The class...

- The midterm was too long...
- Yes, but it was the same length for everyone
- We'll scale appropriately
- Lectures
  - I'll be out of town next week
  - Two options
    - A lecture by Noah
    - An extra lecture on a Friday—"fun" topic
- Homework
  - More frequent, smaller problem sets
- Projects
  - We'll still do two more
  - I'll scale them back

Prelude

- Who are these people?
  - Pierre and Pam Omidyar
  - And we care because...?
    - Pierre Omidyar is the founder of eBay
    - Donated $100 million to Tufts
  - But there are strings...
    - The money must be invested in microfinance
- What is microfinance?
  - Small business loans, typically given to individuals in developing countries
  - Average: $600 (as low as $40)

Code generation

- Overview
  - Walk AST, generate code for each construct
  - Nested constructs:
    - Call generate on children as necessary
    - Use registers to pass values up tree
  - Key: order is important
    - Emitted code goes into central list
    - Bottom-up post-order traversal

Example

```java
if (c == 0) {
    while (c < 20) {
        c = c + 2;
    }
} else {
    c = n * n + 2;
}
```

Example

```java
if (c == 0) {
    while (c < 20) {
        c = c + 2;
    }
} else {
    c = n * n + 2;
}
```
Example

while

\[ t = \text{generate}(\text{expr}) \]
\[ \text{emit}(\text{ifnot_goto } t, E) \]
\[ \text{generate}(\text{statement}) \]
\[ \text{emit}(\text{goto } T) \]
\[ \text{emit}(E:) \]
\[ t1 = \text{generate}(\text{expr1}) \]
\[ t2 = \text{generate}(\text{expr2}) \]
\[ r = \text{new_temp}() \]
\[ \text{emit}(r = t1 \text{ op } t2) \]
\[ \text{return } r \]

Example

while

\[ R0 = \text{load } c \]
\[ t = \text{generate}(\text{expr}) \]
\[ \text{emit}(\text{ifnot_goto } t, E) \]
\[ \text{generate}(\text{statement}) \]
\[ \text{emit}(\text{goto } T) \]
\[ \text{emit}(E:) \]
\[ t1 = \text{generate}(\text{expr1}) = R0 \]
\[ t2 = \text{generate}(\text{expr2}) \]
\[ r = \text{new_temp}() \]
\[ \text{emit}(r = t1 \text{ op } t2) \]
\[ \text{return } r \]

Example

while

\[ R0 = \text{load } c \]
\[ R1 = 20 \]
\[ t = \text{generate}(\text{expr}) \]
\[ \text{emit}(\text{ifnot_goto } t, E) \]
\[ \text{generate}(\text{statement}) \]
\[ \text{emit}(\text{goto } T) \]
\[ \text{emit}(E:) \]
\[ t1 = \text{generate}(\text{expr1}) = R0 \]
\[ t2 = \text{generate}(\text{expr2}) = R1 \]
\[ r = \text{new_temp}() \]
\[ \text{emit}(r = t1 \text{ op } t2) \]
\[ \text{return } r \]
Example

while

R0 = load c
R1 = 20
R2 = R0 < R1
not_goto R2,L0

E = new_label()
T = new_label()
emit(T:)
t = generate(expr)=R2
emit(ifnot_goto t, E)
generate(statement)
emit(goto T)
emit(E:)

r = generate(expr2)
l = generate(expr2)
emit(store l = r)
return r

Something like:
R6 = base + offset
**Example**

```plaintext
while c < 20
    if c < 20
        goto L1
    if c = 2
        goto L4
while (c=20)
```

**Code**

```plaintext
L1: R0 = load c
R1 = 20
R2 = R0 < R1
not_goto R2, L0
R3 = load c
R4 = 2
R5 = R3 + R2
R6 = & c
store [R6] = R5
goto L1
L0:
```

**Nesting**

```plaintext
while (c=20)
    c = c + 2
while (c=2)
    c = c + 2
```

**Code**

```plaintext
L1: R0 = load c
R1 = 20
R2 = R0 < R1
not_goto R2, L0
R3 = load c
R4 = 2
R5 = R3 + R2
R6 = & c
store [R6] = R5
goto L1
L0:
```

**Code Shape**

- Definition
  - All those nebulous properties of the code that impact performance & code “quality”
  - Includes:
    - Code for different constructs
    - Cost, storage requirements & mapping
    - Choice of operations
    - Code shape is the end product of many decisions
- Impact
  - Code shape influences algorithm choice & results
  - Code shape can encode important facts, or hide them

**Code Shape**

- An example:
  - x + y + z
  - x + y + z
  - x + z + y
  - y + z + x
- What if x is 2 and z is 3?
- What if x + z is evaluated earlier?
- The “best” shape for x + y + z depends on context
  - There may be several conflicting options

**Code Shape**

- Another example – the switch statement
  - Implement it as a jump table
    - Lookup address in a table & jump to it
    - Uniform (constant) cost
  - Implement it as cascaded if-then-else statements
    - Cost depends on where your case actually occurs
    - O(number of cases)
  - Implement it as a binary search
    - Uniform (log n) cost
- Compiler must choose best implementation strategy
  - No way to convert one into another
Order of evaluation

- Ordering for performance
  - Using associativity and commutativity
    - Very hard problem
  - Operands
    - op1 must be preserved while op2 is computed
    - Emit code for more intensive one first
- Language requirements
  - Sequence points:
    - Places where side effects must be visible to other operations
  - C examples:
    - `f() + g()` may be executed in any order
    - `f() || g()` must be executed first
    - `f(i++)` argument to `f` must be `i+1`

Registers only

- Proliferation of registers and labels
  - Many CPUs have a fast `c == 0` test
  - Can use accumulators:
    - `c += 2`
  - Label leads to another goto; may have multiple labels
- Code
  ```
  R7 = load c
  R8 = 0
  R9 = R7 == R8
  not_goto R9,L3
  L1:
  R0 = load c
  R1 = 20
  R2 = R0 < R1
  not_goto R2,L0
  R3 = load c
  R4 = 2
  R5 = R3 + R2
  c = R5
  goto L1
  L0:
  goto L4
  L3:
  ..
  L4:
  ```
- Efficient lowering
  - Reduce number of temporary registers
  - Don’t copy variable values into registers
  - Accumulate values, when possible
  - Reuse temporaries, where possible
  - Generate more efficient labels
  - Don’t generate multiple adjacent labels
  - Avoid goto-label-goto

Avoiding extra copies

- Basic algorithm
  - Recursive generation traverses to leaves
  - At leaves, generate: `R = V` or `R = c`
- Improvement
  - Stop recursion one level early
  - Check to see if children are leaves
  - Don’t call generate recursively on variables, constants

Avoiding copies

```python
if (expr1 is var) t1 = (Var) expr1
else t1 = generate(expr1)

if (expr2 is var) t2 = (Var) expr2
else t2 = generate(expr2)
r = new_temp()
emit(r = t1 op t2)
return r
```
**Example**

- **Expr1** is \((a+b)\)
  - Not a leaf
  - Recursively generate code
  - Return temp
  
  \[
  \begin{align*}
  (a + b) \cdot c \\
  R0 = a + b \\
  R1 = R0 \cdot c
  \end{align*}
  \]

- **Expr2** is \(c\)
  - Return \(c\)
  
  \[
  \begin{align*}
  R0 = a + b \\
  R0 = R0 \cdot c
  \end{align*}
  \]

- **Emit** \((R0 \cdot c)\)

**Use accumulation**

- **Idea:**
  - We only need 2 registers to evaluate \(\text{expr1 op expr2}\)
  - Reuse temp assigned to one of the subexpressions

  ```
  \text{if (expr1 is var)}
  \text{t1 = (Var) expr1}
  \text{else}
  \text{t1 = generate(expr1)}
  \text{t2 = (Var) expr2}
  \text{else}
  \text{t2 = generate(expr2)}
  \text{emit( t1 = t1 op t2 )}
  \text{return t1}
  ```

**Example**

- Combined:
  - Remove copies
  - Accumulate value
  - Only need one register
  
  \[
  \text{(Original version might have required as many as 6)}
  \]

  \[
  \begin{align*}
  (a + b) \cdot c \\
  R0 = a + b \\
  R0 = R0 \cdot c
  \end{align*}
  \]

**Reuse of temporaries**

- **Idea:**
  - Can \(\text{generate(expr1)}\) and \(\text{generate(expr2)}\) share temporaries?

  ```
  Rn = (Var) expr1
  \text{emit( R = Rn op expr2 )}
  \text{return R}
  ```

**Next time...**

- Introduction to optimization
- Later today: Project stage 3
- A small homework