Administration

• Announcements
  – Midterm scheduled for Thursday, March 1st, in class
  – Final exam is scheduled: Apr 18 2018, 7:00pm

• Minijava project
  – Build a Minijava compiler from a Functions compiler
  – You can start with your own or mine
  – Phase 1 due tomorrow
Anybody want to talk about the midterm?
Emitting Assembly Code

- We now understand how maximal munch can be implemented with “munching rules”:

- A rule:
  - matches a pattern in the IR tree
  - “largest tile” pattern takes priority

- when triggered the rule:
  - recursively calls the matcher on subtrees
  - emits assembly code for the pattern
    (we haven’t discussed this yet)
A peek into the compiler

• Look at X86_64Muncher.java
Where are we at?

- Instruction selection is done by writing tree matching patterns and associated code for emitting instructions
- But how are we going to represent instructions?
• We need a good representation of Assembly code:
  – not target architecture specific
  – must allow assembly code with “fictional temp registers”
  – must have enough information for register allocator to analyze register usage

• The representation must be
  – abstract w.r.t. target architecture
  – concrete and detailed w.r.t register usage and flow of control
Assembly Code – a list of instructions

• What do we care about?
  – Labels
  – Branches
  – Registers
    • Which ones get used
    • Which ones get changed

• What don’t we care about?
  – Instructions (sub vs. add vs. div)
Assembly Code – Instructions

• For each instruction
  – Where might it branch to?
  – What registers does it read?
    • “use” set
  – What registers does it write?
    • “def” set

```java
public class Instr {
    abstract public List<Label> jumps();
    abstract public List<Temp> def();
    abstract public List<Temp> use();
}
```
Assembly Code – Labels

• Labels are special as jump targets.

```java
public class A_LABEL extends Instr {
    private Label label;
    public List<Temp> use() {
        return List.empty();
    }
    public List<Temp> def() {
        return List.empty();
    }
    public List<Label> jumps() {
        return null;
    }
}
```
Assembly Code – Moves

• Moves (register-to-register) are special, too. Stay tuned ...

```java
public class A_MOVE extends Instr {
    private Temp src, dst;
    public List<Temp> use() {
        return list(src);
    }
    public List<Temp> def() {
        return list(dst);
    }
    public List<Label> jumps() {
        return null;
    }
}
```
Assembly Code – Jumps

• Jumps are special – represent control flow

```java
public class A_JMP extends Instr {
    private Label target;
    public List<Temp> use() {
        return List.empty();
    }
    public List<Temp> def() {
        return List.empty();
    }
    public List<Label> jumps() {
        return list(target);
    }
}
```
• Jumps are special – represent control flow

```java
public class A_CJMP extends Instr {
    private Label thn, els;
    public List<Temp> use() {
        return List.empty();
    }
    public List<Temp> def() {
        return List.empty();
    }
    public List<Label> jumps() {
        return list(thn, els);
    }
}
```
Assembly Code – Everything else

• For every other instruction

```java
public class A_OPER extends Instr {
    public List<Temp> dst, src;
    public List<Label> jump;
    public List<Temp> def() {
        return dst;
    }
    public List<Temp> use() {
        return src;
    }
    public List<Label> jumps() {
        return jump;
    }
    ...
}
```
public A_OPER(a, d, s, j) {
    dst = d;
    src = s;
    jump = j;
}

public A_OPER(a, d, s) {
    this(a, d, s, null);
}

Assembly Code – A_OPER constructors
Examples

Instr A_ADD(Temp reg, int i) {
    return new A_OPsER(
        "addq $"+i+"", `d0",
        list(reg),
        list(reg));
}
Examples

Instr A_ADD(Temp dst, Temp src) {
    return new A_OPER(
        "addq `s0, `d0",
        list(dst),
        list(src, dst));
}
Examples

Instr A_MOVE_TO_MEM
   (Temp ptr, Temp src) {
      return new A_OPER(
         "movq `s1, (`s0)",
         List.empty(),
         list(ptr, src));
   }
A peek into the compiler

- Look at the A_ methods in X86_64Muncher.java
OK, Now what?

- After the code generation (instruction selection) phase, we now have a representation of assembly code as a list of instructions.

```java
public abstract class Instr {

    private String assem;
    protected Instr(String assem) { this.assem = assem; }

    public abstract List<Temp> use();
    // Temps “read” by instruction

    public abstract List<Temp> def();
    // Temps “written” by instruction

    public abstract List<Label> jumps();
    // jump targets (or null if not a jump)

    ...
```
Assembly Code Representation

- Uses Temps freely:
  - like infinitely many “fictional” registers.
- Eventually, we decide what actual registers to use
  - This is called register allocation
- Before we can do register allocation we must understand the usage of Temps in the code
  - When are values created in each Temp
  - When is a value in a Temp used for the last time
  - This is called liveness analysis
Overview of the next little while

Code Generation

- Assembly with Temps
- Flow Graph: directed graph
  - Nodes = Assem instructions
  - Edges = Transfer of control
- Liveness: for each node in flowgraph:
  - Compute which Temps are “live”
    - just before that instruction
    - just after that instruction
- Interference Graph: undirected graph
  - Nodes: Temps
  - Edge: $t_1 \leftrightarrow t_2$: $t_1$ and $t_2$ cannot be allocated to the same register
- Register allocation (by Graph Colouring)
Overview of the next little while

**Code Generation**

- Assembly with Temps

**Flow Graph**: directed graph
- Nodes = Assembly instructions
- Edges = Transfer of control

**Liveness**: for each node in flowgraph:
- Compute which Temps are “live”
  - just before that instruction
  - just after that instruction

**Interference Graph**: undirected graph
- Nodes: Temps
- Edge: t1 <-> t2 : t1 and t2 can not be allocated to same register

**Register allocation (by Graph Colouring)**

----

Liveness Analysis (Chapter 10)
Example:

\[
\begin{align*}
movq & \quad \$0, \ t1 \\
L1: & \quad movq \ t1, \ t2 \\
addq & \quad t1, \ t2 \\
addq & \quad \text{t2, t3} \\
imulq & \quad \text{t2, t1} \\
cmpq & \quad \$N, \ t1 \\
jl & \quad L1 \\
movq & \quad \text{t3, } \%rax
\end{align*}
\]
Control Flow Graph

• Not going to spend much time explaining how to construct the flow graph
  – It is pretty straightforward:
    • create a node for each Assem instruction
    • add edges based on the “jumps()” information in the Assem instructions

• Note:
  – In our approach each instruction becomes a flow node.
  – In a more sophisticated implementation
    • basic blocks as flow nodes
    • more efficient for many algorithms (fewer nodes)
Liveness

- Definition: A variable (or Temp) is live at some point in a program if it holds a value that may still be used in the future.

- Determine the liveness of Temps t₁, t₂, and t₃ in the program on the right.
  - What variables are live just before/after each instruction?
  - How do instructions affect liveness?

1: movq $0, t₁
2: movq t₁, t₂
3: addq $1, t₂
4: addq t₂, t₃
5: imulq $2, t₂, t₁
6: cmpq $N, t₁
7: jl L₁
8: movq t₃, %rax

Liveness Analysis (Chapter 10)
Liveness exercise

1: movq $0, t1
2: movq t1, t2
3: addq $1, t2
4: addq t2, t3
5: imulq $2, t2, t1
6: cmpq $N, t1
7: jl L1
8: movq t3, %rax
Flow Graph Terminology

pred[n] = \{ p1, p2 \}

succ[n] = \{ s1, s2 \}
Flow Graph Terminology

• The nodes in the graph are Assembly instructions
• They have "def" and "use" information attached to them
• Example:
  – def(4) = { t3 }  
  – use(4) = { t2, t3 }
Dataflow Equations

• Let’s define the following notations:

\[
\text{in}[n] = \{ x | x \text{ is live just before the execution of } n \} \\
\text{out}[n] = \{ x | x \text{ is live just after the execution of } n \}
\]

• What is the relationship between in[n] and out[n] for a given instruction.
  • In other words, how does an instruction affect the liveness of variables?

• What is the relationship between in and out sets of a node and its predecessors and successors?

• Let’s start with some examples...