CS164 Programming Languages and Compilers **Spring 2024**

Discussion Worksheet 6

Note that the code generation conventions used throughout this worksheet are for the "small language" described in lecture and not *for ChocoPy.*

1 Code Generation

A stack machine is an evaluation model where all data is stored on the stack. There are no variables or registers. Every operation in this model does the following:

1. Pop some number of arguments from the top of the stack.
2. Compute something over these arguments.
3. Push a result onto the stack.

E.g., "add" pops two elements off the stack, computes their sum, and pushes the sum onto the stack.
We can optimize this model by adding an "accumulator" register, which usually contains the top element of the stack. We can then manipulate this register directly, instead of operating only in the stack (in memory).

Exercise 0 (warmup) Try filling in the model code for the pure stack machine and the accumulator stack machine, given the expression $(7 + 5)$:

	Pure Stack Machine	With Accumulator
Store 7	push 7	acc = 7
Store 5	—	—
Compute sum	—	

We will implement this accumulator stack machine in machine code using RISC-V. Here are the code generation conventions we will use, based on those we used for the "small language" in lecture.

1. We keep the accumulator in register $a0$.
2. We use $t1$ to load elements from the stack for operations, since we cannot operate directly on the stack.
3. The next writeable stack location is at address sp. When we write (push), we decrease sp by 4.
4. The last readable stack location is at address $sp + 4$.

For example, here is the `cgen` function for a few different constructs:

```
cgen(e1 + e2) =
    cgen(e1)
    push a0
    cgen(e2)
    t1 <- top
    add a0, t1, a0
    pop
```

```
cgen(f(e1,...,en)) =
    push fp
    cgen(en)
    push a0
    ...
    cgen(e1)
    push a0
    jal f_entry
```

```
cgen(def f(x1,...,xn) = e) =
f_entry:
    mv fp, sp
    push ra
    cgen(e)
    ra <- top
    addi sp, sp, z
    lw fp, 0(sp)
    jr ra
```

Exercise 1 Let's convert the following pseudocode into RISC-V using the accumulator model:

```
factorial 0 = 1
factorial x = x * factorial (x - 1)
```

1.1 What elements will be in each Activation Record for each call to factorial?

1.2 Fill in the RISC-V code for the factorial function:

```
    factorial_entry:
    #entry: set up our FP, and save the return address for later
        -                           #Set new FP
        -                           #Store RA
    #body
        -                           #acc = x
        -                           #push acc
        -                           #acc = 0
        -                           #load top of stack to temporary
        -                           #pop to maintain stack hygiene
        -                           #jump to right branch
    false_branch:
        -                           #acc = x
        -                           #push acc
        -                           #start evaluation of (factorial(x-1))
        -                           #acc = x
        -                           #push acc
        -                           #acc = 1
        -                           #load top of stack to temporary
        -                           #perform the subtraction (x - 1)
        -                           #pop to maintain stack hygiene
        -                           #push the last argument, (x-1)
        -                           #recursively call ‘factorial‘
        -                           #start computing the product (x * factorial(x - 1))
        -                           #perform the multiplication
        -                           #maintain stack hygiene
        -                           #skip over the true_branch
    true_branch:
        -                           #compute (0)
    end_if:
    #exit
        -                           #Load RA which we saved earlier
        addi sp,sp,z                #Jump to old FP. What is z? z=___
        -                           #Set FP to be the old FP, which the callee saved on the stack
        -                           #Return to caller
```

Here is an iterative definition for factorial:

```
def factorial(x):
    total = 1
    while (x > 0):
        total = total * x
        x = x - 1
    return total
```

1.3 Fill in the RISC-V code for the iterative factorial. Again, use the code generation conventions for the small language described in lecture, as outlined on page 1 of the worksheet.
Note that local variables also are stored in the stack, like function arguments. Let's put any local variables on the stack just below the RA (i.e. lower addresses); they can be accessed using a fixed offset from `fp`, just like any parameters. Which offset(s) should we use?

```
    factorial_entry:
        -                           # set up our FP
        -                           # save RA to the stack
    #body
        -                           # evaluate the initial value for ‘total‘
        -                           # push ‘total‘ to the stack, for storage throughout the
                                    # execution of this function
    while_loop:
        -                           # evaluate (x > 0) and branch in the next few lines
        -
        -
        -
        -
        -                           # branch as necessary. hint: ‘bge‘ = branch if greater or equal
        -                           # evaluate (total * x) in the next few lines
        -
        -
        -
        -
        -
        -                           # store accumulator in ‘total‘
        -                           # evaluate (x - 1) in the next few lines
        -
        -
        -
        -
        -
        -                           # store accumulator in ‘x‘
        j while_loop                # continue to the next iteration
    exit_while:
    #exit
        -                           # load return value into a0
        -                           # maintain stack hygiene: pop our storage for ‘total‘
        -                           # load return address
        addi sp,sp,z                # restore stack. z=___?
        -                           # restore old FP
        -                           # return to caller
```

2 Temporaries

Instead of pushing and popping elements from the stack, it is sometimes more efficient to precompute locations for temporary variables, allocate stack space at the beginning, and then access them directly. A function can perform the preallocation before evaluating its body expression, by checking $NT(expr)$ and allocating that much space on the stack.
Let's consider the expression $3 + (5 + 7) + 9$. This requires two temporaries. (What change could we make to (5+7) to make it require three temporaries?) Therefore, we will preallocate two spaces on the stack, accessible via $-4(fp)$ and $-8(fp)$.

Exercise 2 Recall that $cgen(e_1 + e_2, nt)$ is defined as

```
cgen(e1 + e2, nt) =
    cgen(e1, nt)
    sw a0, -nt(fp)
    cgen(e2, nt + 4)
    lw t1, -nt(fp)
    add a0, t1, a0
```

2.1 Fill out the blanks in the "Preallocate" column, so that they are the equivalents of the instructions in the "Push/Pop" column, but using `fp`-relative addressing instead of `push` and `pop`. What should each offset from `fp` be? How can we compute this ahead of time?

Push/Pop	Preallocate	cgen(3+(5+7)+9, nt=4)
======	======	cgen(3+(5+7), nt=4)
a0 = 3	a0 = 3	cgen(3, nt=4)
push a0	_	Save temp
======	======	cgen(5+7, nt=8)
a0 = 5	a0 = 5	cgen(5, nt=8)
push a0	_	Save temp
a0 = 7	a0 = 7	cgen(7, nt=12)
pop t1	_	Load temp 5
add a0, t1, a0	add a0, t1, a0	Compute 5+7
pop t1	_	Load temp 3
add a0, t1, a0	add a0, t1, a0	Compute 3+(5+7)
push a0	_	Save temp
a0 = 9	a0 = 9	cgen(9, nt=8)
pop t1	_	Load temp 3+(5+7)
add a0, t1, a0	add a0, t1, a0	Compute full sum

2.2 Why isn't it a problem that we have 'cgen(7, nt=12)' even though we only allocated 8 bytes on the stack?