1.

My Program implements a PDA simulator. Call the main function to begin running the program where you will be prompted for an input file for the instructions of your desired PDA. If a valid instruction file is given you can then begin entering inputs for the PDA to test. Entering at this stage “.quit” will exit the program while “.machine” will prompt the user for a new PDA instruction file.

2.

```haskell
type State = String
type TransitionFunc = (State, State, Char, Char, String)
data Machine = Machine { input :: String
                       , stack :: String
                       , states :: [State]
                       , start :: State
                       , final :: [State]
                       , transition :: [TransitionFunc]
                       } deriving (Show, Eq)
--Tokens used by parse
data Tokens = LBra | RBra | Comma | NL | Word String | Equals | Arrow
            | InputAlpha | StackAlpha | States | Start | Finals | Transitions
            | Err String | LPar | RPar | CharT Char | WordList [String]
            | TranFunc TransitionFunc | TranList [TransitionFunc] | StackStart
  deriving (Show, Eq)
```

```haskell
main :: IO ()
main = do
  putStr "Please enter filename: "
  str <- getLine
  contents <- readFile str
  let parseT = sr [] (parse contents)
  let machine = buildMachine contents
  if checkMachine machine then repl machine else do
    putStrLn "This is not a valid PDA. Please try a different PDA." >> main

repl :: Machine -> IO ()
repl machine = do
        input <- getLine
        case input of
          ".quit" -> return ()
          ".machine" -> main
          e -> do
            let outcome = runMachine (start machine) input "" machine
            case outcome of
              True -> putStrLn "The string was accepted." >> repl machine
              False -> putStrLn "The string was rejected." >> repl machine
```

My program can be broken down into two main components the Construction of the PDA and the Execution of the PDA.

The construction component contains the following functions:
parse :: String -> [Tokens]
reads the PDA instruction file and converts it into tokens

sr :: [Tokens] -> [Tokens] -> [Tokens]
A shift reduction of the tokens to reduce the tokens into 6 tokens, one for each field of the Machine data type.

buildMachine :: String -> Machine
Runs parse and sr on the instruction file then tries to assemble the machine with the reduced tokens. If it cannot an error will be thrown.

The execution component contains the following functions:
checkMachine :: Machine -> Bool
Checks that a machine is valid. The start and final states must be a subset of the set of states. All states used in the transition functions must be in the set of states. All input and stack characters used in the transition functions must also be in their respective alphabets for the machine.

lookUpTran :: [TransitionFunc] -> State -> Char -> Char -> [TransitionFunc]
Finds valid moves in the current position in a PDA. Searches through the machine's Transition Functions and returns any that match the current state, the first character of the input, and top of the stack.

runMachine :: State -> String -> [Char] -> Machine -> Bool
Acts as the main execution of the PDA. This function takes in the current State, the input string, a stack, and the machine which contains the instructions for the PDA. The function will look up all possible moves from its current state using the lookUpTran function and then recursively call itself taking each move. If any of the recursive calls end on a final state then the function will return true.

Various auxiliary functions:
getTranStates :: Machine -> [State]
Returns all states present in transition functions. Used for checkMachine.

getTranInput :: Machine -> [Char]
Returns all input characters present in the transition functions. Used for checkMachine.

getTranStack::Machine -> [Char]
Returns all stack characters present in the transition functions. Used for checkMachine.

getMoveState :: TransitionFunc -> State
Returns the state that is moved to by a transition function. Used for runMachine.

getMoveStack :: TransitionFunc -> String
Returns the stack character pushed onto a stack by a transition function. Used for runMachine.

3.

No research was required for this project.

4.

During this project, I encountered two big speed bumps. The first occurred with the shift reduction and its need to not only create Char Tokens, Strings Tokens, and [Strings] Tokens but also keep them from being separated when they needed to be. The easiest way that I found to fix this problem for most lines was that each line only needed to be one of these tokens so at the start of each set I would add in an empty token and at every comma it would get added to this empty set. One line that broke this one type rule however was the Transition functions. Transition functions needed its own token which itself would need: a String, Arrow, String, Comma, Char, Comma, Char, Comma, String in that order. The problem here occurs with the String Comma Char which would result in the Char being added to the String. This was fixed by swapping out every occurrence of Arrow String/Char Comma with Arrow String/Char Arrow which would prevent the absorption from occurring. The second speed bump I ran into was that the transition functions needed to be able to have empty stack and input characters. There being no empty [] for Chars in Haskell, I would need to instead use '\0' as a substitute which turned out to be quite an easy fix.

5.

In hindsight, it definitely would have been better to pass off the tokens used for Transition Functions to a second shift reduction to reduce collisions. This in combination with using a record syntax for Transition Functions would have made for easier and cleaner handling of them throughout the project.

6.

In conclusion, my projects implements a parse and lexer to build a PDA which can then be used to test input for said PDA. I gained a better understanding of record syntax, parsers, and the overall process of designing a program in Haskell.