Write a Hmmm program that:
1. reads a number $n$ from the user
2. doubles $n$
3. writes the result ($2n$) to the screen
Take-home midterm #1

**Available:** this Sunday night (9/30)

**Must return by:** next Sunday (10/7) at 5pm

**Time-limit:** one sitting (with small breaks)

**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.
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).
A program that computes $2n$

00  read  r1
01  add  r1  r1  r1  # $n = 2 \times n$
02  write  r1
03  halt
A program that computes 2n

This program uses a “function” to compute 2n

00  read  r1
01  jumpn 04  # double(n)
02  write  r1
03  halt

# double(n)
04  add  r1  r1  r1  # n = 2 * n
05  jumpn 02
A program that computes 4n
Uh, oh. What if we need to call the function twice?
We’ll need a place to save the “return location”...

00  read  r1
01  jumpn 05     # double(n)
02  jumpn 05     # double(n)
03  write  r1
04  halt
     # double(n)
05  add  r1 r1 r1  # n = 2 * n
06  jumpn 02
Human programming practices that help us write correct and readable programs

Hmmm conventions

r0 always contains the value 0

r14 is for the return location

Write lots of comments / documentation!
Clearer is better than shorter (but shorter can be clearer).
Calling a function

Set r14 to be the next line, then jump to the start of the function

```
00  read  r1
01  calln  r14  05
02  ...

...  

05  add  r1  r1  r1
06  ...
```

```
        pc  01  pc  05
r14  0   r14  02
```

```
calln  r14  05
```
Returning from a function

`jumpr r14`

```
00  read  r1
01  calln  r14  05
02  ...

  ...

05  add  r1  r1  r1
06  jumpr  r14
```

```
<table>
<thead>
<tr>
<th>pc</th>
<th>06</th>
</tr>
</thead>
<tbody>
<tr>
<td>r14</td>
<td>02</td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th>pc</th>
<th>02</th>
</tr>
</thead>
<tbody>
<tr>
<td>r14</td>
<td>02</td>
</tr>
</tbody>
</table>

“return location”

`jumpr r14`
A program that computes $4n$

This program saves the return location.

```
00  read  r1
01  calln  r14  05  # double(n)
02  calln  r14  05  # double(n)
03  write  r1
04  halt
# double(n)
05  add  r1  r1  r1  # $n = 2 \times n$
06  jump  r14
```
Hmmm conventions
Human programming practices that help us write correct and readable programs

\textbf{r0} always contains the value 0
\textbf{r13} is for the return value
\textbf{r14} is for the return location

Write lots of comments / documentation! Clearer is better than shorter (but shorter \textit{can be} clearer).
A program that computes 4n

This program saves the return value and return location.

```
00  read  r1
01  calln r14  06
02  copy  r1  r13  # prepare arguments
03  calln r14  06
04  write  r13
05  halt
# double(n)
06  add  r13  r1  r1  # return value = 2 * n
07  jmp   r14
```
A program that computes $4n$

Uh, oh. What if the caller needs the register values after the call? We’ll need a place to save the caller’s state...

```
00  read  r1
01  calln r14 06  # quadruple(n)
02  write r13
03  halt

# double(n)
04  add  r13 r1 r1  # return value = $2 \times n$
05  jump r14

# quadruple(n)
06  calln r14 04  # double(n)
07  copy  r1 r13  # prepare arguments
08  calln r14 04  # double(n)
09  jump r14
```
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)
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
A place in RAM where we can save values for later

stack pointer
the “top” of the stack
(where the next value will go)
Pushing onto the stack
Add a new value to the “top” of the stack

```
pushr  r1  r15
```
Popping from the stack

Take an existing value from the “top” of the stack

<table>
<thead>
<tr>
<th>RAM</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_{1} )</td>
<td>42 ( \rightarrow ) 43 ( \rightarrow ) 44 ( \rightarrow ) 45 ( \rightarrow ) 46 ( \rightarrow ) 47 ( \rightarrow ) 48 ( \rightarrow ) 49 ( \rightarrow ) 50 ( \rightarrow ) 51</td>
</tr>
<tr>
<td>( r_{15} )</td>
<td>47 ( \rightarrow ) 43 ( \rightarrow ) 44 ( \rightarrow ) 45 ( \rightarrow ) 46 ( \rightarrow ) 47 ( \rightarrow ) 48 ( \rightarrow ) 49 ( \rightarrow ) 50 ( \rightarrow ) 51</td>
</tr>
</tbody>
</table>

```
popr  r1  r15
```
The stack

“Last-in, first out” (LIFO): pop values in reverse order

push\(r\)  \(r1\ r15\)
push\(r\)  \(r2\ r15\)

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

pop\(r\)  \(r2\ r15\)
pop\(r\)  \(r1\ r15\)

RAM

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<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></td>
</tr>
<tr>
<td>47</td>
<td></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>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A program that computes $4n$

This program uses the stack to save and restore the caller’s state.

```
00  setn  r15  100  # stack starts at address 100 & grows UP
01   read  r1
02  calln  r14  07  # quadruple(n)
03   write r13
04     halt

# double(n)
05  add  r13  r1  r1  # return value = 2 * n
06  jumper  r14

# quadruple(n)
07  pushr  r14  r15
08  calln  r14  05  # double(n)
09  popr  r14  r15
10  copy  r1  r13  # prepare arguments
11  pushr  r14  r15
12  calln  r14  05  # double(n)
13  popr  r14  r15
14  jumper  r14
```
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
pushr rN r15

# prepare the arguments by assigning values to registers
calln r14 N  # call the function

# restore all the register values that I saved (LIFO!)
popr rN r15

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

[tinyurl.com/hmc-hmmm](http://tinyurl.com/hmc-hmmm)

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

```
10  setn  r13  1
11  jeqzn  r1  15
12  mul  r13  r13  r1
13  addn  r1  -1
14  jumpn  11
15  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 100  # 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 pushr r1 r15

# call the factorial function: \( r13 = n! \)
03 calln r14 10

# restore r1, so we can add it to the result of \( r1! \)
04 popr r1 r15

# compute and print the result
05 add r13 r13 r1  # \( r13 += r1 \)
06 write r13
07 halt

10 r13 = r1!
```
Hmmm conventions

Human programming practices that help us write correct and readable programs

\texttt{r0} always contains the value 0
\texttt{r13} is for the return value
\texttt{r14} is for the return location
\texttt{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

```assembly
# initialize stack pointer
setn r15 S

# save any register value that I’ll need later
pushr rN r15

# prepare the arguments by assigning values to registers
calln r14 N  # call the function

# restore all the register values that I saved (LIFO!)
popr rN r15
```

```assembly
N  # function start
# 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!
What kinds of problems can computers solve?

Decision problems on finite, bitstring inputs.

Can sequential logic solve all the problems that a DFA can? How about a Turing Machine?

Yes! No!