GV Logo

We've seen how languages are created - now it is time to make one!

We are going to remake a classic language called Logo (there is a version built into Python called `turtle`). Logo was a language created for educational purposes, with a focus on drawing. You had a little turtle on the screen, and you moved her around with commands. You imagined a pen being attached to the turtle's tail, and if the pen was down, the turtle drew as she moved.

The language needs the following commands (an "*" before the name means you need to implement it):

- **penup** - Raise the pen so the turtle can move without drawing.
- **pendown** - Lower the pen so movement creates a line.
- **print** - Print (to the console) a quoted string. For instance, `print "Hello world!"`.
- **save** - Save the current picture to a bitmap file. Needs a pathname; i.e. `save picture.bmp`.
- **color** - Change the current draw color. Requires a red, green, and blue value (0-255). For instance, `color 255 255 0` would set the current color to yellow.
- **clear** - Clears the screen to the current draw color. Paints over anything currently drawn.
- **turn** - Requires an angle. Reorients the turtle by a number of degrees (clockwise positive). Is cumulative; `turn 45` and `turn 55` would end up turning the turtle 100 degrees.
- **move** - Takes a number of pixels, and moves the turtle in the current direction that many pixels. For instance, `move 25`. If the pen is up, no drawing occurs.
- ***goto** - Moves the turtle to a particular coordinate.Draws if the pen is down, otherwise does not.
- ***where** - Prints the current coordinates.

In addition, we need some extended functionality. Modify the program to do the following:

- Print out the answer to an expression. If the user types `34 + 8` for instance, print out `42`.
  - Include at least +, -, *, and /.
- Evaluate and printout extended expressions. For instance, `34 + 4 * 2` should print `42`. Don't worry about order of operations; simply process them in the order they come.
- Add variables. You can limit the number of possible variables and simply create a fixed sized array to use, or implement a dynamically sized symbol table.
  - Add a way to store an expression to a variable.
  - Modify the CFG to allow a variable value in the `move`, `turn`, and `goto` commands.

> ***Note:*** *Don't be fooled. This doesn't seem like a lot of work, but implementing these features can be a significant endeavor if you aren't comfortable with the concepts we've discussed in class. Start early; it will likely involve some trial and error from your end. This assignment* ***is NOT to be used with ChatGPT or other coding assistant.*** *Put in the time and effort. If you can't fully explain why you are trying something, don't bother trying it.*

The original Logo had loops and a few other higher-order concepts. We aren't going to go into how to create those sorts of things for our language, but that doesn't mean it won't still be useful!

So... how do we do this?

You aren't going to be required to create this from scratch. We will be using a library called SDL2 for drawing, and it will operate on its own threads. I have given you the skeleton code for some basic features. Your job will be to read and understand the code, then complete it. Part of this will be writing new lexer and

scanner rules, regenerating the parser and lexer with bison and flex, and implementing C code to implement the functionality.

This is not a trivial task, so you may work with a partner. However, as the goal is to really understand how computers derive meaning from the (arbitrary) collections of symbols we use, ChatGPT or other assistant is not allowed.

Grading

First, you may work with a partner. Make sure both names are on the code you submit.

Secondly, I only want you to submit your `.l` and `.y` files. **DO NOT SUBMIT THE CODE GENERATED BY FLEX AND BISON AND DO NOT UPLOAD A .ZIP.** If you submit only these two files, I can mark them up in Bb and regenerate the proper code.

For the rubric below, you get full credit for no errors, half credit for one or two errors, and no credit for a rubric item with three or more errors.

- Flex file regex all correct and returning the correct tokens.
- Flex file capturing and returning the correct types of data (strings for strings, floats for floats, etc.).
- Bison grammar rules correct.
- Bison actions for tokens all specified correctly.
- Flex generates your code with no errors or warnings.
- Bison generates your code with no errors or warnings.
- Compiler generates your language with no errors or warnings.