Lecture #8: More on Functions
Another Recursion Problem: Counting Partitions

• I’d like to know the number of distinct ways of expressing an integer as a sum of positive integer “parts.”

• To make things more interesting, let’s also limit the size of the integer parts to some given value:

   def num_partitions(n, k):
       """Number of distinct ways to express N as a sum of positive integers each of which is <= K, where K > 0. (The empty sum is 0.)""

• Example:

   \[
   \begin{align*}
   x06 & = 3 + 3 \\
       & = 3 + 2 + 1 \\
       & = 3 + 1 + 1 + 1 \\
       & = 2 + 2 + 2 \\
       & = 2 + 2 + 1 + 1 \\
       & = 2 + 1 + 1 + 1 + 1 \\
       & = 1 + 1 + 1 + 1 + 1 + 1
   \end{align*}
   \]

   so num_partitions(6, 3) is 7.
Identifying the Problem in the Problem

• Again, consider `num_partitions(6, 3)`.

• Some partitions will contain the maximum size integer, 3, and the rest won’t.

• Those that do contain 3 then have various ways to partition the remaining 3.
  
  \[
  \begin{align*}
  3 & + 3 \\
  3 & + 2 + 1 \\
  3 & + 1 + 1 + 1
  \end{align*}
  \]

• While those that do not contain 3 partition 6 using integers no larger than 2:
  
  \[
  \begin{align*}
  2 & + 2 + 2 \\
  2 & + 2 + 1 + 1 \\
  2 & + 1 + 1 + 1 + 1 \\
  1 & + 1 + 1 + 1 + 1 + 1
  \end{align*}
  \]

• These observation generalize, and lead immediately to a solution.
def num_partitions(n, k):
    """Number of distinct ways to express N as a sum of positive integers each of which is <= K, where K > 0. (The empty sum is 0.)"""

    if .................................:
        return 0

    elif .................................:
        return 1

    else:
        return .................................:
def num_partitions(n, k):
    """Number of distinct ways to express N>=0 as a sum of positive
    integers each of which is <= K, where K > 0. (The empty sum is 0.)""

    if n < 0:
        return 0

    elif ________________:
        return 1

    else:
        return ________________:
def num_partitions(n, k):
    """Number of distinct ways to express N>=0 as a sum of positive integers each of which is <= K, where K > 0. (The empty sum is 0.)""

    if n < 0:
        return 0

    elif k == 1 or n <= 1:
        return 1

    else:
        return _______________:
def num_partitions(n, k):
    """Number of distinct ways to express N>=0 as a sum of positive
    integers each of which is <= K, where K > 0. (The empty sum is 0.)"""

    if n < 0:
        return 0

    elif k == 1 or n <= 1:
        return 1

    else:
        return num_partitions(n - k, k) + num_partitions(n, k - 1)
Functions and Data

- We tend to think of functions as simply doing or computing something with data.
- In fact, they can also represent or contain data themselves.
- Trivial example:

  ```python
  >>> def const(n):
      ...
      return lambda: n
  >>> x, y = const(5), const(11)
  >>> print(x(), y())
  5 11
  ```

- The functions returned by `const` contain pointers to the local frames created when `const` was called, which in turn contain copies of the argument values (5 and 11).
**Functions and Data (II)**

- **We can get a bit fancier:**

  ```python
  >>> def cons(left, right):
  ...       return lambda which: left if which else right
  >>> P = cons("value", 42)
  >>> print(P(True), P(False))
  value 42
  >>> L = cons(1, cons(2, cons(3, None)))
  >>> print(L(True), L(False)(True), L(False)(False)(True), L(False)(False)(False))
  1 2 3 None
  ```

  *(See the chain example at the end of Lecture #4.)*

- **So, in effect, values returned by `cons` are lists of values.**
The Pair Abstraction

• However, writing \( P(\text{True}) \) for “the left part of \( P \)” is not the clearest code one could imagine.

• Better to express the programmer’s intent:
  
  ```python
  >>> def cons(left, right):
  ...     return lambda which: left if which else right
  >>> def left(pair): return pair(True)
  >>> def right(pair): return pair(False)
  >>> P = cons("value", 42)
  >>> print(left(P), right(P))
  value 42
  ```

• Together, these three functions define a **data type**.

• The data (pairs) are **represented** by functions returned by \( \text{cons} \).

• \( \text{left} \) and \( \text{right} \) are the basic operations on the data type.

• If we use these \( \text{cons}, \text{left}, \text{and right} \) and three functions and ignore the fact that \( \text{cons} \) really produces a function rather than a pair, we are obeying the **abstraction barrier**.
Data Abstraction Philosophy

• In the old days, one described programs as hierarchies of actions: *procedural decomposition*.

• Starting in the 1970’s, emphasis moved to the data that the functions operate on.

• An *abstract data type (ADT)* (like the pair abstraction) represents some kind of thing and the operations upon it.

• Instances of the type are often generically called *objects*.

• We can usefully organize our programs around the ADTs in them.

• For each type, we define an *interface* that describes for users (“clients”) of that type of data what operations are available.

• Typically, the interface consists of functions.

• The collection of specifications (syntactic and semantic—see lecture #6) constitute a *specification of the type*.

• We call ADTs *abstract* because clients ideally need not know internals.
Rational Numbers

- The book uses “rational number” as an example of an ADT:

```python
def make_rat(n, d):
    """The rational number n/d, assuming n, d are integers, d!=0""

def add_rat(x, y):
    """The sum of rational numbers x and y.""

def mul_rat(x, y):
    """The product of rational numbers x and y.""

def numer(r):
    """The numerator of rational number r.""

def denom(r):
    """The denominator of rational number r.""
```

- These definitions pretend that $x$, $y$, and $r$ really are rational numbers.
- But from this point of view, the definitions of `numer` and `denom` are problematic. Why?
A Better Specification

- Problem is that “the numerator (denominator) of $r$" is not well-defined for a rational number.

- If `make_rat` really produced rational numbers, then `make_rat(2, 4)` and `make_rat(1, 2)` ought to be identical. So should `make_rat(1, -1)` and `make_rat(-1, 1)`.

- So a better specification would be

  ```python
  def numer(r):
      """The numerator of rational number r in lowest terms."""

  def denom(r):
      """The denominator of rational number r in lowest terms. Always positive."""
  ```
Rationals as Pairs (I)

- Our pair abstraction (represented by functions) can in turn represent rational numbers.

```python
from math import gcd  # Need Python3.5 actually.

def make_rat(n, d):
    """The rational number n/d, assuming n, d are integers, d!=0""
    g = gcd(n, d) if d > 0 else -gcd(n, d)
    n //= g; d //= g
    return cons(n, d)

def numer(r):
    """The numerator of rational number r.""
    return left(r)

def denom(r):
    """The denominator of rational number r.""
    return right(r)

def add_rat(x, y):
    """The sum of rational numbers x and y.""
    return ?

def mul_rat(x, y):
    """The product of rational numbers x and y.""
    return ?
```
Representation as Functions (II)

• **One possibility for** `add_rat`:

```python
from math import gcd

def make_rat(n, d):
    """The rational number n/d, assuming n, d are integers, d!0"""
    g = gcd(n, d) if d > 0 else -gcd(n, d)
    n //= g; d //= g
    return lambda flag: n if flag == 0 else d
...

def add_rat(x, y):
    n0, n1, d0, d1 = x(0), y(0), x(1), y(1)
    n, d = n0 * d1 + n1 * d0, d0 * d1
    g = gcd(n, d) if d > 0 else -gcd(n, d)
    n //= g; d //= g
    return lambda flag: n if flag == 0 else d
```

• Comments?
Abstraction Violations and DRY

• Having created an abstraction \((\text{make\_rat, numer, denom})\), use it:
  - Then, later changes of representation will affect less code.
  - Code will be clearer, since well-chosen names in the API make intent clear.

\[\]

```python
def add_rat(x, y):
    return make_rat(numer(x) * denom(y) + numer(y) * denom(x),
                    denom(x) * denom(y))
```

```python
def mul_rat(x, y):
    """The product of rational numbers x and y.""
    return make_rat(numer(x) * numer(y), denom(x) * denom(y))
```

...
Changing Representations

- It’s cute that functions can represent pairs (or anything else, for that matter), but it’s not a particularly efficient use of them.

- Suppose that we instead decide to use Python’s tuples. What changes?

  ```python
def cons(left, right):
    return (left, right)
def left(pair):
    return pair[0]
def right(pair):
    return pair[1]
  ```

- Crucial Observation: Nothing else changes!