Oregon State University

CS 352 - TRANSLATORS

Winter 2015

Milestone 3: IBTL Parser

Recursive Descent Parser

Author:
Drake Bridgewater

Professor:
Dr. Jennifer Parham-Mocello

DUE 01/16/15 (11:59pm)
February 16, 2015

Contents

1 Source Code Descriptions

The way I approached this problem was one paper with drawing out what how I would perform each of of the operations for a given string. With a few iterations I was able to create the parser that logically followed the grammar, but some modification we needed to account for the left recursion and to factor out the repeated tokens.

1.1 node.py

Since a tree is just a single node with child nodes I created a node that would allow printing in a familiar format for easy readability

1.2 defines.py

I decided to place all the global variable into a file for easy manipulation. This fill contains the token ID and also defines what a token is.

1.3 myparser.py

The bulk of this project was to develop a parser that will spit out a list of tokens in a fashion that would allow seeing scope. Dr. Jennifer Parham-Mocello recommended that we implement it as a tree therefore the node I created. Every time I saw a object in the grammar including 's', 'expr', 'oper', etc. I would create a token and depending on how it is related to its parent it would be added as a child or as a leaf node along side. Once I had this idea I need to come up with a way of documenting my trails to the node for debugging purposes therefore I added a need node each time a function was called and when a function was called within it would be added as a child.

1.4 lexer.py

The lexer of this assignment was to recognize the chars one at a time and take the one with the longest prefix. This would allow gathering o

2 Report

The purpose of this mile stone was to ensure that we can store, retrieve and redistribute in such a way that the grammar is followed. This ensures that we understand the code that we are writing and verifies we understood what we needed to do. Like many people I did have to do some refactoring to the previous milestone to ensure it would work with this one (which took the first week). **To solve** this problem I looked had to ensure I understood what the input was (milestone 1) and understand what I needed the out put to become. Since I had already developed a tree I had to make sure that I understood what I was needing to do in the middle. This part was just a bunch of if statements but they were intertwined making some operations quite confusing leading me to draw it out on paper multiple time. **To test** this I started with one of the smallest accepted statements '( )' and then moved on to have a test case for each of the lines in the grammar. Overall I learned python more, as I know understand how to create structure like elements and I have successfully implemented a recursion. This ensured that we knew our recursive algorithms and verified we new how to work with a medium sized project.

3 Source Code

3.1 main.py

```
#!/usr/bin/python
__author__ = 'Drake'
import sys
from defines import *
from myparser import *


usage = """
Usage:
    main.py [option] [files]
"""


def prepare_files(argv):
    for arg in argv:
        if arg[0] == '-':
            # collect user options
            options.append(arg)
        elif arg != argv[0]:
            # collect files
            files.append(arg)


def read_file(input_file):
    content = ""
    f = open(input_file)

    lines_raw = f.readlines()
    for i in range(0, len(lines_raw)):
        content += lines_raw[i]

    return content


def print_verbose(selected_file, content):
    print('\n', "input: parsing " + str(selected_file))
    print("-" * 40)
    print(content)
    print('\n', "output: ")
    print("-" * 40)


def main():
    if len(sys.argv) < 2:
        print(usage)
        exit()

    global options
    global files
    prepare_files(sys.argv)

```

```
    filename = sys.argv[1]
    for file in files:
        parser = MyParser(filename)
        parser.control()


if __name__ == '__main__':
    main()
```

main.py

3.2 node.py

```
__author__ = 'Drake'


class Node(object):
    def __init__(self, data):
        if hasattr(data, "value"):
            print("New Node: " + str(data.value))
        else:
            print("NN Str: " + str(data))
        self.data = data
        self.children = []
        self.depth = 0

    def add_child(self, obj):
        if obj is None:
            return obj
        self.children.append(obj)
        return True

    # need to set depth recursively
    def set_depth(self, t):
        if t is not None or t != str:
            if len(t.children) > 0:
                for i in t.children:
                    if i is not None:
                        i.depth = t.depth + 1
                        self.set_depth(i)
        self.set_depth()

    def get_child_at(self, index):
        return self.children[index]

    def get_first_child_at_parent(self, obj):
        if len(obj.children) > 0:
            return obj.children[0]
        else:
            return self.children[0]

    def get_first_child_at_parent_level(self, obj, level):
        if level == 0:
            return self.children[0]
        else:
            if level >= 1:
```

```
                if len(obj.children) > 0:
                    return obj.children[0]
                else:
                    return self.children[0]
            else:
                return self.children[0]

    @staticmethod
    def get_parent_depth(obj):
        return obj.depth

    def print_tree(self):
        print("-" * 40 + "\n\t print tree called")
        # print(self.data)
        self.print_tree_helper(self)

    def print_tree_helper(self, node, indent=0):
        indent += 1
        for child in node.children:
            # if child.get_child_count() > 0:
            # if child.data is not None:
            if isinstance(child, int):

                print("\t" * indent + str(child))
            elif isinstance(child, str):
                print("\t" * indent + str(child))
            elif hasattr(child, "data"):
                if hasattr(child.data, "value"):
                    print("\t" * indent + "[line: " + str(child.data.line) +
                          ", ID: " + child.data.type +
                          ", Value: " + str(child.data.value) + "]")
                else:
                    print("\t" * indent + str(child.data))
                self.print_tree_helper(child, indent)
            elif hasattr(child, "value"):
                print("\t" * indent + "[line: " + str(child.line) +
                      ", ID: " + child.type +
                      ", Value: " + str(child.value) + "]")
            else:
                print("Error in print_tree_helper")
                print(child)
                return
                # else:
                # print("Failed")

    def print_postordered_tree(self):
        print("-" * 80 + "\n\t print post ordered tree called")
        print(self.root.get_value())
        self.post_order_tree_print(self.root)

    def post_order_tree_print(self, node):
        for child in node.children:
            self.post_order_tree_print(child)
            print("[line: " + str(child.data.line) + ", ID: " + child.data.
        type + ", Value: " + str(
```

```
            child.data.value) + "]")
```
node.py

3.3 defines.py

```
__author__ = 'Drake'

files = []
options = []

OPER_EQ = '='
OPER_ASSIGN = ':='
OPER_ADD = '+'
OPER_SUB = '-'
OPER_DIV = '/'
OPER_MULT = '*'
OPER_LT = '<'
OPER_GT = '>'
OPER_LE = '<='
OPER_GE = '>='
OPER_NE = '!='
OPER_NOT = '!'
OPER_MOD = '%'
OPER_EXP = '^'
SEMI = ';'
L_PAREN = '('
R_PAREN = ')'
OPER_AND = 'and'
OPER_OR = 'or'
OPER_NOT = 'not'
OPER_SIN = 'sin'
OPER_TAN = 'tan'
OPER_COS = 'cos'
KEYWORD_STDOUT = 'stdout'
KEYWORD_LET = 'let'
KEYWORD_IF = 'if'
KEYWORD_WHILE = 'while'
KEYWORD_TRUE = "true"
KEYWORD_FALSE = "false"
TYPE_BOOL = 'bool'
TYPE_INT = 'int'
TYPE_REAL = 'float'
TYPE_STRING = 'string'
TYPE_ID = 'ID'


class Token:
    type = ''
    value = ''
    line = ''
```
defines.py

3.4 myparser.py

```
__author__ = 'drakebridgewater'
from lexer import *
from node import *
from defines import *


class MyParser(object):
    def __init__(self, filename):
        temp_token = Token
        temp_token.value = "root"
        temp_token.type = "root"
        temp_token.line = -1
        self.tree = Node(temp_token)
        self.lexer = Lexer(filename)
        self.stack = []
        self.current_state = True
        self.tokens = []
        self.line = 0
        self.current_token_index = 0

    def exit(self):
        self.tree.print_tree()
        exit()

    def parse_error(self, msg=''):
        print("PARSE ERROR: [line: " + str(self.line) + "] " + msg)

    # Function Description:
    # will return a single token as the lexer may spit out multiple
    def get_token(self):
        # if not self.tokens:
        if len(self.tokens) == 0:
            self.tokens.append(self.lexer.get_token())
            if self.tokens[0] == -1:
                self.current_state = False  # Done reading file
                return None
            self.line = self.tokens[0].line
        return self.tokens[self.current_token_index]

    def remove_token(self):
        # TODO instead of removing move integer to point to next value
        if len(self.tokens) > 0:
            self.tokens.pop()

    def restore_tokens(self, idx):
        self.current_token_index = idx

    def print_tokens(self):
        try:
            self.lexer.open_file()
            while self.get_token():
                print("[line: " + str(self.tokens.line) +
                      ", ID: " + self.tokens.type +
                      ", Value: " + str(self.tokens.value) + "]")
        finally:
```

```
            self.lexer.close_file()
57
    def control(self):
59      try:
            self.lexer.open_file()
61          # TODO I need a token!
            # while 1:
63          temp = Node(self.tokens)
            print("-" * 30)
65          self.tree.add_child(self.s())
            self.tree.print_tree()
67          if len(self.tokens) == 0:
                return None
69              # if it was unable to tokenize a float then we get a list of
    tokens
                # TODO where we start putting everything in a huge statement
71      finally:
            self.lexer.close_file()
73
    def is_value(self, token, compare):
75      if not self.current_state:
            return None
77      save = self.current_token_index
        if token.value == compare:
79          self.remove_token()
            return Node(token)
81      else:
            self.restore_tokens(save)
83          return None

85  def s(self):
        if not self.current_state:
87          return None
        # s -> expr S' | ( S"
89      new_node = Node("S")
        save = self.current_token_index
91      if new_node.add_child(self.is_value(self.get_token(), L_PAREN)):
            new_node.add_child(self.s_double_prime())
93      elif new_node.add_child(self.expr()):
            new_node.add_child(self.s_prime())
95      else:
            self.restore_tokens(save)
97          print("ERROR")
            self.current_state = False
99      # if len(new_node.children) > 0:
        # return new_node
101     # else:
        #     return None
103     return new_node

105 def s_prime(self):
        if not self.current_state:
107         return None
        # s' -> S S' | epsilon
109     new_node = Node("S'")
```

```
        save = self.current_token_index
        if new_node.add_child(self.s()):
            new_node.add_child(self.s_prime())
        else:
            self.restore_tokens(save)
            new_node.add_child("epsilon")
        return new_node

    def s_double_prime(self):
        if not self.current_state:
            return None
        # S" ->  )S' | S)S'
        new_node = Node('S"')
        save = self.current_token_index
        if new_node.add_child(self.is_value(self.get_token(), R_PAREN)):
            if new_node.add_child((self.s_prime())):
                pass
        elif new_node.add_child((self.s())):
            if new_node.add_child(self.is_value(self.get_token(), R_PAREN)):
                new_node.add_child(self.s_prime())
        else:
            self.restore_tokens(save)
            return None
        return new_node

    def expr(self):
        if not self.current_state:
            return None
        # expr -> oper | stmts
        new_node = Node("expr")
        save = self.current_token_index
        if new_node.add_child(self.oper()):
            pass
        elif new_node.add_child((self.stmts())):
            pass
        else:
            self.restore_tokens(save)
            return None
        return new_node

    def oper(self):
        if not self.current_state:
            return None
        # oper ->   ( := name oper )
        # ( binops oper oper )
        # ( unops oper )
        # constants
        # name
        new_node = Node("oper")
        save = self.current_token_index
        if new_node.add_child(self.is_value(self.get_token(), L_PAREN)):
            new_node.add_child(self.tokens[0])
            self.remove_token()
            if new_node.add_child(self.is_value(self.get_token(), OPER_ASSIGN)
    ):
```

```
                new_node.add_child(self.tokens[0])
                self.remove_token()
                if self.get_token().type == "keyword":
                    new_node.add_child(self.tokens[0])
                    self.remove_token()
                    if new_node.add_child(self.oper()):
                        if new_node.add_child(self.is_value(self.get_token(),
    R_PAREN)):
                            new_node.add_child(self.tokens[0])
                            self.remove_token()
                        else:
                            self.parse_error('missing right paren')
                            self.restore_tokens(save)
                            return None
                    else:
                        self.parse_error("missing oper")
                        self.restore_tokens(save)
                        return None
                else:
                    self.parse_error("missing keyword")
                    self.restore_tokens(save)
                    return None
            elif new_node.add_child(self.binops()):
                if new_node.add_child(self.oper()):
                    if new_node.add_child(self.oper()):
                        if new_node.add_child(self.is_value(self.get_token(),
    R_PAREN)):
                            new_node.add_child(self.tokens[0])
                            self.remove_token()
                        else:
                            self.parse_error("missing expected right paren")
                            self.restore_tokens(save)
                            return None
                    else:
                        self.parse_error("missing expected oper")
                        self.restore_tokens(save)
                        return None
                else:
                    self.parse_error("missing expected oper")
            elif new_node.add_child(self.unops()):
                if new_node.add_child(self.oper()):
                    if new_node.add_child(self.is_value(self.get_token(),
    R_PAREN)):
                        new_node.add_child(self.tokens[0])
                        self.remove_token()
                    else:
                        self.parse_error("missing expected right paren")
                        self.restore_tokens(save)
                        return None
                else:
                    self.parse_error("missing expected oper")
                    self.restore_tokens(save)
                    return None
            else:
                self.parse_error("missing assignment oper or binop or unop")
```

```
                self.restore_tokens(save)
                return None
        elif new_node.add_child(self.constants()):
            pass
        elif new_node.add_child(self.name()):
            pass
        else:
            self.parse_error("missing left paren constant or name")
            self.restore_tokens(save)
            return None
        return new_node

    def binops(self):
        # binops -> + | - | * | / | % | ^ | = | > | >= | < | <= | != | or | and
        if not self.current_state:
            return None
        new_node = Node("binops")
        save = self.current_token_index
        if new_node.add_child(self.is_value(self.get_token(), OPER_ADD)):
            new_node.add_child(self.tokens[0])
            self.remove_token()
        elif new_node.add_child(self.is_value(self.get_token(), OPER_SUB)):
            new_node.add_child(self.tokens[0])
            self.remove_token()
        elif new_node.add_child(self.is_value(self.get_token(), OPER_MULT)):
            new_node.add_child(self.tokens[0])
            self.remove_token()
        elif new_node.add_child(self.is_value(self.get_token(), OPER_DIV)):
            new_node.add_child(self.tokens[0])
            self.remove_token()
        elif new_node.add_child(self.is_value(self.get_token(), OPER_MOD)):
            new_node.add_child(self.tokens[0])
            self.remove_token()
        elif new_node.add_child(self.is_value(self.get_token(), OPER_EXP)):
            new_node.add_child(self.tokens[0])
            self.remove_token()
        elif new_node.add_child(self.is_value(self.get_token(), OPER_EQ)):
            new_node.add_child(self.tokens[0])
            self.remove_token()
        elif new_node.add_child(self.is_value(self.get_token(), OPER_LT)):
            new_node.add_child(self.tokens[0])
            self.remove_token()
        elif new_node.add_child(self.is_value(self.get_token(), OPER_LE)):
            new_node.add_child(self.tokens[0])
            self.remove_token()
        elif new_node.add_child(self.is_value(self.get_token(), OPER_GT)):
            new_node.add_child(self.tokens[0])
            self.remove_token()
        elif new_node.add_child(self.is_value(self.get_token(), OPER_GE)):
            new_node.add_child(self.tokens[0])
            self.remove_token()
        elif new_node.add_child(self.is_value(self.get_token(), OPER_NE)):
            new_node.add_child(self.tokens[0])
            self.remove_token()
```

```
            elif new_node.add_child(self.is_value(self.get_token(), OPER_OR)):
271             new_node.add_child(self.tokens[0])
                self.remove_token()
273         elif new_node.add_child(self.is_value(self.get_token(), OPER_AND)):
                new_node.add_child(self.tokens[0])
275             self.remove_token()
            else:
277             self.parse_error("missing binop")
                self.restore_tokens(save)
279             return None
            return new_node
281
    def unops(self):
283     # unops -> - | not | sin | cos | tan
        if not self.current_state:
285         return None
        new_node = Node("unops")
287     save = self.current_token_index
        if new_node.add_child(self.is_value(self.get_token(), OPER_NOT)):
289         new_node.add_child(self.tokens[0])
            self.remove_token()
291     elif new_node.add_child(self.is_value(self.get_token(), OPER_SIN)):
            new_node.add_child(self.tokens[0])
293         self.remove_token()
        elif new_node.add_child(self.is_value(self.get_token(), OPER_COS)):
295         new_node.add_child(self.tokens[0])
            self.remove_token()
297     elif new_node.add_child(self.is_value(self.get_token(), OPER_TAN)):
            new_node.add_child(self.tokens[0])
299         self.remove_token()
        else:
301         self.restore_tokens(save)
            self.parse_error("missing unop")
303         return None
        return new_node
305
    def constants(self):
307     # constants -> string | ints | floats
        if not self.current_state:
309         return None
        new_node = Node("constant")
311     save = self.current_token_index
        if new_node.add_child(self.strings()):
313         pass
        elif new_node.add_child(self.ints()):
315         pass
        elif new_node.add_child(self.floats()):
317         pass
        else:
319         self.restore_tokens(save)
            return None
321     return new_node

323 def strings(self):
        # strings ->    reg_ex for str literal in C (  any alphanumeric )
```

```
325         # true | false
            if not self.current_state:
327             return None
            new_node = Node("string")
329         save = self.current_token_index
            if self.get_token().type == TYPE_STRING:
331             new_node.add_child(self.tokens[0])
                self.remove_token()
333         elif self.get_token().type == TYPE_BOOL:
                new_node.add_child(self.tokens[0])
335             self.remove_token()
            else:
337             self.restore_tokens(save)
                return None
339         return new_node

341     def name(self):
            # name -> reg_ex for ids in C (any lower and upper char
343         # or underscore followed by any combination of lower,
            # upper, digits, or underscores)
345         if not self.current_state:
                return None
347         new_node = Node("name")
            save = self.current_token_index
349         if self.get_token().type == TYPE_ID:
                new_node.add_child(self.tokens[0])
351             self.remove_token()
            else:
353             self.restore_tokens(save)
                return None
355         return new_node

357     def ints(self):
            # ints -> reg ex for positive/negative ints in C
359         if not self.current_state:
                return None
361         new_node = Node("int")
            save = self.current_token_index
363         if self.get_token().type == TYPE_INT:
                new_node.add_child(self.tokens[0])
365             self.remove_token()
            else:
367             self.restore_tokens(save)
                return None
369         return new_node

371     def floats(self):
            # floats -> reg ex for positive/negative doubles in C
373         if not self.current_state:
                return None
375         new_node = Node("float")
            save = self.current_token_index
377         if self.get_token().type == TYPE_REAL:
                new_node.add_child(self.tokens[0])
379             self.remove_token()
```

```
            else:
381             self.restore_tokens(save)
                return None
383         return new_node

385     def stmts(self):
            # stmts -> ifstmts | whilestmts | letstmts |printsmts
387         if not self.current_state:
                return None
389         new_node = Node("stmts")
            save = self.current_token_index
391         if new_node.add_child(self.ifstmts()):
                pass
393         elif new_node.add_child(self.whilestmts()):
                pass
395         elif new_node.add_child(self.letstmts()):
                pass
397         elif new_node.add_child(self.printstmts()):
                pass
399         else:
                self.parse_error("missing if, while, let or print statment")
401             self.restore_tokens(save)
                return None
403         return new_node

405     def printstmts(self):
            # printstmts -> (stdout oper)
407         if not self.current_state:
                return None
409         new_node = Node("printstmts")
            save = self.current_token_index
411         if new_node.add_child(self.is_value(self.get_token(), L_PAREN)):
                new_node.add_child(self.tokens[0])
413             self.remove_token()
                if new_node.add_child(self.is_value(self.get_token(),
    KEYWORD_STDOUT)):
415                 new_node.add_child(self.tokens[0])
                    self.remove_token()
417                 if new_node.add_child(self.oper()):
                        if new_node.add_child(self.is_value(self.get_token(),
    R_PAREN)):
419                         new_node.add_child(self.tokens[0])
                            self.remove_token()
421                     else:
                            self.parse_error("missing right paren")
423                         self.restore_tokens(save)
                            return None
425                 else:
                        self.parse_error("missing oper")
427                     self.restore_tokens(save)
                        return None
429             else:
                    self.parse_error("missing keyword stdout")
431                 self.restore_tokens(save)
                    return None
```

```
433             else:
                    self.parse_error("missing left paren")
435                 self.restore_tokens(save)
                    return None
437             return new_node

439         def ifstmts(self):
                # ifstmts -> (if expr expr expr) | (if expr expr)
441             if not self.current_state:
                    return None
443             new_node = Node("ifstmts")
                save = self.current_token_index
445             if new_node.add_child(self.is_value(self.get_token(), L_PAREN)):
                    new_node.add_child(self.tokens[0])
447                 self.remove_token()
                    if new_node.add_child(self.expr()):
449                     if new_node.add_child(self.expr()):
                            if new_node.add_child(self.expr()):
451                             if new_node.add_child(self.is_value(self.get_token(),
    R_PAREN)):
                                    new_node.add_child(self.tokens[0])
453                                 self.remove_token()
                                else:
455                                 self.parse_error("missing right paren")
                                    self.restore_tokens(save)
457                                 return None
                            elif new_node.add_child(self.is_value(self.get_token(),
    R_PAREN)):
459                             new_node.add_child(self.tokens[0])
                                self.remove_token()
461                         else:
                                self.parse_error("missing 3 expression in if statement
     or right paren")
463                             self.restore_tokens(save)
                                return None
465                 else:
                        self.parse_error("missing 2 expression in ifstmts")
467                     self.restore_tokens(save)
                        return None
469             else:
                    self.parse_error("missing first expression in ifstmt")
471                 self.restore_tokens(save)
                    return None
473         else:
                self.parse_error("missing left paren in if statment")
475             self.restore_tokens(save)
                return None
477         return new_node

479     def whilestmts(self):
            # whilestmts -> (while expr exprlist)
481         if not self.current_state:
                return None
483         new_node = Node("whilestmts")
            save = self.current_token_index
```

```
485             if new_node.add_child(self.is_value(self.get_token(), L_PAREN)):
                    new_node.add_child(self.tokens[0])
487                 self.remove_token()
                    if new_node.add_child(self.is_value(self.get_token(),
    KEYWORD_WHILE)):
489                     new_node.add_child(self.tokens[0])
                        self.remove_token()
491                     if new_node.add_child(self.expr()):
                            if new_node.add_child(self.exprlist()):
493                             if new_node.add_child(self.is_value(self.get_token(),
    R_PAREN)):
                                    new_node.add_child(self.tokens[0])
495                                 self.remove_token()
                                else:
497                                 self.parse_error("missing right paren")
                                    self.restore_tokens(save)
499                                 return None
                            else:
501                             self.parse_error("missing exprlist")
                                self.restore_tokens(save)
503                             return None
                        else:
505                         self.parse_error("missing expression")
                            self.restore_tokens(save)
507                         return None
                else:
509                 self.parse_error("missing while clause")
                    self.restore_tokens(save)
511                 return None
            else:
513             self.parse_error("missing left paren")
                self.restore_tokens(save)
515             return None
            return new_node
517
        def exprlist(self):
519         # exprlist -> expr | expr exprlist
            if not self.current_state:
521             return None
            new_node = Node("exprlist")
523         save = self.current_token_index
            if new_node.add_child(self.expr()):
525             if new_node.add_child(self.exprlist()):
                    pass
527         else:
                self.parse_error("missing expr")
529             self.restore_tokens(save)
                return None
531         return new_node

533     def letstmts(self):
            # letstmts -> (let (varlist))
535         if not self.current_state:
                return None
537         new_node = Node("letstmts")
```

```
        save = self.current_token_index
539     if new_node.add_child(self.is_value(self.get_token(), L_PAREN)):
            new_node.add_child(self.tokens[0])
541         self.remove_token()
            if new_node.add_child(self.is_value(self.get_token(), KEYWORD_LET)
    ):
543             new_node.add_child(self.tokens[0])
                self.remove_token()
545             if new_node.add_child(self.is_value(self.get_token(), L_PAREN)
    ):
                    new_node.add_child(self.tokens[0])
547                 self.remove_token()
                    if new_node.add_child(self.varlist()):
549                     if new_node.add_child(self.is_value(self.get_token(),
    R_PAREN)):
                            new_node.add_child(self.tokens[0])
551                         self.remove_token()
                            if new_node.add_child(self.is_value(self.get_token
    (), R_PAREN)):
553                             new_node.add_child(self.tokens[0])
                                self.remove_token()
555                         else:
                                self.parse_error("missing right paren")
557                             self.restore_tokens(save)
                                return None
559                     else:
                            self.parse_error("missing right paren")
561                         self.restore_tokens(save)
                            return None
563                 else:
                        self.parse_error("missing varlist")
565                     self.restore_tokens(save)
                        return None
567             else:
                    self.parse_error("missing left paren")
569                 self.restore_tokens(save)
                    return None
571         else:
                self.parse_error("missing keyword let")
573             self.restore_tokens(save)
                return None
575     else:
            self.parse_error("missing left paren")
577         self.restore_tokens(save)
            return None
579     return new_node

581 def varlist(self):
        # varlist -> (name type) | (name type) varlist
583     if not self.current_state:
            return None
585     new_node = Node("varlist")
        save = self.current_token_index
587     if new_node.add_child(self.is_value(self.get_token(), L_PAREN)):
            new_node.add_child(self.tokens[0])
```

```
589             self.remove_token()
                if self.get_token().type == TYPE_ID:
591                 new_node.add_child(self.tokens[0])
                    self.remove_token()
593                 if new_node.add_child(self.type()):
                        if new_node.add_child(self.is_value(self.get_token(),
    R_PAREN)):
595                         new_node.add_child(self.tokens[0])
                            self.remove_token()
597                         if new_node.add_child(self.varlist()):
                                pass
599                         return new_node
                        else:
601                         self.parse_error("missing right paren")
                            self.restore_tokens(save)
603                         return None
                    else:
605                     self.parse_error("missing type")
                        self.restore_tokens(save)
607                     return None
                else:
609                 self.parse_error("missing name ")
                    self.restore_tokens(save)
611                 return None
            else:
613             self.parse_error("missing left paren")
                self.restore_tokens(save)
615             return None

617     def type(self):
            # type -> bool | int | real | string
619         if not self.current_state:
                return None
621         new_node = Node("type")
            save = self.current_token_index
623         if new_node.add_child(self.is_value(self.get_token(), TYPE_BOOL)):
                new_node.add_child(self.tokens[0])
625             self.remove_token()
            elif new_node.add_child(self.is_value(self.get_token(), TYPE_INT)):
627             new_node.add_child(self.tokens[0])
                self.remove_token()
629         elif new_node.add_child(self.is_value(self.get_token(), TYPE_REAL)):
                new_node.add_child(self.tokens[0])
631             self.remove_token()
            elif new_node.add_child(self.is_value(self.get_token(), TYPE_STRING)):
633             new_node.add_child(self.tokens[0])
                self.remove_token()
635         else:
                self.parse_error("missing type")
637             self.restore_tokens(save)
                return None
639         return new_node

641     def print_stack(self):
            for child in self.stack:
```

```
643             print("[line: " + str(child.line) + ", ID: " + child.type + ",
    Value: " + str(child.value) + "]")

645     def add_to_stack(self, token):
            if token.value is L_PAREN:
647             self.stack.append(token)
            elif token.value is R_PAREN:
649             if self.stack.pop() is L_PAREN:
                    pass
651             else:
                    print("Syntax Error: [Line: " + str(token.line) + "] missing
    right parentheses")
653         pass
```

myparser.py

3.5 lexer.py

```
1  __author__ = 'drakebridgewater'
   import string
3
   from defines import *
5

7  class Lexer():
       def __init__(self, filename):
9          self.line = 1
           self.filename = filename
11         self.file = ''
           self.current_char = ' '
13         self.pointer = 0
           self.token_list = []
15         self.current_state = True  # When false throw error
           self.accepted_ops = ('=', '+', '-', '/', '*', '<', '>', '!', ';', ':',
       '%', '(', ')', '^')
17         # tokens is a dictionary where each token is a list
           self.tokens = \
19             {"keywords": [KEYWORD_STDOUT, KEYWORD_LET, KEYWORD_IF,
    KEYWORD_WHILE,
                             KEYWORD_TRUE, KEYWORD_FALSE, OPER_ASSIGN],
21             "ops": [OPER_ASSIGN, OPER_ADD, OPER_SUB, OPER_DIV, OPER_MULT,
                       OPER_LT, OPER_GT, OPER_NOT, OPER_MOD, OPER_EXP,
23                     OPER_AND, OPER_OR, OPER_NOT, OPER_NE, R_PAREN, L_PAREN],
               'type': [TYPE_BOOL, TYPE_INT, TYPE_REAL, TYPE_STRING]
25             }

27     def open_file(self):
           self.file = open(self.filename, 'r')
29
       def close_file(self):
31         self.file.close()

33     def has_token(self, value, key=''):
           # if subgroup given check it first
35         if key != '':
               if value in self.tokens[key]:
```

```
37                 return key

39             # if subgroup checking fails check all entries
               for x in self.tokens:
41                 if value in self.tokens[x]:
                       return x
43             return -1

45         def get_next_char(self):
               try:
47                 self.current_char = self.file.read(1)
               except EOFError:
49                 print("Reached end of file")

51         def get_token(self):
               self.get_next_char()
53             while True and self.current_state:
                   if not self.current_char:
55                     return -1
                   if self.current_char == ' ' or self.current_char == '\t':
57                     self.get_next_char()
                       pass
59                 elif self.current_char == '\n':
                       self.get_next_char()
61                     self.line += 1
                   elif self.current_char in self.accepted_ops:
63                     return self.is_op()
                   elif self.is_letter():
65                     return self.identify_word()  # identify the string and add to
    the token list
                   elif self.is_digit():
67                     return self.is_number()  # identify the number and add to the
    token list
                   elif self.current_char == '"':
69                     return self.create_token(("ops", '"'))
                       # self.parse_string()                 # parse a string
71                 else:
                       print("Line:ERROR: Could not identify on line: " + str(
73                         self.line) + " near char: '" + self.current_char + "'")
                       return None
75
                   # TODO have all functions return to a state that has the next
    char
77
           # Function Description:
79         # General function to do something with the tokens once we have classified
            them.
           def create_token(self, token):
81             new_token = Token()
               new_token.line = self.line
83             new_token.type = token[0]
               new_token.value = token[1]
85             return new_token

87         def add_token(self, token):
```

```
        new_token = Token()
89      new_token.line = self.line
        new_token.type = token[0]
91      new_token.value = token[1]
        self.token_list.append(new_token)
93
    def print_tokens(self):
95      for x in self.token_list:
            print("[line: " + x.line + ", ID: " + x.type + ", Value: " + x.
    value + "]")
97
    def is_op(self):
99      item = self.current_char
        # If we see an op look to see if we see another. If we see another add
     the previous
101     # found op
        if self.current_char is '+':
103         self.get_next_char()
            return self.create_token((self.has_token(item), item))
105     elif self.current_char is '-':
            self.get_next_char()
107         # if self.current_char is '-':
            # item += self.current_char  # Seen -- make new token
109         #    self.get_next_char()
            return self.create_token((self.has_token(item), item))
111     elif self.current_char in ('<', '>', '!'):
            self.get_next_char()
113         if self.current_char == '=':
                item += self.current_char
115             self.get_next_char()
            return self.create_token((self.has_token(item), item))
117     elif self.current_char in ':':
            self.get_next_char()
119         if self.current_char is '=':
                item += self.current_char
121             return self.create_token((self.has_token(item), item))
            else:
123             print("Lexer Error [Line: " + str(
                    self.line) + '] the "' + self.current_char + '" symbol not
     recognized after colon [:]')
125     elif self.current_char in '=':
            return self.create_token((self.has_token(item), item))
127     elif self.current_char in ('*', '/', '(', ')', '%', '^'):
            self.get_next_char()
129         return self.create_token((self.has_token(item), item))
        else:
131         print("Lexer Error: [Line: " + str(self.line) + "] could not
    intemperate: " +
                  self.current_char)
133         return -1

135 def parse_string(self):
        accepted_chars = ['"']
137     new_string = ''
        self.get_next_char()
```

```
139             while self.current_char in accepted_chars:
                    new_string += self.current_char
141                 self.get_next_char()
                self.token_list.append(("string", new_string))
143
            def identify_word(self):
145             accepted_chars = list(string.ascii_letters) + list(string.digits) +
        list('_')
                acceptable_first_chars = list(string.ascii_letters)
147
                word = ''
149             if self.current_char in acceptable_first_chars:
                    word += self.current_char
151                 self.get_next_char()
                    while self.current_char in accepted_chars:
153                     word += self.current_char
                        # TODO if part of the token is in the token list what do we do
        ??
155                     self.get_next_char()
                token_value = word
157             token_type = self.has_token(token_value)
                if token_type == -1:
159                 token_type = "ID"
                return self.create_token((token_type, token_value))
161
            # Function Description:
163         # This function should be called when a word identifier or keyword is
        started
            # and will return the full word upon seeing invalid characters.
165         def parse_word(self, accepted_chars, acceptable_first_chars=[]):
                if self.current_char not in acceptable_first_chars:
167                 return -1
                else:
169                 word = ''
                    while self.current_char in accepted_chars:
171                     word += self.current_char
                        self.get_next_char()
173                 return word

175         def is_int(self):
                word = ''
177             while self.is_digit(exclude=['.', 'e']):
                    word += self.current_char
179                 self.get_next_char()

181             return word

183         # Function Description:
            # This function should be called after seeing the start of a number
185         # If a period is present the number is converted to a float and returned
            def is_number(self, value=''):
187             if value == '':
                    word = self.current_char
189             else:
                    word = value
```

```
191         self.get_next_char()

193         other_accepted = ['.'] # accept additional chars if we have seen
    certain chars
            while self.is_digit(other_accepted):
195             if self.current_char is '.':
                    if '.' in other_accepted:
197                     other_accepted.remove('.')
                    if '.' not in word:
199                     # this number is a decimal
                        word += self.current_char
201                     self.get_next_char()
                    else:
203                     # word already contains a dot. don't get next char
                        return self.create_token(('float', float(word)))
205             elif self.current_char is 'e': # once you 'e' has been seen no
    decimal can be used
                    if '.' in other_accepted:
207                     other_accepted.remove('.')
                    self.get_next_char()
209                 if self.current_char is '+':
                        self.get_next_char()
211                     exp = self.is_int()
                        try:
213                         self.get_next_char()
                            exp = int(exp)
215                         word += 'e+'
                            word += str(exp)
217                         try:
                                return self.create_token(("float", float(word)))
219                         except ValueError:
                                print("Fatal parse error: [row: " + str(self.line)
    + "] when parsing char '" +
221                                   str(self.current_char) + "' for: \n\t\t" +
    str(word))
                    except ValueError:
223                     return [self.create_token(("int", word)),
                                self.create_token(("ID", "e")),
225                             self.create_token((self.has_token("+"), "+"))]

227                 elif self.current_char is '-':
                        self.get_next_char()
229                     exp = self.is_int()
                        try:
231                         self.get_next_char()
                            exp = int(exp)
233                         word += 'e-'
                            word += str(exp)
235                         try:
                                return self.create_token(("float", float(word)))
237                         except ValueError:
                                print("Fatal parse error: [row: " +
239                                   str(self.line) + "] when parsing char '" +
                                   str(self.current_char) + "' for: \n\t\t" +
    str(word))
```

```
241                     except ValueError:
                            return [self.create_token(("int", word)),
243                                 self.create_token(("ID", "e")),
                                    self.create_token((self.has_token("-"), "-"))]
245                 else:
                        exp = self.is_int()
247                     try:
                            exp = int(exp)
249                         word += str(exp)
                            return self.create_token(("float", float(word)))
251                     except ValueError:
                            print("Lexer Error: [row: " + str(self.line) + "]
    Unable to parse '" +
253                               str(self.current_char) + "' in: " + str(exp))
            elif self.is_digit(other_accepted):
255                 word += self.current_char
                    self.get_next_char()
257         else:
                break
259         if 'e' not in other_accepted:
                other_accepted.append('e')
261
    if '.' in word or 'e' in word:
263         try:
                return self.create_token(("float", float(word)))
265         except ValueError:
                print("Lexer Error (line: " + str(self.line) +
267                   "): could not determine numerical token of: " + str(word
    ))
    else:
269         try:
                return self.create_token(("int", int(word)))
271         except ValueError:
                print("Lexer Error (line: " + str(self.line) +
273                   "): could not determine numerical token of: " + str(word
    ))

275 # Function Description:
    # checks to see if the current token in peek is a digit or '.'
277 # return true if it is
    def is_digit(self, others=[], exclude=[]):
279     digits = ['.', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
        for x in others:
281         if x not in digits:
                digits.append(x)
283     for x in exclude:
            if x in digits:
285             digits.remove(x)
        if self.current_char in digits:
287         return True
        return False
289
    # Function Description:
291 # checks to see if the current token in peek is a letter
    # return true if it is
```

```
293     def is_letter(self, others=[]):
            letters = list(string.ascii_letters)
295         for x in others:
                if x not in letters:
297                 letters.append(x)
            if self.current_char in letters:
299             return True
            return False
```

lexer.py