Space Consumption

Which environment frames do we need to keep during evaluation? Each step of evaluation has a set of active environments. Values and frames referenced by active environments are kept. Memory used for other values and frames can be reclaimed.

Active environments:
- The environment for the current expression being evaluated
- Environments for calls that depend upon the value of the current expression
- Environments associated with functions referenced by active environments
Active Environments for Returned Functions

```python
make_adder:
def make_adder(n):
    def adder(k):
        return k + n
    return adder

add1 = make_adder(1)
```

Associated with an environment

Therefore, all frames in this environment must be kept.

Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases.

- \( n \): size of the problem
- \( R(n) \): Measurement of some resource used (time or space)

\[ R(n) = \Theta(f(n)) \]

means that there are constants \( k_1 \) and \( k_2 \) such that

\[ k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n) \]

for sufficiently large values of \( n \).

Comparing orders of growth

- \( \Theta(b^n) \): Exponential growth! Recursive fib takes \( \Theta(b^n) \) steps, where \( b = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \)
- \( \Theta(\varphi^n) \): Incrementing the problem scales \( R(n) \) by a factor.
- \( \Theta(n) \): Linear growth. Resources scale with the problem.
- \( \Theta(\log n) \): Logarithmic growth. These functions scale well.
- \( \Theta(1) \): Constant. The problem size doesn’t matter.

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

```python
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n-1):
        prev, curr = curr, prev + curr
    return curr

@memo
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)
```

Exponentiation

**Goal**: one more multiplication lets us double the problem size.

```python
def exp(b, n):
    if n == 0:
        return 1
    if n % 2 == 0:
        return b * exp(b, n//2)
    else:
        return square(fast_exp(b, n//2))
```

Exponentiation

**Goal**: one more multiplication lets us double the problem size.

```python
def exp(b, n):
    if n == 0:
        return 1
    if n % 2 == 0:
        return square(exp(b, n//2))
    else:
        return b * exp(b, n-1)
```