Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [92]:
```
NAME = ""
COLLABORATORS = ""
```

CSE 30 Spring 2022 - Homework 9

Instructions

Please disregard the YOUR NAME and COLLABORATORS above. They are put there atomatically by the grading tool. You can find instructions on how to work on a homework on Canvas. Here is a short summary:

Submitting your work

To submit your work:

- First, click on "Runtime > Restart and run all", and check that you get no errors. This enables you to catch any error you might have introduced, and not noticed, due to your running cells out of order.
- Second, download the notebook in .ipynb format (File > Download .ipynb) and upload the .ipynb file to this form.

You can submit multiple times; the last submission before the deadline is the one that counts. This homework is due by **11:59pm on Tuesday 10 May, 2022**.

Homework format

For each question in this notebook, there is:

- A text description of the problem.
- One or more places where you have to insert your solution. You need to complete every place marked:

  `# YOUR CODE HERE`

  and you should not modify any other place.

- One or more test cells. Each cell is worth some number of points, marked at the top. You should not modify these tests cells. The tests pass if no error is printed out: when there is a statement that says, for instance:

  ```
  assert x == 2
  ```

  then the test passes if `x` has value 2, and fails otherwise. You can insert a `print(x)` (for this case!) somewhere if you want to debug your work; it is up to you.

Notes:

- Your code will be tested both according to the tests you can see (the `assert` statements you can see), *and* additional tests. This prevents you from hard-coding the answer to the particular questions posed. Your code should solve the *general* intended case, not hard-code the particular answer for the values used in the tests.
- **Please do not delete or add cells!** The test is autograded, and if you modify the test by adding or deleting cells, even if you re-add cells you delete, you may not receive credit.
- **Please do not import modules that are not part of the standard library.** You do not need any, and they will likely not available in the grading environment, leading your code to fail.
- **If you are inactive too long, your notebook might get disconnected from the back-end.** Your work is never lost, but you have to re-run all the cells before you continue.
- You can write out print statements in your code, to help you test/debug it. But remember: the code is graded on the basis of what it outputs or returns, not on the basis of what it prints.
- **TAs and tutors have access to this notebook,** so if you let them know you need their help, they can look at your work and give you advice.

Grading

Each cell where there are tests is worth a certain number of points. You get the points allocated to a cell only if you pass *all* the tests in the cell.

The tests in a cell include both the tests you can see, and other, similar, tests that are used for grading only. Therefore, you cannot hard-code the solutions: you really have to solve the essence of the problem, to receive the points in a cell.

Code of Conduct

- Work on the test yourself, alone.

- You can search documentation on the web, on sites such as the Python documentation sites, Stackoverflow, and similar, and you can use the results.
- You cannot share your work with others or solicit their help.

We will develop a data structure to represent arithmetic expressions containing variables, such as $3 + 4$ or $2 + x * (1 - y)$.

What is an expression? An expression consists of one of these:

1. A number
2. A variable
3. If e_1 and e_2 are expressions, then $e_1 + e_2$, $e_1 - e_2$, $e_1 * e_2$, and e_1/e_2 are also expressions.

Formally, the set of expressions is the *least* set constructed according to the rules above.

Thus, an expression can be either a constant, representing numbers and variables, or a composite expression, consisting of an operator, a left expression, and a right expression.

There are (at least) two ways of representing expressions. The simplest way is to represent expressions as trees, and define operations on them. The more sophisticated way consists in representing expressions via classes: there will be one class for variable and constants, and one class representing composite expressions; both of these classes will be subclasses of a generic "expression" class.

In this chapter, we will represent expression as trees, to gain experience with writing recursive functions on trees; in the next chapter, we will show how to represent them more elegantly as classes.

We will represent expressions as trees. A number will be represented via a number; a variable via a string, and the expression $e_1 \odot e_2$ via the tuple $(\odot, e_1, e_2)$, for $\odot \in \{+, -, *, /\}$.

For example, we will represent $2 * (x + 1)$ via:

```
('*', 2, ('+', 'x', 1))
```

In [93]:
```
e = ('*', 2, ('+', 'x', 1))
```

In particular, we will consider expressions built out of the four arithmetic operators `"+"`, `"-"`, `"*"`, `"/"`.

A compute function

Let us define a function compute() that takes one such expression, and returns the expression obtained by performing all possible numerical computation. We consider first the

simple case of an expression where the only operators that can appear are `+` and `-`, and where there are no variables.

Let us create an exception to raise when we cannot interpret an expression.

In [94]:
```
class IllegalOperator(Exception):
    pass
```

Let us define a helper function `calc`, which takes as argument an operator and two numbers, and computes the required operation. It will make it easier to write the rest of the code.

In [95]:
```
def calc(op, left, right):
    if op == "+":
        return left + right
    elif op == "-":
        return left - right
    elif op == "*":
        return left * right
    elif op == "/":
        return left / right
    else:
        raise IllegalOperator(op)
```

With this, we can write our compute method as follows.

In [96]:
```
def compute(e):
    if isinstance(e, tuple):
        # We have an expression.
        op, l, r = e
        # We compute the subexpressions.
        ll = compute(l)
        rr = compute(r)
        # And on the basis of those, the whole expression.
        return calc(op, ll, rr)
    else:
        # base expression; just return the number.
        return e
```

In [97]:
```
compute(("+", 4, 5))
```

Out[97]: 9

In [98]:
```
compute(("+", ("-", 3, 1), ("+", 4, 9)))
```

Out[98]: 15

Expressions with variables

If an expression can have variables, we can distinguish three types of expressions:

- Numbers
- Variables
- Composite expressions.

To facilitate writing code, let us define for you three helper functions that tell us the type of an expression.

In [99]:
```
from numbers import Number # The mother class of all numbers.

def isnumber(e):
    return isinstance(e, Number)

def isvariable(e):
    return isinstance(e, str)

def iscomposite(e):
    return isinstance(e, tuple)
```

The idea we use to simplify an expression is the following:

- If the expression is a Number, you return a number: it's already simplified.
- If the expression is a variable, you return the variable (that is, the expression unchanged); there is nothing to be done.
- If the expression is an operation, such as "+", "-", ..., then you consider the right and left children, and you reason:
  - If all the two children are numbers, then you can compute the operation and return the result.
  - Otherwise, again, there is nothing that can be done, and you return the expression unchanged.

In [100…
```
def simplify(e):
    if isinstance(e, tuple):
        op, l, r = e
        # We simplify the children expressions.
        ll = simplify(l)
        rr = simplify(r)
        # We compute the expression if we can.
        if isnumber(ll) and isnumber(rr):
            return calc(op, ll, rr)
        else:
            return (op, ll, rr)
    else:
        # Leaf. No simplification is possible.
        return e
```

Let's see how this works.

In [101…
```
simplify(3)
```

Out[101… 3

In [102…
```
simplify(("+", "x", 1))
```

Out[102…
```
('+', 'x', 1)
```

Yes, there was nothing we could simplify. Let's try now with something we can simplify:

In [103…
```
simplify(('-', 5, 4))
```

Out[103…
```
1
```

Can we simplify bigger expressions?

In [104…
```
simplify( ('+', 6, ('-', 7, 2)) )
```

Out[104…
```
11
```

Question 1: Evaluating expressions with respect to a variable valuation.

The function `simplify` above can perform all numerical computations, but stops whenever it encounters a variable. It cannot do any better, in fact, because it does not know the values of variables.

If we specify values for variables, we can then use those values in the computation, replacing each variable whose value is specified with the value itself.

A *variable valuation* is a mapping from variables to their values; we can represent it simply as a dictionary associating to each variable a number:

In [105…
```
varval = {'x': 3, 'y': 8}
```

You can extend the evaluation function to take as input a variable valuation. The idea is that, when you find a variable, you try to see whether its value is specified in the variable valuation. If it is, you can replace the variable with the value, and carry on. If it is not, you leave the variable as it is, since you cannot evaluate it.

To check if a variable (a string) `s` is in a dictionary `d`, you can test

```
s in d
```

and to get the value, in case it is present, you can just do `d[s]` of course. We let you develop the code.

In [106…
```
### Evaluating an expression with respect to a variable evaluation

def compute(e, varval={}):
    # YOUR CODE HERE
```

```
    se = simplify(e)
    if isinstance(se, str) and se in varval:
        return varval[se]
    elif isinstance(e, tuple):
        op, l, r = e
        cl, cr = compute(l, varval), compute(r, varval)
        return simplify((op, cl, cr))
    else:
        # base expression; just return the number.
        return e
```

Let us play with this.

In [107…
```
## You can modify this cell to test your code.

e = ('+', 'x', 4)
print(compute(e))
compute(e, varval={'x': 3})
print(compute(e, varval={'x': 6}))
```

```
('+', 'x', 4)
10
```

If we provide the values for only some of the variables, the compute function defined above, will plug in the values for those variables and perform all computations possible. Of course, if the expression contains variables for which the valuation does not specify a value, the resulting expression will still contain those variables: it will not be simply a number. In computer science, evaluating an expression as far as possible using the values for a subset of the variables is knwon as *partial evaluation*.

In [108…
```
## Tests for compute.  10 points.

e = 'x'
assert compute(e) == 'x'
assert compute(e, varval={'x': 3}) == 3

e = ('*', 2, ('+', 'x', ('-', 3, 2)))
assert compute(e) == ('*', 2, ('+', 'x', 1))
assert compute(e, varval={'x': 6}) == 14
assert compute(e, varval={'y': 10}) == ('*', 2, ('+', 'x', 1))

e = ('+', ('-', 'yy', 3), ('*', 'x', 4))
assert compute(e, varval={'x': 2}) == ('+', ('-', 'yy', 3), 8)
assert compute(e, varval={'yy': 3}) == ('+', 0, ('*', 'x', 4))
assert compute(e, varval={'x': 2, 'yy': 3}) == 8
```

In [109…
```
### Hidden tests for compute.  10 points.
```

When are two expressions equal?

> *Or: it's better to be lucky than to be smart.*

> *Or: if you don't know how to do it right, do it at random.*

> *Or: the power of randomization.*

We now consider the following problem: given two expressions e and f, how can we decide whether they are equal in value, that is, whether they yield always the same value for all values of the variables?

This "*value equality*" is a different notion from the structural equality we defined before. For instance, the two expressions `("+", "x", 1)` and `("-", (*, 2, "x"), "x")` are not structurally equal, but they are equal in values.

How can we test for value equality of expressions? There are two ways: the high road, and the pirate road. Of course, we take the pirate road.

The high-road approach consists in trying to demonstrate, in some way, that the two expressions are equal. One way of doing so would be to define a set of rewriting rules for expressions, that try to transform one expression into the other; this would mimick the process often done by hand to show that two expressions are equal. Another way would be to use theorem provers that can reason about expressions and real numbers, such as PVS. The problem is that these approaches are a lot of work. Is there a way to be lazy, and still get the job done?

There is, it turns out. Suppose you have two expressions f, g containing variable x only. The idea is that if f and g are built with the usual operators of algebra, it is exceedingly unlikely for f and g to give the same value many values of x, and yet not be always equal. This would not be true if our expressions could contain if-then-else statements, but for the operators we defined so far, it holds. Indeed, one could be more precise, and try to come up with a theorem of the form:

> If f and g have "zerosity" n, and are equal for $n+1$ values of x, then they are equal for all values of x.

We could then try to define the "zerosity" of an expression to make this hold: for example, for two polynomials of degree at most d, once you show that they are equal for $d+1$ points, they must be equal everywhere (why?). But this again would be a smart approach, and we are trying to see if we can solve the problem while being as stupid as possible. So our idea will simply be: pick 1000 values of x at random; if the two expressions are equal for all the values, then they must be equal everywhere. This is a somewhat special case of a Monte Carlo method, a method used to estimate the probability of complex phenomena (where expression equality is our phenomenon).

There are only two wrinkles with this. The first is that an expression can contain many variables, and we have to try to value assignments for all of the variables. This is easy to overcome; we just need some helper function that gives us the set of variables in a function.

The second wrinkle is: how do we generate the possible value assignments? How big do these values need to be on average? According to what probability distribution? We could dive into a lot of theory and reasoning about how to compute appropriate probability distributions, but since our goal is to be stupid, we will use one of the simplest distributions with infinite domain: the Gaussian one.

Question 2: Variable Occurrences

Let us start by writing the function `variables` such that, if `e` is an expression, `variables(e)` is the set of variables that appear in it.

In [110…
```
### Exercise: define `variables`

# YOUR CODE HERE
def variables(e):
    varset = set()
    se = simplify(e)
    if isvariable(se):
        varset.add(se)
    elif iscomposite(se):
        op, l, r = se
        ll, rr = variables(l), variables(r)
        varset.update(ll)
        varset.update(rr)
    return varset
```

In [111…
```
### Here you can write your own test code.
```

In [112…
```
### 10 points. Tests for `Expr.variables`

e = ('*', ('+', 'x', 2), ('/', 'x', 'yay'))
assert variables(e) == {'x', 'yay'}
```

Question 3: Value equality

Now we let you write the `value_equality` method.
Given the two expressions e and f, first compute the set of variables V that appear in either e or f. Then, the idea consists in performing `num_sample` times the following test for equality:

- First, produce a variable assignment (a dictionary) mapping each variable in V to a random value. Choose these random values from the gaussian distribution centered around 0 and with standard deviation 10 (for instance; any continuous distribution with infinite domain would work). You can obtain such numbers using `random.gauss(0, 10)`.

- Then, compute the values of e and f with respect to that variable evaluation. If the values are closer than a specified tolerance `tolerance`, you consider e and f equal (for that variable valuation). Otherwise, you can stop and return that e and f are different.

If you can repeat the process `num_sample` times, and e and f are considered equal every time, then you declare them equal.

In [113…
```
### Exercise: implementation of value equality

import random

def value_equality(e, f, num_samples=1000, tolerance=1e-6):
    """Return True if the two expressions self and other are numerically
    equivalent.  Equivalence is tested by generating
    num_samples assignments, and checking that equality holds
    for all of them.  Equality is checked up to tolerance, that is,
    the values of the two expressions have to be closer than tolerance.
    It can be done in less than 10 lines of code."""
    # YOUR CODE HERE
    varset = variables(e)
    varset.update(variables(f))
    for i in range(num_samples):
        varval = {}
        for v in varset:
            varval[v] = random.gauss(0,10)
        ce = compute(e, varval)
        cf = compute(f, varval)
        if tolerance<abs(ce-cf):
            return False
    return True
```

In [114…
```
### Here you can test your solution.
```

In [115…
```
### 5 points: Tests for value equality

assert value_equality('x', 'x')
assert value_equality(3, 3)
assert not value_equality(3, 'x')

e1 = ('+', ('*', 'x', 1), ('*', 'y', 0))
e2 = 'x'
assert value_equality(e1, e2)

e3 = ('/', ('*', 'x', 'x'), ('*', 'x', 1))
e3b = ('/', ('*', 'x', 'y'), ('*', 'x', 1))
assert value_equality(e1, e3)
assert not value_equality(e1, e3b)

e4 = ('/', 'y', 2)
assert not value_equality(e1, e4)
assert not value_equality(e3, e4)
```

```
e5 = ("+", "cat", ("-", "dog", "dog"))
assert value_equality(e5, "cat")
```

In [116…

```
## 10 points: hidden tests for value equality.
```

Symbolic Expressions

The notation we developed enables the representation of *symbolic expressions:* expressions in which not only numbers appear, but also symbols. Accordingly, we can perform *symbolic* operations on the expressions: operations that encode what you do with pencil and paper when you work on an expression.

Our first symbolic operations will be the implementation of *derivatives*.

Derivatives

Given an expression f and a variable x, we can compute symbolically the (partial) derivative

$$\frac{\partial f}{\partial x}$$

of f with respect to x. For instance:

$$\frac{\partial}{\partial x}(x^2 + 3y + 4) = 2x$$

$$\frac{\partial}{\partial x}(x^2y + xy^2 + z) = 2yx + y^2 + z\,.$$

Here, the "partial" in partial derivative simply means: if you need to take the derivative with respect to a specific variable, simply treat all other variables as constants.

Computing symbolic derivatives seems complicated, but we can take it in steps. Our expression trees have leaves, consisting of

- numerical constants
- variables

and operators, corresponding to `+`, `-`, `*`, `/`. Therefore, we break down the task into the computation of the symbolic derivative for numerical constants, variables, and arithmetic operators. The cases for the leaf nodes are:

- For a constant c, $\partial c/\partial x = 0$.
- For a variable $y \neq x$, $\partial y/\partial x = 0$.
- $\partial x/\partial x = 1$.

For operators, we can use:

$$\frac{\partial}{\partial x}(f \pm g) = \frac{\partial f}{\partial x} \pm \frac{\partial g}{\partial x},$$
$$\frac{\partial}{\partial x}(f \cdot g) = \frac{\partial f}{\partial x} \cdot g + f \cdot \frac{\partial g}{\partial x},$$
$$\frac{\partial}{\partial x}\left(\frac{f}{g}\right) = \frac{\frac{\partial f}{\partial x} \cdot g - f \cdot \frac{\partial g}{\partial x}}{g^2}.$$

Question 4: Derivative of a leaf expression

Let's start from a leaf expression. The function `derivate_leaf` takes as argument an expression that is a leaf, and a variable, and returns the symbolic derivative of the leaf writh respect to the variable.

In [126…
```
### Derivation of a leaf expression

def derivate_leaf(e, x):
    """This function takes as input an expression e and a variable x,
    and returns the symbolic derivative of e wrt. x, as an expression."""
    # YOUR CODE HERE
    if isnumber(e):
        return 0
    if isvariable(e) and e == x:
        return 1
    if isvariable(e):
        return 0
```

In [118…
```
### You can play here with your code.
```

Here are some tests.

In [127…
```
# 5 points.

assert derivate_leaf("x", "x") == 1
assert derivate_leaf("x", "y") == 0
assert derivate_leaf("y", "z") == 0
assert derivate_leaf(4, "x") == 0
```

Question 5: Derivative of an expression

We now let you implement the derivative of a general expression. The function is recursive: for instance, if the expression is (`*` , f, g), to compute its derivative with respect to a variable x, we use:

$$\frac{\partial}{\partial x}(f \cdot g) = \frac{\partial f}{\partial x} \cdot g + f \cdot \frac{\partial g}{\partial x}$$

and so we need to recursively call symbolic derivation on the f and g sub-expressions, to obtain $\partial f/\partial x$ and $\partial g/\partial x$, and then produce and return an expression representing the result.

Important: The code for checking expression equality is *not* able to cope with the commutativity of `+` and `*`. So, in your solution, please use *exactly* these forms:

$$\frac{\partial}{\partial x}(f \pm g) = \frac{\partial f}{\partial x} \pm \frac{\partial g}{\partial x},$$
$$\frac{\partial}{\partial x}(f \cdot g) = \frac{\partial f}{\partial x} \cdot g + f \cdot \frac{\partial g}{\partial x},$$
$$\frac{\partial}{\partial x}\left(\frac{f}{g}\right) = \frac{\frac{\partial f}{\partial x} \cdot g - f \cdot \frac{\partial g}{\partial x}}{g^2}.$$

and not, for instance, $\frac{\partial}{\partial x}(f \cdot g) = g \cdot \frac{\partial f}{\partial x} + f \cdot \frac{\partial g}{\partial x}$, which is mathematically equivalent, but would be considered different by the tests.

In [131…
```
### Implement `derivate`

def derivate(e, x):
    """Returns the derivative of e wrt x.
    It can be done in less than 15 lines of code."""
    # YOUR CODE HERE
    if iscomposite(e):
        op, fx, gx = e
        if op == "/":
            n = ("-", ("*", derivate(fx, x), gx), ("*", fx, derivate(gx, x)))
            d = ("*", gx, gx)
            return ("/", n, d)
        if op == "*":
            return ("+", ("*", derivate(fx, x), gx), ("*", fx, derivate(gx, x)))
        if op == "+" or op == "-":
            return (op, derivate(fx, x), derivate(gx, x))
    else:
        return derivate_leaf(e, x)
```

In [129…
```
### You can play here with your code.
```

In [132…
```
### Tests for `derivate` for single-operator expressions

assert derivate(('+', 'x', 'x'), 'x') == ('+', 1, 1)
assert derivate(('-', 4, 'x'), 'x') == ('-', 0, 1)
assert derivate(('*', 2, 'x'), 'x') == ('+', ('*', 0, 'x'), ('*', 2, 1))
assert derivate(('/', 2, 'x'), 'x') == ('/', ('-', ('*', 0, 'x'), ('*', 2, 1)), ('*
```

In [133…
```
### 10 points: Hidden tests for `derivate` for single-operator expressions
```

In [134…
```
### 10 points: Tests for `derivate` for composite expressions

e1 = ('*', 'x', 'x')
```

```
e2 = ('*', 3, 'x')
num = ('-', e1, e2)
e3 = ('*', 'a', 'x')
den = ('+', e1, e3)
e = ('/', num, den)

f = ('/',
 ('-',
  ('*',
   ('-',
    ('+', ('*', 1, 'x'), ('*', 'x', 1)),
    ('+', ('*', 0, 'x'), ('*', 3, 1))),
   ('+', ('*', 'x', 'x'), ('*', 'a', 'x'))),
  ('*',
   ('-', ('*', 'x', 'x'), ('*', 3, 'x')),
   ('+',
    ('+', ('*', 1, 'x'), ('*', 'x', 1)),
    ('+', ('*', 0, 'x'), ('*', 'a', 1))))),
 ('*',
  ('+', ('*', 'x', 'x'), ('*', 'a', 'x')),
  ('+', ('*', 'x', 'x'), ('*', 'a', 'x'))))

assert derivate(e, 'x') == f
```

In [134…