Hmmm Functions
It never ends!

Write a Hmmm program that:
1. never halts
2. is as short as possible
Take-home midterm #1

**Available:** this Sunday night (9/25)

**Must return by:** next Sunday (10/2) at 5pm

**Time-limit:** 75 minutes

**Covers:** everything up to and including this week
automata, circuits, and assembly

**Resources:** one, 8½ x 11 sheet of notes (double-sided)

**Honor code:** don’t discuss exam questions

There will be **assignments**, too.
It never ends!

Write a Hmmm program that:
1. never halts
2. is as short as possible

0 jumpn 0
Today’s goal:

Understand how functions work
(in a stored-program machine)
Hmmm conventions
Human programming practices that help us write correct and readable programs

r0 always contains the value 0

Write lots of comments / documentation!
Clearer is better than shorter (but shorter can be clearer).
Hmmm conventions
Human programming practices that help us write correct and readable programs

r0 always contains the value 0

r13 is for result values

Write lots of comments / documentation!
Clearer is better than shorter (but shorter can be clearer).
Practice writing Hmmm (well)

Write a Hmmm program that reads two integers from input into registers r1 and r2. The program then computes $r13 = r1^2 + r2$ and writes that value out.

A suggested algorithm-algorithm

1. Write a few test cases
2. Write pseudocode (i.e., a sketch of the algorithm)
3. Write comments
4. Write Hmmm instructions for the math
5. Write Hmmm instructions for the control flow
6. Write the line numbers
A program that computes $r1^2 + r2$

# read the inputs from the user
0 read r1
1 read r2

# compute $r1^2 + r2$
2 copy r13 r1
3 mul r13 r13 r13
4 add r13 r13 r2

# print the result
5 write r13
6 halt
A program that computes $r_1^{r_2}$

Lots of comments have been omitted, so that the program fits on this slide!

```plaintext
# read the inputs from the user
0 read r1    # base
1 read r2    # exponent

# initialize the result (r13)
2 se0n r13 1

# if the exponent (r2) is 0, the result is 1, so we can finish
3 jeqzn r2 7

4 mul r13 r13 r1    # result *= base
5 addn r2 -1        # decrement exponent
6 jgtzn r2 4        # do we need to keep iterating?

# we're done, print the output
7 write r13
8 halt
```
A program that computes $\mathbf{r1}^{\mathbf{r2}}$

Lots of comments have been omitted, so that the program fits on this slide!

```
0 read r1  # base
1 read r2  # exponent

# initialize the result (r13)
2 setn r13 1

# if the exponent (r2) is 0, the result is 1, so we can finish
3 jeqzn r2 7

2 r13 = r1^{r2}

4 mul r13 r13 r1  # result *= base
5 addn r2 -1  # decrement exponent
6 jgtzn r2 4  # do we need to keep iterating?

7 write r13
8 halt
```
A function that computes $r1^{r2}$?

0  read  r1  # base
1  read  r2  # exponent

# initialize the result (r13)
2  setn  r13  1

# if the exponent (r2) is 0, the result is 1, so we can finish
3  jeqzn  r2  0

2  r13 = r1^{r2}

4  mul  r13  r13  r1  # result *= base
5  addn  r2  -1  # decrement exponent
6  jgtzn  r2  4  # do we need to keep iterating?

7  write  r13
8  halt
A function that computes $r_1^{r_2}$?

0 read r1  # base
1 read r2  # exponent

2 $r_{13} = r_1^{r_2}$  # how can we say this?

3 write r13
4 halt

# initialize the result (r13)
6 setn r13 1

# if the exponent (r2) is 0, the result is 1, so we can finish
7 $r_{13} = r_1^{r_2}$

8 mul  r13 r13 r1  # result *= base
9 addn  r2  -1      # decrement exponent
10 jgtzn  r2  4     # do we need to keep iterating?
A function that computes \( r_1^{r_2} \)?

0 read r1  # base
1 read r2  # exponent
2 jumpn 5
3 write r13
4 halt

# initialize the result (r13)
6 setn r13 1

# if the exponent (r2) is 0, the result is 1, so we can finish
5 r13 = r1^{r_2}

8 mul r13 r13 r1  # result *= base
9 addn r2 -1  # decrement exponent
10 jgtzn r2 4  # do we need to keep iterating?
A function that computes $r_1^{r_2}$?

Oops! We’ve re-ordered the code. How can we fix it?

0 read r1  # base
1 read r2  # exponent

2 jumpn 5

3 write r13
4 halt

# initialize the result (r13)
5 setn r13 1

# if the exponent (r2) is 0, the result is 1, so we can finish
6 jeqzn r2 7

7 mul r13 r13 r1  # result *= base
8 addn r2 -1  # decrement exponent
9 jgtzn r2 4  # do we need to keep iterating?
A function that computes $r_1^{r_2}$

That’s better.

```
0 read r1      # base
1 read r2      # exponent
2 jumpn 5
3 write r13
4 halt

# initialize the result (r13)
5 setn r13 1

# if the exponent (r2) is 0, the result is 1, so we can finish
6 jeqzn r2 10

7 mul r13 r13 r1      # result *= base
8 addn r2 -1          # decrement exponent
9 jgtzn r2 7           # do we need to keep iterating?
10 jumpn 3             # return to location 3
```
Hmmm conventions
Human programming practices that help us write correct and readable programs

r0 always contains the value 0
r13 is for the return value
r14 is for the return location

Write lots of comments / documentation!
Clearer is better than shorter (but shorter can be clearer).
A function that computes $r1^{r2}$

Calls and returns in Hmmm: use `calln r14` # and `jumpn r14`

```
0 read r1  # base
1 read r2  # exponent
2 jumpn calln r14 5  # the same as:
3     setn r14 3
4     jumpn 5
5 write r13
6 halt

# initialize the result (r13)
7 setn r13 1

# if the exponent (r2) is 0, the result is 1, so we can finish
8 jeqzn r2 10

9 mul r13 r13 r1  # result *= base
10 addn r2 -1  # decrement exponent
11 jgtzn r2 7  # do we need to keep iterating?
12 jumpn 5  # return
```
A function that computes $r1^{r2}$

Calls and returns in Hmmm: use `calln r14 # and `jumpn r14`

```plaintext
0 read r1    # base
1 read r2    # exponent

2 calln r14 5  # r13 = r1^{r2}

3 write r13
4 halt

# initialize the result (r13)
5 setn r13 1

# if the exponent (r2) is 0, the result is 1, so we can finish
6 jeqzn r2 10

r13 = r1^{r2}

7 mul r13 r13 r1    # result *= base
8 addn r2 -1    # decrement exponent
9 jgtzn r2 7    # do we need to keep iterating?
10 jumpn r14    # return
```
A function that computes $r_1^{r_2}$

Uh, oh... What if we want to call the function lots of times, with different arguments?

```
0 read r1  # base
1 read r2  # exponent
2 calln r14 7  # r13 = r1^{r2}
3 write r13

4 calln r14 7  # r13 = r1^{r3}  ← This doesn’t work!!!!
5 write r13

6 halt
```

$r_{13} = r_1^{r_2}$
A function that computes $r_1^{r_2}$

Uh, oh... What if we want to call the function lots of times, with different arguments?

```plaintext
0  read  r1  # base
1  read  r2  # exponent
2  calln  r14  8  # r13 = r1^{r2}
3  write  r13
4  setn  r2  3  # prepare the argument
5  calln  r14  8  # r13 = r1^{r3}
6  write  r13
7  halt
```

$r13 = r1^{r2}$
A function that computes $r_1^{r_2}$
Uh, oh. What if we need the register values after the call?

```
0 read r1    # base
1 read r2    # exponent

2 # r13 = r1^{r_2} + r2

3 write r13
4 halt

# initialize the result (r13)
5 setn r13 1

# if the exponent (r2) is 0, the result is 1, so we can finish
6 jeqzn r2 10

7 mul r13 r13 r1  # result *= base
8 addn r2 -1      # decrement exponent
9 jgtzn r2 7      # do we need to keep iterating?
10 jump r14        # return
```
Function calls in Hmmm

Caller (outside the function): assume the function writes to every register

Callee (inside the function): assume every register is yours

```plaintext
# save any register values that I’ll need later
caller

# prepare the arguments

# call the function

calln r14 N  # call the function

# restore all the register values that I saved
caller

N  # function start
callee

write to registers with gleeful abandon
if the function should return a value, save it in r13

M jump r14  # return
callee

# treat register values like local variables!
```
The stack
A place in RAM where we can save values for later

stack pointer
the “top” of the stack (where the next value will go)
Loads and stores

Registers

- r0
- r1: 100
- r2: 43
- r3
- r4
- r5
- r6
- r7: -1
- r8: 51
- r9
- r10
- r11

RAM

- 42
- 43
- 100
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56

Operations:
- store: store
- load: load
- store
  - store
  - r1: 43
- store
  - r1
  - r2
- load
  - load
  - r7: 51
- load
  - r7
  - r8
Hmmm conventions

Human programming practices that help us write correct and readable programs

r0 always contains the value 0
r13 is for the return value
r14 is for the return location
r15 is for the stack pointer

Write lots of comments / documentation!
Clearer is better than shorter (but shorter can be clearer).
The stack

“Last-in, first out” (LIFO)

storer r1 r15
addn r15 1
storer r2 r15
addn r15 1

stack pointer
the “top” of the stack
(where the next value will go)

addn r15 -1
loadr r2 r15
addn r15 -1
loadr r1 r15

RAM

<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>19</td>
</tr>
<tr>
<td>44</td>
<td>23</td>
</tr>
<tr>
<td>45</td>
<td>154</td>
</tr>
<tr>
<td>46</td>
<td>r1</td>
</tr>
<tr>
<td>47</td>
<td>r2</td>
</tr>
<tr>
<td>48</td>
<td>...</td>
</tr>
<tr>
<td>49</td>
<td>...</td>
</tr>
<tr>
<td>50</td>
<td>...</td>
</tr>
<tr>
<td>51</td>
<td>...</td>
</tr>
</tbody>
</table>
Function calls in Hmmm

Caller (outside the function): assume the function writes to every register

Callee (inside the function): assume every register is yours

```asm
# initialize stack pointer
setn   r15  S

# save any register value that I’ll need later
storer rN  r15
addn   r15  1

# prepare the arguments

calln r14  N  # call the function

# restore all the register values that I saved (LIFO!)
addn   r15  -1
loadr rN  r15

N  # function start
callee
write to registers with gleeful abandon
if the function should return a value, save it in r13
M jump r14  # return
caller

# function start
callee
write to registers with gleeful abandon
treat register values like local variables!
```
Let’s practice!

Write a Hmmm program that reads a positive integer value $n$ into r1, then writes the value $n! + n$ to the screen.

*Ask yourself: which register(s) do I need to save / restore?*

Here is a function that computes $r1!$ and stores the result in r13:

```assembly
42 setn r13 1
43 jeqzn r1 47
44 mul r13 r13 r1
45 addn r1 -1
46 jumpn 43
47 jump r14
```

$r13 = r1!$

*Bonus: can you write a recursive factorial?*
Let’s practice!
A Hmmm program that reads a positive integer value n into r1, then writes the value n! + n to the screen.

```hmmm
# initialization
00  setn  r15  17  # start the stack pointer at address 100

# read the input from the user
01  read  r1

# save r1 so we can use it after the function call
02  storer  r1  r15
03  addn  r15  1

# call the factorial function: r13 = n!
04  calln  r14  10

# restore r1, so we can add it to the result of r1!
05  addn  r15  -1
06  loadr  r1  r15

# compute and print the result
07  add  r13  r13  r1  # r13 += r1
08  write  r13
09  halt
```

10  r13  =  r1!
Human programming practices that help us write correct and readable programs

Hmmm conventions

- `r0` always contains the value 0
- `r13` is for the return value
- `r14` is for the return location
- `r15` is for the stack pointer

Write lots of comments / documentation!
Clearer is better than shorter (but shorter can be clearer).
Function calls in Hmmm

Caller (outside the function): assume the function writes to every register

Callee (inside the function): assume every register is yours

# initialize stack pointer
setn r15 S

# save any register value that I’ll need later
caller
storer rN r15
addn r15 1

# prepare the arguments

calln r14 N # call the function

# restore all the register values that I saved (LIFO!)
callee
addn r15 -1
loadr rN r15

N # function start
callee
write to registers with gleeful abandon
if the function should return a value, save it in r13

M jump r14 # return

treat register values like local variables!