METU Computer Engineering

Ceng 111 – Fall 2021
Week 10b

Credit: Some slides are from the “Invitation to Computer Science” book by G. M. Schneider, J. L. Gersting and some from the “Digital Design” book by M. M. Mano and M. D. Ciletti.

Nested Functions in Python

Previously on CENG111!

```
def f(N):
        Number = N
        def g():
                C = 20
                return N * Number
        print "Number", N, "and its square:", g()
```

- Function `g()` can access all the local variables as well as the parameters of function `f()`.
- Function `f()` cannot access the local variables of function `g()`!
- Function `g()` cannot be used before it is defined! For example, the second line could not have been `Number = 10 * g(10)`.
- The indentation is extremely important to understand which statement belongs to which function! For example, the last line is part of function `f()` since they are at the same indentation!

Previously on CENG111!

Global Variables in Python

- To access variables in the global workspace, you should use “global <varname>”

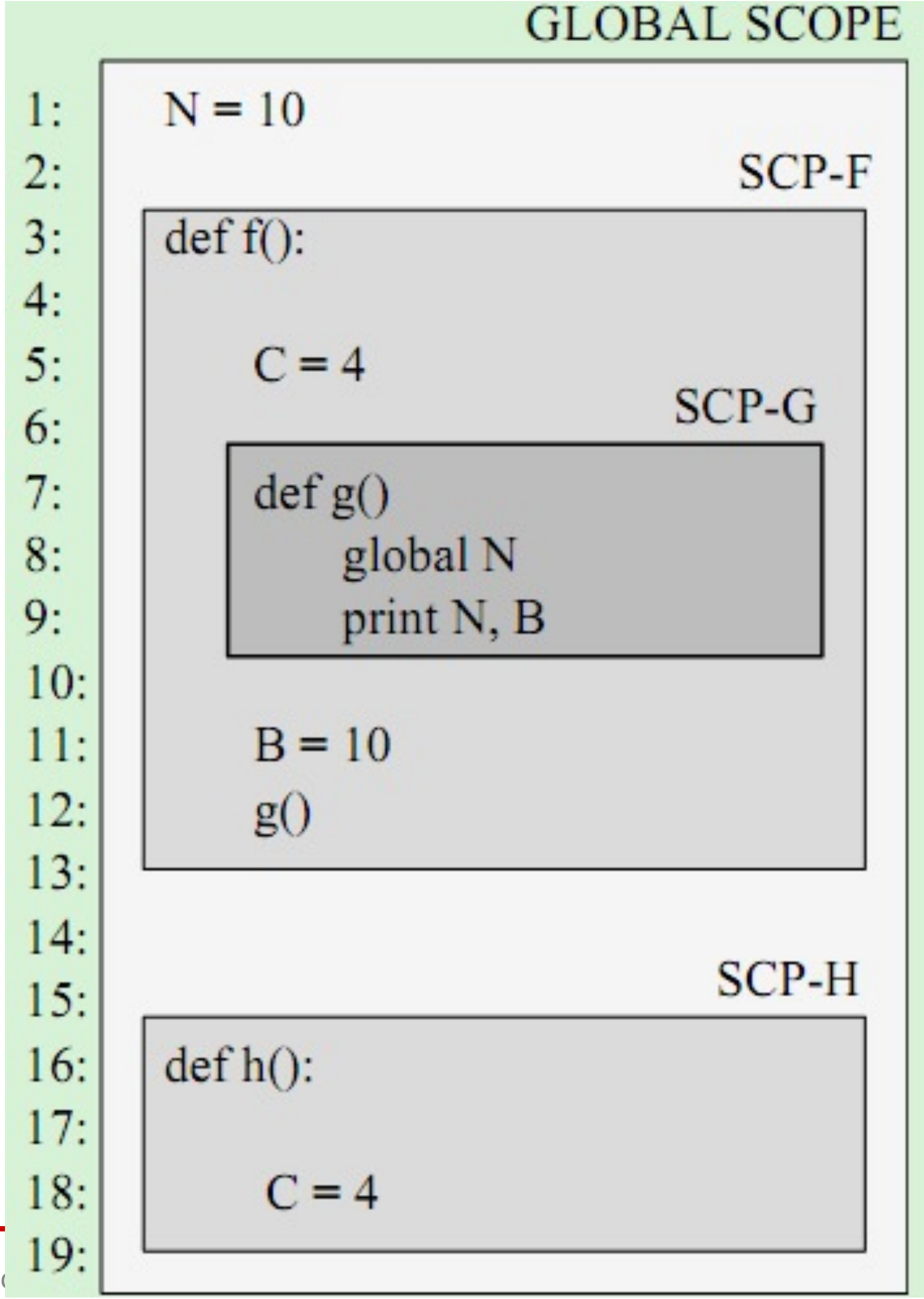
```
N = 10
def f():
        global N
        def g(Number):
                C = 20
                return N * Number
        N = g(N)
```

METU Computer Engineering

Scope in Python

Previously on CENG111!

- Since you can nest functions in Python, understanding scope is important



METU Computer Engineering

METU Computer Engineering

Previously on CENG111!

- Updating variables of an outer function

```python
# Method using the function like an object.
def f():
    f.a = 10
    def m():
        f.a  = 20
    m()
    print("a (M1): ", f.a)

# Method using the nonlocal keyword (only with v3).
def g():
    a = 10
    def m():
        nonlocal a
        a  = 20
    m()
    print("a (M2): ", a)

# Method using a mutable datatype
def h():
    a = [1]
    def m():
        a[0]  = 20
    m()
    print("a (M3): ", a[0])

# Call the functions
f()
g()
h()
```

Previously on CENG111!

Default Parameters in Python

METU

```
def reverse_num(Number=123):
  """reverse_num: Reverse
                  the digits in a number"""
  str_num = str(Number)
  print "Reverse of", Number, "is", str_num[::-1]
```

- We can now call this function with reverse_num() in which case Number is assumed to be 123.
- If we supply a value for Number, that value is used instead.

```
def f(Str, Number=123, Bst="Some"):
  print Str, Number, Bst
```

Previously on CENG111!

While we are at it…
let's have a look at commenting in Python

```
def reverse_num(Number=123):
    """reverse_num: Reverse
                the digits in a number"""
    str_num = str(Number)
    print "Reverse of", Number, "is", str_num[::-1]
```

There are two different ways to put comments in Python: (1) You can use # in which case the rest of the line is not interpreted. (2) You can enclose multiple lines like """ <lines of text> """. The comments that are written using the second option are basically documentation strings and available through the **help** page.

Functional Programming in Python

Previously on CENG111!

METU Computer Engineering

- List Comprehension

```
[<expr> for <var> in <list>]
```

Example:

```
[3*x for x in [1, 2, 3, 4, 5]]
```

```
[3, 6, 9, 12, 15]
```

Functional Programming in Python

Previously on CENG111!

METU Computer Engineering

Lambda Expression

- lambda *arguments* : *expression*
- Examples:

```
x = lambda a : a + 10
print x(5)

x = lambda a, b : a * b
print x(5, 6)
```

Functional Programming in Python

Previously on CENG111!

- filter(function, list)

```
def Positive(N):
        if N > 0: return True
        return False
```

filter(Positive, [-10, 20, -2, 5, 6, 8, -3]) → [20, 5, 6, 8]

- map(function, list)

```
def Mod4(N):
        return N % 4
```

map(Mod4, range(1, 10)) → [1, 2, 3, 0, 1, 2, 3, 0, 1]

METU Computer Engineering

Previously on CENG111!

Functional Programming in Python

- reduce(function, list)

```
def greater(A, B):
        return A if A > B else B
```

```
reduce(greater, [1, 20, 2, -30])
```

→

```
20
```

Today

- Recursion

Administrative Notes

- THE2 announced:
  - Due date: 26 December, 23:59
- Midterm:
  - 22 December, Wednesday, 18:00

METU Computer Engineering

DEALING WITH BULKY PROBLEMS

What happens when the problem gets bigger?

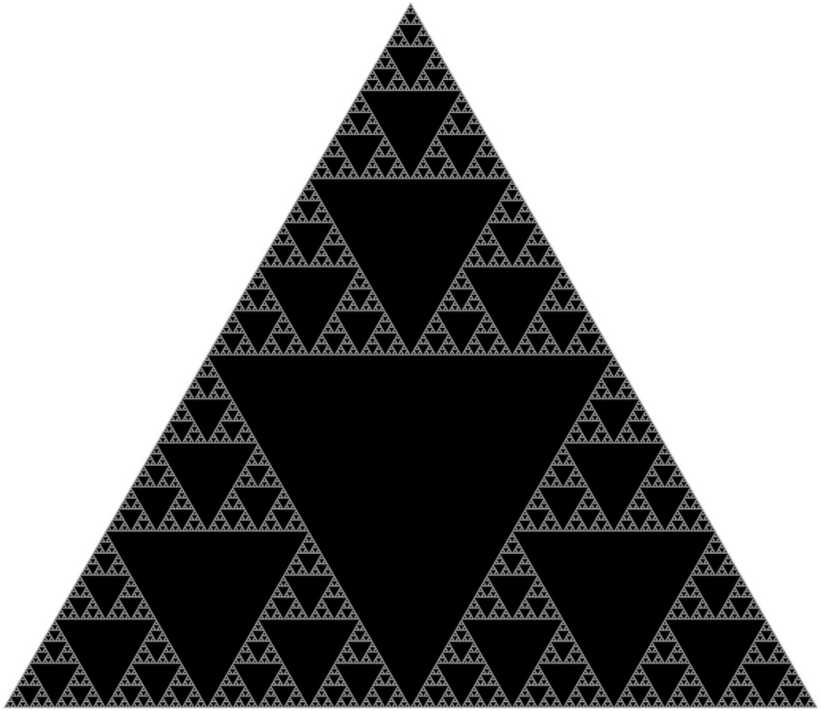
- What do we mean by “bigger”?
- “bigger” means more in size.
- For example, it is very easy to compute the letter grade of a student given his midterm, homework and final grades.
  - How about computing the letter grades of a few hundred students?
- We call these “bigger” cases, “bulky problems”.

What do we do for “bulky” problems?

1. Recursion
   - A powerful tool of the functional paradigm
2. Iteration
   - A powerful tool of the imperative paradigm

Recursion: An action wizard

- What is recursion?



Recursion: An example

- Definition of factorial:

Except for zero, the factorial of a natural number is defined to be the number obtained by multiplication of all the natural numbers lesser or equal to it. The factorial of zero is defined, and is set to be 1.

- More formally:

$$\begin{aligned} N! &= 1 \times 2 \times \cdots \times (N-1) \times N \qquad N \in \mathbf{N},\ \ N > 0 \\ 0! &= 1 \end{aligned}$$

Recursion: an example (cont'd)

$$\begin{aligned} N! &= 1 \times 2 \times \cdots \times (N-1) \times N \qquad N \in \mathbf{N},\;\; N > 0 \\ 0! &= 1 \end{aligned}$$

- A careful look at the formal definition:

$$N! = \underbrace{1 \times 2 \times \cdots \times (N-1)}_{(N-1)!} \times N \qquad N \in \mathbf{N},\;\; N > 0$$

$$\begin{aligned} N! &= (N-1)! \times N \qquad N \in \mathbf{N},\;\; N > 0 \\ 0! &= 1 \end{aligned}$$

Factorial uses its own definition!

Recurrence & Recursion

$$
\begin{aligned}
N! &= (N-1)! \times N \qquad N \in \mathbf{N},\ N > 0 \\
0! &= 1
\end{aligned}
$$

- This is called recurrence relation/rule.
- Algorithms which make use of recurrence relations are called recursive.

Recursion: an example (cont'd)

- Let us look at the pseudo-code:

$$
\begin{aligned}
N! &= (N-1)! \times N \qquad N \in \mathbf{N},\ \ N > 0 \\
0! &= 1
\end{aligned}
$$

$\Downarrow$

define $factorial(n)$
  if $n \stackrel{?}{=} 0$ **then**
    return 1
  else
    return $n \times factorial(n-1)$

Recursion: another example

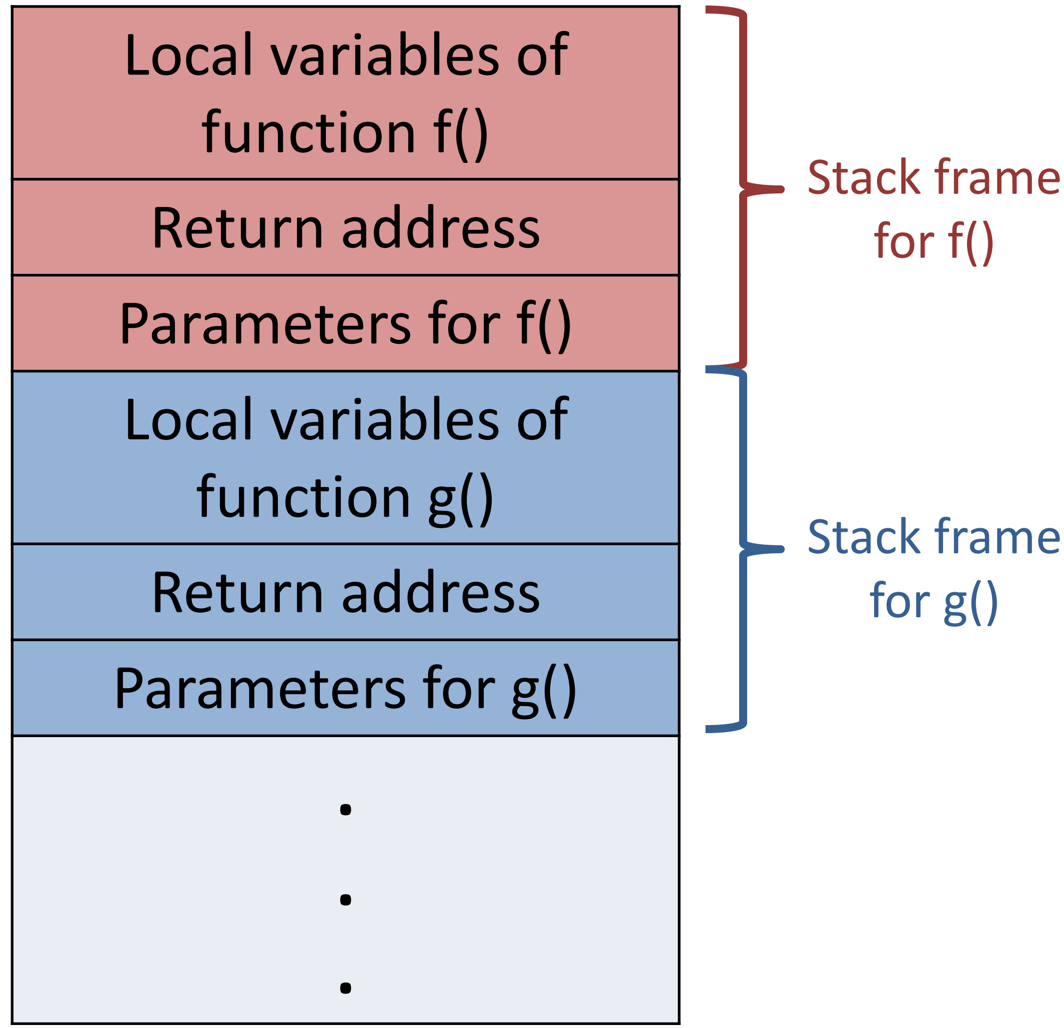
- Recursive definition of even or odd (a mutually recursive example):

  - 0 is even.
  - 1 is odd.
  - A number is even if one less the number is odd.
  - A number is odd if one less the number is even.

What happens when we call a function?

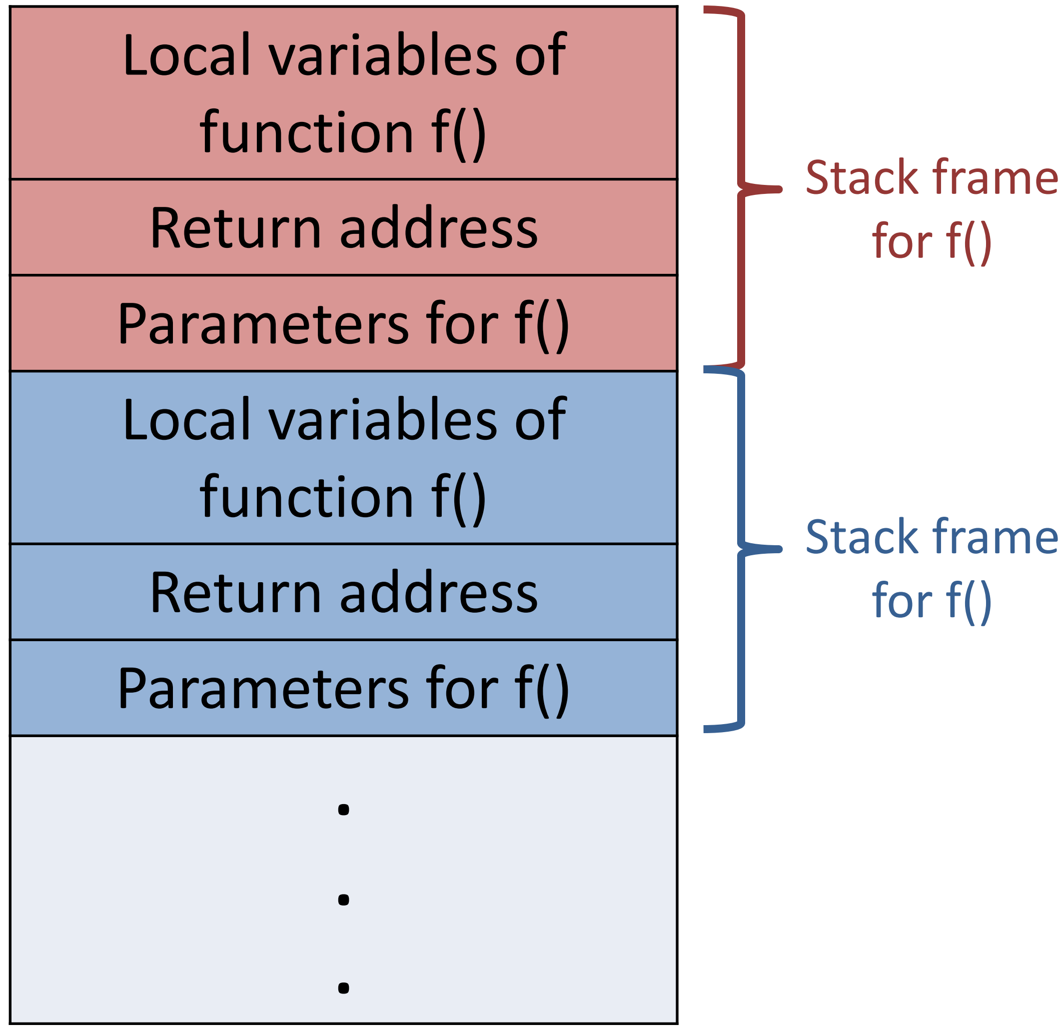
```python
def f(a):
    b = 10
    return b+a

def g(c):
    d = 3
    return c + d + f(c)
```



What happens with recursive functions?

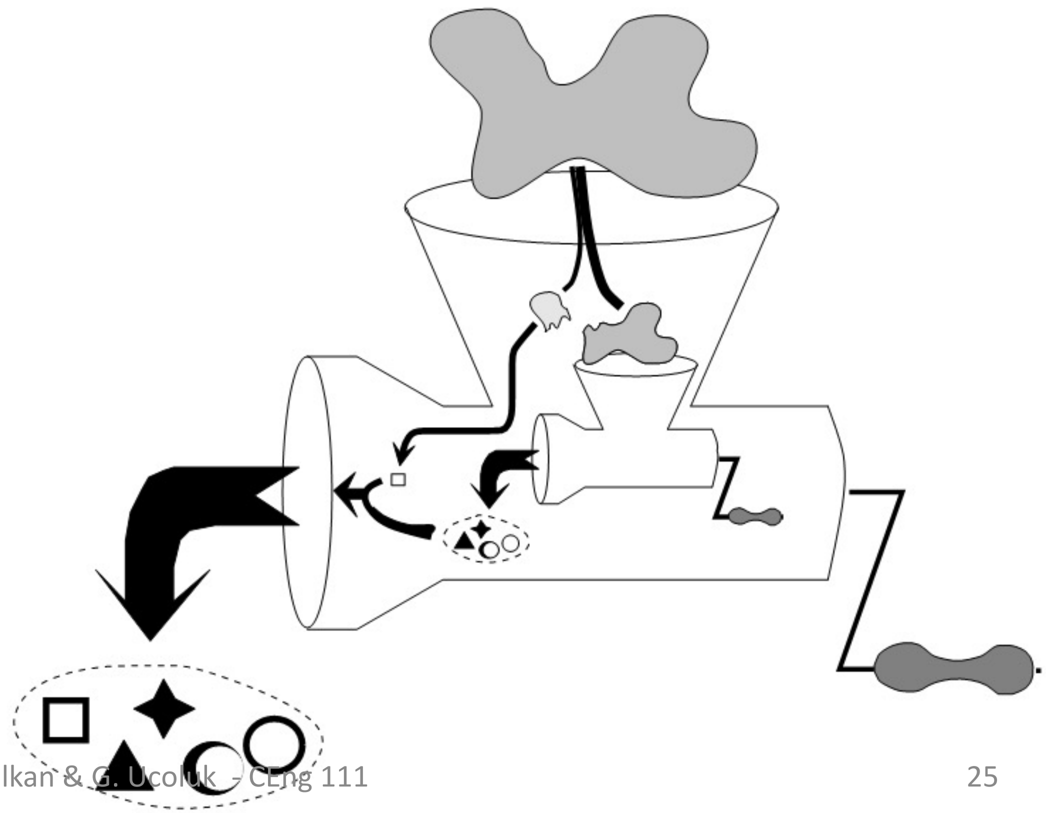
```python
def f(a):
    if a==0:
        return a
    return a+f(a-1)
```



In other words, each call to f() is treated like a different function call.

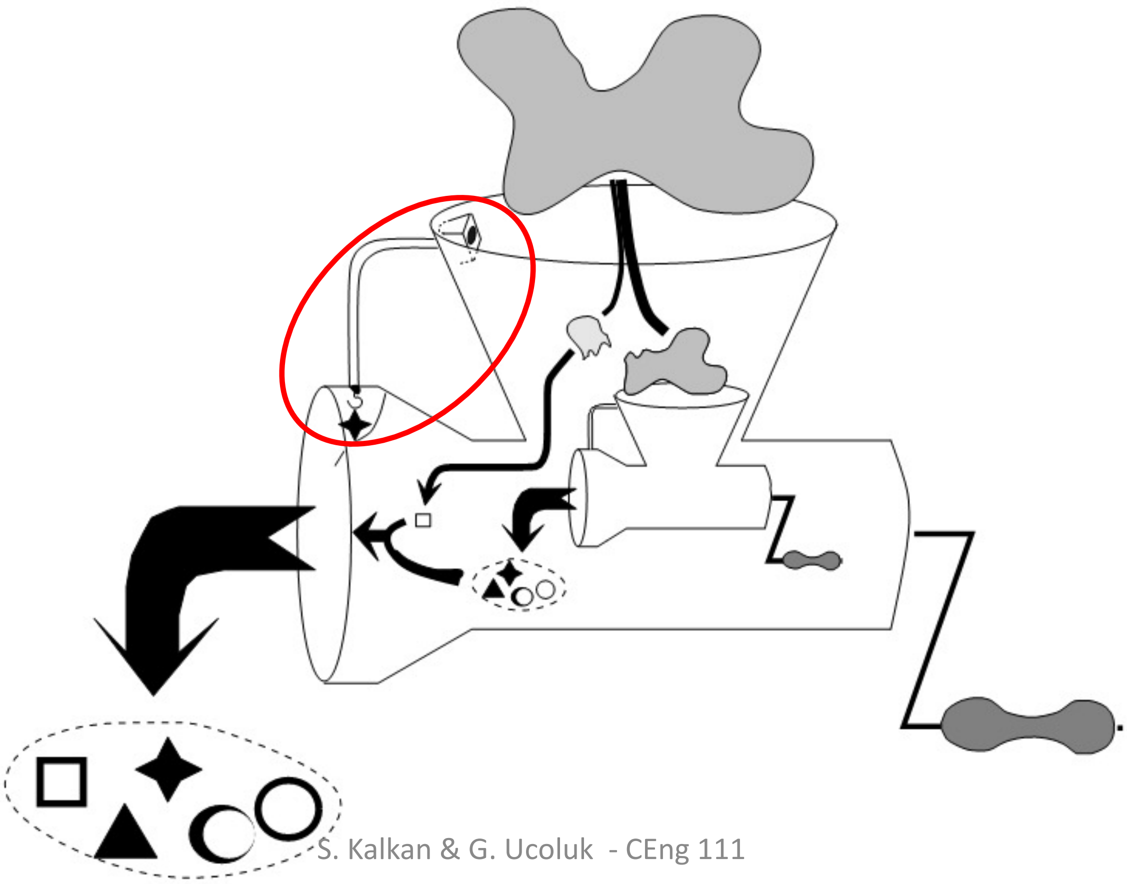
For what can we use recursion?

- Not all problems are **suited** to be solved recursively.
- The required properties for the problem:
  1. Scalability
  2. Downsize-ability
  3. Constructability

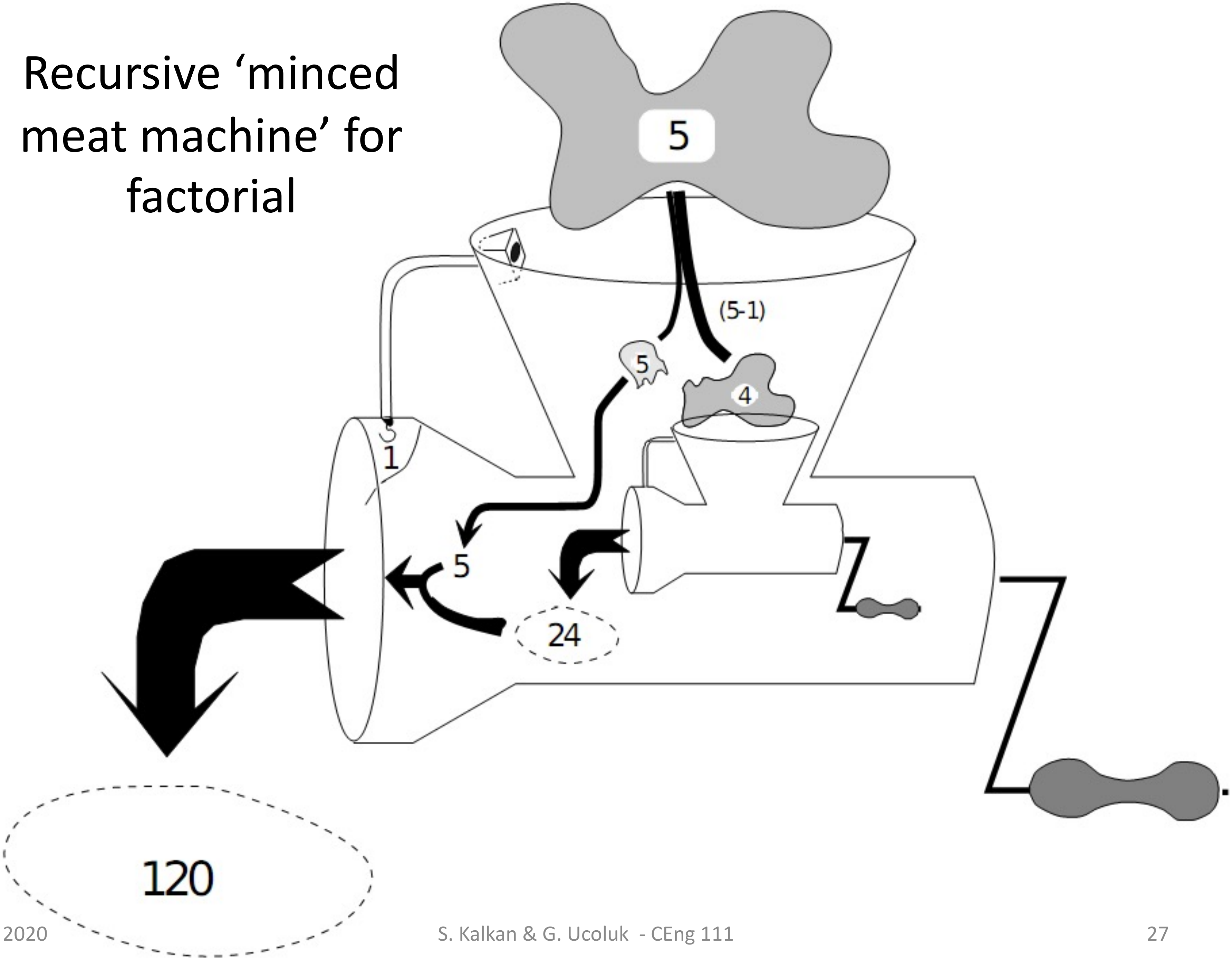


The recursive 'minced meat machine'

- We need a sensor to detect 'no meat'



Recursive ‘minced meat machine’ for factorial



Now let us lay down some rules while/for using recursion:

ter Engineering

Rule I

Looking at the problem's data content decide on a suitable data representation. This data representation shall allow easy shrinking and expansion with the change in problem's data.

Now let us lay down some rules while/for using recursion:

Rule II

Start with the *terminating condition*. Here you inspect the input data to the function to be minimal (as small as it is allowed). If it is the case, perform what the function is expected to do for this situation. [Depending on what the function is devised for, either you return an appropriate value or you take an appropriate action.]

Now let us lay down some rules while/for using recursion:

RULE III

Now it is turn to handle the non-minimal condition of the input data. You have to imagine that you partition this input data. The key idea of this partitioning is that *at least* one of the pieces shall be of the type of the partitioned data. **You have the right (without any cause) to assume that a further call of the same function with the piece(s) that remained to be of the same type will return/do the correct result/action.** It might be possible to do the partitioning in different ways. Bearing in mind the alternatives apply the next rule.

Now let us lay down some rules while/for using recursion:

Rule IV

Having in hand one hand the correct values/actions returned/-done for the pieces that conserved the input type and in the other hand the pieces themself which have not conserved the type (if any), you will seek to find the answer to the following question:

> *With those in my hand how can I construct the desired value/action for the original input data?*

If you cannot figure out a solution consider the other alternatives that surfaced with the application of Rule III (if any).

Applications of the rules

Application of Rule I

First question: What data-type shall we use?

The subject to the factorial operation is a *natural number* and the result is again a *natural number*. So the suitable data-type for the representation of the input data is the *integer* type. But, due to the nature of the problem we will restrict all our integers to be natural numbers only.

Second question: Is this data-type partitionable in the terms explained above?

Yes, indeed. There exist operations like *subtraction* and *division* defined over the *integers* which produce *smaller* pieces (in this case all of integer types as well).

Applications of the rules

Engineering

Application of Rule II

Having decided about the data-type (integer in this case), it is time to set the terminating condition. We know that the smallest integer the factorial function would allow as argument is 0. And the value to be returned for this input value is 1. So the first line of

define $factorial(n)$
  if $n \stackrel{?}{=} 0$ **then**
    return 1
  else
    $\ldots$

Applications of the rules

Application of Rule III

The question is: How can we partition the argument (in this case n), so that at least one of the pieces remains to be an integer?

1. Partitioning where the parts are of the same type (for a number n):
   a. $n = n_1 + n_2$ such that $n_1 = n_2$ or $n_1 = 1 + n_2$
   b. $n = n_1 + n_2 + \ldots n_k$
2. Partitioning where one part is small and the other part is big:
   a. $n_1 = n - 1$, $n_2 = 1$
   b. $n_1 = n-2$, $n_2 = 2$

Applications of the rules

- Application of the rule IV (i.e., constructing the result from the results of the partitions).
  - Consider using equal partitioning for factorial (i.e., $n = n_1 + n_2$ such that $n_1 = n_2$ or $n_1 = 1 + n_2$)

Given $n, n_1, n_2, n_1!, n_2!$ can we compute $n!$ in an easy manner?

No! So => Partitioning is important.
Partition in a way that you can construct later.

Applications of the rules

- Consider the partitioning $n_1 = n - 1$, $n_2 = 1$
- Then, we have the following for Rule IV (for construction):

  - n
  - 1
  - $n - 1$
  - $1!$
  - $(n-1)!$

$$\Rightarrow \quad n! = n \times (n-1)!$$

Coming back to the factorial example

$$
\begin{array}{l}
\textbf{define } factorial(n) \\
\quad \textbf{if } n \stackrel{?}{=} 0 \textbf{ then} \\
\qquad \textbf{return} \quad 1 \\
\quad \textbf{else} \\
\qquad \textbf{return} \quad n \times factorial(n-1)
\end{array}
$$

Some operations that we will use

```
def head(L): return L[0]

def tail(L): return L[1:]

def last(L): return L[-1]
```

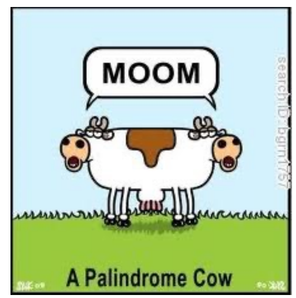
Another example for recursion

- Reversing a list / string
  - [a, b, c] → [c, b, a]
- Now, **reverse([a, b, c]) = reverse([b, c]) + [a]**.

define $reverse(S)$
  if $S \stackrel{?}{=} \emptyset$ then
    return $\emptyset$
  else
    return $reverse(tail(S)) \oplus [head(S)]$

Another example: Palindromes

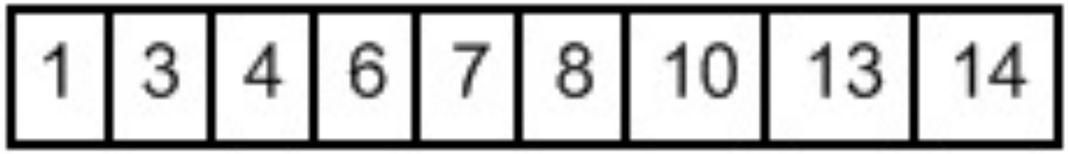
- Remember palindromes?
  - kek, mom, elele, …

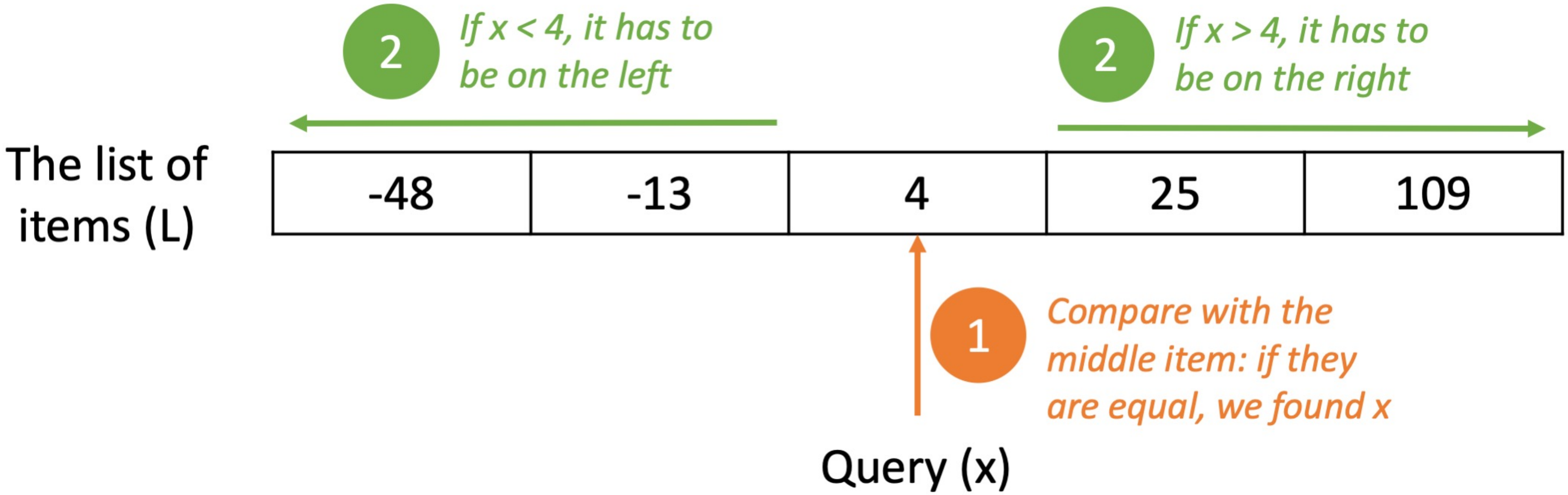



```python
def palindrome(S):
    if len(S) <= 1:
        return True
    else:
        return head(S) == last(S) and palindrome(S[1:-1])
```

Searching for a number in a list

- Binary search





Binary search

```python
def bin_search(x, L):
    if len(L) == 0: return False
    Mid_index = len(L) // 2
    if x == L[Mid_index]: return True
    if x < L[Mid_index]: return bin_search(x, L[0:Mid_index])
    if x > L[Mid_index]: return bin_search(x, L[Mid_index+1:])
```

Another example: insert an item into ordered list

```python
def insert(x, L):
    if len(L) == 0:
        return [x]
    if x < head(L):
        return [x] + L
    else:
        return [head(L)] + insert(x, tail(L))
```