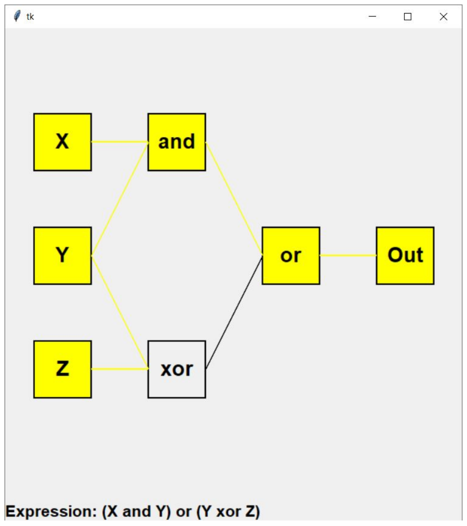
15-110 Hw6 - Circuit Simulator

Hw6 and its checks are organized differently from the other assignments. If you haven't already done so, you should read the **Hw6 General Guide** to understand how this assignment works.

Project Description



Expression: (X and Y) or (Y xor Z)

In this project, you will build a simple circuit simulator that can take a boolean expression in text format, generate an interactive circuit for the expression using simulation, and create a truth table for the expression automatically. This will allow you to explore what different boolean expressions output.

In the first week, you will write code that can parse a boolean expression into a circuit tree. In the second week, you will evaluate that circuit tree on different inputs and generate truth tables. In the third week, you will visualize that circuit tree using simulation, and allow the user to interact with the circuit by clicking on the inputs.

Click on the following links to read the instructions for each week's assignment:

Check6-1 - due Wednesday 4/15 at noon EDT

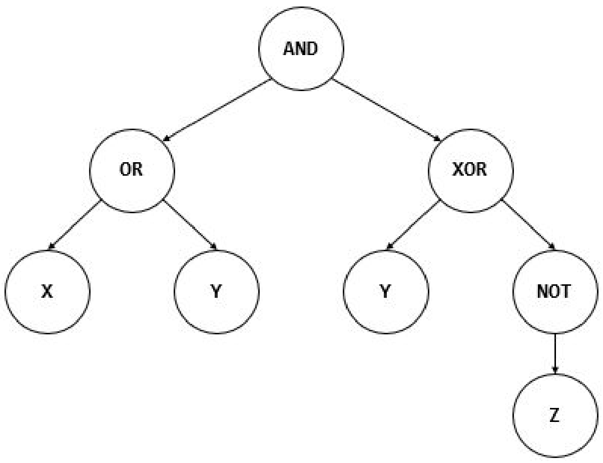
Check6-2 - due Wednesday 4/22 at noon EDT

Hw6 - due Wednesday 4/29 at noon EDT

Check6-1 - due Wednesday 4/15 at noon EDT

In the first stage, you will write code that can convert a boolean expression as a string into a circuit tree, as described below. You will also validate that the produced circuit trees are valid circuits. This conversion will make it possible to make truth tables and simulated circuits in the following stages of the project.

A **circuit tree** is like an expression tree- it's a way of representing a circuit based on operation order For example, in the expression (X OR Y), OR is the root node of the tree, and it has two children- leaf node X and leaf node Y. By representing expressions as trees, we can evaluate complex nested expressions, like ((X OR Y) AND (Y XOR (NOT Z))), which parses to the tree shown below.



Step 0: Written Assignment **[45pts]**

In addition to completing the steps described below, there is a short written assignment on the week's material. You can find the written assignment on the course website.

Step 1: Find Matching Parentheses **[10pts]**

First, we need to write two helper functions that will be used by our primary parsing function. These functions will identify parts of a boolean expression by A) recognizing which part of the text is part of a pair of parentheses, and B) finding the starting and ending indexes of tokens (like variables and operators) in the text.

First, let's write the helper function for Task A. Implement the function `findMatchingParen(expr, index)` in the starter file, which takes a boolean expression string and a starting index. The function returns the index of the parenthesis that is *paired* with the parenthesis at the starting index.

For example, the expression **`((X OR Y) AND (Y XOR (NOT Z)))`** has had parentheses color-coded to demonstrate where the pairs are. If we call the function on this expression with the index `1` (the orange starting paren), it should return `8` (the orange ending paren's index).

To solve this problem, keep track of how many open and closed parentheses have been seen so far as you iterate over the string. When these counts are equal, you have found the closing paren; return the index you're on.

To test this function, run testFindMatchingParen().

Step 2: Find Token Bounds **[10pts]**

Now let's write the second helper function, which will help us identify tokens. A *token* in a string is a part of the text that is interpreted together, like a word. We'll separate tokens in this project based on spaces and parentheses.

Implement the function `getTokenBounds(expr, start)` which takes a boolean expression as a string and a starting index, and identifies the bounds of the next token that occurs at or after that starting index. You should return a two-element list where the first element is the start index of the token, and the second element is the last index of the token. For example, given the expression `((X OR Y) AND (Y XOR (NOT Z)))`, the next token starting at index `3` would have start index 4 and end index 5; it's the token `OR`. So the function would return the list `[4, 5]`.

To solve this, keep iterating over indexes in the string (starting at the start index) until you find something that isn't a space. Then keep iterating until you find something that IS a space. That gives you your start and end indexes. Note that the end index should be *inclusive*; in the example above, `5` is the index of 'R', not the space after the 'R'.

To test this function, run `testGetTokenBounds()`.

Step 3: Parse Boolean Expressions **[15pts]**

Now we have everything that we need to parse a boolean expression into a tree. Some boolean operations (like `not`) don't have a sense of a `"left"` or `"right"` child, so we'll use a **general tree format** instead of a binary tree format. This format will still represent a tree as a dictionary, but will only have two keys - `"value"`, which maps to the value, and `"children"`, which maps to a **list of the children** (which are also trees). A node with no children (in other words, a leaf) would map `children` to an empty list.

Implement the function `parseExpr(expr)` in the starter file, which takes a string and returns a tree (as a dictionary).

First, run `.strip()` on the expression to remove any extra whitespace. Then, recursively parse the expression. To do this, we'll need to consider a few cases.

- If the string begins with a left parenthesis, call `findMatchingParen`. If the index of the paired parenthesis is the end of the string, there are paired parens wrapping the whole expression. Recursively call `parseExpr` on everything but the first and last characters, and return the result.

- If there are no spaces in the string, this is a single token, a variable. Return a leaf, where the value is the token name and children is an empty list.

- If the expression begins with `"NOT"`, this is a not operation. Recursively call the function on all but the first three characters of the string to get the value of the expression. Then return a tree with the value `"NOT"` and a children list with one node, the recursive result.

- Otherwise, this is a two-element operation. We need to find the left value, the operator, and the right value.
    - To get the left value, you have to determine whether this value is an expression within parentheses, or a single token. We'll assume that there are no ambiguous expressions (like `X AND Y OR Z`).
        - If the first token of the expression is an open paren, call `findMatchingParen` to find the index of the closing paren. This gives you the start and end indexes of the entire left side.
        - Otherwise, assume it's a variable. Call `getTokenBounds` to get the start and end indexes of the entire left side.
    - Once we have the left side, we can call `getTokenBounds` starting from the space after the left side to get the bounds of the operator.
    - Then the right side is the rest of the string.

- Recursively call the function on the left side and right side to get the two child trees. Then return a tree with the operator as the value, and a children list with two nodes, which are the two sides returned before.

To test this function, run `testParseExpr()`.

Step 4: Validate the Circuit Tree **[10pts]**

Now that we can generate trees from strings, we need to ensure that every circuit we make is valid. The circuits we test should only have four types of operators - `NOT`, `AND`, `OR`, and `XOR` gates.

Implement the function `validateExpr(expr)` in the starter file. This takes a tree (generated by `parseExpr`) and returns `True` if that tree is valid, or `False` otherwise.

In general, to validate a circuit tree, you first check that it's a valid node (ie, does it have `"value"` and `"children"` keys). Then check whether all of the tree's children are valid. If they are, verify that the number of children matches the value. `AND`, `OR`, and `XOR` gates should have two children; `NOT` gates should have one child; and anything else should have zero children (as anything else is a variable).

To test this function, run `testValidateExpr()`.

Step 5: Put it All Together **[10pts]**

Now we just need to put everything together, and we have a working Stage 1 program! Implement the function `runProgram()` from the starter file. This function should ask the user to input a boolean expression (using the `input()` command), parse the expression into a tree, and check if the tree is valid. If it is, the program should print out the tree; if it is not, the program should report that the tree is invalid. The program should also report an invalid tree if the string cannot be parsed.

How can we tell if a string can't be parsed? We could try to check for mistakes (like missing parentheses or bad gate names) as we parse, but that would be difficult. An easier approach is to let the program crash if the parsing doesn't work, but then **catch** the error before it can reach the user, and display a nice error message instead.

We can do this by using a **try-except** statement. This takes the following format:

```
try:
    <put code to run here>
except:
    <put code to run if an error happens here>
```

If the code in the try block runs successfully, the program will exit normally. If it crashes, the statement will catch the exception and redirect the program to the except block. So the following code would print `"Not a number"` instead of crashing:

```
try:
    int("foo")
except:
    print("Not a Number")
```

If you put your call to `parseExpr` in the try statement, you can catch any errors that occur and print a message that tells the user their expression was invalid.

To test your program, run `runProgram()` and try giving it a few different boolean expressions. Make sure to include one valid expression, one invalid (but parseable) expression, and one non-parsing expression.

Check6-2 - due Wednesday 4/22 at noon EDT

In the second stage, you will write code to evaluate your circuit tree on any possible input, and use this code to generate a truth table in text format. This code will also help support the circuit simulation in Stage 3.

Step 0: Written Assignment **[45pts]**

In addition to completing the steps described below, there is a short written assignment on the week's material. You can find the written assignment on the course website.

Step 1: Make Leaf List **[10pts]**

First, we need to determine what the inputs of the circuit are. We can do this by making a list of all the leaves in the tree. When we remove the duplicate leaves, this gives us all the possible inputs!

Implement the function `getLeaves(t)` from the starter file. This takes a circuit tree and returns a sorted list of all the leaf values of that tree, with no repeats. You will need to implement this function recursively (as it parses a tree).

In the base case, the result is simple- a tree with no children is a leaf, so you can return a list holding that leaf's value.

In the recursive case, call the function recursively on each child of the tree. For each leaf in the resulting lists, add it to a result list only if it has not already been added.

To test this function, run `testGetLeaves()`.

Step 2: Generate All Input Combinations **[10pts]**

Next, we need to generate every possible combination of inputs that could be used to evaluate the boolean expression. Each input combination will be a list of n `False` and `True` values, where n is the number of inputs in the circuit.

We'll solve this problem recursively. If we can generate all possible input combinations for n-1 inputs, we can solve the problem for n inputs by making two versions of each input from the preliminary result- one version that adds a `False` value to the end, and one version that adds a `True` value to the end.

Implement the function `generateAllInputs(n)` in the starter file, which takes an integer `n` and returns a list of all possible input lists that have `n` elements. You should use the following algorithm:

- In the base case (when `n` is 0), return a 2D list with one element - an empty list.
- Otherwise…
    - Recursively generate all possible input combinations for `n-1` elements.
    - Make a new result list
    - For each input list in the `n-1` result:
        - Add that input list plus a `False` element to the new result list
        - Add that input list plus a `True` element to the new result list
    - Return the new result list

To test this function, run `testGenerateAllInputs()`.

Step 3: Evaluate Circuit **[15pts]**

Next, implement the function `evalTree(t, inputs)` in the starter file, which takes a tree and a dictionary, and returns a boolean value. This function will evaluate a circuit tree based on a dictionary that maps input variables to booleans (`False` or `True`).

We will do this by recursively evaluating each of the children of the circuit tree, then applying the appropriate boolean operator to the child results. If the input tree is a leaf (a variable), look it up in the input dictionary to find its value.

To test this function, run `testEvalTree()`.

Step 4: Make Truth Table **[15pts]**

Now we can finally generate an actual truth table! In this step, you will use the functions you have written so far to write out each line of a truth table, by determining the inputs of a tree and testing each possible mapping of `False` and `True` to the inputs.

Implement the function `makeTruthTable(tree)` in the starter file. This takes a tree (generated by `parseExpr`) and prints out a truth table showing all the possible inputs to the tree, and the resulting outputs. You may personalize this truth table however you wish, as long as it shows every input combination and every possible output.

To write this function, start by using `getLeaves()` to get the inputs and `generateAllInputs()` to get the rows of the truth table. For each row, create an input dictionary that maps each input variable name to its value (`False` or `True`). Then run `evalTree` on the tree and the input dictionary, and display the result.

To test this function, run `testMakeTruthTable()` and check the table that is displayed. Here is the table we generated; yours may look different than ours, as long as the same general information is displayed. Note that we replaced `False`/`True` with `0`/`1` to make the table more readable.

```
X | Y | Z | Out
0 | 0 | 0 |  0
0 | 0 | 1 |  0
0 | 1 | 0 |  1
0 | 1 | 1 |  1
1 | 0 | 0 |  1
1 | 0 | 1 |  1
1 | 1 | 0 |  1
1 | 1 | 1 |  1
```

Step 5: Update Run Function **[5pts]**

For the last step, update your `runProgram()` function from the first check-in. When the tree is valid, instead of printing the tree dictionary, call `makeTruthTable` on the tree. Now you can generate truth tables for any circuit you want!

Hw6 - due Wednesday 4/29 at noon EDT

In the final stage of the project, you will use your circuit-solving code to program a circuit-solving simulator. You will do this using Tkinter and the class's simulation framework.

In the simulation, the user will type the boolean expression into the window; when they press Enter, the circuit will be drawn on the screen (with all inputs starting in the Off position). If the user clicks on an input to the circuit, it should turn On. At all times, any nodes and wires in the circuit that are on (or powered) should be drawn as yellow. And finally, if the user presses the Tab button, it should produce a truth table for the circuit in the interpreter.

In order to track which parts of the circuit are powered, we will add one variable to each node in the circuit tree- `"powered"`, which maps to `False` if the node evaluates to `False`, or `True` otherwise.

Step 0-A: Complete Check6-1 **[20pts]**

If you got a perfect score on Check6-1 (the project part), congratulations; this step is already done! Go to the next step.

Otherwise, go back to your Gradescope feedback on Check6-1 and use it to update your Check6-1 code. This is your chance to implement any features you might have missed before, and fix any code that isn't working.

Step 0-B: Complete Check6-2 **[20pts]**

If you got a perfect score on Check6-2 (the project part), congratulations; this step is already done! Go to the next step.

Otherwise, go back to your Gradescope feedback on Check6-2 and use it to update your Check6-2 code. This is your chance to implement any features you might have missed before, and fix any code that isn't working.

Step 1: Review Provided Code **[0pts]**

Drawing a circuit such that it fits nicely in the window is fairly complicated, so we've provided some code for you to handle the finicky bits. You will write some functions that call the functions we've provided, and you will also write some functions that are called by our provided functions.

You don't need to understand how each of the following functions work, but knowing what they do will be helpful!

- **`getTreeDepth`:** returns the depth of the tree, the max length from root to leaf
- **`getTreeWidth`:** returns the width of the tree, the max number of nodes on the same level
- **`drawInputs`:** draws all the inputs of the circuit. They should all go on the left side of the screen.
- **`drawTree`:** draws a circuit tree within the specified bounding box. It returns the location where the node was drawn, to make drawing wires easier.
- **`drawCircuit`:** draws the entire circuit. It first determines the size of each circuit node by measuring the width/height of the tree. Then it draws the inputs and outputs. Then it recursively draws the circuit tree.

Step 2: Type Boolean Expression in Simulation **[20pts]**

First, we need to update the simulation code to let the user type an expression into the window, then parse that expression into a tree.

To do this, we will need to update three simulation functions: `makeModel`, `keyPressed`, and `makeView`. We will also need to implement one additional function, `runInitialCircuit`.

In `makeModel(data)`, we need to store three pieces of information in `data`: the string expression that the user is typing, the tree that is generated, and an input dictionary. Store these in `data["expression"]`, `data["tree"]`, and `data["inputs"]`. These should all begin as default values- expression should be an empty string, tree should be `None`, and inputs should be an empty list.

In `keyPressed(data, event)`, we need to accept user input from the keyboard. For most keys the user types, we simply want to add the typed key to the end of the expression in data. However, there are two exceptions:

- If the user presses the backspace key and there is at least one character in the expression, we should delete the last character. The keysym name for this character is `"BackSpace"`.
- If the user presses the Enter key, we should parse their expression into a tree (using `parseExpr`) and store the result in `data["tree"]`. Recall that the keysym name for this character is `"Return"`.

In `makeView(data, canvas)`, we should draw the current expression at the bottom of the screen. Draw the text `"Expression: "` followed by the text in `data["expression"]` in the bottom-left corner of the window. You should use `canvas.create_text` to do this. Using an anchor will help place the text correctly!

Finally, in `runInitialCircuit(data)`, set up the input dictionary for the circuit (which is needed to draw the circuit). You can do this by generating an input list with `getLeaves`, then making a dictionary that maps each leaf to the value `False`, just like in `makeTruthTable`. Set `data["inputs"]` to that dictionary. Once the function is working, call it in `keyPressed` in the Return condition, after you set `data["tree"]`.

To test this step, run the simulation and try typing an expression. You should see the expression appear in the bottom-left corner of the window.

Now, just for the fun of it, let's add in a cheat code to let the user generate truth tables for the circuits they create. Again in `keyPressed`, detect if the user has clicked Tab (which has the keysym name `"Tab"`). If they did, run `makeTruthTable` on `data["tree"]` to print out the truth table.

Step 3: Draw the Circuit **[20pts]**

Next, we want to draw the circuit parsed from the expression. Most of this is provided in the starter code, but you will implement two new functions for it: `drawNode` and `drawWire`.

Implement the function `drawNode(canvas, value, x, y, size, lit)` in the starter file. `canvas` is the Tkinter canvas; `value` is the name that should be written in the node (like `X` or `AND`). `x` and `y` are the center coordinate of the node, and `size` is its size. If `lit` is `True`, the node is powered (and should be filled with yellow); otherwise, the node is off (and should be clear, or filled with `None`). Draw a square centered at (`x`, `y`) with `size` size, then draw the `value` variable as text in the center of the square.

Now update `makeView(data, canvas)` so that if `data["tree"]` is not `None`, the function calls `drawCircuit(data)` (a function that is provided in the starter file). This function calls your `drawNode` function.

If you run the simulation now and enter a valid boolean expression, you should see the nodes of the expression appear! Now we just need to add the connections, or wires.

Implement the function `drawWire(canvas, x1, y1, x2, y2, lit)` in the starter file. `canvas` is the Tkinter canvas; `(x1, y1)` and `(x2, y2)` are the two endpoints of the wire. If `lit` is `True`, the wire is powered (and colored yellow); otherwise, the wire is off (and should be black). Draw a line between `(x1, y1)` and `(x2, y2)`.

Now when you run your simulation and parse a valid tree, you should see the whole circuit!

Step 4: Make the Circuit Interactive **[20pts]**

Now we've reached the final step: making the circuit interactive. We want to change the code so that users can click on the inputs to switch them On/Off and see how the whole circuit changes as a result.

As was mentioned above, we'll need to keep track of which parts of the circuit are powered, and which aren't. Update your function `evalTree` to make one change: before you return the result of the evaluation, set `t["powered"]` equal to that result. This will add a key to every level of the dictionary, and will store all the temporary results.

Now, in `runInitialCircuit`, add a call to the end of the function that sets `data["output"]` equal to the result of calling `evalTree` on `data["tree"]` and `data["inputs"]`. This will set the initial `"powered"` values for every level of the tree, and will also store the overall output (which is used in `drawCircuit()`).

This finally leads us to the interesting part: clicking on inputs. You'll want to implement this in `mousePressed(data, event)`. To detect whether the user has clicked on an input, you need to determine if the click location is within the bounds of the input box. The `drawInputs()` function creates a dictionary `data["inputLocations"]` which maps each input variable to a dictionary. That inner dictionary has four keys- `"left"`, `"right"`, `"top"`, and `"bottom"`. These are the bounds of the input's box.

For each input in the input dictionary, check whether the click location in `event` is within the bounds of that input's box. This is true if `event.x` is between the left and right bounds, **and** if `event.y` is between the top and bottom bounds. If this is true, set `data["inputs"]` of that input to the opposite of itself (to flip the switch)

Once you've gone through all the inputs, set `data["output"]` to `evalTree` of `data["tree"]` and `data["inputs"]` again, to update the results.

Now when you run your simulation, you should be able to click on different inputs and see the whole circuit light up in response. Try it out on a few different circuits, and have fun!

You can watch the following video for an example of a working circuit at the end of Hw6: https://www.cs.cmu.edu/~110/hw/hw6_circuit_final_demo.mp4