Lecture 10, 07 September 2022

Mutual recursion

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

In [1]:
```
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 [2]:
```
zigzag([0,1,0,1,0])
```

Out[2]:
```
True
```

- Function must be defined before it can be called
- Reading a function definition is different from executing it
- Similar to an undefined value -- the error is flagged only when the function is executed

In [3]:
```
def fnwitherror():
  return(thisisanewname)
```

- The function definition above does not generate an error, though `thisisanewname` is undefined
- The function call below generates the error

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

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_353557/3639571702.py in <module>
----> 1 fnwitherror()

/tmp/ipykernel_353557/618273130.py in fnwitherror()
      1 def fnwitherror():
----> 2   return(thisisanewname)

NameError: name 'thisisanewname' is not defined
```

- Similarly, if we try to execute `updown` before we define `downup` we get an `NameError`

In [5]:
```
# Remove the earlier definitions and redefine
del(zigzag)
del(updown)
del(downup)

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:]))

updown([1,3,1])

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

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_353557/3340148618.py in <module>
     13         return(l[0] < l[1] and downup(l[1:]))
     14
---> 15 updown([1,3,1])
     16
     17 def downup(l):

/tmp/ipykernel_353557/3340148618.py in updown(l)
     11         return(True)
     12     else:
---> 13         return(l[0] < l[1] and downup(l[1:]))
     14
     15 updown([1,3,1])

NameError: name 'downup' is not defined
```

In [9]:
```
zigzag([1,0,1,0,1])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_353557/1999970637.py in <module>
----> 1 zigzag([1,0,1,0,1])

/tmp/ipykernel_353557/3340148618.py in zigzag(l)
      5
      6 def zigzag(l):
----> 7     return(updown(l) or downup(l))
      8
      9 def updown(l):

NameError: name 'downup' is not defined
```

More recursive functions on lists

`find(l,v)`

Check `v` is a member of `l` -- like built-in `v in l`

- Base case, if `l == []` then `v` is not found
- If `l[0] == v` then `v` is found
- Otherwise, inductively search for `v` in `l[1:]`

In [10]:
```
def find(l,v):

  if l == []:
    return(False)
  if l[0] == v:
    return(True)
  else:
    return(find(l[1:],v))
```

Short cut evaluation of boolean expressions

- If we write A or B. we evaluate A and B and then check if at least one is true

- In what order are A and B evaluated?
- Python (and other languages) **always** evaluate left to right
- And stop when then answer is known
  - `True or x` is `True` whatever the value of `x`, so no need to evaluate `x`
  - `False and y` is `False` whatever the value of `y`, so no need to evaluate `y`
- Here is a version of `find` in which the two cases of the inductive step are combined using `or`

In [11]:
```
def find2(l,v):
  if l == []:
    return(False)
  else:
    return((l[0] == v) or find2(l[1:],v))
    # Unwinds as l[0] == v or l[1] == v or l[2] == v or ... or l[len(l)-1] == v
```

In [12]:
```
l1 = list(range(0,100,3))
```

In [13]:
```
l2 = [j for j in range(0,100,5) if find(l1,j)]
print(l2)
```

```
[0, 15, 30, 45, 60, 75, 90]
```

In [14]:
```
l2 = [j for j in range(0,100,5) if find2(l1,j)]
print(l2)
```

```
[0, 15, 30, 45, 60, 75, 90]
```

`insert(l,v)`

Insert `v` in `l`, assume `l` is sorted in ascending order

- If `l == []` return singleton list `[v]`
- If `v < l[0]` return `[v] + l`
- If `l[0] <= v`, inductively insert `v` in `l[1:]` and stick `l[0]` before this list

In [15]:
```
def insert(l,v): # Assume l sorted in ascending order
  if l == []:
    return([v])
  if v < l[0]:
    return([v] + l)
  else:
    return(l[:1] + insert(l[1:],v))
    # same as
    # return([l[0]] + insert(l[1:],v))
```

In [16]:
```
l3 = insert(l1,1000)
print(l3)
```

```
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78,
81, 84, 87, 90, 93, 96, 99, 1000]
```

`delete(l,v)`

Delete first occurrence `v` from `l`, if `v` exists

- Similar structure to `insert(l,v)`
- If `l == []` nothing to be done
- If `l[0] == v`, return `l[1:]`
- Otherwise, inductively delete `v` from `l[1:]` and stick `l[0]` before this list

In [17]:
```
def delete(l,v):
  if l == []:
    return(l)
  if l[0] == v:
    return(l[1:])
  else:
    return(l[:1] + delete(l[1:],v))
```

In [18]:
```
l3 = delete(delete(insert(l1,15),13),15)
print(l3)
```

```
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78,
81, 84, 87, 90, 93, 96, 99]
```

`findpos(l,v)`

- Like `find(l,v)` but report position of first `v` in `l`
- If `v` is not found in `l` return `-1`
- If `l == []`, return `-1`
- If `l[0] == v`, return `0`
- Otherwise, inductively find the first position of `v` in `l[1:]` and add `1` to account for `l[0]`
- Unless `v` is not found in `l[1:]` in which case the recursive call returns `-1` and this should be passed on untouched

In [19]:
```
def findpos(l,v):  # Returns -1 if v not in l
  if l == []:
    return(-1)
  if l[0] == v:
    return(0)
  else:
    z = findpos(l[1:],v)
    if z >= 0:
      return(1+z)
    else:
      return(z)
```

Alternative `findpos(l,v)`

- Return `len(l) + 1` if `v` is not found in `l`
- The recursive case becomes simpler: just add `1` to the recursive call, which works whether or not `v` is found in `l[1:]`

In [20]:
```
def findpos2(l,v):  # Returns len(l)+1 if v not in l
  if l == []:
    return(1)
  if l[0] == v:
    return(0)
  else:
    z = findpos2(l[1:],v)
    return(1+z)
```

In [21]:
```
findpos(l1,17),findpos2(l1,17),len(l1)
```

Out[21]:
```
(-1, 35, 34)
```

Defining our own data structures

- In Lecture 8, we implemented a "linked" list using dictionaries
- The fundamental functions like `listappend`, `listinsert`, `listdelete` modify the underlying list
- Instead of `mylist = {}`, we wrote `mylist = createlist()`
- To check empty list, use a function `isempty()` rather than `mylist == {}`
- Can we clearly separate the **interface** from the **implementation**
- Define the data structure in a more "modular" way

Object oriented approach

- Describe a datatype using a template, called a **class**
- Create independent instances of a class, each is an **object**
- Each object has its own internal *state* -- the values of its local variables
- All objects in a class share the same functions to query/update their state
- `l.append(x)` vs `append(l,x)`
  - Tell an object what to do vs passing an object to a function
- Each object has a way to refer to itself