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

(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.

(c) [5 points] The algorithm used in Chapter 5 to compute liveness information for code generated from $\mathcal{L}_{\text{If}}$ 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.

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.

3. [15 points] Root Stack.

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

  (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)
```

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:

(b) [15 points] Show the final 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.

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.

Concrete Syntax of Languages

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

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

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

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

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

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

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

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

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

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