CMPS 10 Lecture Notes: Lecture 9 (2-2-2016)

PARDIS/SNAP SEGMENT

As always, the video that Pardis takes (available on the course webpage) is probably a more accessible resource for these Snap! Segments than written notes!

Week 5: Advanced Buildings, Abstraction, and Testing
Part 1: Type and Boundary Safety Checks (things like making sure your input is of the acceptable bound, and of the correct type)
Part 2: Accumulator Coding Example
Part 3: Blocks to Turn in
Part 4: Brick Wall Drawing Few Elements
Part 5: Week 5 Homework

Part 1: Type Safety Check Coding Example
Betweeness: man, dog, and cat does not make any sense.

- If you type in cat, dog, tree in the three values, it says false, but it still feels weird that it even let you type those things in the first place.
- You can click on the arrow next to 'input name' when clicking on a variable in a custom block, and you can then choose what type of input you are allowed to have.
  - Number is what we want in this situation. And when we do that, we can now only type numbers.
- another thing you can is give a default number to a block, e.g. give an example as to what could go inside of it.
  - At the bottom, there is a 'default value' but it doesn't seem to work as expected. Paris will look into it.

Type Safety Check Coding Example

- Sum of the two smallest numbers given three numbers.
  - How do we make sure that we only output the two smallest ones.
    - Well, we can find the largest number, put it aside, and then find the other two.
  - We can do this with three if statements.
    - if a > b and a > c then a is the max.
    - if b > a and b > c then b is the max.
    - if c > a and c > b then c is the max.
      - And note: && is another way of saying AND is a mathematical way.
- So we make a new block, sum of the two smallest numbers, and set the three inputs to be of type number.
  - We use an if else
    - In first if condition, we check if a is bigger than b and c, then we report the sum of b + c.
    - Then inside of the else, we put another if else. In THAT if condition we check to see if b is the biggest (and if so report a + c)
      - And so then we know that we are safe in the final else block to simply report a + b.
- Then we showed off what happens if you make the inputs equal to each other and it handles it correctly!

Let's do another version, this time using three if statements instead of using if else.

- So we use the three ifs instead, and it seems to work still!
- And the reason why is because we know that the three ifs actually *are* all encompassing.
  - But what if you put in all three inputs that are exactly the same.

Boundary Safety Check Coding Example

- Accurate Date in 2016? It accepts three inputs, a year, a month, and a day
- You want to check if those three inputs (year month and day) are valid.

    - So we make a new block, is Year Month Date valid, and make three input variables, all of type number.
        * We want our year to only look at 1900 or above.
            · So if year is less than 1900 or greater than 2016 then we are not interested.
            · Similar with month and date
            · (I'm a little concerned... I think we want to be using 'or' instead of 'and' here.)
            · Ah, yes, we ended up changing all of our And Blocks to Or blocks. Perfect!
- So we have our giant if statements to see if the boundaries are ok
    - If they aren't, we go into the if and return false.
    - And if they are then we go into our else statement

Part 2: Let's talk about accumulators!

- Let's say we want to add four coins, and we want to do it in a loop.
- You go through a loop, and you keep on dumping these coins into a bag.
    - If the bag appears on the right side, that is the old value, if you put it on the left side, that means it is the new value.
    - And you just keep on doing that again.
- We need to get comfortable with the left hand side having the new value, and the right hand side having the old value.

One of the programming questions has three different implementations of accumulator.

- The goal is to sum up numbers between x and y.
    - That INCLUDES x and y themselves!
    - So if x and y are 5 and 7, then that means you want to sum 5 + 6 + 7

To use 'for' click on the page and then select import tools. And then the for block will show up under control.

- So here, the for implementation, if given 1 to 3, repeats three times. So it does the last one as well.
- To compute the total amount of times it will loop is y - x + 1.

Looking at the first 'broken' example, we see that we set the sum to i, but that overwrites what sum used to contain.

- We're not adding it into the sum, we're just placing it into the sum!
- In order for sum to carry things around, we need to carry the notion of sum = sum + something.

And note, there is the "change Variable by x" block and that automatically does the sum = sum + i thing.

Part 3: Blocks to Turn In

- The three accumulators, a tic tac toe board, a summing of three numbers, finding the sum of the two smallest numbers, and finding if anything is equal (7 blocks)
- And one large other project (Drawing a brick wall).

Part 4: Block Drawing Few Elements

- One trick, is to make it so that instead of drawing a zillion little lines to make a rectangle, we instead just make our pen thick enough so that a single line is a brick.
- You can go into settings (the little sunburst/gear icon in snap) and check the setting "make flat line ends"
    - And now it is starting to look more like a brick wall!
- After making one row, you want to turn 90 degrees, move 25 steps (for example, plus a little bit for an extra gap)
    - And then you want to make a small block
        * and then you'll need yet a DIFFERENT small block to take care of the case where you finish your second row
            · And then make your wall taller and taller.

Hints on brick wall drawing part:

- First think high level
  - Think about the broad, general things that you will need, and then break them down into smaller and smaller components until you reach the point where you can code it up!
- When you start, skip ahead to the Sum Things Up step and use that as the base project to get those blocks
  - Then you can go back and do everything as a single project.
- Clean Nested Blocking is a *must* for this assignment.
- Safety types and boundary checks are a *must* for this assignment

DIMITRIS SEGMENT

An Adding Machine

Let's say we have two numbers:

57293 + 68461

What does it mean to add these numbers together.

- You can line them up so that the ones place matches the one's place, etc. That's a good first step.
- But you want to make it so that someone can perform the task of addition, armed only with a set of instructions.
- Let's say that we want to be able to add arbitrarily long numbers.
- So the numbers are offered to us on a tape that has a beginning.
  - The assumption will be that our tape will have distinct cells, that take up exactly the same amount of space.
  - And the only numbers allowed in our tape are the numbers 0 through 9, and a *.
- So tapes like that will be given to us, and we need to do the addition that corresponds to them.
  - So, let's say that our tape looks like 57293*68461* (the special symbol * means that the number ended)
  - Note that numbers don't need to be of the same length
  - Note that the bigger number doesn't have to be first.
    - So we've given this to someone from Mars. They don't know they are numbers. They have no notion of meaning.
      - From their point of view, the * could have been a digit, we are just making the contract that * means 'end of number'
  - So we have specified the input: we would like to add these two numbers.
    - And then we can also create an addition table.
    - Even though we can add up infinitely large numbers, our addition table can be finite.
    - So it ranges from 0 to 9 in both the rows and the columns.
    - And we fill it up very nicely. However, something funny happens when we get to 9 + 1
      - Normally 9 + 1 is 10, of course.
      - But when doing "addition" when we get to 9 + 1, we know that we write a 0 underneath it, and we need to carry!
    - to handle the carry, we need ANOTHER table (with the same number of rows and columns) that represents if there is going to be a carry there or not.
      - We can see that in our main, primary sum table, each row is the cyclic shift of the one above it.
      - Note that, in essence, we didn't even need to know addition! All we needed was the first row, and then be told that it's the cyclic shift to fill it in.
      - There is a similar pattern drawn with the carry table (it looks like a big triangle! by the last row everything but first column is a carry).

So, alright, you've given someone from Mars these tables.

- They don't actually need to understand anything yet, they can just follow these rules.

- Now, let's say that you've given this alien a bunch of different tapes that they can work on.
- And now let's say that there is a notion of a head. And the head has a lot of little arms jumping to certain points on the tapes.
  - So the 'reading' arm of the machine is above a certain point on each of those tapes.
- And then, you can think of being given a long list that is divided up into three segments: the STATE segment, the READ segment and the ACTION segment.
- Initial configuration, you have all of the tapes, and the reading arms of the head are all going to be on the far left; i.e. the beginning of the tapes.
- And with the head, you can think of it as having a variety of different STATES.
  - At any given moment, there will be one of five different states, one of five different letters (A, B, C, D, or E).
- And going back to our big list
  - If you are in state BLAH (say, A) and the input of tape 1 is BLAH, and tape 2 is BLAH BLAH and tape 3 is BLAHBLAHBLAH and tape 4 is BLAHBLAHBLAHBLAH etc. THEN you should perform the following action.
    - So to look at this picture, it means to look at the state, and it means looking at each individual head, but it doesn't, crucially, require you to understand where the head is in the context of everything else.

We all agree that this is a completely mechanistic device

- someone can operate this without understanding anything.
- and the actions are: you can CHANGE THE STATE of the machine
- OR you can change the value of what is beneath one of the heads (or not)
- OR you can move the header forward.
  - Our plan is to read from the original input tape (i.e. the two numbers we want to add together) that will be tape 1
  - Then we'll put number one in Tape 2
  - We'll put number two in Tape 3
  - We'll put whether we have a carry or not in Tape 4.
  - And then we'll put the final answer in Tape 5.
- One thing that we'll do is we are going to write our result in the wrong direction as it makes it a little easier to add.
  - So what are the instructions to do it!
    - Hint: we can have more than 5 states if we want!

Coming up with Good State Names

- Let's call A: INITIAL. So whenever we start, we begin in INITIAL State
- Then let's make a new state: SEARCHING FOR END OF INPUT
  - that is, if in initial state, leave tentacles alone, and move to a new state called "search for end of input"
- So what do we want to do for SEARCHING FOR END OF INPUT
  - The instruction can essentially be the same for 0 through 9
  - But we need a special instruction for when it is a *.
    - So what should we tell the machine to do in each of these situations.
    - And remember, for now, we are just trying to find the end of the input.
      - And we need to find TWO stars. Because we are definitely looking for the end of our 'input' (in terms of the adding two numbers, not just looking for the first number)
  - So let's introduce a new state: I HAVE ALREADY READ THE FIRST NUMBER
    - And then, inside of that state, if it sees a star, it enters a new state called "I am at the end of the input"
  - then you move to state "COPY FIRST NUMBER"
    - And you move the head to the left again and again and again
      - And as you do that, you are writing it down in the second tape!

you can right click on the block and search for any block! Very useful!