```python
# TODO type checking for built-in funcitons
# TODO check bool operator for control flow statements

# Scoping done using a universal count, which is a unique number for every single scope

from SymbolTable import *
import Parser

table = None
count = 0
function = None
scopeList = [0]
errorList = []
firstPass = True

def analyzeStart(ast):
  # this block for testing purposes
  global table, count, function, scopeList, errorList, firstPass
  table = SymbolLookupTable()
  count = 0
  function = None
  scopeList = [0]
  errorList = []
  firstPass = True

  analyze(ast)
  # TODO uncomment after removing testing code
  # global firstPass, count, function, scopeList
  count = 0
  function = None
  firstPass = False
  scopeList = [0]
  analyze(ast)
  return errorList

def analyze(ast):
   '''Traverse the AST and check for semantic errors.'''

   # potential AST values and their associated analysis functions
   # use astAnalzyers.get() instead of a long chain of else-ifs
   astAnalyzers = {
       "assignment_command": assignmentCommandAnalyzer,
       "comparison": comparisonAnalyzer,
       "declaration_command": declarationCommandAnalyzer,
       "define_command": defineCommandAnalyzer,
       "drive_command": driveCommandAnalyzer,
       "empty": emptyAnalyzer,
       "function_command": functionCommandAnalyzer,
       "if_command": ifCommandAnalyzer,
       "opt_else": optElseAnalyzer,
       "opt_else_if": optElseIfAnalyzer,
       "opt_extra_params": optExtraParamsAnalyzer,
       "opt_param_list": optParamListAnalyzer,
       "plus_expression": plusExpressionAnalyzer,
       "print": printAnalyzer,
       "repeat_if_command": repeatIfAnalyzer,
       "repeat_times_command": repeatTimesAnalyzer,
       "statement_block": statementBlockAnalyzer,
       "statements": statementsAnalyzer,
       "times_expression": timesExpressionAnalyzer,
       "turn_command": turnCommandAnalyzer,
       "word_expression": wordExpressionAnalyzer,
   }

   # Fetch the appropriate analyzer function from astAnalyzers
   # If there is no analyzer for ast.value, then just let the
   # "analyzer" be ast.value
   analyzer = astAnalyzers.get(ast.value, ast.value)


   # If the "anaylzer" is just a string (inherits from basestring)
   if isinstance(analyzer, basestring):
      # this should only be useful for evaluating the type of an expression
      if (ast.type == "WORD"):
         return "word"
      elif (ast.type == "NUMBER"):
         return "number"
      elif (ast.type == "ID"):
         # do existence and scope checking right here
         # return type if passes
         id = ast.value
         idEntry = table.getEntry(SymbolTableEntry(id, None, list(scopeList), function,
None))
         if idEntry == None:
            # ID does not exist or exists but the scoping is wrong
            # this is to be returned to binaryOperatorAnalyzer
            return "ERROR"
         if function == None and idEntry.initialized == False:
            return "ERROR"
         return idEntry.type

   # if the translator is a real function, then invoke it
   else:
       return analyzer(ast)


def statementsAnalyzer(ast):
  if firstPass:
      if ast.children[0].value == "define_command" or ast.children[0].value ==
"statements":
          analyze(ast.children[0])
      if ast.children[1].value == "define_command" or ast.children[1].value ==
"statements":
          analyze(ast.children[1])
  else:
      analyze(ast.children[0])
      analyze(ast.children[1])


def driveCommandAnalyzer(ast):
   # for "plus_expression"
   result = analyze(ast.children[1])
   if result != "number":
      errorList.append("Error in drive command: need to use valid variable or number")


def turnCommandAnalyzer(ast):
   # nothing to do here
   return

def emptyAnalyzer(ast):
  return []

def comparisonAnalyzer(ast):
  result = binaryOperatorAnalyzer(ast)
  if result == "number":
      return
  elif result == "word":
      if ast.children[1].type == "IS" or ast.children[1].type == "IS NOT":
         return
      else:
         errorList.append("Error in comparison: words must be compared using 'is' or 'is not'")
  else:
      errorList.append("Error in comparison: use only words or only numbers; cannot mix both")


def optElseIfAnalyzer(ast):
  # for "expression"
  if ast.children[1].value != "empty":
      analyze(ast.children[1])
   # for "statement_block"
  if ast.children[3].value != "empty":
      analyze(ast.children[3])
   # for "optional_else_if"
  if ast.children[4].value != "empty":
      analyze(ast.children[4])


def optElseAnalyzer(ast):
  # for "statement_block"
  if ast.children[2].value != "empty":
      analyze(ast.children[2])


def ifCommandAnalyzer(ast):
  # for "expression"
  analyze(ast.children[1])
  # for "statement_block"
  analyze(ast.children[3])

  if ast.children[4].value != "empty":
      analyze(ast.children[4])

  if ast.children[5].value != "empty":
      analyze(ast.children[5])


def repeatTimesAnalyzer(ast):
  # for "plus_expression"
  if analyze(ast.children[1]) != "number":
      errorList.append("Error in repeat loop: need to use valid variable or number")
  # for "statement_block"
  analyze(ast.children[4])


def repeatIfAnalyzer(ast):
   # for "expression"
   analyze(ast.children[2])
   # for "statement_block"
   analyze(ast.children[4])


def declarationCommandAnalyzer(ast):
   # Note ast.children[3].type is word
   table.addEntry(SymbolTableEntry(ast.children[0].value, ast.children[3].value, list(scopeList), function, None))


def assignmentCommandAnalyzer(ast):
   # check for the existence of ID - child 1
   # and that it can be accessed in this block
   idNoneBool = False
```

```python
   id = ast.children[1].value
   idEntry = table.getEntry(SymbolTableEntry(id, None, list(scopeList), function, None))
   if idEntry == None:
      idNoneBool = True
      # ID does not exist or exists but the scoping is wrong
      errorList.append("Error1 in assignment: variable does not exist or cannot be used here")
   else:
      idEntry.initialized = True


   # do type checking
   # child 3 is an expression - it needs to be evaluated to a type
   child3Evaluation = analyze(ast.children[3])
   if child3Evaluation == "ERROR":
      # type check in expression failed
      errorList.append("Error2 in assignment: use only words or only numbers; cannot mix both")
   else:
      if (not idNoneBool) and idEntry.type != child3Evaluation:
         # type check failed
         errorList.append("Error3 in assignment: variable and value must have the same type")


def printAnalyzer(ast):
   # for the word or identifier
   if analyze(ast.children[1]) == "ERROR":
      errorList.append("Error in an expression: use only words or only numbers; cannot mix both")
   # check will be done in analyze


def defineCommandAnalyzer(ast):
   global function
   id = ast.children[0].value
   if scopeList[-1] != 0:
      errorList.append("Error in function creation: functions cannot be created in other functions or a nested block")
   function = id
   if firstPass:
      paramList = []
      if ast.children[1].value != "empty":
          paramList = optParamListAnalyzer(ast.children[1])
          scopeList.pop()
      table.addEntry(SymbolTableEntry(id, "function", list(scopeList), None, paramList))
      return
   # for "statement_block"
   analyze(ast.children[2])
   function = None


def optParamListAnalyzer(ast):
   scopeList.append(count+1)
   parameterTypeList = []
   toAdd = SymbolTableEntry(ast.children[1].value, ast.children[3].value, list(scopeList), function, None)
   toAdd.functionParamBool = True
   table.addEntry(toAdd)
   parameterTypeList.append(ast.children[3].value)
   if ast.children[5].value == "opt_extra_params":
       return optExtraParamsAnalyzer(ast.children[5], parameterTypeList)
   else:
       return parameterTypeList

```

```

def optExtraParamsAnalyzer(ast, parameterTypeList):
   toAdd = SymbolTableEntry(ast.children[1].value, ast.children[3].value, list(scopeList), function, None)
   toAdd.functionParamBool = True
   table.addEntry(toAdd)
   parameterTypeList.append(ast.children[3].value)
   if ast.children[5].value == "opt_extra_params":
       return optParametersAnalyzer(ast.children[5], parameterTypeList)
   else:
       return list(parameterTypeList)


def statementBlockAnalyzer(ast):
   global count
   count += 1
   scopeList.append(count)
   analyze(ast.children[0])
   scopeList.pop()


def functionCommandAnalyzer(ast):
  # check existence of function ID
  idEntry = table.getEntry(SymbolTableEntry(ast.children[0].value, "function", list(scopeList), function, None))
  if idEntry == None:
      errorList.append("Error in attempt to use function: function does not exist")
      return
  elif idEntry.type == "function":
            parameterTypeList = list(idEntry.functionParameterTypes)
  else:
            errorList.append("Error in attempt to use function: function does not exist")
            return
  if len(ast.children) == 2:
      optParametersAnalyzer(ast.children[1], parameterTypeList)


# this is for user-defined function parameters
def optParametersAnalyzer(ast, parameterTypeList):
  if len(ast.children) == 1:
      if len(parameterTypeList) != 1 or analyze(ast.children[0]) != parameterTypeList[0]:
          # Type checking error
          errorList.append("Error in attempt to use function: wrong type of parameter used")
  else:
      # More parameters left
      if analyze(ast.children[1]) != parameterTypeList.pop():
          # Type checking error
          errorList.append("Error1 in attempt to use function: wrong type of parameter used")
      else:
          optParametersAnalyzer(ast.children[0], parameterTypeList)


def binaryOperatorAnalyzer(ast):
   result1 = analyze(ast.children[0])
   result3 = analyze(ast.children[2])

   if result1 == "ERROR" or result3 == "ERROR":
      return "ERROR"

   elif result1 == result3:
      return result1

   elif ast.children[1].type == "CONCAT":
```

```python
      return "word"

   else:
      return "ERROR"


def plusExpressionAnalyzer(ast):
   return binaryOperatorAnalyzer(ast)


def timesExpressionAnalyzer(ast):
   return binaryOperatorAnalyzer(ast)

def wordExpressionAnalyzer(ast):
   return binaryOperatorAnalyzer(ast)

if __name__ == "__main__":
    inputString = ''
    while True:

        inputString = raw_input('enter expression > ')

        if inputString == 'exit':
            break

        else:
            # first parse the string
            ast = Parser.parseString(inputString)

            ast.printTree()
            print

            # then check for errors
            if len(ast.errors) > 0:
                print ast.errors
                break

            analyzeStart(ast)

            print errorList
```