Administration

• Announcements
  – Final exam is scheduled: Apr 18 2018, 7:00pm

• Minijava project
  – Phase 3 due next Wednesday
• 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?

Liveness Analysis (Chapter 10)
Flow Graph Terminology

\[ \text{pred}[n] = \{ p1, p2 \} \]
\[ \text{succ}[n] = \{ s1, s2 \} \]
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:

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

• 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...
Dataflow Equations

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

- What is the relationship between \text{in}[n] and \text{out}[n] for a given instruction.

\[
\begin{align*}
\text{addq} & \quad t2, t3 & \text{use} & = \{ \} \\
& & \text{def} & = \{ \} \\
t3 & = t3 + t2 & \text{in} & = \{ \} \\
& & \text{out} & = \{ \}
\end{align*}
\]
Dataflow Equations

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

- What is the relationship between \text{in}[n] and \text{out}[n] for a given instruction.

\text{imulq} \ $2, t2, t1 \quad \text{use} = \{ \}
\text{def} = \{ \}
\text{t1} = \$2 \ast t2 \quad \text{in} = \{ \}
\text{out} = \{ \}

Dataflow Equations

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

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

[1] addq t2, t3
\[ \text{use}[1] = \{ \text{t2, t3} \} \]
\[ \text{def}[1] = \{ \text{t3} \} \]
\[ \text{use}[2] = \{ - \} \]
\[ \text{def}[2] = \{ \text{t3} \} \]
\[ \text{out}[1] = \{ \} \]
\[ \text{in}[2] = \{ \} \]
Dataflow Equations

• What is the relationship between in[n] and out[n] for a given instruction.

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

• What is the relationship between $\text{in}[n]$ and $\text{out}[n]$ for a given instruction.

$$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$$

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

$$\text{out}[n] = \bigcup_{s \text{ in } \text{succ}[n]} \text{in}[s]$$
Solving Dataflow Equations

• We can solve these equations with an iterative approach:
  1. Start with empty sets for in\([n]\) and out\([n]\) for all nodes in the graph
  2. (Re)compute new in\([n]\) and out\([n]\) using the equations.
  3. If there was a change in any in\([n]\) or out\([n]\), repeat step 2
Solving Dataflow Equations

- Does this algorithm work?
  - Does it terminate?
  - If it terminates, does it give a correct solution?
  - If there are multiple solutions, which one does it give?
Formally

• The algorithm is correct because:
  – the equations are “monotonic” functions (on sets).
  – there is an upper bound (on the size of the sets, because there are only so many Temps in program)
  – On each iteration the sets can only grow, not shrink.
  – Because there is an upper bound the algorithm must terminate!

• If the algorithm terminates, recomputing the sets had “no effect.”
  – This means that the equations are satisfied!
Monotonicity?

- **Definition:**
  - A function $f$ is monotonic if, for all $x, y$:
    - $x \leq y \implies f(x) \leq f(y)$
  - In English: “Making the input bigger can only make the output bigger”

- **For our equations we have two functions:**
  - The two equations for the in and out sets.
    - in[$x$] = ...
    - out[$x$] = ...

- The $\leq$ relation is the subset relation.
Formally

• The solution that the algorithm finds is the “least possible” solution.
  – This is what you want, larger solutions include “live” variables that are not really live.
  – Least fixed point
  – Can you think of a “bigger” solution that is also a solution?
Dynamic vs. Static Liveness

1: \texttt{imulq t2, t2, t1}

2: \texttt{movq t2, t3}

3: \texttt{addq t1, t3}

4: \texttt{t3 >= t2}

5: \texttt{movq t1, %eax}

6: \texttt{movq t3, %eax}

What is really live after instr 4?
If we are going to be sloppy (inaccurate), which direction is ok?
Implementing Liveness Analysis

- The iterative algorithm is relatively straightforward to implement.
- It is also possible to implement analysis “one temp at a time”.
- In Java, we can encapsulate the iterative approach in a data structure.
“Active” Sets

• An active set is a set of elements. The elements in the set may be computed from, or dependent on, other active sets, including the set itself.

• The set is called “active” because the implementation is done in such a way that if any of the dependent sets change (get more or fewer elements) the active set is also automatically updated.

• We can use these sets to solve data flow equations in a straight-forward way.
  – Assuming the equations are monotonic and the sets are finite.
Active Sets are just Sets

ActiveSet<Integer> oneTwo = new ActiveSet<Integer>();
oneTwo.add(1);
oneTwo.add(2);
oneTwo.add(2);
oneTwo.add(2);

System.out.println("oneTwo = "+oneTwo);

Prints:

oneTwo = ActiveSet { 2, 1 }
Active Sets are “active” (always up to date)

ActiveSet<Integer> oneTwo = new ActiveSet<Integer>();
oneTwo.add(1);

ActiveSet<Integer> threeFour = new ActiveSet<Integer>();

ActiveSet<Integer> allFour = new ActiveSet<Integer>();
allFour.addAll(oneTwo); // addAll creates a dependency
allFour.addAll(threeFour);

oneTwo.add(2);
threeFour.add(3); threeFour.add(4);

System.out.println("oneTwo = "+oneTwo);
System.out.println("threeFour = "+threeFour);
System.out.println("allFour = "+allFour);

Prints:
   oneTwo = ActiveSet { 2, 1 }
   threeFour = ActiveSet { 4, 3 }
   allFour = ActiveSet { 4, 3, 2, 1 }
ActiveSet<Integer> oneTwoThree = new ActiveSet<Integer>();
ActiveSet<Integer> oneTwo = oneTwoThree.remove(list(3));

oneTwoThree.add(1);
oneTwoThree.add(2);
oneTwoThree.add(3);

System.out.println("oneTwo = "+oneTwo);
System.out.println("oneTwoThree = "+oneTwoThree);

Prints:

oneTwo = ActiveSet { 2, 1 }
oneTwoThree = ActiveSet { 3, 2, 1 }
Circular Dependencies are OK!

Problem:

Solve the following equations

\[ A = \{1,2\} \cup (B - \{4\}) \]
\[ B = \{3,4\} \cup (C - \{5\}) \]
\[ C = \{4,5\} \cup A \]

Note: These equations are very similar to the dataflow equations for liveness analysis.

Exercise: Write a piece of Java code that uses our ActiveSet implementation to solve these equations and print out the result.
Circular Dependencies are OK!

```java
ActiveSet<Integer> a = new ActiveSet<Integer>();
ActiveSet<Integer> b = new ActiveSet<Integer>();
ActiveSet<Integer> c = new ActiveSet<Integer>();

// A = \{1,2\} U ( B \setminus \{4\} )
a.add(1); a.add(2);
a.addAll(b.remove(list(4)));

// B = \{3,4\} U (C \setminus \{5\})
b.add(3); b.add(4);
b.addAll(c.remove(list(5)));

// C = \{4,5\} U A
c.add(4); c.add(5);
c.addAll(a);

// print result:
System.out.println("A = " + a);
System.out.println("B = " + b);
System.out.println("C = " + c);
```

Prints:

A = ActiveSet { 3, 2, 1 }
B = ActiveSet { 1, 2, 4, 3 }
C = ActiveSet { 1, 2, 3, 5, 4 }

Liveness Analysis (Chapter 10)
Using ActiveSets to implement liveness analysis

This is up to you... for the project :)

You can also implement liveness using the iterative approach that we discussed previously.