CS410P/510 Programming Language Compilation Practice Midterm

Name: _____

Instructions

- This exam has 6 questions, for a total of 80 points.
- You may spend up to 1 hour, 50 minutes (110 minutes) on the exam.
- The exam is closed-book, closed-notes, except that one 8.5"x11" single-sided sheet of handwritten notes is permitted.
- No computing devices (laptops, tablets, cell phones, etc.) may be used.

Concrete syntax for all the intermediate languages mentioned in the exam can be found on the last two pages.

1. [10 points] Compiler Structure

   The compiler for the language with booleans and conditionals in Chapter 5 involves multiple intermediate and target languages, namely: $\mathcal{L}_{\mathsf{If}}$ (the source language for the whole compiler), $\mathcal{L}_{\mathsf{If}}^{mon}$, $\mathcal{C}_{\mathsf{If}}$, $\text{x86}_{\mathsf{If}}^{\mathsf{Var}}$, and x86_{If} (the target language for the whole compiler).

   The compiler involves multiple passes, which are listed here (with abbreviations) in no particular order:

   ```
   prelude_and_conclusion (P&C)
   shrink (S)
   select_instructions (SE)
   explicate_control (EC)
   remove_complex_operands (RCO)
   patch_instructions (PI)
   assign_homes (AH)
   ```

   Draw a diagram that shows the order in which the passes actually execute, and indicates which language is the source and target of each pass. (Use the abbreviations to save writing.)

2. [15 points] Compile the following $\mathcal{L}_{\mathsf{lf}}$ program to an equivalent program in the $\mathcal{L}_{\mathsf{if}}^{mon}$ language.

```
a = input_int()
b = 3 + ((- a) - 7)
c = 42 if (b < 10) else (a + input_int())
print(c)
```

3. [15 points] Given the following code for the body of an $\text{x86}_{\text{If}}^{\text{Var}}$ program written using symbolic variable names, write down the full assembly code for the x86_{If} program obtained by assigning distinct `%rbp`-relative stack locations (not registers!) to the variables `x`, `t0`, and `t1`, in the style of Chapter 2. Your answer should be in the form of a single `main` function definition, given in the syntax of x86_{If}, i.e. the usual assembler syntax of .s files. Be sure to give the *complete* function code, including entry and exit sequences, and consisting entirely of *legal* instructions.

```
callq _read_int
movq %rax, x
movq $-7, t0
movq t0, t1
addq x, t1
movq t1, %rdi
callq _print_int
```

4. [10 points] Recall that numeric comparisons on the X86 are perfomed by setting the condition codes (typically using a `cmpq` instruction) and then testing them using one of the `set`*cc* or `j`*cc* instructions. Our compiler finds it useful to generate both `set`*cc* and `j`*cc* instructions in different situations.

   Illustrate why, by giving a *short* $\mathcal{L}_{\mathsf{If}}$ source program fragment and its translation into $\text{x86}_{\mathsf{If}}^{\mathsf{Var}}$, where the translated program uses both kinds of *cc*-testing instructions.

5. [15 points] For the following $\mathcal{C}_{\mathsf{If}}$ program, fill in the live-after and live-before sets at each specified point in the program. (Note: although in our compiler we compute liveness information for X86 code, exactly the same ideas can be used to compute liveness for $\mathcal{C}_{\mathsf{If}}$ code.)

```
                              live-before =
start:
  a = 1
                              live-after =
  b = 2
                              live-after =
  t3 = input_int()
                              live-after =
  if t3 == 0: goto block1
  else: goto block2

                              live-before =
block1:
  t2 = a
                              live-after =
  t3 = -t2
                              live-after =
  goto block3

                              live-before =
block2:
  t3 = b
                              live-after =
  t4 = 20
                              live-after =

  goto block3

                              live-before =
block3:
  x = t3
                              live-after =
  t5 = x + 10
                              live-after =
  print(t5)
                              live-after =
  return 0
```

6. [15 points] Consider the following results from liveness analysis on a $\text{x86}_{\mathsf{Var}}$ 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) Draw the interference graph for the variables `x,y,z,w,t`. (You can ignore `%rax`.)

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

[This page deliberately left blank.]

Concrete Syntax of Languages

$\mathcal{L}_{\mathsf{If}}$

$$
\begin{array}{rcl}
\mathit{cmp} & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\mathit{exp} & ::= & \mathit{int} \mid \mathit{bool} \mid \mathit{var} \\
 & \mid & \texttt{input\_int()} \mid \texttt{-}\ \mathit{exp} \mid \texttt{not}\ \mathit{exp} \mid \mathit{exp}\ \texttt{+}\ \mathit{exp} \mid \mathit{exp}\ \texttt{-}\ \mathit{exp} \\
 & \mid & \mathit{exp}\ \texttt{and}\ \mathit{exp} \mid \mathit{exp}\ \texttt{or}\ \mathit{exp} \mid \texttt{(}\mathit{exp}\texttt{)} \\
 & \mid & \mathit{exp}\ \mathit{cmp}\ \mathit{exp} \mid \mathit{exp}\ \texttt{if}\ \mathit{exp}\ \texttt{else}\ \mathit{exp} \\
\mathit{stmt} & ::= & \texttt{print(}\mathit{exp}\texttt{)} \mid \mathit{exp} \mid \mathit{var}\ \texttt{=}\ \mathit{exp} \mid \texttt{if}\ \mathit{exp}\texttt{:}\ \mathit{stmt}^{+}\ \texttt{else:}\ \mathit{stmt}^{+} \\
\mathcal{L}_{\mathsf{If}} & ::= & \mathit{stmt}^{*}
\end{array}
$$

$\mathcal{L}_{\mathsf{if}}^{mon}$

$$
\begin{array}{rcl}
\mathit{atm} & ::= & \mathit{int} \mid \mathit{bool} \mid \mathit{var} \\
\mathit{cmp} & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{input\_int()} \mid \texttt{-}\ \mathit{atm} \mid \texttt{not}\ \mathit{atm} \mid \mathit{atm}\ \texttt{+}\ \mathit{atm} \mid \mathit{atm}\ \texttt{-}\ \mathit{atm} \\
 & \mid & \mathit{atm}\ \mathit{cmp}\ \mathit{atm} \mid \mathit{exp}\ \texttt{if}\ \mathit{exp}\ \texttt{else}\ \mathit{exp} \mid \{\mathit{stmt}^{*}\ \texttt{produce(}\mathit{exp}\texttt{)}\} \\
\mathit{stmt} & ::= & \texttt{print(}\mathit{atm}\texttt{)} \mid \mathit{exp} \mid \mathit{var}\ \texttt{=}\ \mathit{exp} \mid \texttt{if}\ \mathit{exp}\texttt{:}\ \mathit{stmt}^{+}\ \texttt{else:}\ \mathit{stmt}^{+} \\
\mathcal{L}_{\mathsf{if}}^{mon} & ::= & \mathit{stmt}^{*}
\end{array}
$$

Note: the concrete expression $\{\mathit{stmt}^{*}\ \texttt{produce(}\mathit{exp}\texttt{)}\}$ corresponds to the AST form $\texttt{Begin(}\mathit{stmt}^{*}\texttt{,}\ \mathit{exp}\texttt{)}$.

$\mathcal{C}_{\mathsf{If}}$

$$
\begin{array}{rcl}
\mathit{atm} & ::= & \mathit{int} \mid \mathit{bool} \mid \mathit{var} \\
\mathit{cmp} & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{input\_int()} \mid \texttt{-}\ \mathit{atm} \mid \texttt{not}\ \mathit{atm} \mid \mathit{atm}\ \texttt{+}\ \mathit{atm} \mid \mathit{atm}\ \texttt{-}\ \mathit{atm} \\
 & \mid & \mathit{atm}\ \mathit{cmp}\ \mathit{atm} \\
\mathit{stmt} & ::= & \texttt{print(}\mathit{atm}\texttt{)} \mid \mathit{exp} \mid \mathit{var}\ \texttt{=}\ \mathit{exp} \\
\mathit{tail} & ::= & \texttt{return}\ \mathit{exp} \mid \texttt{goto}\ \mathit{label} \mid \texttt{if}\ \mathit{atm}\ \mathit{cmp}\ \mathit{atm}\texttt{:}\ \texttt{goto}\ \mathit{label}\ \texttt{else:}\ \texttt{goto}\ \mathit{label} \\
\mathcal{C}_{\mathsf{If}} & ::= & (\mathit{label}\texttt{:}\ \mathit{stmt}^{*}\ \mathit{tail})\ldots
\end{array}
$$

$\text{x86}_{\mathsf{If}}^{\mathsf{Var}}$

```
      reg ::= rsp | rbp | rax | rbx | rcx | rdx | rsi | rdi |
              r8 | r9 | r10 | r11 | r12 | r13 | r14 | r15
  bytereg ::= ah | al | bh | bl | ch | cl | dh | dl
      arg ::= $int | %reg | %bytereg | int(%reg) | var
       cc ::= e | ne | l | le | g | ge
    instr ::= addq arg,arg | subq arg,arg | negq arg | movq arg,arg
            | pushq arg | popq arg | callq label | retq
            | xorq arg, arg | cmpq arg, arg | setcc arg | movzbq arg, arg
            | jmp label | jcc label | label: instr
x86_If^Var ::= .globl main
              main: instr ...
```

Note: this is the same as x86_{If}, below, except that *var* is allowed as an *arg*.

x86_{If}

```
      reg ::= rsp | rbp | rax | rbx | rcx | rdx | rsi | rdi |
              r8 | r9 | r10 | r11 | r12 | r13 | r14 | r15
  bytereg ::= ah | al | bh | bl | ch | cl | dh | dl
      arg ::= $int | %reg | %bytereg | int(%reg)
       cc ::= e | ne | l | le | g | ge
    instr ::= addq arg,arg | subq arg,arg | negq arg | movq arg,arg
            | pushq arg | popq arg | callq label | retq
            | xorq arg, arg | cmpq arg, arg | setcc arg | movzbq arg, arg
            | jmp label | jcc label | label: instr
x86_If ::= .globl main
           main: instr ...
```