CS410P/510 Programming Language Compilation — Practice Final

Instructions

- This exam has 5 questions, for a total of 75 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 languages mentioned in the exam can be found on the last two pages.

1. [20 points] Loops.

   Consider the following $\text{x86}^{\mathsf{Var},\mathsf{Def}}_{\mathsf{callq}*}$ program.

```
.globl main
                              live-before =
main:
        movq    $0, a
                              live-after =
        movq    $0, b
                              live-after =
        movq    $0, x
                              live-after =
        jmp     block1

                              live-before =
block1:
        cmpq    $4, x
                              live-after =
        jl      block2

        jmp     block3

                              live-before =
block2:
        movq    b, a
                              live-after =
        addq    $14, b
                              live-after =
        movq    $1, c
                              live-after =
        addq    c, x
                              live-after =
        jmp     block1

                              live-before =
block3:
        movq    a, %rax
                              live-after =
        retq
```

(a) [5 points] Draw the control-flow graph for this program.

Solution:

```
vertices: main, block1, block2, block3
edges: main -> block1
       block1 -> block2
       block1 -> block3
       block2 -> block1
```

(b) [10 points] On the previous page, fill in the live-after and live-before sets at each specified point in the program. Ignore `%rax`, `%rsp`, and `%rbp`.

Solution:

```
  .globl main
                              live-before = {}
  main:
          movq    $0, a
                              live-after = a
          movq    $0, b
                              live-after = a,b
          movq    $0, x
                              live-after = a,b,x
          jmp     block1

                              live-before = a,b,x
  block1:
          cmpq    $4, x
                              live-after = a,b,x
          jl      block2

          jmp     block3

                              live-before = b,x
  block2:
          movq    b, a
                              live-after = a,b,x
          addq    $14, b
                              live-after = a,b,x
          movq    $1, c
                              live-after = a,b,x,c
          addq    c, x
                              live-after = a,b,x
          jmp     block1

                              live-before = a
  block3:
```

```
        movq    a, %rax
                                live-after = {}
        retq
```

(c) [5 points] The algorithm used in Chapter 5 to compute liveness information for code generated from $\mathcal{L}_{\mathsf{If}}$ programs has the nice property that it considers each block and each instruction just once. Explain, briefly but clearly, why that algorithm does not work for programs like this one.

Solution:
In programs with loops, the liveness analyses for different blocks can be mutually dependent, so no single pass through the blocks can suffice.

2. [10 points] Heap.

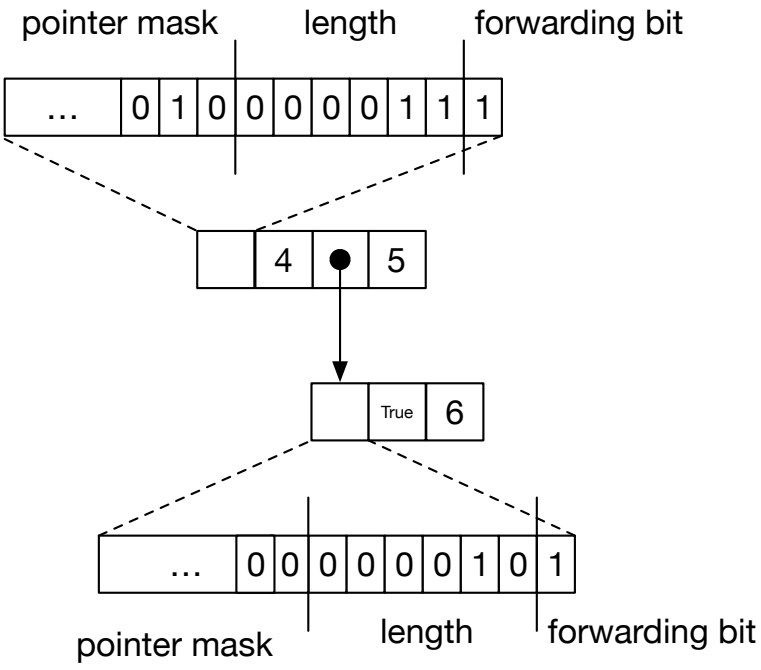
   Draw a graph that represents the heap as it would look just after the following $\mathcal{L}_{\mathsf{Tup}}$ expression is evaluated.

   ```
   (4,(True,6),5)
   ```

   The nodes in your graph are allocated objects (draw them as rectangles) and the directed edges are pointers. The rectangles should be subdivided into a box for each 64-bit element. If a box represents a pointer, then draw an arrow from it to the object it points to. If the box represents an integer or Boolean, simple write the value in the box. If the box contains tag information, then show the individual bits and annotate to describe which bits represent what.

   Reminder: The least-significant bit of the tag is the forwarding bit; the next 6 bits are the length (in binary); the remaining bits are the pointer mask.

Solution:



3. [15 points] Root Stack.

   (a) [5 points] Briefly but clearly explain the purpose of the root stack and what is stored in it.

   Solution: The purpose of the root stack is to make live pointers into the heap visible to the garbage collector. Any variable that holds a heap pointer and is potentially live across a call to the collector is assigned to a root stack slot (rather than to a register or ordinary stack slot).

   (b) [10 points] Draw a picture of the root stack and its contents at the moment when variable `v` is assigned into during execution of the following $\mathcal{L}_{\mathsf{Tup}}$ program. For each slot in the root stack, state which variable is stored in the slot, and the current value of that variable. (You do not have to describe the tag fields of the values.) Make it clear which direction the stack is growing (i.e., which slot represents the top-of-stack).

```
def f (x:int) -> int:
  t = (x,x)
  u = (t,t)
  v = u[0]        # what does the root stack contain at this point?
  return v[1]

a = (30,40)
b = (a[1],a[0])
c = f(2)
print (b[0] + c)
```

   Solution:

   At the point where `v` is assigned into, the root stack contains two entries:

```
top-of-stack:   | t | --> (2,2)
base-of-stack:  | b | --> (40,30)
```

4. [20 points] Functions.

   Consider the following $\mathcal{C}_{\mathsf{Tup}}$ program.

```
def f(x:int, y:int) -> int:
f_start:
    print(x)
    z = y + y
    return z
def g(h:Callable[[int, int], int]) -> None:
g_start:
    a = h(42, 21)
    print(a)
    return
def main() -> int:
main_start:
    t = {f}
    {g}(t)
    return 0
```

   (a) [5 points] Give an assignment of each variable (parameter, local, temporary) to a register, using the standard X86-64 calling conventions. You do not have to match the exact details of our compiler's register allocation scheme, but, as in that scheme, your solution should *not* require saving caller-save registers over calls, and should otherwise minimize the use of callee-save registers.

   As a reminder:

   - First six arguments go in registers `rdi`,`rsi`,`rdx`,`rcx`,`r8`,`r9`, in that order.
   - Return register is `rax`.
   - Caller-save registers are: `rax rcx rdx rsi rdi r8 r9 r10 r11`.
   - Callee-save registers are: `rsp rbp rbx r12 r13 r14 r15`. Of these, `rsp` is reserved for the stack pointer and `rbp` for the base pointer.

   Assignment:

   `x:`

   `y:`

   `z:`

   `h:`

   `a:`

   `t:`

Solution:

One solution that minimizes moves:

```
x: rdi  (or some other caller-save register, avoiding rsi)
y: rbx  (or some other callee-save register)
z: rbx  (or some caller-save register.
         It's OK to use this particular callee-save register here,
         because it is already in use to store y in this function)
h: rdx  (or some other caller-save register, avoiding rdi, rsi)
a: rdi  (or some other caller-save register)
t: rdi  (or some other caller-save register)
```

(b) [15 points] Show the final $\text{x86}^{\mathsf{Def}}_{\mathsf{callq}*}$ assembly code for the `start` block of each function, using the register assignment you gave for part (a). You do *not* need to show the code for entry or exit blocks of the functions. Again, you do not have to match the exact details of our compiler's instruction selection scheme, but your code should be at least as efficient as what is produced by that scheme. In particular, you should prefer direct calls over indirect calls wherever possible.

Solution:

```
f_start:
    movq %rsi, %rbx
    callq print_int
    addq %rbx, %rbx
    movq %rbx, %rax
    jmp f_conclusion

g_start:
    movq %rdi, %rdx
    movq $21, %rdi
    movq $42, %rsi
    callq *%rdx
    movq %rax, %rdi
    callq print_int
    jmp g_conclusion

 main_start:
    leaq f(%rip), %rdi
    callq g
    movq $0, %rax
    jmp main_conclusion
```

5. [10 points] Interpreters.

   Interpretating a virtual machine instruction set (such as JVM bytecode) in software is inherently less efficient than executing a real machine instruction set in hardware. Name two fundamental interpreter tasks that can be expensive compared to their hardware equivalents. For each task, briefly describe an interpreter optimization technique that may help to reduce the cost.

   Solution: Two fundamental costs are (i) instruction dispatch; (ii) operand access. The first can be improved by using threaded code and/or introducing super-instructions; the second by arranging for operands to live in machine registers, e.g. using stack caching.

Concrete Syntax of Languages

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

$$
\begin{array}{rcl}
\mathit{cmp} & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{is} \\
\mathit{exp} & ::= & \mathit{int} \mid \mathit{bool} \mid \mathit{var} \\
 & \mid & \texttt{input\_int()} \mid \texttt{-}\ \mathit{exp} \mid \texttt{not}\ \mathit{exp} \mid \mathit{exp}\ \texttt{+}\ \mathit{exp} \mid \mathit{exp}\ \texttt{-}\ \mathit{exp} \\
 & \mid & \mathit{exp}\ \texttt{and}\ \mathit{exp} \mid \mathit{exp}\ \texttt{or}\ \mathit{exp} \mid \texttt{(}\mathit{exp}\texttt{)} \\
 & \mid & \mathit{exp}\ \mathit{cmp}\ \mathit{exp} \mid \mathit{exp}\ \texttt{if}\ \mathit{exp}\ \texttt{else}\ \mathit{exp} \\
 & \mid & \mathit{exp}\texttt{(}\mathit{exp}\texttt{,...)} \mid \mathit{exp}\texttt{,...,}\mathit{exp} \mid \texttt{()} \mid \mathit{exp}\texttt{[}\mathit{int}\texttt{]} \\
\mathit{stmt} & ::= & \texttt{print(}\mathit{exp}\texttt{)} \mid \mathit{exp} \mid \mathit{var}\ \texttt{=}\ \mathit{exp} \mid \texttt{if}\ \mathit{exp}\texttt{:}\ \mathit{stmt}^{+}\ \texttt{else:}\ \mathit{stmt}^{+} \\
 & \mid & \texttt{while}\ \mathit{exp}\texttt{:}\ \mathit{stmt}^{+} \mid \texttt{return}\ \mathit{exp} \mid \texttt{return} \mid \mathit{exp}\texttt{[}\mathit{int}\texttt{]}\ =\ \mathit{exp} \\
\mathit{type} & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{tuple[}\mathit{type}\texttt{, ...]} \mid \texttt{Callable[[}\mathit{type}\texttt{,...],}\ \mathit{rtype}\texttt{]} \\
\mathit{rtype} & ::= & \mathit{type} \mid \texttt{None} \\
\mathit{def} & ::= & \texttt{def}\ \mathit{var}\texttt{(}\mathit{var}\texttt{:}\mathit{type}\texttt{,...) ->}\ \mathit{rtype}\texttt{:}\ \mathit{stmt}^{+} \\
\mathcal{L}_{\mathsf{Tup}} & ::= & \mathit{def}\ldots\mathit{stmt}\ldots
\end{array}
$$

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

$$
\begin{array}{rcl}
\mathit{atm} & ::= & \mathit{int} \mid \mathit{bool} \mid \mathit{var} \mid \mathit{global} \\
\mathit{cmp} & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{is} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{input\_int()} \mid \texttt{-}\ \mathit{atm} \mid \texttt{not}\ \mathit{atm} \mid \mathit{atm}\ \texttt{+}\ \mathit{atm} \mid \mathit{atm}\ \texttt{-}\ \mathit{atm} \\
 & \mid & \mathit{atm}\ \mathit{cmp}\ \mathit{atm} \mid \{\mathit{label}\} \mid \mathit{atm}\texttt{(}\mathit{atm}\texttt{,...)} \mid \{\mathit{label}\}\texttt{(}\mathit{atm}\texttt{,...)} \\
 & \mid & \mathit{atm}\texttt{[}\mathit{int}\texttt{]} \mid \texttt{allocate(}\mathit{int}\texttt{,}\mathit{type}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{print(}\mathit{atm}\texttt{)} \mid \mathit{exp} \mid \mathit{var}\ \texttt{=}\ \mathit{exp} \mid \mathit{atm}\texttt{[}\mathit{int}\texttt{]}\ =\ \mathit{atm} \mid \texttt{collect(}\mathit{int}\texttt{)} \\
\mathit{tail} & ::= & \texttt{return}\ \mathit{exp} \mid \texttt{return} \mid \texttt{goto}\ \mathit{label} \\
 & \mid & \texttt{if}\ \mathit{atm}\ \mathit{cmp}\ \mathit{atm}\texttt{:}\ \texttt{goto}\ \mathit{label}\ \texttt{else:}\ \texttt{goto}\ \mathit{label} \\
\mathit{type} & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{tuple[}\mathit{type}\texttt{, ...]} \mid \texttt{Callable[[}\mathit{type}\texttt{,...],}\ \mathit{rtype}\texttt{]} \\
\mathit{rtype} & ::= & \mathit{type} \mid \texttt{None} \\
\mathit{block} & ::= & \mathit{label}\texttt{:}\ \mathit{stmt}^{*}\ \mathit{tail} \\
\mathit{def} & ::= & \texttt{def}\ \mathit{label}\texttt{(}\mathit{var}\texttt{:}\mathit{type}\texttt{,...) ->}\ \mathit{rtype}\texttt{:}\ \mathit{block}^{*} \\
\mathcal{C}_{\mathsf{Tup}} & ::= & \mathit{def}\ldots
\end{array}
$$

Note: the concrete expression $\{\mathit{label}\}$ corresponds to the AST form `FunRef`(label).

$\text{x86}^{\mathsf{Var,Def}}_{\mathsf{callq*}}$

$$
\begin{array}{rcl}
\textit{reg} & ::= & \texttt{rsp} \mid \texttt{rbp} \mid \texttt{rax} \mid \texttt{rbx} \mid \texttt{rcx} \mid \texttt{rdx} \mid \texttt{rsi} \mid \texttt{rdi} \mid \\
 & & \texttt{r8} \mid \texttt{r9} \mid \texttt{r10} \mid \texttt{r11} \mid \texttt{r12} \mid \texttt{r13} \mid \texttt{r14} \mid \texttt{r15} \\
\textit{bytereg} & ::= & \texttt{ah} \mid \texttt{al} \mid \texttt{bh} \mid \texttt{bl} \mid \texttt{ch} \mid \texttt{cl} \mid \texttt{dh} \mid \texttt{dl} \\
\textit{arg} & ::= & \texttt{\$}\textit{int} \mid \texttt{\%}\textit{reg} \mid \texttt{\%}\textit{bytereg} \mid \textit{int}\texttt{(\%}\textit{reg}\texttt{)} \mid \textit{label}\texttt{(\%rip)} \mid \textit{var} \\
\textit{cc} & ::= & \texttt{e} \mid \texttt{ne} \mid \texttt{l} \mid \texttt{le} \mid \texttt{g} \mid \texttt{ge} \\
\textit{instr} & ::= & \texttt{addq}\ \textit{arg}\texttt{,}\textit{arg} \mid \texttt{subq}\ \textit{arg}\texttt{,}\textit{arg} \mid \texttt{negq}\ \textit{arg} \mid \texttt{movq}\ \textit{arg}\texttt{,}\textit{arg} \\
 & \mid & \texttt{pushq}\ \textit{arg} \mid \texttt{popq}\ \textit{arg} \mid \texttt{callq}\ \textit{label} \mid \texttt{callq *}\textit{arg} \mid \texttt{retq} \\
 & \mid & \texttt{xorq}\ \textit{arg}\texttt{,}\ \textit{arg} \mid \texttt{cmpq}\ \textit{arg}\texttt{,}\ \textit{arg} \mid \texttt{set}\textit{cc}\ \textit{arg} \mid \texttt{movzbq}\ \textit{arg}\texttt{,}\ \textit{arg} \\
 & \mid & \texttt{jmp}\ \textit{label} \mid \texttt{j}\textit{cc}\ \textit{label} \mid \textit{label}\texttt{:}\ \textit{instr} \mid \texttt{leaq}\ \textit{arg}\texttt{,}\ \texttt{\%}\textit{reg} \\
\textit{block} & ::= & \textit{instr}^{+} \\
\textit{def} & ::= & \texttt{.globl}\ \textit{label}\ (\textit{label}\texttt{:}\ \textit{block})^{*} \\
\text{x86}^{\mathsf{Var,Def}}_{\mathsf{callq*}} & ::= & \textit{def}^{*}
\end{array}
$$

Note: this is the same as $\text{x86}^{\mathsf{Def}}_{\mathsf{callq*}}$, below, except that *var* is allowed as an *arg*.

$\text{x86}^{\mathsf{Def}}_{\mathsf{callq*}}$

$$
\begin{array}{rcl}
\textit{reg} & ::= & \texttt{rsp} \mid \texttt{rbp} \mid \texttt{rax} \mid \texttt{rbx} \mid \texttt{rcx} \mid \texttt{rdx} \mid \texttt{rsi} \mid \texttt{rdi} \mid \\
 & & \texttt{r8} \mid \texttt{r9} \mid \texttt{r10} \mid \texttt{r11} \mid \texttt{r12} \mid \texttt{r13} \mid \texttt{r14} \mid \texttt{r15} \\
\textit{bytereg} & ::= & \texttt{ah} \mid \texttt{al} \mid \texttt{bh} \mid \texttt{bl} \mid \texttt{ch} \mid \texttt{cl} \mid \texttt{dh} \mid \texttt{dl} \\
\textit{arg} & ::= & \texttt{\$}\textit{int} \mid \texttt{\%}\textit{reg} \mid \texttt{\%}\textit{bytereg} \mid \textit{int}\texttt{(\%}\textit{reg}\texttt{)} \mid \textit{label}\texttt{(\%rip)} \\
\textit{cc} & ::= & \texttt{e} \mid \texttt{ne} \mid \texttt{l} \mid \texttt{le} \mid \texttt{g} \mid \texttt{ge} \\
\textit{instr} & ::= & \texttt{addq}\ \textit{arg}\texttt{,}\textit{arg} \mid \texttt{subq}\ \textit{arg}\texttt{,}\textit{arg} \mid \texttt{negq}\ \textit{arg} \mid \texttt{movq}\ \textit{arg}\texttt{,}\textit{arg} \\
 & \mid & \texttt{pushq}\ \textit{arg} \mid \texttt{popq}\ \textit{arg} \mid \texttt{callq}\ \textit{label} \mid \texttt{callq *}\textit{arg} \mid \texttt{retq} \\
 & \mid & \texttt{xorq}\ \textit{arg}\texttt{,}\ \textit{arg} \mid \texttt{cmpq}\ \textit{arg}\texttt{,}\ \textit{arg} \mid \texttt{set}\textit{cc}\ \textit{arg} \mid \texttt{movzbq}\ \textit{arg}\texttt{,}\ \textit{arg} \\
 & \mid & \texttt{jmp}\ \textit{label} \mid \texttt{j}\textit{cc}\ \textit{label} \mid \textit{label}\texttt{:}\ \textit{instr} \mid \texttt{leaq}\ \textit{arg}\texttt{,}\ \texttt{\%}\textit{reg} \\
\textit{block} & ::= & \textit{instr}^{+} \\
\textit{def} & ::= & \texttt{.globl}\ \textit{label}\ (\textit{label}\texttt{:}\ \textit{block})^{*} \\
\text{x86}^{\mathsf{Def}}_{\mathsf{callq*}} & ::= & \textit{def}^{*}
\end{array}
$$