CS164 Programming Languages and Compilers **Fall 2024**

Written Assignment 4

Assigned: October 18 **Due:** October 31 at 11:59pm

Instructions: This a[illegible]repare written answers to questions on code generation. Each of the questions has a s[illegible] discuss this assignment with other students and work on the problems together. How[illegible]uld be your own individual work.

Please write your n[illegible]d discussion section on your homework. ***Please start each question on a new p[illegible]nments must be submitted as a PDF via Gradescope: https://gradescope.[illegible]*** [illegible]ow to submit assignments to Gradescope can be found at the following links: `https://gradescope.com/get_started#student-submission`

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


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

We say that this par[illegible] *[illegible]l call* if the call is the last thing `f` does before returning. For example, consider th[illegible]nctions for computing positive powers of 2:

```
def f(x : int, ac[illegible]
    if x > 0:
        return f([illegible]
    else:
        return ac[illegible]
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.)

(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 [illegible]

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

ii. Explain the [illegible] ore each activation record is removed for both `num_odd` and `num_odd2`.

iii. Is there an[illegible] 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?

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`).

(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
```

(c) Explain what happens in part (b) if `obj` references an object that has dynamic type `B`.

(d) Explain what happens in part (b) if `obj` references an object that has dynamic type `C`.

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

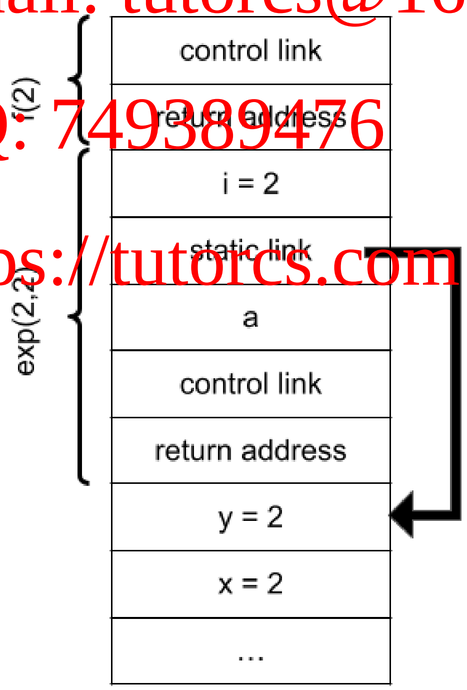

```
def exp(x: int, y: int) -> int:
    a: int = 1
    def f(i: int) -> int:
        nonlo
        def g
            r
        if i
            r
        else:
            a
            r
    return f(
exp(2, 2)
```

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.


f(2)
exp(2,2)
control link
return address
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.)

```
exp.f:
    ... # code to compute a * x leaving the result in a0



    ...
```

(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.)


```
exp.f.geta:
    addi  sp, sp, -8
    sw    fp,
    sw    ra,
    addi  fp,




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


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

```
exp_f:
    ...



    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:
    ...




    {code to compute i - 1 in a0}




    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 [illegible] for this solver function which uses strictly fp-relative accesses to and from the stack [illegible] body of the function. That is, the assembly code of the function will use sp only on e[illegible] other functions (such as sqrt), and all temporary values will be stored in the stac[illegible] ck at fixed offsets from fp. For this question, please use the calling convention fo[illegible] fied in lecture.

(a) Give a definiti[illegible] en$(f(e_1, \ldots, e_n), nt)$. This function should generate code which evaluates [illegible] ..., 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.)

(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.)

```
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(
    lw fp, 8(
    jr ra

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



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