Midterm Exam

CS410P/510 Programming Language Compilation

Feb. 13, 2024

Name: ______________________________

Instructions

- This exam has 6 questions, for a total of 85 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.

The concrete syntax for all the intermediate languages mentioned in the exam and the X86 register usage conventions can be found on the last two pages.

1. [15 points] Consider the languages $\mathcal{L}_{\mathsf{If}}$ (the source language for the whole compiler) and $\mathcal{L}_{\mathsf{if}}^{mon}$ (the intermediate language that is the target of the Remove Complex Operands pass). For each of the following code examples, indicate whether the example is a syntactically valid program in $\mathcal{L}_{\mathsf{If}}$ only, $\mathcal{L}_{\mathsf{if}}^{mon}$ only, both languages, or neither language. No explanations are required, but brief ones might help you get partial credit even if your answer is wrong.

   (a)
```
print (- (10 if input_int() == 0 else 20))
```

   (b)
```
x = 0
y = input_int()
x = y if y > 0 else x
print(y)
```

   (c)
```
x = input_int()
y = 2 * x
print(y)
```

   (d)
```
x = input_int()
z = { y = input_int()
      w = y + 1
      produce -w }
    if x == 0
    else x + 1
print(z)
```

   (e)
```
x = input_int()
if x < 0:
  print (x = 100 - x)
else:
  print (x = x - 100)
print(x)
```

Solution:

(a) $\mathcal{L}_{\mathsf{If}}$ only (arguments to `print`, `-` and `==` are not atomic).

(b) $\mathcal{L}_{\mathsf{If}}$ and $\mathcal{L}_{\mathsf{if}}^{mon}$.

(c) Neither (`*` is not a binary operator).

(d) $\mathcal{L}_{\mathsf{if}}^{mon}$ only (contains a `Begin`).

(e) Neither (argument to `print` must be an *exp*, not a *stmt*).

2. [10 points] Consider this fragment of the code implementing Remove Complex Operands:

```
def rco_stmt(self, s: stmt) -> list[stmt]:
  match s:
    case Expr(e):
      e_rco,temps = self.rco_exp(e,False)
      return [Assign([x], rhs) for (x, rhs) in temps] + [Expr(e_rco)]
    ...

def rco_exp(self, e:expr, need_atomic: bool) -> tuple[expr, list[tuple[Name,expr]]]:
  def atomize(e:expr,temps:list[tuple[Name,expr]]):
    if need_atomic:
      tmp = Name(generate_name('tmp'))
      return (tmp,temps + [(tmp,e)])
    else:
      return (e,temps)
  match e:
    ...
    case Compare(left,[cmpr],[right]):
      left_rco, temps1 = self.rco_exp(left,True)
      right_rco, temps2 = self.rco_exp(right,True)
      return atomize(Compare(left_rco,[cmpr],[right_rco]), temps1 + temps2)
    ...
```

Suppose we were to change `temps1 + temps2` into `temps2 + temps1` in the last line shown. Write a *short* test that can distinguish between the behavior of the original compiler (#1) and the version with the order swapped (#2).

Your test should consist of a source program `ex.py` (written in $\mathcal{L}_{\mathsf{If}}$), an input file `ex.in`, a `ex.golden` file showing the output expected from compiler #1, and a `ex.out` file showing the output that will be produced by compiler #2.

Solution: There are many possible solutions, but they all rely on using `input_int()`, which is the sole expression that has a side-effect. Here is one simple example:

`ex.py`:

```
print (1 if input_int() < input_int() else 0)
```

`ex.in`:

```
1
2
```

`ex.golden`:

```
1
```

`ex.out`:

```
0
```

3. [15 points] Translate the following $\mathcal{L}_{\mathsf{if}}^{mon}$ program into $\mathcal{C}_{\mathsf{If}}$.

```
x = input_int()
z = { y = input_int()
      produce -y }
    if x == 0
    else 42
print(z)
```

Solution: (Approx. 1 point per statement.)

```
start:
    x = input_int()
    if x == 0:
      goto block.4
    else:
      goto block.5

block.4:
    y = input_int()
    z = -y
    goto block.3

block.5:
    z = 42
    goto block.3

block.3:
    print(z)
    return 0
```

4. [15 points] Translate the following $\mathcal{C}_{\mathsf{If}}$ program into $\text{x86}_{\mathsf{If}}^{\mathsf{Var}}$.

```
start:
  b = True
  x = input_int()
  y = x <= 10
  z = 42
  if y == b:
    goto block.1
  else:
    goto block.2

block.1:
  z = -z
  goto block.0

block.2:
  z = x - 10
  goto block.0

block.0:
  print(z)
  return 0
```

Solution:

```
start:
    movq $1, b
    callq read_int
    movq %rax, x
    cmpq $10, x
    setle %al
    movzbq %al, y
    movq $42, z
    cmpq b, y
    je block.1
    jmp block.2

block.1:
    negq z
    jmp block.0

block.2:
    movq x, z
    subq $10, z
    jmp block.0

block.0:
    movq z, %rdi
    callq print_int
    movq $0, %rax
    jmp conclusion
```

5. [15 points] For the following $\mathcal{C}_{\mathsf{If}}$ program, fill in the live variable 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.)

```
start:
                                    live =
  a = 1
                                    live =
  b = 2
                                    live =
  c = a + b
                                    live =
  d = input_int()
                                    live =
  if d > 0: goto block2
  else: goto block3

block2:
                                    live =
  a = 2 + b
                                    live =
  goto block1

block3:
                                    live =
  c = -a
                                    live =
  a = c + 2
                                    live =
  goto block1

block1:
                                    live =
  print(a)
                                    live =
  return 0
```

Solution:

```
start:
                                 live = {}
  a = 1
                                 live = {a}
  b = 2
                                 live = {a,b}
  c = a + b
                                 live = {a,b}
  d = input_int()
                                 live = {a,b,d}
  if d > 0: goto block2
  else: goto block3

block2:
                                 live =  {b}
  a = 2 + b
                                 live = {a}
  goto block1

block3:
                                 live = {a}
  c = -a
                                 live = {c}
  a = c + 2
                                 live = {a}
  goto block1

block1:
                                 live = {a}
  print(a)
                                 live = {}
  return 0
```

6. [15 points] Consider the following results from liveness analysis on a $\text{x86}^{\mathsf{Var}}_{\mathsf{If}}$ program.

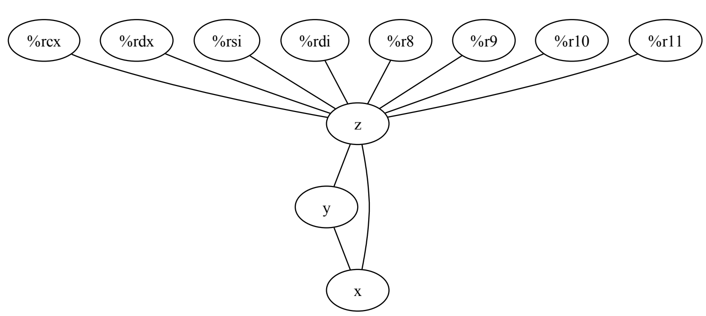
```
start:
                {}
  callq read_int
                {%rax}
  movq %rax, x
                {x}
  movq x, y
                {y, x}
  addq $1, y
                {y, x}
  movq y, z
                {y, x, z}
  addq $1, z
                {y, z, x}
  cmpq $0, x
                {y, x, z}
  je block.1
                {y, x, z}
  jmp block.2
                {y, z, x}
```

```
block.1:
                {x, z}
  movq x, %rdi
                {%rdi, z}
  callq print_int
                {z}
  jmp block.0
                {z}
block.2:
                {y, z}
  movq y, %rdi
                {%rdi, z}
  callq print_int
                {z}
  jmp block.0
                {z}
block.0:
                {z}
  movq z, %rdi
                {%rdi}
  callq print_int
                {}
  movq $0, %rax
                {%rax}
  jmp conclusion
                {%rax}
```

(a) Draw the corresponding interference graph. Assume that `%rax` will not be used as an assignable register, so it can be omitted from the graph.

(b) Suppose we assign all the variables of this program to registers (not stack slots). What is the minimum number of registers needed? How many of these must be *callee-save* registers?

Solution: (a)



(b) Three registers are needed. Of these, one needs to be a callee-save register, because `z` is in conflict with all the caller-save registers.

[This page deliberately left blank.]

Concrete Syntax of Languages

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

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

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

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

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

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

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

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

```
      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) | 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^Var_If ::= .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 ...
```

The caller-saved registers are:

`rax rcx rdx rsi rdi r8 r9 r10 r11`

The callee-saved registers are:

`rsp rbp rbx r12 r13 r14 r15`

The argument registers are:

`rdi rsi rdx rcx r8 r9`

The result register is:

`rax`