CS164 Programming Languages and Compilers **Spring 2024**

Written Assignment 4 Solutions

Assigned: March 5 **Due:** March 20 at 11:59pm

1. Suppose `f` is a function with a call to `g` somewhere in the body of `f`:

```
def f(...):
 ... g(...) ...
```

We say that this particular call to `g` is a *tail call* if the call is the last thing `f` does before returning. For example, consider the following ChocoPy functions for computing positive powers of 2:

```
def f(x : int, acc : int) -> int:
    if x > 0:
        return f(x-1, acc*2)
    else:
        return acc
def g(x : int) -> int:
    if x > 0:
        return 2 * g(x-1)
    else:
        return 1
```

Here `f(x, 1)` = `g(x)` $= 2^x$ for $x \geq 0$. The recursive call to `f` is a tail call, while the recursive call to `g` is not. A function in which all recursive calls are tail calls is called *tail recursive*.

(a) Here is a non-tail recursive function for computing the number of odd digits of a non-negative integer.

```
def num_odd(n : int) -> int:
    if n < 10:
        if n % 2 == 1:
            return 1
        else:
            return 0
    else:
        if n % 2 == 1:
            return num_odd(n // 10) + 1
        else:
            return num_odd(n // 10) + 0
```

Write a tail recursive function `num_odd2` that computes the same result. (Hint: Your function will most likely need two arguments, or it may need to invoke a function of two arguments.)

Here is a tail-recursive function that computes the number of odd digits:

```
def num_odd2(n : int, total : int) -> int:
    if n < 10:
        if n % 2 == 1:
            return 1 + total
        else:
            return total
    else:
```

```
if n % 2 == 1:
    return num_odd2(n // 10, total + 1)
else:
    return num_odd2(n // 10, total)
```

(b) Recall from lecture that function calls are usually implemented using a stack of activation records.

i. Trace the execution of `num_odd` and `num_odd2` computing the result for 123456, writing out the stack of activation records at each step i.e., draw the stack of activation records. You don't need to draw everything in each activation frame. Just label each activation frame with the function name and argument(s) for that frame. An example is given below, where foo(3) calls foo(2) which calls foo(1).

AR for `foo(1)`
AR for `foo(2)`
AR for `foo(3)`

If we trace out the execution of `num_odd` and `num_odd2` computing 123456, we see that they both require at most six activation records on the stack. Below we show the stack during the evaluation of the last recursive call:

AR for `num_odd(1)`
AR for `num_odd(12)`
AR for `num_odd(123)`
AR for `num_odd(1234)`
AR for `num_odd(12345)`
AR for `num_odd(123456)`

AR for `num_odd2(1, 2)`
AR for `num_odd2(12, 2)`
AR for `num_odd2(123, 1)`
AR for `num_odd2(1234, 1)`
AR for `num_odd2(12345, 0)`
AR for `num_odd2(123456, 0)`

ii. Explain the computation done before each activation record is removed for both `num_odd` and `num_odd2`.

For `num_odd`, before a new activation record is pushed onto the stack we do one comparison and 1 modulo by 10. Just before we remove an activation record from the stack (i.e., just before we return) we do one addition. For `num_odd2`, we do all of these before each new record is pushed onto the stack. When we remove an activation record from the stack all we do is take the result of the recursive call to `num_odd2` and return it. Thus the only part of the stack frame for `num_odd2` we are using after a recursive call is the return address.

iii. Is there any place where you can see potential for making the execution of the tail-recursive `num_odd2` more time-efficient or space-efficient than `num_odd` (without changing `num_odd2`'s source code)? What could you do?

We can implement `num_odd2` more efficiently by reusing the same activation record for a recursive call, rather than pushing a new record on the stack. To recursively call `num_odd2`, we compute the new values for `n` and `total` and then place them on the stack as if they were the arguments for the current activation record. Then we jump back to the beginning of `num_odd2` and execute the same code again.

2. Consider the following ChocoPy classes:

```
class A(object):
    x : int = 0
    y : int = 1
    def getX(self:"A") -> int:
        return self.x
    def getY(self:"A") -> int:
        return self.y

class B(A):
    z : int = 2
    def getX(self:"B") -> int:
        return self.x + 1
    def getZ(self:"B") -> int:
        return self.z

class C(B):
    s : str = "c"
    def getY(self:"C") -> int:
        return self.y - 1
    def toString(self:"C") -> str:
        return self.s
```

(a) Draw a diagram that illustrates the layout of objects of type `A`, `B` and `C`, including their dispatch tables. You can assume the existence of labels that point to the code containing method definitions (e.g. label `A.getX` for the method `getX` defined in class `A`).

Offset/Class	**A**	**B**	**C**
0	ATag	BTag	CTag
4	5	6	7
8	*	*	*
12	x	x	x
16	y	y	y
20		z	z
24			s

Dispatch tables:

	A	**B**	**C**
0	__init__	__init__	__init__
4	A.getX	B.getX	B.getX
8	A.getY	A.getY	C.getY
12		B.getZ	B.getZ
16			C.toString

(b) Let `obj` be a variable whose static type is `A`. Assume that the contents of `obj` have been loaded into register `a0`. The contents may be an address of an object in memory or the address `0`, which represents the `None` value. Write RISC-V code for the method invocation `obj.getY()`. You may assume the existence of a label `error_dispatch_None` that contains logic for aborting the program with an error due to a dispatch on a `None` value (you just need to jump to the label appropriately). You may use temporary registers such as `t1` if you wish. Use ChocoPy's calling convention (the caller should push and pop arguments on the stack). As an example of this convention, here is the RISC-V code for the function invocation `g(1)` assuming there already exists the label `g` that contains callee code for the corresponding function:

```
    li a0, 1    # load argument 1
    push a0     # push argument on stack
    jal g      # jump to function
    pop         # pop argument on stack
```

```
beq  a0, x0, error_dispatch_None  # check for null/None obj
push a0                           # push argument on stack
lw   t1, 8(a0)                    # load dispatch pointer
lw   t1, 8(t1)                    # load method2 ptr from table
jalr t1                           # jump to method
pop                               # pop argument on stack
```

(c) Explain what happens in part (b) if `obj` references an object that has dynamic type `B`.
If `obj` has dynamic type `B`, then we correctly invoke `getY` on the object. All objects have a dispatch pointer at offset 8, so we correctly fetch the dispatch pointer. Furthermore, all classes that inherit from class `A` will have a pointer to the appropriate version of `getY` at offset 8 (in bytes) of their dispatch table. In this case, we will call the version of `m2` supplied by class `A`, since class `B` did not override it. Note that it is legal to pass an object of type `B` as the `self` object, since the layout of `A` is a prefix of the layout of `B`.

(d) Explain what happens in part (b) if `obj` references an object that has dynamic type `C`.
Since class `C` overrides the `getY` method, we will call the version of `getY` supplied by class `C`. This will still be at offset 8 in the dispatch table for class `C` since the offset should not change for the same method, regardless of whether or not the method is overridden.

3. (a) Consider the following ChocoPy program with nested function definitions:

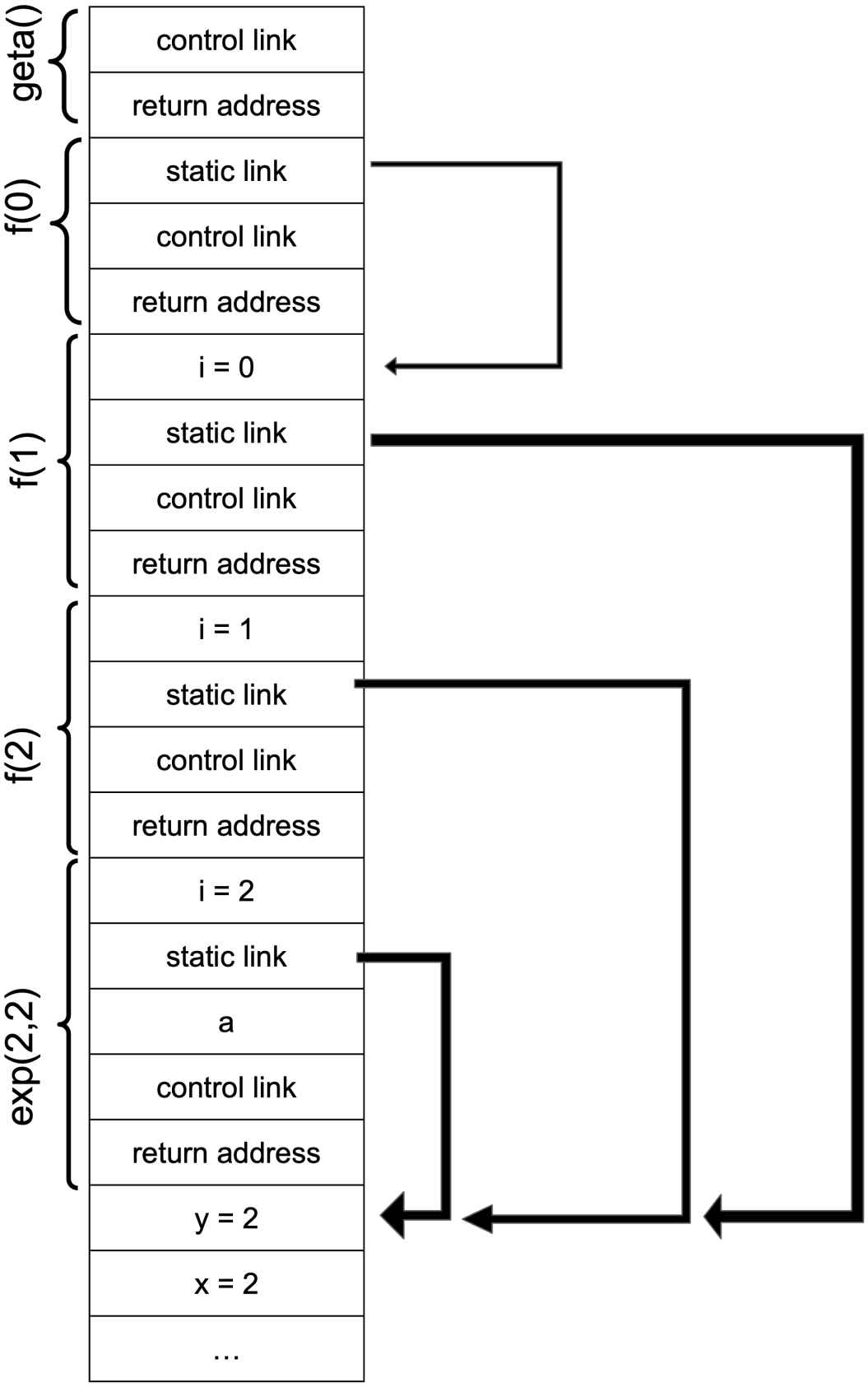
```
def exp(x: int, y: int) -> int:
    a: int = 1
    def f(i: int) -> int:
        nonlocal a
        def geta() -> int:
            return a
        if i <= 0:
            return geta()
        else:
            a = a * x
            return f(i-1)
    return f(y)
exp(2, 2)
```

This language feature causes some complications in code generation because nested functions may need to use variables defined in enclosing functions/methods. Consider the above example. The function `f` not only needs access to its own activation record (for variable `i`) but also the activation record for `exp` (for variables `x` and `a`).

One way to implement this feature is to use a different type of activation record for nested functions (as opposed to the activation record used for global functions and methods). This new activation record contains an extra entry known as a *static link*, that is passed as an extra argument when calling a nested function. The static link is a pointer to the activation record of the latest dynamic instance of the nearest statically enclosing function/method.

The first two activation records for a method call to `exp(2,2)` are given below (one for `exp(2,2)` and one for the function call to `f(2)`). In the diagram, we have noted with an arrow that the static link for `f(2)` points to the word below the return address in the activation record for `exp(2,2)`.

Complete the stack of activation records at the time of the call to `geta()` for `exp(2,2)` (i.e., having three calls to `f`). Include the activation record for `geta()`. Then, draw arrows to show where the static links point.


geta()
control link
return address
f(0)
static link
control link
return address
f(1)
i = 0
static link
control link
return address
f(2)
i = 1
static link
control link
return address
exp(2,2)
i = 2
static link
a
control link
return address
y = 2
x = 2
…

(b) Generate RISC-V code to store the result of the assignment `a = a * x` in function `f`. Assume that the code for the multiplication has already been generated with the result in `a0`. (Hint: refer to the diagram given on the previous page. You can use temporaries to generate code.)
Solution:

```
exp.f:
    ...              # code to compute a * x leaving the result in a0
    lw t0, 4(fp)     # Load static link from ‘f‘ to ‘exp‘ in t0
    sw a0, -12(t0)   # Store result a0 into variable ‘exp.a‘
    ...
```

(c) Complete the generated RISC-V code for `geta()`. Return the result in register `a0`. The function prologue and epilogue has been provided for you. (Hint: use your completion of the above diagram.)

Solution:

```
exp.f.geta:
    addi  sp, sp, -8
    sw    fp, 0(sp)
    sw    ra, 4(sp)
    addi  fp, sp, 8

    lw t0, 0(fp)     # Load static link from ‘geta‘ to ‘f‘ in t0
    lw t0, 4(t0)     # Load static link from ‘f‘ to ‘exp‘ in t0
    lw a0, -12(t0)   # Load variable ‘exp.a‘ into a0

    lw    ra, -4(fp)
    lw    fp, -8(fp)
    addi  sp, sp, 8
    jr    ra
```

Note: the epilogue ideally should have used sp-relative indexing to be consistent with the prologue, although this should still produce the same result. This was left unchanged because it was discovered after already releasing the updated pdf.

(d) Complete the generated RISC-V code for the call to `geta()` from `f`.

```
exp_f:
    ...
    addi sp, sp, -4   # Increment stack pointer
    sw fp, 0(sp)      # Push static link from ‘geta‘ to ‘f‘ on stack top


    jal exp.f.geta
    ...
```

(e) Complete the generated RISC-V code for the call to `f(i - 1)` from `f`. Assume that the code for the subtraction is generated in the printed line {`code to compute i - 1 in a0`}, after which the result of subtraction will be present in register `a0`.

```
exp.f:
    ...
    # First part of solution
    lw t0, 4(fp)       # Get static link from ‘f‘ to ‘exp‘ in t0
    addi sp, sp, -4    # Increment stack pointer
    sw t0, 0(sp)       # Push static link from ‘f‘ to ‘exp‘ on stack

    {code to compute i - 1 in a0}

    # Second part of solution
    addi sp, sp, -4    # Increment stack pointer
    sw a0, 0(sp)       # Push i - 1 to stack


    jal exp.f
    ...
```

4. Consider the following function defined in the small language from our Simple Code Generation lectures (suitably extended with multiplication, division, and unary negation operations):

```
def func(a, b, c):
    return sqrt(a*b) / (a+b) * -c
```

We want to produce RISC-V assembly code for this solver function which uses strictly `fp`-relative accesses to and from the stack when evaluating the body of the function. That is, the assembly code of the function will use `sp` only on entry and when calling other functions (such as `sqrt`), and all temporary values will be stored in the stack and read from the stack at fixed offsets from `fp`. For this question, please use the calling convention for this language as specified in lecture.

(a) Give a definition for the function cgen$(f(e_1, \ldots, e_n), nt)$. This function should generate code which evaluates the expression `f(e_1, ..., e_n)` while only using temporaries whose addresses are `fp - nt` or lower. Ensure that the generated code writes the current frame pointer and the arguments for `f` to the stack in such a way that `f` can properly read its arguments and correctly restore the frame pointer as it returns.

Note that this generated code must properly set `sp` *just before* calling `f`, i.e. just before the `jal` instruction. No other accesses to `sp` are necessary or allowed.

(Hint: modify the function cgen$(f(e_1, \ldots, e_n))$ from lecture slides as appropriate.)

```
sw fp, -nt(fp)
cgen(e_n, nt + 4)
sw a0, (-nt - 4)(fp)
cgen(e_(n-1), nt + 8)
sw a0, (-nt - 8)(fp)
...
cgen(e_1, nt + 4n)
sw a0, (-nt - 4n)(fp)
addi sp, fp, -nt - 4(n+1)
jal f_entry
```

(b) Fill in RISC-V code for the function `func`, so that your code uses a fixed (`fp`-relative) location in the stack for each stored temporary. The stack pointer should not be accessed anywhere in your assembly code except to set the stack pointer just before calling `sqrt` with the `jal` instruction, as mentioned in part (a). In particular, you may not use the macros `push reg`, `pop`, or `ra <- top`, since these macros all access the `sp` register.

You will need to use the following RISC-V instructions:

- `mul r1, r2, r3` multiplies registers `r2` and `r3` and stores the result in `r1`.
- `div r1, r2, r3` divides the register `r2` by `r3` and stores the result in `r1`. Don't worry about division by zero.
- `sub r1, x0, r2` may be used to compute $-$`r2` and store the result in register `r1`. (Recall that `x0` is the always-zero register.)

(Hint: use the cgen(e, nt) function from lecture, along with your implementation for cgen$(f(e_1, \ldots, e_n), nt)$. You may need to determine how to implement cgen$(e_1 * e_2, nt)$, cgen$(e_1/e_2, nt)$, and cgen$(-e, nt)$ as well.)

Fill in your code between the `body` and `exit` comments under `func`. (You may need more room than is given here.)

We would very much recommend drawing a stack diagram and using it to help in tracing the staff solution.

```
sqrt_entry:
    # entry
    mv fp, sp
    sw ra, 0(fp) # note that even this uses fp-relative addressing!
    # body: reads argument at 4(fp), and places result in a0
    ...
    # exit
    lw ra, 0(fp)
    lw fp, 8(fp)
    jr ra

func:
    # entry
    mv fp, sp
    sw ra, 0(fp)
    # body

    # prepare to call sqrt: store fp and calculate arg
    sw fp, -4(fp) # store current fp to temporary 1
    lw a0, 4(fp) # load a
    sw a0, -8(fp) # store a to temporary 2
    lw a0, 8(fp) # load b
    lw t1, -8(fp) # load temporary 2 = a
    mul a0, t1, a0 # calculate a * b
    sw a0, -8(fp) # store a * b to temporary 2
    # about to call sqrt: set sp, and jal to it
    addi sp, fp, -12
    jal sqrt_entry # a0 will contain the result
    sw a0, -4(fp) # store sqrt(a * b) to temporary 1
    lw a0, 4(fp) # load a
    sw a0, -8(fp) # store a to temporary 2
    lw a0, 8(fp) # load b
    lw t1, -8(fp) # load temporary 2 = a
    add a0, t1, a0 # calculate a + b
    lw t1, -4(fp) # load temporary 1 = sqrt(a * b)
    div a0, t1, a0 # calculate sqrt(a * b) / (a + b)
    sw a0, -4(fp) # store sqrt(a * b) / (a + b) to temporary 1
    lw a0, 12(fp) # load c
    sub a0, x0, a0 # calculate -c
    lw t1, -4(fp) # load temporary1 = sqrt(a * b) / (a + b)
    mul a0, t1, a0


    # exit
    lw ra, 0(fp)
    lw fp, 16(fp)
    jr ra
```