What’s 51 about?

Programming isn’t hard.

Programming well is very hard.

We want you to write code that is:

- Reliable, efficient, readable, testable, provable, maintainable… beautiful!

Expand your problem-solving skills:

- Recognize problems & map them onto the right languages, abstractions, & algorithms.
Prime Directive

Good programmers are lazy.

  – Never write the same code twice.
  – Reuse libraries.
  – Keep interfaces small & simple.
  – Pick a language that makes writing & maintaining the code easy.
Language & Code

• Language & abstractions matter.
  – Try formulating an algorithm to multiply Roman numerals.

• Often, don’t have the luxury of choosing the language.
  – We can still conceptualize & prototype using the right language abstractions.
  – If we understand relationships between linguistic abstractions, we can realize the code in any language.
Better Example: Red-Black Trees

- A particular kind of balanced search tree [Guibas & Sedgewick 1978].
Key Invariants:

1. No red node has a red child
2. Every path from root has same number of black nodes
Must Rebalance

```
7
/    \
4      11
|      /
1      5      15
|  |  |  |
0 3 12 17
```

25
Must Rebalance
**ML Code for Insert**

```ml
fun balance ((Blk, T(Red, T(Red, a, x, b), y, c), z, d)
  | (Blk, T(Red, a, x, T(Red, b, y, c)), z, d)
  | (Blk, a, x, T(Red, T(Red, b, y, c), z, d))
  | (Blk, a, x, T(Red, b, y, T(Red, c, z, d)))) =
    T(Red, T(Blk, a, x, b), y, T(Blk, c, z, d))
  | balance x = T x

fun ins x Empty = T(R, Empty, x, Empty)
  | ins x (T(color, a, y, b)) =
    if x <= y then balance(color, ins x a, y, b)
    else if x > y then balance(color, a, y, ins x b)
```

void rb_insert( Tree T, node x ) {
    tree_insert( T, x );
    x->colour = red;
    while ( (x != T->root) && (x->parent->colour == red) ) {
        if ( x->parent == x->parent->parent->left ) {
            y = x->parent->parent->right;
            if ( y->colour == red ) {
                x->parent->colour = black;
                y->colour = black;
                x->parent->parent->colour = red;
                x = x->parent->parent;
            } else {
                if ( x == x->parent->right ) {
                    x = x->parent;
                    left_rotate( T, x );
                }
                x->parent->colour = black;
                x->parent->parent->colour = red;
                right_rotate( T, x->parent->parent );
            }
        } else {
            ...
            /* repeat above with red/black swapped */
        }
    }
}
void left_rotate( Tree T, node x ) {
    node y;
    y = x->right;
    x->right = y->left;
    if ( y->left != NULL )
        y->left->parent = x;
    y->parent = x->parent;
    if ( x->parent == NULL )
        T->root = y;
    else if ( x == (x->parent)->left )
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
    y->left = x;
    x->parent = y;
}

/* repeat above for right_rotate with “obvious” changes */
A Key Outcome

- Master Key Linguistic Abstractions:
  - procedural abstraction
  - control: iteration, recursion, pattern matching, laziness, exceptions, events, threads, continuations
  - encapsulation: closures, ADTs, objects, modules
  - Parameterization: higher-order procedures, modules; classes, inheritance
More Outcomes

• Exposure to software eng. techniques:
  – modular design.
  – unit tests, integration tests.
  – critical code reviews.

• Exposure to abstract models:
  – models for design & communication.
  – models & techniques for proving correctness of code.
  – models for space & time.