```
import ply.lex as lex
import ply.yacc as yacc
from Tree import *

reserved = {
    'drive': 'DRIVE',
    'forward': 'FORWARD',
    'forwards': 'FORWARD',
    'backward': 'BACKWARD',
    'backwards': 'BACKWARD',
    'number': 'NUMBER_TYPE',
    'word': 'WORD_TYPE',
    'step': 'STEP',
    'steps': 'STEP',
    'turn': 'TURN',
    'left': 'LEFT',
    'right': 'RIGHT',
    'canDrive': 'CAN_DRIVE',
    'getCarPosition': 'GET_CAR_POSITION',
    'define': 'DEFINE',
    'using': 'USING',
    'and': 'AND',
    'print': 'PRINT',
    'elseIf': 'ELSE_IF',
    'if': 'IF',
    'else': 'ELSE',
    'repeat': 'REPEAT',
    'times': 'TIMES',
    'a': 'A',
    'is': 'IS',
    'not': 'NOT',
    'set': 'SET',
    'to': 'TO',
}


tokens = [
    "NUMBER",
    "WORD",
    "ID",
    "GT",
    "LT",
    "GEQ",
    "LEQ",
    "CONCAT",
    "NEWLINE",
    "SINGLE_LINE_COMMENT",
] + list(set(reserved.values()))

literals = "{}()+-*/"

#t_NUMBER = r'[0-9]+'
#t_WORD = r'".*?"'
t_GT = r'>'
t_LT = r'<'
t_GEQ = r'>='
t_LEQ = r'<='
t_CONCAT = r'\+\+'
t_SINGLE_LINE_COMMENT = r':\).*$'
t_ignore = ' \t'


def t_ID(t):
    r'[A-Za-z][A-Za-z0-9]*'
```

```
    t.type = reserved.get(t.value, 'ID')
    t.value = (t.value, t.type, t.lexer.lineno)
    return t


def t_NUMBER(t):
    r'[0-9]+'
    t.value = (t.value, t.type, t.lexer.lineno)
    return t


def t_WORD(t):
    r'".*?"'
    t.value = (t.value, t.type, t.lexer.lineno)
    return t


def t_NEWLINE(t):
    r'\n|;|:-\((.|\n)*?:-\)'
    # \n is for actual newlines
    # ; is for debugging use
    # the next expression is for multiline comments. it is an adaptation of
    # hw1, problem 2.
    # the last expression :\).* matches single-line comments
    t.lexer.lineno += 1
    return t


def t_error(t):
    print "Illegal character '%s' at line '%s'" % (t.value[0], t.lexer.lineno)
    t.lexer.skip(1)
    t.value = (t.value, "ERROR", t.lexer.lineno)
    return t



def p_error(p):
    if p is None:
        raise SyntaxError("Reached end of file unexpectedly!")
    elif p.value[0] is None:
        print "Lexing Error with character ", p.value[1]
        p.value = p.value[1]
    else:
        print "Syntax error at token ", p.type


def makeParseTreeNode(p, value):
    '''Returns a Tree object containing
        as children p[1:] and a value of value'''
    toReturn = Tree()
    for element in p[1:]:
        if type(element) == type(toReturn):
            toReturn.children.append(element)
            toReturn.errors += element.errors
        else:
            # the element is not a tree. wrap it in a tree
            newElement = Tree()
            if isinstance(element, tuple):
                newElement.value = element[0]
                newElement.type = element[1]
            else:
                newElement.value = element
            toReturn.children.append(newElement)

    if isinstance(value, tuple):
```

```
        toReturn.value = value[0]
        toReturn.type = value[1]
    else:
        toReturn.value = value
    if value == "error":
        errorMessage = str(p[1][2]) + ": " + p[1][0]
        toReturn.errors.append(errorMessage)

    return toReturn


def p_statements(p):
    '''statements : statements statement'''
    p[0] = makeParseTreeNode(p, "statements")


def p_error_statement(p):
    '''statement : error NEWLINE'''
    if not isinstance(p[1], tuple):
        p[1] = p[1].value
    p[0] = makeParseTreeNode(p, "error")


def p_statements_empty(p):
    '''statements : empty'''
    p[0] = p[1]


def p_statement_block(p):
    """statement_block : '{' statements '}' newline_opt_comment"""
    p[0] = makeParseTreeNode([p[0], p[2]], "statement_block")


def p_empty(p):
    '''empty :'''
    p[0] = Tree()
    p[0].value = "empty"


def p_newline_opt_comment(p):
    '''newline_opt_comment : opt_comment NEWLINE'''
    p[0] = p[2]


def p_opt_comment(p):
    '''opt_comment : SINGLE_LINE_COMMENT
       | empty'''
    p[0] = p[1]


def p_statement_simple_compound(p):
    '''statement : simple_statement
                 | compound_statement'''
    p[0] = p[1]


def p_simple_statement_command(p):
    '''simple_statement : statement_contents newline_opt_comment'''
    p[0] = p[1]


def p_statement_newline(p):
    '''simple_statement : newline_opt_comment'''
    p[0] = Tree()
    p[0].value = "empty"
```

```
195.
196.
197. def p_statement_contents_drive(p):
198.     '''statement_contents : drive_command'''
199.     p[0] = p[1]
200.
201.
202. def p_statement_contents_turn(p):
203.     '''statement_contents : turn_command'''
204.     p[0] = p[1]
205.
206.
207. def p_compound_statement_define(p):
208.     '''compound_statement : define_command'''
209.     p[0] = p[1]
210.
211.
212. def p_compound_statement_repeat_if(p):
213.     '''compound_statement : repeat_if_command'''
214.     p[0] = p[1]
215.
216.
217. def p_compound_statement_repeat_times(p):
218.     '''compound_statement : repeat_times_command'''
219.     p[0] = p[1]
220.
221.
222. def p_compound_statement_if(p):
223.     '''compound_statement : if_command'''
224.     p[0] = p[1]
225.
226.
227. def p_statement_contents_print(p):
228.     '''statement_contents : print_command'''
229.     p[0] = p[1]
230.
231.
232. def p_statement_contents_assignment(p):
233.     '''statement_contents : assignment_command'''
234.     p[0] = p[1]
235.
236.
237. def p_statement_contents_declaration(p):
238.     '''statement_contents : declaration_command'''
239.     p[0] = p[1]
240.
241.
242. def p_statement_contents_function(p):
243.     '''statement_contents : function_command'''
244.     p[0] = p[1]
245.
246.
247. def p_expression_can_drive(p):
248.     '''expression : can_drive_expression'''
249.     p[0] = p[1]
250.
251.
252. def p_expression_comparison(p):
253.     '''expression : comparison'''
254.     p[0] = p[1]
255.
256.
257. def p_can_drive(p):
258.     '''can_drive_expression : CAN_DRIVE drive_direction primary_expression opt_steps'''
259.     p[0] = makeParseTreeNode([p[0], p[2], p[3]], "can_drive_expression")
```

```
260.
261.
262. def p_comparison_with_operator(p):
263.     '''comparison : plus_expression comparison_operator plus_expression'''
264.     p[0] = makeParseTreeNode(p, "comparison")
265.
266.
267. def p_comparison_plus(p):
268.     '''comparison : plus_expression'''
269.     p[0] = p[1]
270.
271.
272. def p_comparison_operator(p):
273.     '''comparison_operator : IS
274.           | IS NOT
275.           | GT
276.           | LT
277.           | GEQ
278.           | LEQ'''
279.     if len(p) == 3:  # i.e. token is IS NOT
280.         p[0] = (p[1][0] + " " + p[2][0], p[1][1] + " " + p[2][1])
281.     else:  # any other token
282.         p[0] = p[1]
283.
284.
285. def p_plus_expression_plus_minus(p):
286.     '''plus_expression : plus_expression '+' times_expression
287.         | plus_expression '-' times_expression'''
288.     p[0] = makeParseTreeNode(p, "plus_expression")
289.
290.
291. def p_plus_expression_times_expression(p):
292.     '''plus_expression : times_expression'''
293.     p[0] = p[1]
294.
295.
296. def p_times_expression_times_divide(p):
297.     '''times_expression : times_expression '*' word_expression
298.       | times_expression '/' word_expression'''
299.     p[0] = makeParseTreeNode(p, "times_expression")
300.
301.
302. def p_times_expression_word_expression(p):
303.     '''times_expression : word_expression'''
304.     p[0] = p[1]
305.
306.
307. def p_word_expression_concat(p):
308.     '''word_expression : word_expression CONCAT primary_expression'''
309.     p[0] = makeParseTreeNode(p, "word_expression")
310.
311.
312. def p_word_expression_primary_expression(p):
313.     '''word_expression : primary_expression'''
314.     p[0] = p[1]
315.
316.
317. def p_primary_expression_parens(p):
318.     """primary_expression : '(' expression ')'"""
319.     p[0] = p[2]
320.
321.
322. def p_primary_expression_token(p):
323.     '''primary_expression : NUMBER
324.       | WORD
```

```
      | GET_CAR_POSITION
      | ID'''
    p[0] = p[1]


def p_function_command(p):
    '''function_command : ID opt_parameters'''
    if p[2].value == "empty":
        p[0] = makeParseTreeNode([p[0], p[1]], "function_command")
    else:
        p[0] = makeParseTreeNode(p, "function_command")


def p_opt_parameters(p):
    '''opt_parameters : opt_parameters primary_expression'''
    if p[1].value == "empty":
        p[0] = makeParseTreeNode([p[0], p[2]], "opt_parameters")
    else:
        p[0] = makeParseTreeNode(p, "opt_parameters")


def p_opt_parameters_empty(p):
    '''opt_parameters : empty'''
    p[0] = p[1]


def p_drive_command(p):
    '''drive_command : DRIVE drive_direction plus_expression opt_steps'''
    p[0] = makeParseTreeNode([p[0], p[2], p[3]], "drive_command")


def p_drive_direction(p):
    '''drive_direction : FORWARD
      | BACKWARD'''
    p[0] = p[1]


def p_opt_steps(p):
    '''opt_steps : STEP
      | empty'''
    p[0] = p[1]


def p_turn_command(p):
    '''turn_command : TURN turn_direction'''
    p[0] = makeParseTreeNode(p, "turn_command")


def p_turn_direction(p):
    '''turn_direction : LEFT
      | RIGHT'''
    p[0] = p[1]


def p_define_command(p):
    """define_command : DEFINE ID opt_param_list \
   newline_opt_comment statement_block"""
    p[0] = makeParseTreeNode([p[0], p[2], p[3], p[5]], "define_command")


def p_opt_param_list(p):
    '''opt_param_list : USING ID '(' type_enum ')' opt_extra_params'''
    p[0] = makeParseTreeNode(p, "opt_param_list")
```

```
390. def p_opt_param_list_empty(p):
391.     '''opt_param_list : empty'''
392.     p[0] = p[1]
393.
394.
395. def p_opt_extra_params(p):
396.     '''opt_extra_params : AND ID '(' type_enum ')' opt_extra_params'''
397.     p[0] = makeParseTreeNode(p, "opt_extra_params")
398.
399.
400. def p_opt_extra_params_empty(p):
401.     '''opt_extra_params : empty'''
402.     p[0] = p[1]
403.
404.
405. def p_type_enum(p):
406.     '''type_enum : WORD_TYPE
407.       | NUMBER_TYPE'''
408.     p[0] = p[1]
409.
410.
411. def p_repeat_if_command(p):
412.     """repeat_if_command : REPEAT IF expression newline_opt_comment \
413.    statement_block"""
414.     p[0] = makeParseTreeNode(p, "repeat_if_command")
415.
416.
417. def p_repeat_times_command(p):
418.     """repeat_times_command : REPEAT plus_expression \
419.    TIMES newline_opt_comment statement_block"""
420.     p[0] = makeParseTreeNode(p, "repeat_times_command")
421.
422.
423. def p_if_command(p):
424.     """if_command : IF expression newline_opt_comment statement_block \
425.    opt_else_if opt_else"""
426.     p[0] = makeParseTreeNode(p, "if_command")
427.
428.
429. def p_opt_else_if(p):
430.     """opt_else_if : ELSE_IF expression newline_opt_comment \
431.    statement_block opt_else_if
432.       | empty"""
433.
434.     if len(p) == 2:
435.         p[0] = p[1]
436.     else:
437.         p[0] = makeParseTreeNode(p, "opt_else_if")
438.
439.
440. def p_opt_else(p):
441.     """opt_else : ELSE newline_opt_comment statement_block
442.       | empty"""
443.
444.     if len(p) == 2:
445.         p[0] = p[1]
446.     else:
447.         p[0] = makeParseTreeNode(p, "opt_else")
448.
449.
450. def p_print_command(p):
451.     """print_command : PRINT word_expression"""
452.     p[0] = makeParseTreeNode(p, "print")
453.
454.
```

```
455. def p_declaration_command(p):
456.     """declaration_command : ID IS A type_enum"""
457.     p[0] = makeParseTreeNode(p, "declaration_command")
458. 
459. 
460. def p_assignment_command(p):
461.     """assignment_command : SET ID TO expression"""
462.     p[0] = makeParseTreeNode(p, "assignment_command")
463. 
464. 
465. 
466. def parseString(stringToParse):
467.     '''Returns the parse tree for the given string'''
468.     lexer = lex.lex()
469.     parser = yacc.yacc()
470.     return parser.parse(stringToParse)
471. 
472. if __name__ == "__main__":
473.     lexer = lex.lex()
474.     parser = yacc.yacc()
475.     inputString = ''
476.     while True:
477. 
478.         inputString = raw_input('enter expression > ')
479. 
480.         if inputString == 'exit':
481.             break
482. 
483.         else:
484.             try:
485.                 result = parser.parse(inputString)
486.             except SyntaxError as e:
487.                 print "Error: ", e
488.             else:
489.                 result.printTree()
490.                 print
491.                 print "errors: ", result.errors
```