

```

1. # TODO type checking for built-in functions
2. # TODO check bool operator for control flow statements
3.
4. # Scoping done using a universal count, which is a unique number for every single scope
5.
6. from SymbolTable import *
7. import Parser
8.
9. table = None
10. count = 0
11. function = None
12. scopeList = [0]
13. errorList = []
14. firstPass = True
15.
16. def analyzeStart(ast):
17.     # this block for testing purposes
18.     global table, count, function, scopeList, errorList, firstPass
19.     table = SymbolLookupTable()
20.     count = 0
21.     function = None
22.     scopeList = [0]
23.     errorList = []
24.     firstPass = True
25.
26.     analyze(ast)
27.     # TODO uncomment after removing testing code
28.     # global firstPass, count, function, scopeList
29.     count = 0
30.     function = None
31.     firstPass = False
32.     scopeList = [0]
33.     analyze(ast)
34.     return errorList
35.
36. def analyze(ast):
37.     '''Traverse the AST and check for semantic errors.'''
38.
39.     # potential AST values and their associated analysis functions
40.     # use astAnalyzers.get() instead of a long chain of else-ifs
41.     astAnalyzers = {
42.         "assignment_command": assignmentCommandAnalyzer,
43.         "comparison": comparisonAnalyzer,
44.         "declaration_command": declarationCommandAnalyzer,
45.         "define_command": defineCommandAnalyzer,
46.         "drive_command": driveCommandAnalyzer,
47.         "empty": emptyAnalyzer,
48.         "function_command": functionCommandAnalyzer,
49.         "if_command": ifCommandAnalyzer,
50.         "opt_else": optElseAnalyzer,
51.         "opt_else_if": optElseIfAnalyzer,
52.         "opt_extra_params": optExtraParamsAnalyzer,
53.         "opt_param_list": optParamListAnalyzer,
54.         "plus_expression": plusExpressionAnalyzer,
55.         "print": printAnalyzer,
56.         "repeat_if_command": repeatIfAnalyzer,
57.         "repeat_times_command": repeatTimesAnalyzer,
58.         "statement_block": statementBlockAnalyzer,
59.         "statements": statementsAnalyzer,
60.         "times_expression": timesExpressionAnalyzer,
61.         "turn_command": turnCommandAnalyzer,
62.         "word_expression": wordExpressionAnalyzer,
63.     }
64.

```

```
65. # Fetch the appropriate analyzer function from astAnalyzers
66. # If there is no analyzer for ast.value, then just let the
67. # "analyzer" be ast.value
68. analyzer = astAnalyzers.get(ast.value, ast.value)
69.
70.
71. # If the "analyzer" is just a string (inherits from basestring)
72. if isinstance(analyzer, basestring):
73.     # this should only be useful for evaluating the type of an expression
74.     if (ast.type == "WORD"):
75.         return "word"
76.     elif (ast.type == "NUMBER"):
77.         return "number"
78.     elif (ast.type == "ID"):
79.         # do existence and scope checking right here
80.         # return type if passes
81.         id = ast.value
82.         idEntry = table.getEntry(SymbolTableEntry(id, None, list(scopeList), function,
None))
83.         if idEntry == None:
84.             # ID does not exist or exists but the scoping is wrong
85.             # this is to be returned to binaryOperatorAnalyzer
86.             return "ERROR"
87.         if function == None and idEntry.initialized == False:
88.             return "ERROR"
89.         return idEntry.type
90.
91. # if the translator is a real function, then invoke it
92. else:
93.     return analyzer(ast)
94.
95.
96. def statementsAnalyzer(ast):
97.     if firstPass:
98.         if ast.children[0].value == "define_command" or ast.children[0].value ==
"statements":
99.             analyze(ast.children[0])
100.         if ast.children[1].value == "define_command" or ast.children[1].value ==
"statements":
101.             analyze(ast.children[1])
102.     else:
103.         analyze(ast.children[0])
104.         analyze(ast.children[1])
105.
106.
107. def driveCommandAnalyzer(ast):
108.     # for "plus_expression"
109.     result = analyze(ast.children[1])
110.     if result != "number":
111.         errorList.append("Error in drive command: need to use valid variable or number")
112.
113.
114. def turnCommandAnalyzer(ast):
115.     # nothing to do here
116.     return
117.
118. def emptyAnalyzer(ast):
119.     return []
120.
121. def comparisonAnalyzer(ast):
122.     result = binaryOperatorAnalyzer(ast)
123.     if result == "number":
124.         return
125.     elif result == "word":
126.         if ast.children[1].type == "IS" or ast.children[1].type == "IS NOT":
```

```
127.         return
128.     else:
129.         errorList.append("Error in comparison: words must be compared using 'is' or 'is
not'")
130.     else:
131.         errorList.append("Error in comparison: use only words or only numbers; cannot mix
both")
132.
133.
134. def optElseIfAnalyzer(ast):
135.     # for "expression"
136.     if ast.children[1].value != "empty":
137.         analyze(ast.children[1])
138.     # for "statement_block"
139.     if ast.children[3].value != "empty":
140.         analyze(ast.children[3])
141.     # for "optional_else_if"
142.     if ast.children[4].value != "empty":
143.         analyze(ast.children[4])
144.
145.
146. def optElseAnalyzer(ast):
147.     # for "statement_block"
148.     if ast.children[2].value != "empty":
149.         analyze(ast.children[2])
150.
151.
152. def ifCommandAnalyzer(ast):
153.     # for "expression"
154.     analyze(ast.children[1])
155.     # for "statement_block"
156.     analyze(ast.children[3])
157.
158.     if ast.children[4].value != "empty":
159.         analyze(ast.children[4])
160.
161.     if ast.children[5].value != "empty":
162.         analyze(ast.children[5])
163.
164.
165. def repeatTimesAnalyzer(ast):
166.     # for "plus_expression"
167.     if analyze(ast.children[1]) != "number":
168.         errorList.append("Error in repeat loop: need to use valid variable or number")
169.     # for "statement_block"
170.     analyze(ast.children[4])
171.
172.
173. def repeatIfAnalyzer(ast):
174.     # for "expression"
175.     analyze(ast.children[2])
176.     # for "statement_block"
177.     analyze(ast.children[4])
178.
179.
180. def declarationCommandAnalyzer(ast):
181.     # Note ast.children[3].type is word
182.     table.addEntry(SymbolTableEntry(ast.children[0].value, ast.children[3].value,
list(scopeList), function, None))
183.
184.
185. def assignmentCommandAnalyzer(ast):
186.     # check for the existence of ID - child 1
187.     # and that it can be accessed in this block
188.     idNoneBool = False
```

```

189.     id = ast.children[1].value
190.     idEntry = table.getEntry(SymbolTableEntry(id, None, list(scopeList), function, None))
191.     if idEntry == None:
192.         idNoneBool = True
193.         # ID does not exist or exists but the scoping is wrong
194.         errorList.append("Error1 in assignment: variable does not exist or cannot be used
here")
195.     else:
196.         idEntry.initialized = True
197.
198.
199.     # do type checking
200.     # child 3 is an expression - it needs to be evaluated to a type
201.     child3Evaluation = analyze(ast.children[3])
202.     if child3Evaluation == "ERROR":
203.         # type check in expression failed
204.         errorList.append("Error2 in assignment: use only words or only numbers; cannot mix
both")
205.     else:
206.         if (not idNoneBool) and idEntry.type != child3Evaluation:
207.             # type check failed
208.             errorList.append("Error3 in assignment: variable and value must have the same
type")
209.
210.
211. def printAnalyzer(ast):
212.     # for the word or identifier
213.     if analyze(ast.children[1]) == "ERROR":
214.         errorList.append("Error in an expression: use only words or only numbers; cannot mix
both")
215.     # check will be done in analyze
216.
217.
218. def defineCommandAnalyzer(ast):
219.     global function
220.     id = ast.children[0].value
221.     if scopeList[-1] != 0:
222.         errorList.append("Error in function creation: functions cannot be created in other
functions or a nested block")
223.     function = id
224.     if firstPass:
225.         paramList = []
226.         if ast.children[1].value != "empty":
227.             paramList = optParamListAnalyzer(ast.children[1])
228.             scopeList.pop()
229.             table.addEntry(SymbolTableEntry(id, "function", list(scopeList), None, paramList))
230.         return
231.     # for "statement_block"
232.     analyze(ast.children[2])
233.     function = None
234.
235.
236. def optParamListAnalyzer(ast):
237.     scopeList.append(count+1)
238.     parameterTypeList = []
239.     toAdd = SymbolTableEntry(ast.children[1].value, ast.children[3].value, list(scopeList),
function, None)
240.     toAdd.functionParamBool = True
241.     table.addEntry(toAdd)
242.     parameterTypeList.append(ast.children[3].value)
243.     if ast.children[5].value == "opt_extra_params":
244.         return optExtraParamsAnalyzer(ast.children[5], parameterTypeList)
245.     else:
246.         return parameterTypeList
247.

```

```

248.
249. def optExtraParamsAnalyzer(ast, parameterTypeList):
250.     toAdd = SymbolTableEntry(ast.children[1].value, ast.children[3].value, list(scopeList),
function, None)
251.     toAdd.functionParamBool = True
252.     table.addEntry(toAdd)
253.     parameterTypeList.append(ast.children[3].value)
254.     if ast.children[5].value == "opt_extra_params":
255.         return optParametersAnalyzer(ast.children[5], parameterTypeList)
256.     else:
257.         return list(parameterTypeList)
258.
259.
260. def statementBlockAnalyzer(ast):
261.     global count
262.     count += 1
263.     scopeList.append(count)
264.     analyze(ast.children[0])
265.     scopeList.pop()
266.
267.
268. def functionCommandAnalyzer(ast):
269.     # check existence of function ID
270.     idEntry = table.getEntry(SymbolTableEntry(ast.children[0].value, "function",
list(scopeList), function, None))
271.     if idEntry == None:
272.         errorList.append("Error in attempt to use function: function does not exist")
273.         return
274.     elif idEntry.type == "function":
275.         parameterTypeList = list(idEntry.functionParameterTypes)
276.     else:
277.         errorList.append("Error in attempt to use function: function does not exist")
278.         return
279.     if len(ast.children) == 2:
280.         optParametersAnalyzer(ast.children[1], parameterTypeList)
281.
282.
283. # this is for user-defined function parameters
284. def optParametersAnalyzer(ast, parameterTypeList):
285.     if len(ast.children) == 1:
286.         if len(parameterTypeList) != 1 or analyze(ast.children[0]) != parameterTypeList[0]:
287.             # Type checking error
288.             errorList.append("Error in attempt to use function: wrong type of parameter
used")
289.         else:
290.             # More parameters left
291.             if analyze(ast.children[1]) != parameterTypeList.pop():
292.                 # Type checking error
293.                 errorList.append("Error1 in attempt to use function: wrong type of parameter
used")
294.             else:
295.                 optParametersAnalyzer(ast.children[0], parameterTypeList)
296.
297.
298. def binaryOperatorAnalyzer(ast):
299.     result1 = analyze(ast.children[0])
300.     result3 = analyze(ast.children[2])
301.
302.     if result1 == "ERROR" or result3 == "ERROR":
303.         return "ERROR"
304.
305.     elif result1 == result3:
306.         return result1
307.
308.     elif ast.children[1].type == "CONCAT":

```

```
309.         return "word"
310.
311.     else:
312.         return "ERROR"
313.
314.
315. def plusExpressionAnalyzer(ast):
316.     return binaryOperatorAnalyzer(ast)
317.
318.
319. def timesExpressionAnalyzer(ast):
320.     return binaryOperatorAnalyzer(ast)
321.
322. def wordExpressionAnalyzer(ast):
323.     return binaryOperatorAnalyzer(ast)
324.
325. if __name__ == "__main__":
326.     inputString = ''
327.     while True:
328.
329.         inputString = raw_input('enter expression > ')
330.
331.         if inputString == 'exit':
332.             break
333.
334.         else:
335.             # first parse the string
336.             ast = Parser.parseString(inputString)
337.
338.             ast.printTree()
339.             print
340.
341.             # then check for errors
342.             if len(ast.errors) > 0:
343.                 print ast.errors
344.                 break
345.
346.             analyzeStart(ast)
347.
348.             print errorList
```