Announcements

• HW6: Dataflow Analysis
  – Due: Weds. April 26\textsuperscript{th}

  NOTE: See Piazza for an update…
  TLDR: "simple" regalloc should not suffice.
  Change gradedtests.ml \texttt{>=} to \texttt{>}

• FINAL EXAM
  – Thursday, May 4\textsuperscript{th} noon – 2:00p.m.
  – Location: DRLB A4
LOOPS AND DOMINATORS
Loops in Control-flow Graphs

- Taking into account loops is important for optimizations.
  - The 90/10 rule applies, so optimizing loop bodies is important

- Should we apply loop optimizations at the AST level or at a lower representation?
  - Loop optimizations benefit from other IR-level optimizations and vice-versa, so it is good to interleave them.

- Loops may be hard to recognize at the quadruple / LLVM IR level.
  - Many kinds of loops: while, do/while, for, continue, goto...

- Problem: How do we identify loops in the control-flow graph?
Definition of a Loop

- A loop is a set of nodes in the control flow graph.
  - One distinguished entry point called the header

- Every node is reachable from the header & the header is reachable from every node.
  - A loop is a strongly connected component

- No edges enter the loop except to the header
- Nodes with outgoing edges are called loop exit nodes
Nested Loops

- Control-flow graphs may contain many loops
- Loops may contain other loops:

**Control Tree:**

The *control tree* depicts the nesting structure of the loops in the program.
Control-flow Analysis

- Goal: Identify the loops and nesting structure of the CFG.

- Control flow analysis is based on the idea of *dominators*:
  - Node A *dominates* node B if the only way to reach B from the start node is through node A.

- An edge in the graph is a *back edge* if the target node dominates the source node.

- A loop contains at least one back edge.
Dominator Trees

- Domination is transitive:
  - if A dominates B and B dominates C then A dominates C
- Domination is anti-symmetric:
  - if A dominates B and B dominates A then A = B
- Every flow graph has a dominator tree
  - The Hasse diagram of the dominates relation
DOMINATOR DATAFLOW ANALYSIS

• We can define Dom[n] as a forward dataflow analysis.
  – Using the framework we saw earlier: \( \text{Dom}[n] = \text{out}[n] \) where:
• “A node B is dominated by another node A if A dominates all of the predecessors of B.”
  – \( \text{in}[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n'] \)
• “Every node dominates itself.”
  – \( \text{out}[n] := \text{in}[n] \cup \{n\} \)

• Formally: \( L \) = set of nodes ordered by \( \subseteq \)
  – \( T = \{ \text{all nodes} \} \)
  – \( F_n(x) = x \cup \{n\} \)
  – \( \cap \) is \( \cap \)
• Easy to show monotonicity and that \( F_n \) distributes over meet.
  – So algorithm terminates and is MOP
Improving the Algorithm

- Dom[b] contains just those nodes along the path in the dominator tree from the root to b:
  - e.g. Dom[8] = \{1,2,4,8\}, Dom[7] = \{1,2,4,5,7\}
  - There is a lot of sharing among the nodes

- More efficient way to represent Dom sets is to store the dominator tree.
  - doms[b] = immediate dominator of b

- To compute Dom[b] walk through doms[b]
- Need to efficiently compute intersections of Dom[a] and Dom[b]
  - Traverse up tree, looking for least common ancestor:

- See: “A Simple, Fast Dominance Algorithm” Cooper, Harvey, and Kennedy
Completing Control-flow Analysis

- Dominator analysis identifies **back edges**: 
  - Edge $n \rightarrow h$ where $h$ dominates $n$
- Each back edge has a **natural loop**: 
  - $h$ is the header
  - All nodes reachable from $h$ that also reach $n$ without going through $h$
- For each back edge $n \rightarrow h$, find the natural loop: 
  - $\{n' \mid n$ is reachable from $n'$ in $G - \{h\}\} \cup \{h\}$

- Two loops may share the same header: merge them

- Nesting structure of loops is determined by set inclusion 
  - Can be used to build the control tree
The control tree depicts the nesting structure of the program.
Uses of Control-flow Information

• Loop nesting depth plays an important role in optimization heuristics.
  – Deeply nested loops pay off the most for optimization.

• Need to know loop headers / back edges for doing
  – loop invariant code motion
  – loop unrolling

• Dominance information also plays a role in converting to SSA form
  – Used internally by LLVM to do register allocation.
Phi nodes
Alloc “promotion”
Register allocation
Single Static Assignment (SSA)

- LLVM IR names (via `%uids`) all intermediate values computed by the program.
- It makes the order of evaluation explicit.
- Each `%uid` is assigned to only once
  - Contrast with the mutable quadruple form
  - Note that dataflow analyses had these kill[n] sets because of updates to variables...
- Naïve implementation of backend: map `%uids` to stack slots
- Better implementation: map `%uids` to registers (as much as possible)

- Question: How do we convert a source program to make maximal use of `%uids`, rather than alloca-created storage?
  - two problems: control flow & location in memory

- Then: How do we convert SSA code to x86, mapping `%uids` to registers?
  - Register allocation.
Allocating Variables

- Current compilation strategy:
  - Directly map source variables into `%uids`?

```c
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```asm
%x = alloca i64
%y = alloca i64
store i64* %x, 3
store i64* %y, 0
%x1 = load i64* %x
%tmp1 = add i64 %x1, 1
store i64* %x, %tmp1
%x2 = load i64* %x
%tmp2 = add i64 %x2, 2
store i64* %y, %tmp2
```

- Does this always work?

```c
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```c
int x1 = 3;
int y1 = 0;
x2 = x1 + 1;
y2 = x2 + 2;
```

```asm
%x1 = add i64 3, 0
%y1 = add i64 0, 0
%x2 = add i64 %x1, 1
%y2 = add i64 %x2, 2
```
What about If-then-else?

• How do we translate this into SSA?

```
int y = ...
int x = ...
int z = ...
if (p) {
    x = y + 1;
} else {
    x = y * 2;
}
z = x + 3;
```

```
entry:
    %y1 = ...%x1 = ...%z1 = ...
    %p = icmp ...
br i1 %p, label %then, label %else
then:
    %x2 = add i64 %y1, 1
    br label %merge
else:
    %x3 = mult i64 %y1, 2
merge:
    %z2 = %add i64 ???, 3
```

• What do we put for ???
Phi Functions

• Solution: \( \phi \) functions
  – Fictitious operator, used only for analysis
    • implemented by Mov at x86 level
  – Chooses among different versions of a variable based on the path by which control enters the phi node.

\[
\%uid = \phi <ty> v_1, <label_1>, ..., v_n, <label_n>
\]

```c
int y = ... 
int x = ... 
int z = ... 
if (p) {
  x = y + 1;
} else {
  x = y * 2;
} 
z = x + 3;
```

entry:

```c
  %y1 = ... 
  %x1 = ... 
  %z1 = ... 
  %p = icmp ... 
  br il %p, label %then, label %else 
then:
  %x2 = add i64 %y1, 1 
  br label %merge 
else:
  %x3 = mult i64 %y1, 2 
merge:
  %x4 = phi i64 %x2, %then, %x3, %else 
  %z2 = %add i64 %x4, 3
```
Phi Nodes and Loops

• Importantly, the $\texttt{uids}$ on the right-hand side of a phi node can be defined “later” in the control-flow graph.
  – Means that $\texttt{uids}$ can hold values “around a loop”
  – Scope of $\texttt{uids}$ is defined by dominance (discussed soon)

entry:
entry:
%y1 = …
%x1 = …
br label %body

body:
body:
%x2 = phi i64 %x1, %entry, %x3, %body
%x3 = add i64 %x2, %y1
%p = icmp slt i64, %x3, 10
br i1 %p, label %body, label %after

after:
after:
...
Allocated Promotion

- Not all source variables can be allocated to registers
  - If the address of the variable is taken (as permitted in C, for example)
  - If the address of the variable “escapes” (by being passed to a function)
- An alloca instruction is called promotable if neither of the two conditions above holds

```
entry:
  %x = alloca i64        // %x cannot be promoted
  %y = call malloc(i64 8)
  %ptr = bitcast i8* %y to i64**
  store i65** %ptr, %x     // store the pointer into the heap

entry:
  %x = alloca i64        // %x cannot be promoted
  %y = call foo(i64* %x)  // foo may store the pointer into the heap
```

- Happily, most local variables declared in source programs are promotable
  - That means they can be register allocated
Converting to SSA: Overview

- Start with the ordinary control flow graph that uses allocas
  - Identify “promotable” allocas
- Compute dominator tree information
- Calculate def/use information for each such allocated variable
- Insert $\phi$ functions for each variable at necessary “join points”

- Replace loads/stores to alloc’ed variables with freshly-generated %uids

- Eliminate the now unneeded load/store/alloca instructions.
Where to Place φ functions?

- Need to calculate the “Dominance Frontier”

- Node A *strictly dominates* node B if A dominates B and $A \neq B$.
  - Note: A does not strictly dominate B if A does not dominate B or $A = B$.

- The *dominance frontier* of a node B is the set of all CFG nodes y such that B dominates a predecessor of y but does not strictly dominate y
  - Intuitively: starting at B, there is a path to y, but there is another route to y that does not go through B

- Write $DF[n]$ for the dominance frontier of node n.
Dominance Frontiers

- Example of a dominance frontier calculation results
Algorithm For Computing DF[n]

- Assume that \( \text{doms}[n] \) stores the dominator tree (so that \( \text{doms}[n] \) is the *immediate dominator* of \( n \) in the tree)

- Adds each \( B \) to the DF sets to which it belongs

\[
\text{for all nodes } B \\
\text{if } \#(\text{pred}[B]) \geq 2 \quad // \text{(just an optimization)} \\
\text{for each } p \in \text{pred}[B] \{ \\
\quad \text{runner} := p \quad // \text{start at the predecessor of } B \\
\quad \text{while } (\text{runner} \neq \text{doms}[B]) \quad // \text{walk up the tree adding } B \\
\quad \quad \text{DF}[\text{runner}] := \text{DF}[\text{runner}] \cup \{B\} \\
\quad \quad \text{runner} := \text{doms}[\text{runner}] \\
\} 
\]
Insert $\phi$ at Join Points

- Lift the $DF[n]$ to a set of nodes $N$ in the obvious way:
  \[ DF[N] = \bigcup_{n \in N} DF[n] \]
- Suppose that at variable $x$ is defined at a set of nodes $N$.
- $DF_0[N] = DF[N]$
  $DF_{i+1}[N] = DF[DF_i[N] \cup N]$
- Let $J[N]$ be the least fixed point of the sequence:
  $DF_0[N] \subseteq DF_1[N] \subseteq DF_2[N] \subseteq DF_3[N] \subseteq \ldots$
  - That is, $J[N] = DF_k[N]$ for some $k$ such that $DF_k[N] = DF_{k+1}[N]$
- $J[N]$ is called the “join points” for the set $N$
- We insert $\phi$ functions for the variable $x$ at each such join point.
  - $x = \phi(x, x, \ldots, x)$; (one “$x$” argument for each predecessor of the node)
  - In practice, $J[N]$ is never directly computed, instead you use a worklist algorithm that keeps adding nodes for $DF_k[N]$ until there are no changes.

- Intuition:
  - If $N$ is the set of places where $x$ is modified, then $DF[N]$ is the places where phi nodes need to be added, but those also “count” as modifications of $x$, so we need to insert the phi nodes to capture those modifications too…
Example Join-point Calculation

• Suppose the variable x is modified at nodes 3 and 6
  – Where would we need to add phi nodes?

• \( DF_0[{3,6}] = DF[{3,6}] = DF[3] \cup DF[6] = \{2,8\} \)

• \( DF_1[{3,6}] \)
  = \( DF[DF_0{3,6} \cup \{3,6\}] \)
  = \( DF[{2,3,6,8}] \)
  = \( \{1,2\} \cup \{2\} \cup \{8\} \cup \{0\} = \{1,2,8,0\} \)

• \( DF_2[{3,6}] \)
  = \( ... \)
  = \( \{1,2,8,0\} \)

• So \( J[{3,6}] = \{1,2,8,0\} \) and we need to add phi nodes at those four spots.
Phi Placement Alternative

• Less efficient, but easier to understand:

• Place phi nodes "maximally" (i.e. at every node with > 2 predecessors)

• If all values flowing into phi node are the same, then eliminate it:

\[
\%x = \text{phi } t \%y, \%\text{pred1 } t \%y \%\text{pred2 } \ldots t \%y \%\text{predK}
\]

// code that uses %x
⇒

// code with %x replaced by %y

• Interleave with other optimizations
  – copy propagation
  – constant propagation
  – etc.
Example SSA Optimizations

- How to place phi nodes without breaking SSA?

- Note: the “real” implementation combines many of these steps into one pass.
  - Places phis directly at the dominance frontier

- This example also illustrates other common optimizations:
  - Load after store/alloca
  - Dead store/alloca elimination
Example SSA Optimizations

- How to place phi nodes without breaking SSA?
- Insert — Loads at the end of each block

l₁: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x₁ = load %p
    br %b, %l₂, %l₃

l₂:
    store 1, %p
    %x₂ = load %p
    br %l₃

l₃:
    %x = load %p
    ret %x

Find alloca
max φs
LAS/LAA
DSE
DAE
elim φs
Example SSA Optimizations

- How to place phi nodes without breaking SSA?
  - Insert
    - Loads at the end of each block
    - Insert $\phi$-nodes at each block

l₁: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x₁ = load %p
    br %b, %l₂, %l₃

l₂: %x₃ = $\phi$[%x₁, %l₁]
    store 1, %p
    %x₂ = load %p
    br %l₃

l₃: %x₄ = $\phi$[%x₁; %l₁, %x₂; %l₂]
    %x = load %p
    ret %x
Example SSA Optimizations

- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert \( \phi \) -nodes at each block
  - Insert stores after \( \phi \) -nodes

```c
l_1: \%p = alloca i64
    store 0, \%p
    \%b = \%y > 0
    \%x_1 = load \%p
    br \%b, \%l_2, \%l_3

l_2: \%x_3 = \phi[\%x_1, \%l_1]
    store \%x_3, \%p
    store 1, \%p
    \%x_2 = load \%p
    br \%l_3

l_3: \%x_4 = \phi[\%x_1; \%l_1, \%x_2; \%l_2]
    store \%x_4, \%p
    \%x = load \%p
    ret \%x
```
Example SSA Optimizations

For loads after stores (LAS):
- Substitute all uses of the load by the value being stored
- Remove the load

\[ \begin{align*}
\text{l}_1: & \quad %p = \text{alloca i64} \\
& \quad \text{store 0, } %p \\
& \quad %b = %y > 0 \\
& \quad %x_1 = \text{load } %p \\
& \quad \text{br } %b, %l_2, %l_3 \\
\end{align*} \]

\[ \begin{align*}
\text{l}_2: & \quad %x_3 = \phi[ %x_1, %l_1 ] \\
& \quad \text{store } %x_3, %p \\
& \quad \text{store } 1, %p \\
& \quad %x_2 = \text{load } %p \\
& \quad \text{br } %l_3 \\
\end{align*} \]

\[ \begin{align*}
\text{l}_3: & \quad %x_4 = \phi[ %x_1; %l_1, %x_2; %l_2 ] \\
& \quad \text{store } %x_4, %p \\
& \quad %x = \text{load } %p \\
& \quad \text{ret } %x \\
\end{align*} \]
Example SSA Optimizations

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

```
l_1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x_1 = load %p
    br %b, %l_2, %l_3

l_2: %x_3 = phi[ %x_1, %l_1 ]
    store %x_3, %p
    store 1, %p
    %x_2 = load %p
    br %l_3

l_3: %x_4 = phi[ %x_1, %l_1, %x_2:%l_2 ]
    store %x_4, %p
    %x = load %p
    ret %x
```
Example SSA Optimizations

For loads after stores (LAS):
- Substitute all uses of the load by the value being stored
- Remove the load

```
// l1: %p = alloca i64
store 0, %p
%b = %y > 0
%x = load %p
br %b, %l2, %l3
```

```
// l2: %x = phi[0, %l1]
store %x, %p
store 1, %p
%x = load %p
br %l3
```

```
// l3: %x = phi[0; %l1, %x2:%l2]
store %x, %p
%x = load %p
ret %x
```
Example SSA Optimizations

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

```
l_1: %p = alloca i64
     store 0, %p
     %b = %y > 0
     br %b, %l_2, %l_3

l_2: %x_3 = phi[0,%l_1]
     store %x_3, %p
     store 1, %p
     %x_2 = load %p
     br %l_3

l_3: %x_4 = phi[0;%l_1, %x_2, %l_2]
     store %x_4, %p
     %x = load %p
     ret %x
```
Example SSA Optimizations

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

```c
l_1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    br %b, %l_2, %l_3

l_2: %x_3 = phi[0, %l_1]
    store %x_3, %p
    store 1, %p
    %x_2 = load %p
    br %l_3

l_3: %x_4 = phi[0; %l_1, 1; %l_2]
    store %x_4, %p
    %x = load %p
    ret %x
```
Example SSA Optimizations

For loads after stores (LAS):

- Substitute all uses of the load by the value being stored
- Remove the load

\[
\begin{align*}
  l_1: & \quad \%p = \text{alloca } \text{i64} \\
       & \quad \text{store } 0, \%p \\
       & \quad \%b = \%y > 0 \\
       & \quad \text{br } \%b, \%l_2, \%l_3 \\

  l_2: & \quad \%x_3 = \phi[0, \%l_1] \\
       & \quad \text{store } \%x_3, \%p \\
       & \quad \text{store } 1, \%p \\
       & \quad \text{br } \%l_3 \\

  l_3: & \quad \%x_4 = \phi[0; \%l_1, 1; \%l_2] \\
       & \quad \text{store } \%x_4, \%p \\
       & \quad \%x = \text{load } \%p \\
       & \quad \text{ret } \%x
\end{align*}
\]
Example SSA Optimizations

For loads after stores (LAS):
- Substitute all uses of the load by the value being stored
- Remove the load

\[ l_1: \%p = \text{alloca i64} \]
\[ \text{store 0, } \%p \]
\[ \%b = \%y > 0 \]
\[ \text{br } \%b, \%l_2, \%l_3 \]

\[ l_2: \%x_3 = \phi[0, %l_1] \]
\[ \text{store } \%x_3, \%p \]
\[ \text{store 1, } \%p \]
\[ \text{br } \%l_3 \]

\[ l_3: \%x_4 = \phi[0, %l_1, 1: %l_2] \]
\[ \text{store } \%x_4, \%p \]
\[ \%x = \text{load } \%p \]
\[ \text{ret } \%x_4 \]
Example SSA Optimizations

- **Dead Store Elimination (DSE)**
  - Eliminate all stores with no subsequent loads.

- **Dead Alloca Elimination (DAE)**
  - Eliminate all allocas with no subsequent loads/stores.

```
l_1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    br %b, %l_2, %l_3

l_2: %x_3 = phi[0,%l_1]
    store %x_3, %p
    store 1, %p
    br %l_3

l_3: %x_4 = phi[0;%l_1, 1;%l_2]
    store %x_4, %p
    ret %x_4
```

Find alloca

max φs

LAS/LAA

DSE

DAE

elim φs
Example SSA Optimizations

- **Dead Store Elimination (DSE)**
  - Eliminate all stores with no subsequent loads.

- **Dead Alloca Elimination (DAE)**
  - Eliminate all allocas with no subsequent loads/stores.

Code example:

```
l_1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    br %b, %l_2, %l_3

l_2: %x_3 = phi[0, %l_1]
    store %x_3, %p
    store 1, %p
    br %l_3

l_3: %x_4 = phi[0; %l_1, 1:%l_2]
    store %x_4, %p
    ret %x_4
```
Example SSA Optimizations

$l_1$: 
\[
%b = %y > 0
\]
\[
br %b, %l_2, %l_3
\]

$l_2$: 
\[
%x_3 = \phi[0, %l_1]
\]
\[
br %l_3
\]

$l_3$: 
\[
%x_4 = \phi[0; %l_1, 1:%l_2]
\]
\[
ret %x_4
\]

- Eliminate $\phi$ nodes:
  - Singletons
  - With identical values from each predecessor
  - See Aycock & Horspool, 2002
Example SSA Optimizations

1. \( b = y > 0 \)
   
   \( \text{br } b, l_2, l_3 \)

2. \( x_3 = \phi[0,l_1] \)
   
   \( \text{br } l_3 \)

3. \( x_4 = \phi[0;l_1, l_2] \)
   
   \( \text{ret } x_4 \)

---

- Eliminate \( \phi \) nodes:
  - Singletons
  - With identical values from each predecessor

---

Find alloca

max \( \phi \)s

LAS/LAA

DSE

DAE

elim \( \phi \)s
Example SSA Optimizations

\[ l_1: \%
b = \%y > 0 \]
\[ \text{br } \%b, \%l_2, \%l_3 \]

\[ l_2: \]
\[ \text{br } \%l_3 \]

\[ l_3: \%x_4 = \phi[0;\%l_1, 1:\%l_2] \]
\[ \text{ret } \%x_4 \]

- Done!

Find alloca
max $\phi$s
LAS/LAA
DSE
DAE
elim $\phi$
LLVM Phi Placement

• This transformation is also sometimes called register promotion
  – older versions of LLVM called this “mem2reg” memory to register promotion

• In practice, LLVM combines this transformation with scalar replacement of aggregates (SROA)
  – i.e. transforming loads/stores of structured data into loads/stores on register-sized data

• These algorithms are (one reason) why LLVM IR allows annotation of predecessor information in the .ll files
  – Simplifies computing the DF
Thursday, May 4th noon – 2:00p.m.
Location: DRLB A4

FINAL EXAM
Final Exam

• Will cover material since the midterm almost exclusively
  – Starting from Lecture 14
  – Typechecking
  – Objects, inheritance, types, implementation of dynamic dispatch
  – Basic optimizations
  – Dataflow analysis (forward vs. backward, fixpoint computations, etc.)
    • Liveness
  – Graph-coloring Register Allocation
  – Control flow analysis
    • Loops, dominator trees

• Will focus more on the theory side of things
• Format will be similar to the midterm
  – Simple answer, computation, multiple choice, etc.
  – Sample exam from last time is on the web
What have we learned?
Where else is it applicable?
What next?
Why CIS 341?

• You will learn:
  – Practical applications of theory
  – Parsing
  – How high-level languages are implemented in machine language
  – (A subset of) Intel x86 architecture
  – A deeper understanding of code
  – A little about programming language semantics
  – Functional programming in OCaml
  – How to manipulate complex data structures
  – How to be a better programmer

• Did we meet these goals?
Stuff we didn’t Cover

- We skipped stuff at every level…
- Concrete syntax/parsing:
  - Much more to the theory of parsing…
  - Good syntax is art not science!
- Source language features:
  - Exceptions, recursive data types (easy!), advanced type systems, type inference, concurrency
- Intermediate languages:
  - Intermediate language design, bytecode, bytecode interpreters, just-in-time compilation (JIT)
- Compilation:
  - Continuation-passing transformation, efficient representations, scalability
- Optimization:
  - Scientific computing, cache optimization, instruction selection/optimization
- Runtime support:
  - Memory management, garbage collections
Related Courses

• CIS 500: Software Foundations
  – Prof. Pierce
  – Theoretical course about functional programming, proving program properties, type systems, lambda calculus. Uses the theorem prover Coq.

• CIS 501: Computer Architecture
  – Prof. Devietti
  – 371++: pipelining, caches, VM, superscalar, multicore,…

• CIS 552: Advanced Programming
  – Prof. Weirich
  – Advanced functional programming in Haskell, including generic programming, metaprogramming, embedded languages, cool tricks with fancy type systems

• CIS 670: Special topics in programming languages
Where to go from here?

• Conferences (proceedings available on the web):
  – Programming Language Design and Implementation (PLDI)
  – Principles of Programming Languages (POPL)
  – Object Oriented Programming Systems, Languages & Applications (OOPSLA)
  – International Conference on Functional Programming (ICFP)
  – European Symposium on Programming (ESOP)
  – …

• Technologies / Open Source Projects
  – Yacc, lex, bison, flex, …
  – LLVM – low level virtual machine
  – Java virtual machine (JVM), Microsoft’s Common Language Runtime (CLR)
  – Languages: OCaml, F#, Haskell, Scala, Go, Rust, …?
Where else is this stuff applicable?

• General programming
  – In C/C++, better understanding of how the compiler works can help you generate better code.
  – Ability to read assembly output from compiler
  – Experience with functional programming can give you different ways to think about how to solve a problem

• Writing domain specific languages
  – lex/yacc very useful for little utilities
  – understanding abstract syntax and interpretation

• Understanding hardware/software interface
  – Different devices have different instruction sets, programming models
Thanks!

- To the TAs: Dmitri, Richard, J.J. and Vivek
  - for doing an amazing job putting together the projects for the course.

- To you for taking the class!

- How can I improve the course?
  - Feedback survey posted to Piazza (soon!)