Name:

Midterm 2

CMSC 430

Introduction to Compilers

Spring 2012

April 18, 2012

Instructions

This exam contains 10 pages, including this one. Make sure you have all the pages. Write your name on the top of this page before starting the exam.

Write your answers on the exam sheets. If you finish at least 15 minutes early, bring your exam to the front when you are finished; otherwise, wait until the end of the exam to turn it in. Please be as quiet as possible.

If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

Question	Score	Max
1		28
2		15
3		22
4		35
Total		100

Question 1. Short Answer (28 points).

a. (7 points) Explain briefly what *bitvectors* are used for in data flow analysis, and why they may be worth using.

> **Answer:** Bitvectors are an efficient representation of sets of data flow facts in which each fact is represented by a bit in the vector. Bitvector union and intersection operations are done using bitwise-or and -and, respectively, and so they are much faster (approximately $32\times$ or $64\times$) than representations in which each fact is, say, a node in a data structure.

b. (7 points) Explain briefly what an *activation record* is, and list four items an activation record is likely to contain.

> **Answer:** An activation record provides local storage for a function invocation. Activation records may include the function's parameters, the return address it should jump to on exit, and space for the return value, saved registers, caller's frame pointer (or activation record pointer), and the callee's local variables.

c. (7 points) In class, we discussed some recent tools that use dataflow analysis to find bugs in programs (rather than for optimization). Explain what *false positives* and *false negatives* are in this context. (You will not lose any points if you swap the definitions of these terms.)

> **Answer:** False positives are bug reports (generated by the tools) that do not correspond to actual bugs. False negatives are bugs in the programs whose presence is not detected by the tools.

d. (7 points) Explain in at most 3 sentences what *dynamic software updating* is.

> **Answer:** Dynamic software updating allows programs to be updated on-the-fly with new code and data representations, without shutting the program down or restarting.

Question 2. Definite assignment analysis (15 points). A variable is *definitely assigned to* at a program point if it is guaranteed to have been be written to (possibly more than once) on all paths from the start of the program to that point.

a. (4 points) What C programming mistake could we use definite assignment analysis to detect?

Answer: Local variables that are read but not initialized.

a. (8 points) Suppose we want to implement definite assignment analysis as a data flow analysis.

i. What is the *direction* of this analysis?

Answer: Forward

ii. Is this a *may* or a *must* analysis?

Answer: Must

iii. What are `entry_or_exit_facts` for this analysis, in general terms?

Answer: The empty set

iv. What are the `initial_facts` for this analysis, in general terms?

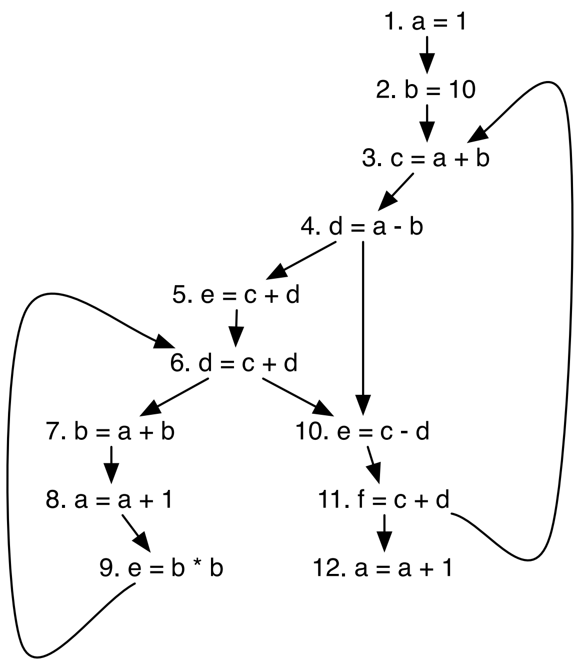
Answer: The set of all variables used in the program.

b. (3 points) Write down Gen and Kill sets for the following statements for definite assignment analysis.

Statement	Gen	Kill
`a = b`	a	∅
`a = b + c`	a	∅
`a = a + 1`	a	∅

This page intentionally left blank.

Question 3. Data flow analysis (22 points). Consider the following control-flow graph.



a. (5 points) Write down the Gen/Kill sets for **live variable analysis** for the following statements in the control flow graph. Write ∅ for the empty set.

Statement	Gen	Kill
1	∅	a
3	a,b	c
6	c,d	d
8	a	a
9	b	e

b. (12 points) For each statement in the control-flow graph, show the results of **available expressions**, i.e., show the set of expressions that are available at the *end* of the statement. Write ∅ for the empty set.

Statement	Expressions
1	∅
2	∅
3	a+b
4	a+b, a-b
5	a+b, a-b, c+d
6	∅
7	∅
8	∅
9	b*b
10	c-d
11	c-d, c+d
12	c-d, c+d

c. (5 points) Write down the set of statements s such that the **definition** at statement 4 **may reach** the end of statement s. (Don't forget to include statement 4 in your list.)

Answer: 3, 4, 5, 10, 11, 12

Question 4. Code generation, register allocation, and optimization (35 points). Below is a snippet of `codegen-2.ml` from class, showing the input expression language, the "bytecode" instruction language, and compilation. (Here we've renamed `‘L_Register` to `‘L_Reg` to save some writing.)

```
type expr =
  | EInt of int
  | EAdd of expr * expr
  | EMul of expr * expr
  | EId of string
  | EAssn of string * expr
  | ESeq of expr * expr
  | EIfZero of expr * expr * expr

type reg = [ ‘L_Reg of int ]
type src = [ ‘L_Int of int | ‘L_Ptr of int ]
type dst = [ ‘L_Ptr of int ]

type instr =
  | ILoad of reg * src         (* dst, src *)
  | IStore of dst * reg        (* dst, src *)
  | IAdd of reg * reg * reg (* dst, src1, src2 *)
  | IMul of reg * reg * reg (* dst, src1, src2 *)
  | IIfZero of reg * int       (* guard, target *)
  | IJmp of int                (* target *)
  | IMov of reg * reg          (* dst, src *)

let rec comp_expr (st:(string*int) list) = function
  | EInt n →
      let r = next_reg () in
        (r, [ILoad (‘L_Reg r, ‘L_Int n)])
  | EIfZero (e1, e2, e3) →
      let (r1, p1) = comp_expr st e1 in
      let (r2, p2) = comp_expr st e2 in
      let (r3, p3) = comp_expr st e3 in
      let r = next_reg () in
        (r, p1 @
           [IIfZero (‘L_Reg r1, (2+(List.length p3)))] @
           p3 @
           [IMov (‘L_Reg r, ‘L_Reg r3);
            IJmp (1+(List.length p2))] @
           p2 @
           [IMov (‘L_Reg r, ‘L_Reg r2)]
        )
```

a. (10 points) Suppose we extended the source language with a *while loop*:

```
type expr = ... | EWhile of expr * expr
```

Here, EWhile(e1,e2) has the usual semantics, where `e1` is the loop guard—which is considered false if `0` and true otherwise—and `e2` is the loop body. Loops should be evaluated for their side effects only, and no matter how many times (0 or more) the loop body is evaluated, all while loops should return the value `0`.

Write a case of comp_expr that handles while loops. (Hint: recall that the `IJmp` and `IIfZero` instructions add their offset to the program counter for the *following* instruction; offsets may be negative.)

```
| EWhile (e1, e2) →
```

Answer:

```
let (r1, p1) = comp_expr st e1 in
let (r2, p2) = comp_expr st e2 in
let r = next_reg () in
  (r, p1 @
     [IIfZero (‘L_Reg r1, (1 + List.length p2))] @
     p2 @
     [IJmp (− ((List.length p2) + (List.length p1) + 2))] @
     [ILoad (‘L_Reg r, ‘L_Int 0)]
  )
```

b. (15 points) Next, consider the bytecode program in the left column below.

In the second column below, list the numbers of registers that are live *after* each instruction—we've given a couple of examples to get you started.

Now suppose that we have three phyiscal registers available, *ra*, *rb*, and *rc*. In the rightmost column group below, assign each virtual register to a physical register, and indicate what virtual registers you need to *spill after* executing an instruction or *load before* executing an instruction. For example, if you had an instruction IAdd(‘L_Reg 0, ‘L_Reg 1, ‘L_Reg 2), you could list 0 in the spill column and 1,2 in the load column if you needed to spill and load all possible registers. (Don't forget to also assign any loaded virtual registers to physical registers.)

You may use any spill policy you choose, but you should try to maximize register usage—don't load and spill all registers with every instruction.

		Reg. Assignment				
Instruction	Live registers	ra	rb	rc	spills	loads
ILoad (‘L_Reg 0, ‘L_Int 1)	0	0				
ILoad (‘L_Reg 1, ‘L_Int 42)	0,1	0	1			
IAdd (‘L_Reg 2, ‘L_Reg 0, ‘L_Reg 1)	0,2	0	1	2		
ILoad (‘L_Reg 3, ‘L_Int 2)	0,2,3	0	3	2		
ILoad (‘L_Reg 4, ‘L_Int 41)	0,2,3,4	4	3	2	0	
IMul (‘L_Reg 5, ‘L_Reg 4, ‘L_Reg 2)	0,2,3,5	5	3	2		
IAdd (‘L_Reg 6, ‘L_Reg 3, ‘L_Reg 5)	0,2,6	5	6	2		
IAdd (‘L_Reg 7, ‘L_Reg 2, ‘L_Reg 6)	0,7	5	6	7		
IAdd (‘L_Reg 8, ‘L_Reg 7, ‘L_Reg 0)	8	8	0	7		0
IStore (‘L_Ptr 3, ‘L_Reg 8)	None					

c. (10 points) For each of the following optimizations, write down a short sequence of `instr` s as they might appear before and after the optimization. You may write `` `L_Reg `` n as $\mathtt{r}n$ and `` `L_Int `` n as n. For example, for dead code elimination, you might write

Before	After
ILoad (r0, 42) ILoad (r0, 13)	ILoad (r0, 13)

i. Common subexpression elimination

Answer:

Before	After
IAdd (r0, r1, r2) IAdd (r3, r1, r2)	IAdd (r0, r1, r2) IAdd (r3, r0)

ii. Constant folding

Answer:

Before	After
ILoad (r0,5) ILoad (r1, r6) IAdd (r2, r0, r1)	ILoad (r0,5) ILoad (r1, r6) ILoad (r2,11)

Note that even something as simple as IAdd (r0, 5, 6) being replaced with ILoad(r0, 11) would be considered constant folding, and it's useful because it may enable further constant propagation. Also, several people combined constant folding with copy propagation (e.g., eliminating the writes to r0 and r1 above). This is technically not quite right, as it combines dead code elimination with constant folding (and assumes that r0 and r1 are dead), but such answers received full credit because of the wording of the question.

iii. Copy propagation

Answer:

Before	After
ILoad (r0, r1) ILoad (r2, r0)	ILoad (r0, r1) ILoad (r2, r1)

Note that copy propagation involves copies of registers; copying constants would be constant folding.