Lecture 10, 14 September 2023

Inductive definitions

- Define $f(n)$ in terms of n and $f(m)$ for $m < n$
- Need to define a base case explicitly, typically $f(0)$ or $f(1)$

Factorial

- $0! = 1$
- $n! = n \times (n-1)!$

Fibonacci numbers

- $fib(0) = 0$
- $fib(1) = 1$
- $fib(n) = fib(n-1) + fib(n-2)$

Recursive function calls

- A function can call itself
- Current execution is suspended until recursive call returns a value, like any other function call
- Recursive call will again call itself, so must ensure progress towards a base case for termination

In [1]:
```
def factorial(n):
  if n == 0:
    return(1)
  else:
    return(n*factorial(n-1))  # Recursive call
```

In [2]:
```
def fib(n):
  if n == 0:
    return(0)
  elif n == 1:
    return(1)
  else:
    return(fib(n-1)+fib(n-2))
```

In [3]:
```
factorial(20), factorial(50)
```

Out[3]:
```
(2432902008176640000,
 30414093201713378043612608166064768844377641568960512000000000000)
```

In [4]:
```
fib(4)
```

Out[4]:
```
3
```

In [5]:
```
fib(20)
```

Out[5]:
```
6765
```

In [6]:
```
fib(50)
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In [6], line 1
----> 1 fib(50)

Cell In [2], line 7, in fib(n)
      5   return(1)
      6 else:
----> 7   return(fib(n-1)+fib(n-2))

Cell In [2], line 7, in fib(n)
      5   return(1)
      6 else:
----> 7   return(fib(n-1)+fib(n-2))

    [... skipping similar frames: fib at line 7 (33 times)]

Cell In [2], line 7, in fib(n)
      5   return(1)
      6 else:
----> 7   return(fib(n-1)+fib(n-2))

Cell In [2], line 1, in fib(n)
----> 1 def fib(n):
      2   if n == 0:
      3     return(0)

KeyboardInterrupt:
```

Can also do induction on "structures"

- A list consists of the first element and the rest
- Base case is usually the empty list `[]`
- May occasionally also have a base case for a singleton list

In [7]:
```
def mylength(l):
  if l == []:
    return(0)
  else:
    return(1 + mylength(l[1:]))
```

In [8]:
```
mylength(list(range(900)))
```

Out[8]:
```
900
```

In [9]:
```
def mysum(l):
    if l == []:
        return(0)
    else:
        return(l[0] + mysum(l[1:]))
```

In [10]:
```
mysum(list(range(10)))
```

Out[10]:
```
45
```

In [11]:
```
mysum(['the','long','road'])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [11], line 1
----> 1 mysum(['the','long','road'])

Cell In [9], line 5, in mysum(l)
      3     return(0)
      4 else:
----> 5     return(l[0] + mysum(l[1:]))

Cell In [9], line 5, in mysum(l)
      3     return(0)
      4 else:
----> 5     return(l[0] + mysum(l[1:]))

Cell In [9], line 5, in mysum(l)
      3     return(0)
      4 else:
----> 5     return(l[0] + mysum(l[1:]))

TypeError: can only concatenate str (not "int") to str
```

- Problem is `'the'+'long'+'road'+0`
- Could try to fix this by querying types from within the code, but we won't bother

Querying types

In [12]:
```
x = [3,3,4]
y = 5
type(x) == type([])  # Compare type() with a "known" type
```

Out[12]:
```
True
```

In [13]:
```
type(x)
```

Out[13]:
```
list
```

Can also compare `type(v)` with name of type without quotes

In [14]:
```
type(x) == list, type(y) == int
```

Out[14]:
```
(True, True)
```

Ascending and descending

- Check if a list is in ascending order, `l[0] < l[1] < ...`
- Similarly, descending, `l[0] > l[1] > ...`
- Ascending
  - Base case, `len(l)` is 0 or 1, nothing to check
  - Otherwise, check first pair `l[0] < l[1]`
  - Inductively check that that remaining list `l[1:]` is also ascending

In [15]:
```
def ascending(l):
  if len(l) <= 1:
    return(True)
  else:
    return(l[0] < l[1] and ascending(l[1:]))
    # l[0] < l[1] and l[1] < l[2] and l[2] < l[3] and ...

def descending(l):
  if len(l) <= 1:
    return(True)
  else:
    return(l[0] > l[1] and descending(l[1:]))
```

Zigzag

- Alternate between ascending and descending
- Two possibilities
  - up-down-up-down..., `[1,3,2,7,1,5]`
  - down-up-down-up..., `[8,2,18,-5,7,2,8]`

Up-down

- If `len(l)` is 0 or 1, nothing to do
- Up-down unit repeats after two elements
- 2 element list, check up
- 3 element list, check up-down and recursively check that `l[2:]` is also up-down

Down-up is symmetric

Combine to get zigzag

In [16]:
```
def updown(l):
  if len(l) <= 1:
    return(True)
  elif len(l) == 2:
    return(l[0] < l[1])
  else:
    return(l[0] < l[1] and l[1] > l[2] and updown(l[2:]))

def downup(l):
  if len(l) <= 1:
    return(True)
  elif len(l) == 2:
    return(l[0] > l[1])
  else:
    return(l[0] > l[1] and l[1] < l[2] and downup(l[2:]))

def zigzag(l):
  return(updown(l) or downup(l))
```

In [17]:
```
l1 = [1,2,1,3,1,4,1]
updown(l1), downup(l1), zigzag(l1)
```

Out[17]:
```
(True, False, True)
```

In [18]:
```
l2 = [2,1,3,1,4,1]
updown(l2), downup(l2), zigzag(l2)
```

Out[18]:
```
(False, True, True)
```

Mutual recursion

- Can define `updown` and `downup` in terms of each other
- **Mutual recursion**

In [19]:
```
def zigzag(l):
    return(updown(l) or downup(l))

def updown(l):
    if len(l) < 2:
        return(True)
    else:
        return(l[0] < l[1] and downup(l[1:]))


def downup(l):
    if len(l) < 2:
        return(True)
    else:
        return(l[0] > l[1] and updown(l[1:]))
```

In [20]:
```
zigzag([0,1,0,1,0])
```

Out[20]:
```
True
```