Homework 4 Solutions

hw04.zip (hw04.zip)

Solution Files

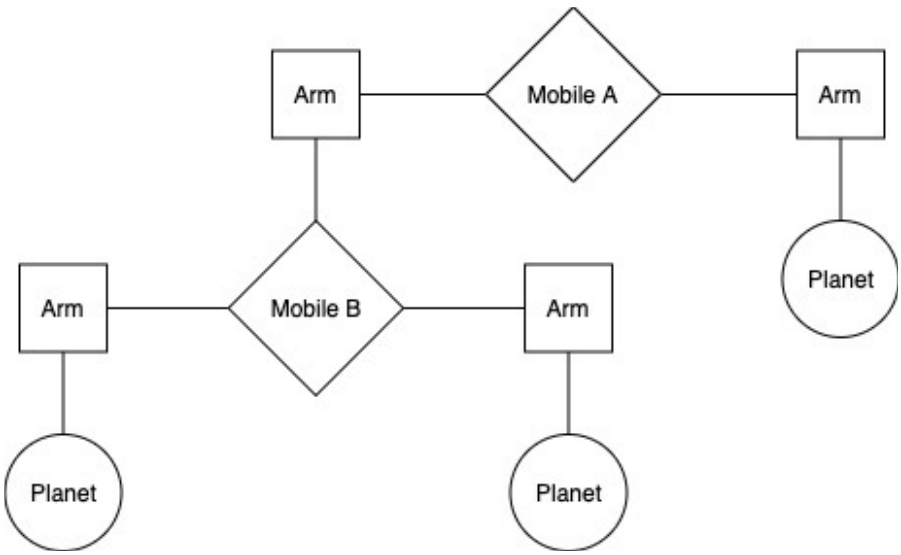
You can find the solutions in hw04.py (hw04.py).

Required questions

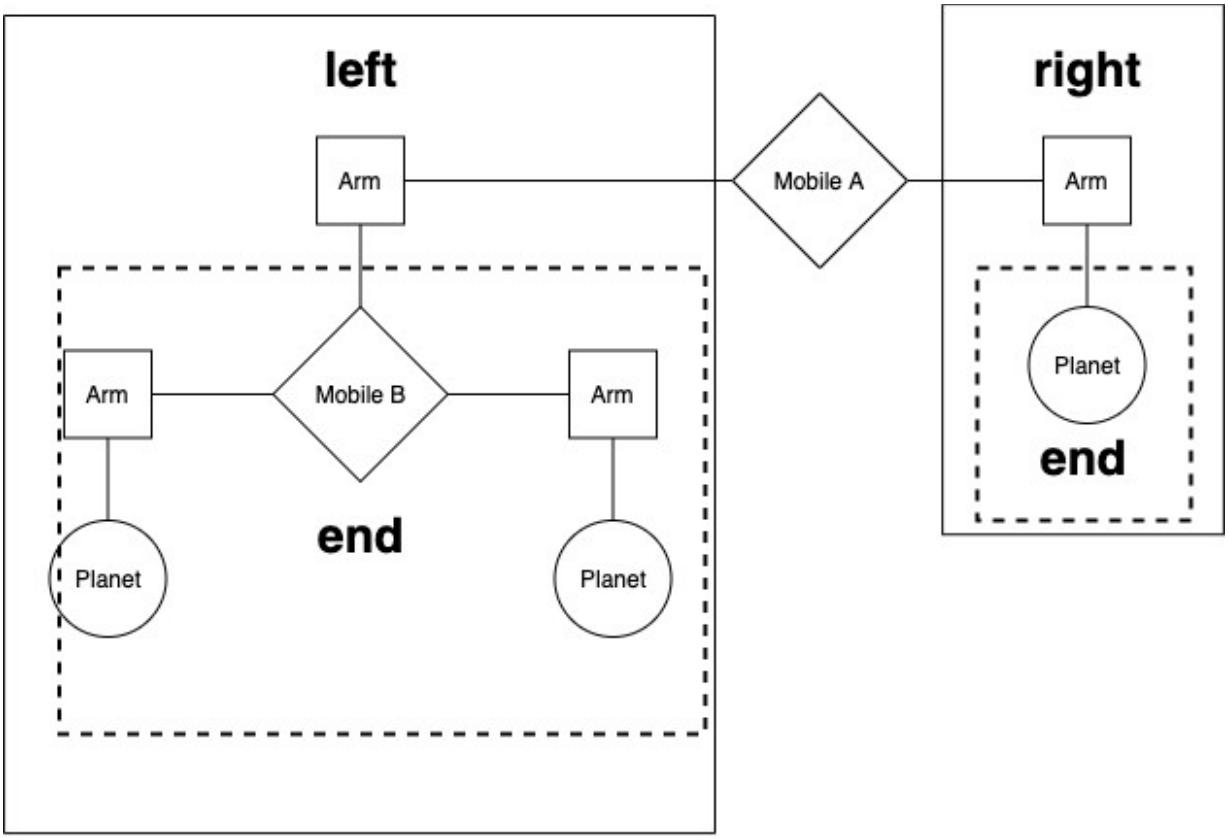
Abstraction

Mobiles

Acknowledgements. This mobile example is based on a classic problem from Structure and Interpretation of Computer Programs, Section 2.2.2 (https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book-Z-H-15.html#%_sec_2.2.2).



We are making a planetarium mobile. A mobile (https://images.zanui.com.au/unsafe/1600x/filters:sharpen(1,0.2,1):quality(80)/production-static.aws.zanui.com.au/p/authentic-models-0459-1.jpg) is a type of hanging sculpture. A binary mobile consists of two arms. Each arm is a rod of a certain length, from which hangs either a planet or another mobile.



We will represent a binary mobile using the data abstractions below.

- A `mobile` has a left `arm` and a right `arm`.
- An `arm` has a positive length and something hanging at the end, either a `mobile` or `planet`.
- A `planet` has a positive size.

Q1: Weights

Implement the `planet` data abstraction by completing the `planet` constructor and the `size` selector so that a planet is represented using a two-element list where the first element is the string `'planet'` and the second element is its size. The `total_weight` example is provided to demonstrate use of the mobile, arm, and planet abstractions.

```
def mobile(left, right):
    """Construct a mobile from a left arm and a right arm."""
    assert is_arm(left), "left must be a arm"
    assert is_arm(right), "right must be a arm"
    return ['mobile', left, right]

def is_mobile(m):
    """Return whether m is a mobile."""
    return type(m) == list and len(m) == 3 and m[0] == 'mobile'

def left(m):
    """Select the left arm of a mobile."""
    assert is_mobile(m), "must call left on a mobile"
    return m[1]

def right(m):
    """Select the right arm of a mobile."""
    assert is_mobile(m), "must call right on a mobile"
    return m[2]
```

```
def arm(length, mobile_or_planet):
    """Construct a arm: a length of rod with a mobile or planet at the end."""
    assert is_mobile(mobile_or_planet) or is_planet(mobile_or_planet)
    return ['arm', length, mobile_or_planet]

def is_arm(s):
    """Return whether s is a arm."""
    return type(s) == list and len(s) == 3 and s[0] == 'arm'

def length(s):
    """Select the length of a arm."""
    assert is_arm(s), "must call length on a arm"
    return s[1]

def end(s):
    """Select the mobile or planet hanging at the end of a arm."""
    assert is_arm(s), "must call end on a arm"
    return s[2]
```

```
def planet(size):
    """Construct a planet of some size."""
    assert size > 0
    return ['planet', size]

def size(w):
    """Select the size of a planet."""
    assert is_planet(w), 'must call size on a planet'
    return w[1]

def is_planet(w):
    """Whether w is a planet."""
    return type(w) == list and len(w) == 2 and w[0] == 'planet'
```

```
def total_weight(m):
    """Return the total weight of m, a planet or mobile.

    >>> t, u, v = examples()
    >>> total_weight(t)
    3
    >>> total_weight(u)
    6
    >>> total_weight(v)
    9
    """
    if is_planet(m):
        return size(m)
    else:
        assert is_mobile(m), "must get total weight of a mobile or a planet"
        return total_weight(end(left(m))) + total_weight(end(right(m)))
```

Use Ok to test your code:

```
python3 ok -q total_weight
```

Q2: Balanced

> **Hint:** for more information on this problem (with more pictures!) please refer to this document (assets/mobiles.pdf#page=3).

Implement the `balanced` function, which returns whether `m` is a balanced mobile. A mobile is balanced if two conditions are met:

1. The torque applied by its left arm is equal to that applied by its right arm. Torque of the left arm is the length of the left rod multiplied by the total weight hanging from that rod. Likewise for the right.
2. Each of the mobiles hanging at the end of its arms is balanced.

Planets themselves are balanced.

```
def balanced(m):
    """Return whether m is balanced.

    >>> t, u, v = examples()
    >>> balanced(t)
    True
    >>> balanced(v)
    True
    >>> w = mobile(arm(3, t), arm(2, u))
    >>> balanced(w)
    False
    >>> balanced(mobile(arm(1, v), arm(1, w)))
    False
    >>> balanced(mobile(arm(1, w), arm(1, v)))
    False
    """
    if is_planet(m):
        return True
    else:
        left_end, right_end = end(left(m)), end(right(m))
        torque_left = length(left(m)) * total_weight(left_end)
        torque_right = length(right(m)) * total_weight(right_end)
        return balanced(left_end) and balanced(right_end) and torque_left == torqu
```

Use Ok to test your code:

```
python3 ok -q balanced
```

The fact that planets are balanced is important, since we will be solving this recursively like many other tree problems (even though this is not explicitly a tree).

- **Base case:** if we are checking a planet, then we know that this is balanced. Why is this an appropriate base case? There are two possible approaches to this:
    1. Because we know that our data structures so far are trees, planets are the simplest possible tree since we have chosen to implement them as leaves.
    2. We also know that from an ADT standpoint, planets are the terminal item in a mobile. There can be no further mobile structures under this planet, so it makes sense to stop check here.
- **Otherwise:** note that it is important to do a recursive call to check if both arms are balanced. However, we also need to do the basic comparison of looking at

the total weight of both arms as well as their length. For example if both arms are a planet, trivially, they will both be balanced. However, the torque must be equal in order for the entire mobile to balanced (i.e. it's insufficient to just check if the arms are balanced).

Q3: Totals

Implement `totals_tree`, which takes a `mobile` (or `planet`) and returns a `tree` whose root is the total weight of the input. For a `planet`, the result should be a leaf. For a `mobile`, the result's branches should be `totals_tree`s for the ends of its arms.

```
def totals_tree(m):
    """Return a tree representing the mobile with its total weight at the root.

    >>> t, u, v = examples()
    >>> print_tree(totals_tree(t))
    3
      2
      1
    >>> print_tree(totals_tree(u))
    6
      1
      5
        3
        2
    >>> print_tree(totals_tree(v))
    9
      3
        2
        1
      6
        1
        5
          3
          2
    """
    if is_planet(m):
        return tree(size(m))
    else:
        branches = [totals_tree(end(f(m))) for f in [left, right]]
        return tree(sum([label(b) for b in branches]), branches)
```

Use Ok to test your code:

```
python3 ok -q totals_tree
```

Trees

Q4: Replace Leaf

Define `replace_leaf`, which takes a tree `t`, a value `old`, and a value `replacement`. `replace_leaf` returns a new tree that's the same as `t` except that every leaf label equal to `old` has been replaced with `replacement`.

```
def replace_leaf(t, old, replacement):
    """Returns a new tree where every leaf value equal to old has
    been replaced with replacement.

    >>> yggdrasil = tree('odin',
    ...                  [tree('balder',
    ...                        [tree('thor'),
    ...                         tree('freya')]),
    ...                   tree('frigg',
    ...                        [tree('thor')]),
    ...                   tree('thor',
    ...                        [tree('sif'),
    ...                         tree('thor')]),
    ...                   tree('thor')])
    >>> laerad = copy_tree(yggdrasil) # copy yggdrasil for testing purposes
    >>> print_tree(replace_leaf(yggdrasil, 'thor', 'freya'))
    odin
      balder
        freya
        freya
      frigg
        freya
      thor
        sif
        freya
      freya
    >>> laerad == yggdrasil # Make sure original tree is unmodified
    True
    """
    if is_leaf(t) and label(t) == old:
        return tree(replacement)
    else:
        bs = [replace_leaf(b, old, replacement) for b in branches(t)]
        return tree(label(t), bs)
```

Use Ok to test your code:

```
python3 ok -q replace_leaf
```

Nonlocal

Q5: Password Protected Account

In lecture, we saw how to use functions to create mutable objects. Here, for example, is the function `make_withdraw` which produces a function that can withdraw money from an account:

```
def make_withdraw(balance):
    """Return a withdraw function with BALANCE as its starting balance.
    >>> withdraw = make_withdraw(1000)
    >>> withdraw(100)
    900
    >>> withdraw(100)
    800
    >>> withdraw(900)
    'Insufficient funds'
    """
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
           return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw
```

Write a version of the `make_withdraw` function that returns password-protected withdraw functions. That is, `make_withdraw` should take a password argument (a string) in addition to an initial balance. The returned function should take two arguments: an amount to withdraw and a password.

A password-protected `withdraw` function should only process withdrawals that include a password that matches the original. Upon receiving an incorrect password, the function should:

1. Store that incorrect password in a list, and
2. Return the string 'Incorrect password'.

If a withdraw function has been called three times with incorrect passwords `<p1>`, `<p2>`, and `<p3>`, then it is locked. All subsequent calls to the function should return:

```
"Your account is locked. Attempts: [<p1>, <p2>, <p3>]"
```

The incorrect passwords may be the same or different:

```
def make_withdraw(balance, password):
    """Return a password-protected withdraw function.

    >>> w = make_withdraw(100, 'hax0r')
    >>> w(25, 'hax0r')
    75
    >>> error = w(90, 'hax0r')
    >>> error
    'Insufficient funds'
    >>> error = w(25, 'hwat')
    >>> error
    'Incorrect password'
    >>> new_bal = w(25, 'hax0r')
    >>> new_bal
    50
    >>> w(75, 'a')
    'Incorrect password'
    >>> w(10, 'hax0r')
    40
    >>> w(20, 'n00b')
    'Incorrect password'
    >>> w(10, 'hax0r')
    "Your account is locked. Attempts: ['hwat', 'a', 'n00b']"
    >>> w(10, 'l33t')
    "Your account is locked. Attempts: ['hwat', 'a', 'n00b']"
    >>> type(w(10, 'l33t')) == str
    True
    """
    attempts = []
    def withdraw(amount, password_attempt):
        nonlocal balance
        if len(attempts) == 3:
            return 'Your account is locked. Attempts: ' + str(attempts)
        if password_attempt != password:
            attempts.append(password_attempt)
            return 'Incorrect password'
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw
```

Use Ok to test your code:

```
python3 ok -q make_withdraw
```

A couple of things to note:

- The `attempts` list does not need to be nonlocal. We're just mutating the list here, not reassigning it.
- The last few lines of our `withdraw` function are the same as the `withdraw` from lecture. There isn't much to add on top of that -- just the list operations and the password checks.

Video walkthrough: https://youtu.be/YyjQoug0Mtg (https://youtu.be/YyjQoug0Mtg)

Q6: Joint Account

Suppose that our banking system requires the ability to make joint accounts. Define a function `make_joint` that takes three arguments.

1. A password-protected `withdraw` function,
2. The password with which that `withdraw` function was defined, and
3. A new password that can also access the original account.

If the password is incorrect or cannot be verified because the underlying account is locked, the `make_joint` should propagate the error. Otherwise, it returns a `withdraw` function that provides additional access to the original account using *either* the new or old password. Both functions draw from the same balance. Incorrect passwords provided to either function will be stored and cause the functions to be locked after three wrong attempts.

Hint: The solution is short (less than 10 lines) and contains no string literals! The key is to call `withdraw` with the right password and amount, then interpret the result. You may assume that all failed attempts to withdraw will return some string (for incorrect passwords, locked accounts, or insufficient funds), while successful withdrawals will return a number.

Use `type(value) == str` to test if some `value` is a string:

```
def make_joint(withdraw, old_pass, new_pass):
    """Return a password-protected withdraw function that has joint access to
    the balance of withdraw.

    >>> w = make_withdraw(100, 'hax0r')
    >>> w(25, 'hax0r')
    75
    >>> make_joint(w, 'my', 'secret')
    'Incorrect password'
    >>> j = make_joint(w, 'hax0r', 'secret')
    >>> w(25, 'secret')
    'Incorrect password'
    >>> j(25, 'secret')
    50
    >>> j(25, 'hax0r')
    25
    >>> j(100, 'secret')
    'Insufficient funds'

    >>> j2 = make_joint(j, 'secret', 'code')
    >>> j2(5, 'code')
    20
    >>> j2(5, 'secret')
    15
    >>> j2(5, 'hax0r')
    10

    >>> j2(25, 'password')
    'Incorrect password'
    >>> j2(5, 'secret')
    "Your account is locked. Attempts: ['my', 'secret', 'password']"
    >>> j(5, 'secret')
    "Your account is locked. Attempts: ['my', 'secret', 'password']"
    >>> w(5, 'hax0r')
    "Your account is locked. Attempts: ['my', 'secret', 'password']"
    >>> make_joint(w, 'hax0r', 'hello')
    "Your account is locked. Attempts: ['my', 'secret', 'password']"
    """
    error = withdraw(0, old_pass)
    if type(error) == str:
        return error
    def joint(amount, password_attempt):
        if password_attempt == new_pass:
            return withdraw(amount, old_pass)
        return withdraw(amount, password_attempt)
    return joint
```

Use Ok to test your code:

```
python3 ok -q make_joint
```

The hint should alert you to the fact that returned strings should indicate an error, which is useful for the initial setup phase.
To make sure that we correctly created the joint account, we attempt to withdraw `0` from it using the supplied password. If this failed, we should exit immediately without creating the account, following the guidance in the doctests.

Otherwise, we have successfully created the joint account! We now know the old password is valid, but remember that the original password-protected account does not know about any new passwords that a joint account might accept. Therefore, when we see something matching the new password, we still have to access the account using the old password.

Video walkthrough: https://youtu.be/h5MvIM1k1Il (https://youtu.be/h5MvIM1k1Il)

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Extra Questions

Q7: Interval Abstraction

Alyssa's program is incomplete because she has not specified the implementation of the interval abstraction. She has implemented the constructor for you; fill in the implementation of the selectors.

```
def interval(a, b):
    """Construct an interval from a to b."""
    assert a <= b, 'Lower bound cannot be greater than upper bound'
    return [a, b]

def lower_bound(x):
    """Return the lower bound of interval x."""
    return x[0]

def upper_bound(x):
    """Return the upper bound of interval x."""
    return x[1]
```

Use Ok to unlock and test your code:

```
python3 ok -q interval -u
python3 ok -q interval
```

Louis Reasoner has also provided an implementation of interval multiplication. Beware: there are some data abstraction violations, so help him fix his code before someone sets it on fire (https://youtu.be/QwoghxwETng).

```
def mul_interval(x, y):
    """Return the interval that contains the product of any value in x and any
    value in y."""
    p1 = x[0] * y[0]
    p2 = x[0] * y[1]
    p3 = x[1] * y[0]
    p4 = x[1] * y[1]
    return [min(p1, p2, p3, p4), max(p1, p2, p3, p4)]
```

Use Ok to unlock and test your code:

```
python3 ok -q mul_interval -u
python3 ok -q mul_interval
```

Video walkthrough: https://youtu.be/RiLrqgBm4Dk (https://youtu.be/RiLrqgBm4Dk)

Q8: Sub Interval

Using reasoning analogous to Alyssa's, define a subtraction function for intervals. Try to reuse functions that have already been implemented if you find yourself repeating code.

```
def sub_interval(x, y):
    """Return the interval that contains the difference between any value in x
    and any value in y."""
    negative_y = interval(-upper_bound(y), -lower_bound(y))
    return add_interval(x, negative_y)
```

Use Ok to unlock and test your code:

```
python3 ok -q sub_interval -u
python3 ok -q sub_interval
```

Video walkthrough: https://youtu.be/s37VvawB0vE (https://youtu.be/s37VvawB0vE)

Q9: Div Interval

Alyssa implements division below by multiplying by the reciprocal of `y`. Ben Bitdiddle, an expert systems programmer, looks over Alyssa's shoulder and comments that it is not clear what it means to divide by an interval that spans zero. Add an `assert` statement to Alyssa's code to ensure that no such interval is used as a divisor:

```
def div_interval(x, y):
    """Return the interval that contains the quotient of any value in x divided by
    any value in y. Division is implemented as the multiplication of x by the
    reciprocal of y."""
    assert not (lower_bound(y) <= 0 <= upper_bound(y)), 'Divide by zero'
    reciprocal_y = interval(1/upper_bound(y), 1/lower_bound(y))
    return mul_interval(x, reciprocal_y)
```

Use Ok to unlock and test your code:

```
python3 ok -q div_interval -u
python3 ok -q div_interval
```

Video walkthrough: https://youtu.be/jNR7tmV1lMk (https://youtu.be/jNR7tmV1lMk)

Q10: Multiple References

Eva Lu Ator, another user, has also noticed the different intervals computed by different but algebraically equivalent expressions. She says that the problem is multiple references to the same interval.

The Multiple References Problem: a formula to compute with intervals using Alyssa's system will produce tighter error bounds if it can be written in such a form that no variable that represents an uncertain number is repeated.

Thus, she says, `par2` is a better program for parallel resistances than `par1`. Is she right? Why? Write a function that returns a string containing a written explanation of your answer:

Note: To make a multi-line string, you must use triple quotes `""" like this """`.

```
def multiple_references_explanation():
    return """The multiple reference problem exists.  The true value
    within a particular interval is fixed (though unknown).  Nested
    combinations that refer to the same interval twice may assume two different
    true values for the same interval, which is an error that results in
    intervals that are larger than they should be.

    Consider the case of i * i, where i is an interval from -1 to 1.  No value
    within this interval, when squared, will give a negative result.  However,
    our mul_interval function will allow us to choose 1 from the first
    reference to i and -1 from the second, giving an erroneous lower bound of
    -1.

    Hence, a program like par2 is better than par1 because it never combines
    the same interval more than once.
    """
```

Video walkthrough: https://youtu.be/H8slb5KCbU4 (https://youtu.be/H8slb5KCbU4)

Q11: Quadratic

Write a function `quadratic` that returns the interval of all values `f(t)` such that `t` is in the argument interval `x` and `f(t)` is a quadratic function (http://en.wikipedia.org/wiki/Quadratic_function):

```
f(t) = a*t*t + b*t + c
```

Make sure that your implementation returns the smallest such interval, one that does not suffer from the multiple references problem.

Hint: the derivative `f'(t) = 2*a*t + b`, and so the extreme point of the quadratic is `-b/(2*a)`:

```
def quadratic(x, a, b, c):
    """Return the interval that is the range of the quadratic defined by
    coefficients a, b, and c, for domain interval x.

    >>> str_interval(quadratic(interval(0, 2), -2, 3, -1))
    '-3 to 0.125'
    >>> str_interval(quadratic(interval(1, 3), 2, -3, 1))
    '0 to 10'
    """
    extremum = -b / (2*a)
    f = lambda x: a * x * x + b * x + c
    l, u, e = map(f, (lower_bound(x), upper_bound(x), extremum))
    if extremum >= lower_bound(x) and extremum <= upper_bound(x):
        return interval(min(l, u, e), max(l, u, e))
    else:
        return interval(min(l, u), max(l, u))
```

Use Ok to test your code:

```
python3 ok -q quadratic
```

Video walkthrough: https://youtu.be/qgSn_RNBs4A (https://youtu.be/qgSn_RNBs4A)

CS 61A (/)

Weekly Schedule (/weekly.html)

Office Hours (/office-hours.html)

Staff (/staff.html)

Resources (/resources.html)

Studying Guide (/articles/studying.html)

Debugging Guide (/articles/debugging.html)

Composition Guide (/articles/composition.html)

Policies (/articles/about.html)

Assignments (/articles/about.html#assignments)

Exams (/articles/about.html#exams)

Grading (/articles/about.html#grading)