CS 5 Black slides invaded by farm animals!

What are these alien creatures doing in MY slides!?
True Story…

Douglas Adams

Joe Costello, HMC ‘74
Which is the *real* Death Adder?

We prefer multiplication!
Homework 6 and Fall Break

Two interesting readings! The second one is on the “Attention Economy”

- **Reading** [5 Points; individual only]
  (filename: hw6pr0.txt)

- **Lab 6: Countdown to Hmmm and Randohmmm numbers** [30 points; individual or pair]
  (hw6pr1.py)

- **Problem 2: Fibonacci Fun** [15 points; individual or pair]
  (hw6pr2.py)

- **Problem 3: The Towers of Hanoi** [20 points; individual or pair]
  (hw6pr3.py)

- **Problem 4: The Game of Nim: Coming Soon!?** [30 points; individual only]
  (hw6pr4.py)

- **Problem 5: Ackermann Function** [Ex. Cr. up to +10 points; individual or pair]
  (hw6pr5.py)

Reading and Problems 2, 4 and 5 due on Monday October 23 (almost two weeks from now!)

- Attend lab or submit by this Wednesday at 11:59 PM
- Bring your laptop to class on Thursday!
  We'll get started on this one!

Some of these problems quacked me up!
the *fetch - execute* cycle

Central Processing Unit (CPU) central processing unit *registers*

Random Access Memory (RAM) random access memory locations

Program Counter

- Holds address of the next instruction

Instruction Register

- Holds the current instruction

General-purpose register r1

- General-purpose register r1

General-purpose register r2

- General-purpose register r2

Von Neumann bottleneck

0. read r1

1. mul r2 r1 r1 r1

2. add r2 r2 r2 r1

3. write r2

4. halt
Hmmm
The Harvey Mudd Miniature Machine

CPU
central processing unit

RAM
random access memory

Von Neumann bottleneck

<table>
<thead>
<tr>
<th>Program Counter</th>
<th>Holds address of the next instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Register</td>
<td>Holds the current instruction</td>
</tr>
</tbody>
</table>

register 0 is “hard-wired” to store 0

<table>
<thead>
<tr>
<th>r0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>r15</td>
<td>...</td>
</tr>
</tbody>
</table>

16 registers, each 16 bits
they can hold values from -32768 upto 32767

read r1
mul r2 r1 r1
add r2 r2 r1
write r2
halt

255 memory locations of 16 bits
Assembly Language

```
add r2 r2 r2
sub r2 r1 r4
mul r7 r6 r2
div r1 r1 r1 r1
setn r1 42
addn r1 -1
```

\[
\text{reg2} = \text{reg2} + \text{reg2}
\]
crazy, perhaps, but used ALL the time

\[
\text{reg2} = \text{reg1} - \text{reg4}
\]
which is why it is written this way in python!

\[
\text{reg7} = \text{reg6} \times \text{reg2}
\]

\[
\text{reg1} = \text{reg1} / \text{reg1}
\]
INTEGER division - no remainders

\[
\text{reg1} = 42
\]
you can replace 42 with anything from -128 to 127

\[
\text{reg1} = \text{reg1} - 1
\]
a shortcut

```
read r0
write r0
```
read from keyboard and write to screen

Each of these instructions (and many more) get implemented for a particular processor and particular machine… .
n is for number...

r1  
add r1 r1 r2

r2  42
addn r1 42
Hmmm, Let's get jumpn!

**CPU**
- Central Processing Unit

**RAM**
- Random Access Memory

---

Program Counter
- Holds address of the next instruction

Instruction Register
- Holds the current instruction

**r0**
- Register 0 is "hard-wired" to store 0

**r1**
- General-purpose register r1

**r2**
- General-purpose register r2

---

**setn r1 42**

**write r1**

**addn r1 2**

**jumpn 1**

**halt**
**jumps**

**Unconditional** jump  
```plaintext
goto
```

### jumpn 42

replaces the PC (program counter) with 42.  
"jumpnto line number 42"

---

**Conditional** jumps  
```plaintext
if
```

### jeqzn r1 93

IF r1 == 0 THEN jump to line number 93

### jgtzn r1 93

IF r1 > 0 THEN jump to line number 93

### jltzn r1 93

IF r1 < 0 THEN jump to line number 93

### jnezn r1 93

IF r1 != 0 THEN jump to line number 93

Notice all the n’s here!
What is this code computing about its input?
Follow this assembly-language program from top to bottom. Use \( r1 = 42 \) and \( r2 = 5 \). Then, try \( r1 = 5 \) and \( r2 = 42 \).

Write an assembly-language program that reads one integer as keyboard input. Then, the program should compute the factorial of that input and write it out. You may assume without checking that the input will be a positive integer.

(1) What does this program compute in general?

**Hint:** Take in an input. Next, set up a “result” register starting with 1 in it. Then modify the “result” until it’s right!
Follow this assembly-language program from top to bottom. Use \( r1 = 42 \) and \( r2 = 5 \). Then, try \( r1 = 5 \) and \( r2 = 42 \).

Write an assembly-language program that reads one integer as keyboard input. Then, the program should compute the \textit{factorial} of that input and write it out. You may assume without checking that the input will be a positive integer.

(1) What does this program compute in general?

\[ \text{read } r1 \]
\[ \text{read } r2 \]
\[ \text{sub } r3 \text{ } r2 \text{ } r1 \]
\[ \text{nop} \]
\[ \text{jltzn } r3 \text{ } 7 \]
\[ \text{write } r1 \]
\[ \text{jumpn } 8 \]
\[ \text{write } r2 \]
\[ \text{halt} \]

\[ \text{write } r2 \]
\[ \text{halt} \]

\textit{Hint}: Take in an input. Next, set up a “result” register starting with 1 in it. Then modify the “result” until it’s right!
Follow this assembly-language program from top to bottom. Use \( r1 = 42 \) and \( r2 = 5 \). Then, try \( r1 = 5 \) and \( r2 = 42 \).

Write an assembly-language program that reads one integer as keyboard input. Then, the program should compute the factorial of that input and write it out. You may assume without checking that the input will be a positive integer.

(1) What does this program compute in general?

Hint: Take in an input. Next, set up a “result” register starting with 1 in it. Then modify the “result” until it’s right!
1. Follow this assembly-language program from top to bottom. Use $r1 = 42$ and $r2 = 5$. Then, try $r1 = 5$ and $r2 = 42$.

2. Write an assembly-language program that reads one integer as keyboard input. Then, the program should compute the **factorial** of that input and write it out. You may assume without checking that the input will be a positive integer.

(1) What does this program compute in general?

**Hint:** Take in an input. Next, set up a “result” register starting with 1 in it. Then modify the “result” until it’s right!
Follow this assembly-language program from top to bottom. Use \( r_1 = 42 \) and \( r_2 = 5 \). Then, try \( r_1 = 5 \) and \( r_2 = 42 \).

Write an assembly-language program that reads one integer as keyboard input. Then, the program should compute the factorial of that input and write it out. You may assume without checking that the input will be a positive integer.

(1) What does this program compute in general?

**Hint:** Take in an input. Next, set up a “result” register starting with 1 in it. Then modify the “result” until it’s right!
Follow this assembly-language program from top to bottom. Use \( r1 = 42 \) and \( r2 = 5 \). Then, try \( r1 = 5 \) and \( r2 = 42 \).

Write an assembly-language program that reads one integer as keyboard input. Then, the program should compute the factorial of that input and write it out. You may assume without checking that the input will be a positive integer.

(1) What does this program compute in general?

**Hint:** Take in an input. Next, set up a “result” register starting with 1 in it. Then modify the “result” until it’s right!
Worksheet

1. Follow this assembly-language program from top to bottom. Use \( r1 = 42 \) and \( r2 = 5 \). Then, try \( r1 = 5 \) and \( r2 = 42 \).

```
Registers - CPU
0  r0  0
1  r1  5
2  r2  42
3  r3  37
4  r4  

Memory - RAM
0  read r1
1  read r2
2  sub r3 r2 r1
3  nop
4  jltzn r3 7
5  write r1
6  jumpn 8
7  write r2
8  halt
9
```

(1) What does this program compute in general?

2. Write an assembly-language program that reads one integer as keyboard input. Then, the program should compute the factorial of that input and write it out. You may assume without checking that the input will be a positive integer.

```
Memory - RAM
0  0
1  1
2  2
3  3
4  4
5  5
6  6
7  7
8  8
9  9
10 10
```

```
Registers - CPU
0  0
1  1
2  2
3  3
4  4
5  5
6  6
7  7
8  8
9
```

```
0! = 1
```

**Hint:** Take in an input. Next, set up a “result” register starting with 1 in it. Then modify the “result” until it’s right!
Factorial...

00  read r1  # read input into r1
01  setn r2 1  # load 1 into r2
02  jeqzn r1 6  # if r1 == 0, jump to line 6
03  mul r2 r2 r1  # r2 = r2 * r1
04  addn r1 -1  # r1 = r1 - 1
05  jumpn 2  # jump to line 2
06  write r2
07  halt

You now have all you need for this week’s lab problem...

Demo Example1!
# System instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>Stop!</td>
</tr>
<tr>
<td>read rX</td>
<td>Place user input in register rX</td>
</tr>
<tr>
<td>write rX</td>
<td>Print contents of register rX</td>
</tr>
<tr>
<td>nop</td>
<td>Do nothing</td>
</tr>
</tbody>
</table>

# Setting register data

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setn rX N</td>
<td>Set register rX equal to the integer N (-128 to +127)</td>
</tr>
<tr>
<td>addn rX N</td>
<td>Add integer N (-128 to 127) to register rX</td>
</tr>
<tr>
<td>copy rX rY</td>
<td>Set rX = rY</td>
</tr>
<tr>
<td>mov</td>
<td></td>
</tr>
</tbody>
</table>

# Arithmetic

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add rX rY rZ</td>
<td>Set rX = rY + rZ</td>
</tr>
<tr>
<td>sub rX rY rZ</td>
<td>Set rX = rY - rZ</td>
</tr>
<tr>
<td>neg rX rY</td>
<td>Set rX = -rY</td>
</tr>
<tr>
<td>mul rX rY rZ</td>
<td>Set rX = rY * rZ</td>
</tr>
<tr>
<td>div rX rY rZ</td>
<td>Set rX = rY / rZ (integer division; no remainder)</td>
</tr>
<tr>
<td>mod rX rY rZ</td>
<td>Set rX = rY % rZ (returns the remainder of integer division)</td>
</tr>
</tbody>
</table>

# Jumps!

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jumpn N</td>
<td>Set program counter to address N</td>
</tr>
<tr>
<td>jump rX</td>
<td>Set program counter to address in rX</td>
</tr>
<tr>
<td>jeqz rX N</td>
<td>If rX == 0, then jump to line N</td>
</tr>
<tr>
<td>jnez rX N</td>
<td>If rX != 0, then jump to line N</td>
</tr>
<tr>
<td>jgtz rX N</td>
<td>If rX &gt; 0, then jump to line N</td>
</tr>
<tr>
<td>jltz rX N</td>
<td>If rX &lt; 0, then jump to line N</td>
</tr>
<tr>
<td>call rX N</td>
<td>Copy the next address into rX and then jump to mem. addr. N</td>
</tr>
</tbody>
</table>

# Interacting with memory (RAM)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadn rX N</td>
<td>Load register rX with the contents of memory address N</td>
</tr>
<tr>
<td>storen rX N</td>
<td>Store contents of register rX into memory address N</td>
</tr>
<tr>
<td>load rY</td>
<td>Load register rX with data from the address location held in reg. rY</td>
</tr>
<tr>
<td>store rY</td>
<td>Store contents of register rX into memory address held in reg. rY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alias</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>loadi</td>
<td>load</td>
</tr>
<tr>
<td>storei</td>
<td>store</td>
</tr>
</tbody>
</table>
Why program in assembly language?

I was wondering that, but felt a bit too sheepish to ask!
A function call in python:

def main():
    r1 = input()
    result = factorial(r1)
    print result

Hmmm's call operation:

puts NEXT line # into r14, then jumps to line 4

def factorial( r1 ):
    # do work
    return result

0  read r1
1  calln r14 4
2  write r13
3  halt

4  do stuff and
5  answer in r13
6  jump r14
def main():
    r1 = input()
    result = factorial(r1)
    print(result)

def factorial(r1):
    # do work
    return result
What does this do?

00  read  r1
01  jeqzn  r1  5  # if r1==0 then jump to line 5
02  calln  r14  6
03  write  r13
04  jumpn  0  # jump to line 0
05  halt
06  setn  r13  1  # r13 = 1
07  setn  r2  2  # r2 = 2
08  jeqzn  r1  12
09  addn  r1  -1
10  mul  r13  r13  r2
11  jumpn  8
12  jumpn  r14  # jump to the address in r14

Demo Example2!
A function call in python:

United Nations Resolution 424242

- Functions always receive their input in register `r1` (and `r2`, `r3`, and so forth if there more inputs)
- Functions return their answers in register `r13`
- Functions `jumpr r14` to return to where they were called

Hmmm's `call` operation:

```
0   read r1
1   calln r14 4
2   write r13
3   halt

4   do stuff...
5   put in r13
6   jumpr r14
```

puts NEXT line # into `r14`, then jumps to line 4

Who ya gonna call?
The Collatz Function!

def collatz(n):
    counter = 0
    while n != 1:
        counter += 1  # increment counter
        if n % 2 == 0:  n = n // 2
        else: n = 3*n + 1
    return counter

>>> collatz(8)
6
>>> collatz(5)
5
>>> collatz(27)
112
def main():
    read r1
    r13 = collatz(r1)
    write r13
    halt

def collatz(r1):
    r13 = 0 # this is the result
TOP: if r1 == 1: jump to END
else:
    r13 = r13 + 1
    if r1 is even:
        r1 = r1 // 2
        jump back to TOP
    else:
        r1 = 3 * r1 + 1
        jump back to TOP
END: return to function that called us!

One tricky thing is that there’s no way to test if r1 == 1.
def main():
    read r1
    r13 = collatz(r1)
    write r13
    halt

def collatz(r1):
    r13 = 0  # this is the result

    TOP:
    if r1 == 1: jump to END
    else:
        r13 = r13 + 1
        if r1 is even:
            r1 = r1 // 2
            jump back to TOP
        else:
            r1 = 3 * r1 + 1
            jump back to TOP
    END: return to function that called us!

Demo (let’s try Collatzing 27)!
Collatz Revisited

From wikipedia!
def main():
    r1= input()  # r1=3
    r13 = emma(r1)
    r13 = r13+ r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

In other words, hare-y up and figure out what main will return…

Chew on this!
Function Calls…

def main():
    r1 = input()  # r1=3
    r13 = emma(r1)  # emma(3)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
def main():
    r1 = input()  # r1=3
    r13 = emma(r1)  # emma(3)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1  # r1=4
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

You should be worried!

emma(3)
def main():
    r1 = input()  # r1 = 3
    r13 = emma(r1)  # emma(3)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1  # r1 = 4
    r13 = kyle(r1)  # kyle(4)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42  # r1 = 46
    r13 = r1 + 1
    return r13
def main():
    r1 = input()  # r1=3
    r13 = emma(r1)  # emma(3)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1  # r1=4
    r13 = kyle(r1)  # kyle(4)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42  # r1=46
    r13 = r1 + 1  # r13=47
    return r13  # return(47)
def main():
    r1=input()
    r13 = emma(r1)    # emma(3)
    r13 = r13+ r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)   # kyle(4)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
def main():
    r1 = input()  # r1=3
    r13 = emma(r1)  # emma(3)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1  # r1=4
    r13 = kyle(r1)  # kyle(4)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42  # r1=46
    r13 = r1 + 1  # r13=47
    return r13  # return(47)
def main():
    r1 = input()  # r1=3
    r13 = emma(r1)  # emma(3)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1  # r1=4
    r13 = kyle(r1)  # kyle(4)  r13=47
    r13 = r13 + r1  # r13=??
    return r13

def kyle(r1):
    r1 = r1 + 42  # r1=46
    r13 = r1 + 1  # r13=47
    return r13  # return(47)
def main():
    r1=input()  
    r13 = emma(r1)  
    r13 = r13+ r1
    print r13
    return

def emma(r1):  
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
Function Calls…

def main():
    r1 = int(input())
    r1 = 3
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r1 = 4
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r1 = 46
    r13 = r1 + 1
    r13 = 47
    return r13 + 1
    return 47
def main():
    r1 = input()  # r1=3
    r13 = emma(r1)  # emma(3)  # r13=51
    r13 = r13 + r1
    print r13
    return

def emma(r1):  # r1=3
    r1 = r1 + 1  # r1=4
    r13 = kyle(r1)  # kyle(4)  # r13=47
    r13 = r13 + r1  # r13=51
    return r13  # return(51)

def kyle(r1):  # r1=4
    r1 = r1 + 42  # r1=46
    r13 = r1 + 1  # r13=47
    return r13  # return(47)
Function Calls…

```python
def main():
    r1 = input()  # r1 = 3
    r13 = emma(r1)  # emma(3)  # r13 = 51
    r13 = r13 + r1  # r13 = ??
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1  # r1 = 4
    r13 = kyle(r1)  # kyle(4)  # r13 = 47
    r13 = r13 + r1  # r13 = 51
    return r13  # return(51)

def kyle(r1):
    r1 = r1 + 42  # r1 = 46
    r13 = r1 + 1  # r13 = 47
    return r13  # return(47)
```

```python
r1 = 3
emma(r1)  # r1 = 3  # r13 = 51
r1 = 3
r1 = 4
kyle(r1)  # kyle(4)  # r13 = 47
r1 = 4
r1 = 46
r13 = 51
return(r13)
```

```python
r13 = 47
```
def main():
    r1 = input()  # r1=3
    r13 = emma(r1)  # emma(3)  # r13=51
    r13 = r13 + r1  # r13=54
    print(r13)  # 54
    return

def emma(r1):
    r1 = r1 + 1  # r1=4
    r13 = kyle(r1)  # kyle(4)  # r13=47
    r13 = r13 + r1  # r13=51
    return r13  # return(51)

def kyle(r1):
    r1 = r1 + 42  # r1=46
    r13 = r1 + 1  # r13=47
    return r13  # return(47)

Cool, but how does this work!?!
def main():
    r1 = input()  # Get input
    r1 = r1 + 1
    r13 = emma(r1)  # Call emma function
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)  # Call kyle function
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r1 = r1 + 1
    return r13

The stack in RAM!

r1 = 3
r13

code up here!

United Nations Resolution 424242

- Functions always receive their input in register \texttt{r1}
- Functions return their answers in register \texttt{r13}
- Functions jump \texttt{r14} to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

• Functions always receive their input in register \texttt{r1}
• Functions return their answers in register \texttt{r13}
• Functions jump \texttt{r14} to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

United Nations Resolution 424242
- Functions always receive their input in register \textbf{r1}
- Functions return their answers in register \textbf{r13}
- Functions \texttt{jumpr \textbf{r14}} to return to where they were called
```python
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
```

United Nations Resolution 424242

- Functions always receive their input in register `r1`
- Functions return their answers in register `r13`
- Functions jump to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
```python
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
```

- Functions always receive their input in register `r1`
- Functions return their answers in register `r13`
- Functions jump to return to where they were called
Function Calls

def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

Hmmm code up here!

The stack in RAM!

United Nations Resolution 424242

- Functions always receive their input in register \texttt{r1}
- Functions return their answers in register \texttt{r13}
- Functions \texttt{jump} \texttt{r14} to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

United Nations Resolution 424242

- Functions always receive their input in register r1
- Functions return their answers in register r13
- Functions jump r14 to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

United Nations Resolution 424242

- Functions always receive their input in register r1
- Functions return their answers in register r13
- Functions jmp r14 to return to where they were called
Function Calls...

```python
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
```

- Functions always receive their input in register `r1`
- Functions return their answers in register `r13`
- Functions `jumpr r14` to return to where they were called
Function Calls…

def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

United Nations Resolution 424242

- Functions always receive their input in register **r1**
- Functions return their answers in register **r13**
- Functions jmp r**14** to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

• Functions always receive their input in register r1
• Functions return their answers in register r13
• Functions jump to return to where they were called
Function Calls...

```python
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
```

The stack in RAM:

- Functions always receive their input in register `r1`
- Functions return their answers in register `r13`
- Functions jump to `r14` to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

United Nations Resolution 424242

• Functions always receive their input in register \texttt{r1}
• Functions return their answers in register \texttt{r13}
• Functions jump to \texttt{r14} to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

United Nations Resolution 424242

- Functions always receive their input in register \texttt{r1}
- Functions return their answers in register \texttt{r13}
- Functions \texttt{jumpr r14} to return to where they were called
```python
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
```

United Nations Resolution 424242

- Functions always receive their input in register \texttt{r1}
- Functions return their answers in register \texttt{r13}
- Functions jump \texttt{r14} to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

Hmmm code up here!

The stack in RAM!

United Nations Resolution 424242

• Functions always receive their input in register \texttt{r1}
• Functions return their answers in register \texttt{r13}
• Functions \texttt{jumpr r14} to return to where they were called
```python
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
```

**United Nations Resolution 424242**

- Functions always receive their input in register `r1`
- Functions return their answers in register `r13`
- Functions jump to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

- Functions always receive their input in register r1
- Functions return their answers in register r13
- Functions jumpr r14 to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

• Functions always receive their input in register r1
• Functions return their answers in register r13
• Functions jmp r14 to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

United Nations Resolution 424242
- Functions always receive their input in register r1
- Functions return their answers in register r13
- Functions jump r14 to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

United Nations Resolution 424242
• Functions always receive their input in register \texttt{r1}
• Functions return their answers in register \texttt{r13}
• Functions \texttt{jumpr r14} to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

United Nations Resolution 424242

- Functions always receive their input in register r1
- Functions return their answers in register r13
- Functions jump to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

United Nations Resolution 424242

- Functions always receive their input in register r1
- Functions return their answers in register r13
- Functions jump to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

United Nations Resolution 424242

- Functions always receive their input in register \texttt{r1}
- Functions return their answers in register \texttt{r13}
- Functions jmp\texttt{r14} to return to where they were called
```python
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
```

United Nations Resolution 424242

- Functions always receive their input in register `r1`
- Functions return their answers in register `r13`
- Functions `jumpr r14` to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

United Nations Resolution 424242

- Functions always receive their input in register r1
- Functions return their answers in register r13
- Functions jmp r14 to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

United Nations Resolution 424242

- Functions always receive their input in register r1
- Functions return their answers in register r13
- Functions jump r14 to return to where they were called
```python
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
```

United Nations Resolution 424242

- Functions always receive their input in register \( r1 \)
- Functions return their answers in register \( r13 \)
- Functions `jumpr \( r14 \)` to return to where they were called
```python
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
```

**United Nations Resolution 424242**
- Functions always receive their input in register **r1**
- Functions return their answers in register **r13**
- Functions jump to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

United Nations Resolution 424242

- Functions always receive their input in register \texttt{r1}
- Functions return their answers in register \texttt{r13}
- Functions jump to return to where they were called
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
Function Calls...

def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print r13
    return

def emma(r1):
    r1 = r1 + 1
    r13 = kyle(r1)
    r13 = r13 + r1
    return r13

def kyle(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

United Nations Resolution 424242

- Functions always receive their input in register r1
- Functions return their answers in register r13
- Functions jump r14 to return to where they were called