Instructions

- This exam has 6 questions, for a total of 75 points.
- You may spend up to three (continuous) hours on the exam. The final deadline is noon, Tuesday, May 4.
- You should put your answers to all questions into a single plain text or PDF file, called `myname.txt` or `myname.pdf` (where `myname` is your name). Generating a PDF by taking photos of hand-written answers is fine, so long as they are legible.
- Submit your exam by attaching it to an email to `tolmach@pdx.edu` with the subject line “compiler exam”.
- The exam is ”open-book;” you may use the text, your notes, the compiler project code, and any other reference materials you want to complete it. However, you must not collaborate, share answers, or otherwise get help from anyone.

1. [10 points] Compile the following $R_{\text{Var}}$ program to an equivalent program in the $C_{\text{Var}}$ language. Express your answer using the concrete grammar for $C_{\text{Var}}$ shown in Figure 1 (the same format used by CVar.ml when dumping out code for debugging purposes).

   \[
   (\text{let } a \\
   \hspace{1em} (\text{let } b \\
   \hspace{2em} (\text{read}) \\
   \hspace{3em} (+ (+ b 10) 2)) \\
   \hspace{2em} (- a))
   \]

   \[
   \begin{array}{l}
   \text{arg ::= int | var} \\
   \text{exp ::= arg | (read) | (- arg) | (+ arg arg)} \\
   \text{asgn ::= var ::= exp} \\
   \text{$C_{\text{Var}}$ ::= asgn* return exp}
   \end{array}
   \]

   Figure 1: The $C_{\text{Var}}$ intermediate language.

   Solution:

   \[
   b := (\text{read}) \\
   t := (+ b 10) \\
   a := (+ t 2) \\
   \text{return } (- a.1)
   \]

2. [10 points] Suppose we did not run the Uniquify pass before running the ExplicateControl pass. Show why this is a bad idea, by giving an example of a $R_{\text{Var}}$ program that will produce the wrong behavior when translated to $C_{\text{Var}}$. (Be sure to show both the original and translated programs.)
Solution: One of many possible examples:

\[
\begin{align*}
\text{let } a & \text{ 1} \\
\text{let } b & \text{ (let a 2 a)} \\
& \text{a})
\end{align*}
\]

produces 1
\[
\begin{align*}
a & := 1 \\
a & := 2 \\
b & := a \\
\text{return a}
\end{align*}
\]

produces 2
3. [15 points] Given the following code for an x86\textsubscript{int} program written using symbolic variable names, write down the full assembly code for the X86 function obtained by assigning stack locations (not registers!) to the variables $x$, $t1$, and $t2$, in the style of Chapter 2. Your answer should be given in the usual syntax of .s files. Be sure to give the complete function code, including entry and exit sequences, and consisting entirely of legal instructions.

```
start:
callq read_int
movq %rax, x
movq $4, t1
negq t1
movq t1, t2
addq x, t2
movq t2, %rax
jmp conclusion
```

Solution: One possible solution. The assignment of the three variables to the three stack slots is arbitrary.

```
.globl main
main:
pushq %rbp
movq %rsp, %rbp
subq $32, %rsp
callq read_int
movq %rax, -24(%rbp)
movq $4, -8(%rbp)
negq -8(%rbp)
movq -8(%rbp), %rax
movq %rax, -16(%rbp)
movq -24(%rbp), %rax
addq %rax, -16(%rbp)
movq -16(%rbp), %rax
addq $32, %rsp
popq %rbp
retq
```

Rubric:

- prelude (3 points)
- correct use of the stack for $x$, $t1$, and $t2$ (6 points)
- legal instructions: at most one memory argument per instruction (3 points)
- conclusion (3 points)
4. [15 points] For the following $C_{IF}$ program, write down the live-after set for each line containing an assignment (i.e. lines 2,3,4,9,10,14,15,19,20), and the live-before set for each of the four blocks. (Note: although in our compiler we compute liveness information for X86 code, exactly the same ideas can be used to compute liveness for $C_{IF}$ code.)

```
start:
  a := 1
  b := 2
  t3 := (read)
  if t3 = 0 goto block1;
  else goto block2

block1:
  t2 := a
  t3 := (- t2)
  goto block3

block2:
  t3 := b
  t4 := 20
  goto block3

block3:
  x := t3
  t5 := (+ x 10)
  return t5
```

Solution:

Live-afters:

```
2: a
3: a, b
4: a, b, t3
9: t2
10: t3
14: t3
15: t3
19: x
20: t5
```

Live-befores:
```
start: empty
block1: a
block2: b
block3: t3
```
5. [15 points] Consider the following results from liveness analysis on a x86 program using symbolic variable names, where the live-after set is listed next to each instruction.

```
start:
callq read_int ; %rax
movq %rax, x ; x
movq $1, y ; x,y
movq $2, z ; x,y,z
movq y, w ; x,w,z
addq $2, w ; x,w,z
movq z, t ; x,w,t
addq w, t ; t,x
movq t, %rax ; %rax,x
addq x, %rax ; %rax
jmp conclusion
```

(a) Give the interference graph for the variables x, y, z, w, t. (You can ignore %rax.) You can either draw the graph or just list its edges.

(b) What is the minimum number of locations (registers or stack slots) needed to hold the five variables in this code?

<table>
<thead>
<tr>
<th>Solution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) [12 pts] Here are the edge adjacency lists:</td>
</tr>
<tr>
<td>t : w x</td>
</tr>
<tr>
<td>w : t x z</td>
</tr>
<tr>
<td>x : t w y z</td>
</tr>
<tr>
<td>y : x z</td>
</tr>
<tr>
<td>z : w x y</td>
</tr>
<tr>
<td>(b) [3 pts] Three locations are necessary and sufficient. Sufficiency is shown by this assignment: loc1: t,z; loc2: w,y; loc3: x. Necessity follows because the graph contains several fully connected subgraphs (cliques), namely t-w-x, x-w-z and x-z-y.</td>
</tr>
</tbody>
</table>

6. [10 points] Given an existing compiler for a language with binary addition and unary negation operators, there are at least three ways to add support for binary subtraction:

(a) Translate subtraction into negation followed by addition during parsing of the concrete program, as we did in Chapter 2.

(b) Introduce subtraction as a primop in the RVar AST, but translate it into negation followed by addition during a separate Shrink pass, as we did in Chapter 4.

(c) Introduce subtraction as a primop in both RVar and CVar, and translate it directly to a subq instruction in X86 assembly code.

Briefly compare the advantages and disadvantages of these three approaches.
Solution:

(a) requires the minimum amount of change in the existing compiler, but can lead to confusing error messages because there is no reliable way to translate negation followed by addition back into subtraction. (b) avoids the error message problem, but adds (slight) complication to the initial $R_{\text{var}}$ AST; the new constructor should not appear after Shrinking, but enforcing this invariant may be tedious. (c) allows us to produce shorter (and probably faster) code sequences for subtraction, but requires changes in all passes; some of these changes will be more than cut-and-paste from the addition case, because subtraction is not commutative.