fun print(INT n) = Int.toString(n)
  | print(UNARY(s, e)) =
      "(" ^ s ^ print(e) ^ ")"
  | print(BINARY(s, e1, e2)) =
      "(" ^ print(e1) ^ s ^
            print(e2) ^ ")"
print(BINARY("*", INT 5, BINARY("+"
      UNARY("-", INT 3), INT 4)))
```

Remarks

- This example is obviously more compact than its C++ counterpart
- However
  - it fits ML particularly well
  - changing it to allow expressions to be mutable would be a nuisance
- Direct language comparisons are hard

Homework (due Monday)

- Get the use-counted expression trees from last lecture to work
- Implement `eval`
- Change the `print` function so that it prints only necessary parentheses
  - you will need a mapping between binary operators and precedence levels
  - add a precedence argument to `print`