Assignment 2: Lexing, Grammars, and Dataflow Analysis

15-411: Compiler Design
Ian Gillis (igillis) and Alex Crichton (acrichto)

Due: Tuesday, September 25, 2012 (1:30 pm)

Reminder: Assignments are individual assignments, not done in pairs. The work must be all your own.

You may hand in a handwritten solution solution at the beginning of lecture on Tuesday, September 25. Please read the late policy for written assignments on the course web page. If you decide not to typeset your answers, make sure the text and pictures are legible and clear.

Problem 1 (20 points)

The following table defines the tokens for a simple language with $\Sigma = \{[a-z], [0-9], ., \oplus\}$.

token	regular expression
$\langle num \rangle$	≡ 0 | [1-9][0-9]*
$\langle var \rangle$	≡ [a-z][a-z0-9]*
$\langle lam \rangle$	≡ lam
$\langle dot \rangle$	≡ .
$\langle binop \rangle$	≡ ⊕

(a) Construct a DFA which accepts tokens from this language. You may draw the states and transitions graphically or use the notation described in the lecture notes.

Here is the grammar for the language whose tokens were defined in part a.

$$\begin{array}{llcll}
\gamma_1 & : & \langle \mathrm{E} \rangle & \rightarrow & \mathrm{x} \\
\gamma_2 & : & \langle \mathrm{E} \rangle & \rightarrow & \overline{n} \\
\gamma_3 & : & \langle \mathrm{E} \rangle & \rightarrow & \mathrm{lam\ x\ .}\ \langle \mathrm{E} \rangle \\
\gamma_4 & : & \langle \mathrm{E} \rangle & \rightarrow & \langle \mathrm{E} \rangle\ \langle \mathrm{E} \rangle \\
\gamma_5 & : & \langle \mathrm{E} \rangle & \rightarrow & \oplus\ \langle \mathrm{E} \rangle \\
\gamma_6 & : & \langle \mathrm{E} \rangle & \rightarrow & \langle \mathrm{E} \rangle \oplus \langle \mathrm{E} \rangle
\end{array}$$

(b) This grammar is ambiguous. Give an example of an ambiguous expression along with two (or more) ways which it can be parsed.

(c) What additional information or rules are needed to fully specify this language? Make a design decision about the language and write down the disambiguating information, then update your parsing table based on your new specification.

Problem 2 (20 points)

One useful dataflow analysis is that of "available expressions." We will say that an expression e is available at a point l in the program if:

- e is computed on every path to l
- the value of e has not changed since it was last computed on a path to l

For example, the following code:

```
l1: t0 <- t1 + t2
l2: t1 <- t2 * t4
```

Has {`t1 + t2`} available at `l1`, and {`t2 * t4`} available at `l2`. The first expression is eliminated because its component, `t1` changed with the result of `l2`.

We can use this information to eliminate redundant computation of expressions. In this problem you will practice performing dataflow analysis on a small amount of code. Note that this is not the entire program in the sense that not all temps are assigned before they are used; you should assume that these have been initialized by previous code.

```
    a <- b + c
    jmpnzero 1 d
    b <- e + 3
    a <- b + c
    label 1
    t1 <- e + 3
    t2 <- b + c
    return t1 + t2
```

(a) Compute the available expression(s) at each instruction.

(d) Use the results of part (a) to give a (slighty) optimized version of the program snippet.

Problem 3 (20 points)

The lecture notes on liveness analysis and dataflow analysis mention the dataflow equation forms of both liveness and reaching definition anylysis. Recall:

- $A_\bullet(l)$ represents the result of the analysis just after statement l.
- $A_\circ(l)$ represents the result of the analysis just before statement l.
- $l \mapsto l'$ means that l' is a successor of l.

For reaching definitions, a forward analysis, we used the equations:

$$A_\circ(l) = \bigcup_{l_i \mapsto l} A_\bullet(l_i)$$

$$A_\bullet(l) = (A_\circ(l) \setminus \mathsf{kill}(l)) \cup \mathsf{gen}(l)$$

For liveness, which is a backward analysis, we instead used:

$$A_\bullet(l) = \bigcup_{l \mapsto l_i} A_\circ(l_i)$$

$$A_\circ(l) = (A_\bullet(l) \setminus \mathsf{kill}(l)) \cup \mathsf{gen}(l)$$

In the assembly language from Problem 1, for reaching definitions, kill and gen are defined as:

	Reaching Definitions		Liveness	
statement l	$\mathsf{gen}(l)$	$\mathsf{kill}(l)$	$\mathsf{gen}(l)$	$\mathsf{kill}(l)$
$l : x \leftarrow a + b$	$\{l\}$	$\{l \mid \mathsf{def}(l, x)\}$	$\{a, b\}$	$\{x\}$
$l : x \leftarrow a - b$	$\{l\}$	$\{l \mid \mathsf{def}(l, x)\}$	$\{a, b\}$	$\{x\}$
$l : x \leftarrow a * b$	$\{l\}$	$\{l \mid \mathsf{def}(l, x)\}$	$\{a, b\}$	$\{x\}$
$l : x \leftarrow a/b$	$\{l\}$	$\{l \mid \mathsf{def}(l, x)\}$	$\{a, b\}$	$\{x\}$
l : jmpnzero l' x	$\emptyset$	$\emptyset$	$\{x\}$	$\emptyset$
l : jmp l'	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
l : return x	$\emptyset$	$\emptyset$	$\{x\}$	$\emptyset$
l : label l'	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

We solve these equations by initializing $A_\bullet(l)$ to $\{\}$ for each l and then stepping the equations until they saturate.

The interesting aspect of this approach is that it is very general. Changing the definitions of gen and kill will clearly yield a different analysis. Similarly, we might replace set union with set intersection to combine the output from previous statements. Finally, reaching definition analysis runs forward; where we used $l_i \mapsto l$ in reaching analysis, we can use $l \mapsto l_i$ to do backwards-flowing analysis. By changing these parameters, we can use the same algorithm to compute a variety of interesting analyses. In this problem, you will develop these parameters for availability analysis.

(a) Is availability a forward analysis or a backward analysis? Why?

(b) Is the set operation to combine analysis from other statements a union or an intersection? Why?

(c) Write down the table defining gen and kill for each instruction. You may use operations which you define outside of the table if you find that easier.