

OREGON STATE UNIVERSITY

CS 352 - TRANSLATORS

WINTER 2015

Milestone 4: Constant Only Calculations

Author:
Drake Bridgewater

Professor:
Dr. Jennifer PARHAM-MOCELLO

DUE 01/16/15 (11:59pm)
March 1, 2015

Contents

1	Source Code Descriptions	2
1.1	node.py	2
1.2	codegen.py	2
1.3	defines.py	2
1.4	myparser.py	2
1.5	lexer.py	2
2	Report	4
3	Source Code	5
3.1	main.py	5
3.2	codegen.py	6
3.3	node.py	8
3.4	defines.py	10
3.5	myparser.py	11
3.6	lexer.py	23

1 Source Code Descriptions

The way I approached this problem was one paper with drawing out what how I would perform each 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 codegen.py

This was created to take the tree that we created in the last assignment. With the tree we would walk through it, post-order, and then as we saw the elements we pushed them out to to file. This allowed the code to be rather simple. Once that was finished I converted the variable to gforth code and wrote that to another file.

1.3 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.

1.4 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.

1.5 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

This assignment was much easier then I thought it was going to be but **the purpose** of this assignment was to ensure that our tree was producing the correct output and that we understand tokens, trees and parsing. To approach this problem I need it to be simple therefore I took concepts that were taught early on, such as the post order tree traversal and the basic parsing, to produce gforth code. With a solid idea I was able to **solve** this problem mostly by a single if statement but then I need to do type checking to ensure that the values were coming in properly. During the entire process I was creating **test** cases that would test the current function allowing me to get really good code coverage.

3 Source Code

3.1 main.py

```
1  #!/usr/bin/python
   __author__ = 'Drake'
3  import sys

5  from myparser import *

7
   usage = """
9  Usage:
       main.py [option] [files]
11 """

13 files = []
   options = []
15 global DEBUG
   DEBUG = 1

17
   global current_token_index
19 current_token_index = 0

21
   def read_file(input_file):
23       content = ""
       f = open(input_file)

25
       lines_raw = f.readlines()
27       for i in range(0, len(lines_raw)):
           content += lines_raw[i]

29
       return content

31

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

39

41 def main():
       sys.setrecursionlimit(100)
43       if '-d' in sys.argv:
           globals()['DEBUG'] = 1
45       if len(sys.argv) > 1:
           filename = sys.argv[len(sys.argv) - 1]
47       else:
           filename = "test1"
49       parser = MyParser(filename)
       parser.control()
51       tree = parser.tree
```

```

53     tree.post_order_tree_print()

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

```

main.py

3.2 codegen.py

```

__author__ = 'drakebridgewater'
2 from defines import *

4
class codeGen():
6     def __init__(self, tree):
            self.tree = tree
8             self.current_token = None
            self.stack = []
10            self.next_tree_item = False

12    def control(self):
        # TODO as we step through the tree convert and push element on to
        stack
14        pass

16    def post_order_walkthrough(self, node):
        for child in node:
18            self.next_tree_item = False
            self.post_order_walkthrough(child)
20            self.do_something(node.data)

22    def do_something(self, data):
        if self.next_tree_item:
24            return
        num_ops = self.oper_count(data)
26        if num_ops == 1:
            if self.is_number(data):
28                self.write_out()
                if self.next_tree_item:
30                    return
            else:
32                self.print_error("num_ops = 1 first value error")
                return
34        if num_ops == 2:
            if self.write_out(self.is_number(data)):
36                if self.write_out(self.is_number(data)):
                    self.write_out(data)
38            else:
                self.print_error("num_ops = 1 second value error")
40        else:
            self.print_error("num_ops = 1 first value error")
42
44        if self.is_number(data):
            if data.type is TYPE_INT:
                self.stack.append()

```

```

46         if data == OPER_ADD:
47             if self.is_number(data):
48                 self.write_out()
49                 if self.is_number(data):
50                     self.stack.pop()

52     # Function Description:
53     # Only write out if data is actually data
54     def write_out(self, data):
55         if data.value is KEYWORD.STDOUT:
56             print('.s')
57         elif data.value is KEYWORD.STDOUT:
58             print('.s')

60         elif data.value is OPER_EQ:
61             pass

62         elif data.value is OPER_ASSIGN:
63             pass

64         elif data.value is OPER_ADD:
65             pass

66         elif data.value is OPER_ADD:
67             pass

68         elif data.value is OPER_SUB:
69             pass

70         elif data.value is OPER_DIV:
71             pass

72         elif data.value is OPER_DIV:
73             pass

74         elif data.value is OPER_MULT:
75             pass

76         elif data.value is OPER_LT:
77             pass

78         elif data.value is OPER_LT:
79             pass

80         elif data.value is OPER_GT:
81             pass

82         elif data.value is OPER_GT:
83             pass

84         elif data.value is OPER_LE:
85             pass

86         elif data.value is OPER_GE:
87             pass

88         elif data.value is OPER_GE:
89             pass

90         elif data.value is OPER_NE:
91             pass

92         elif data.value is OPER_NOT:
93             pass
94             print(data)
95         else:
96             pass

98     def compare(self, value1, value2):
100         pass

```



```

102     def is_number(self, value1):
103         if hasattr(value1, "type") and value1.type in [TYPE_INT, TYPE_REAL]:
104             self.next_tree_item = True
105             return value1
106     return False

```

codegen.py

3.3 node.py

```

__author__ = 'Drake'

2

4 class Node(object):
5     def __init__(self, data):
6         if hasattr(data, "value"):
7             print("New Node: " + str(data.value))
8         else:
9             print("NN Str: " + str(data))
10        self.data = data
11        self.children = []
12        self.depth = 0

14    def add_child(self, obj):
15        if obj is None:
16            globals()['current_token_index'] -= 1
17            return obj
18        self.children.append(obj)
19        globals()['current_token_index'] += 1
20        return True

22    # need to set depth recursively
23    def set_depth(self, t):
24        if t is not None or t != str:
25            if len(t.children) > 0:
26                for i in t.children:
27                    if i is not None:
28                        i.depth = t.depth + 1
29                        self.set_depth(i)
30        self.set_depth()

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

34

35    def get_first_child_at_parent(self, obj):
36        if len(obj.children) > 0:
37            return obj.children[0]
38        else:
39            return self.children[0]

40

41    def get_first_child_at_parent_level(self, obj, level):
42        if level == 0:
43            return self.children[0]
44        else:
45            if level >= 1:

```

```

46         if len(obj.children) > 0:
47             return obj.children[0]
48         else:
49             return self.children[0]
50     else:
51         return self.children[0]
52
53     @staticmethod
54     def get_parent_depth(obj):
55         return obj.depth
56
57     def print_tree(self):
58         print("-" * 40 + "\n\t print tree called")
59         # print(self.data)
60         self.print_tree_helper(self)
61
62     def print_tree_helper(self, node, indent=0):
63         indent += 1
64         for child in node.children:
65             # if child.get_child_count() > 0:
66             # if child.data is not None:
67             if isinstance(child, int):
68
69                 print("\t" * indent + str(child))
70             elif isinstance(child, str):
71                 print("\t" * indent + str(child))
72             elif hasattr(child, "data"):
73                 if hasattr(child.data, "value"):
74                     print("\t" * indent + "[line: " + str(child.data.line) +
75                               ", ID: " + child.data.type +
76                               ", Value: " + str(child.data.value) + "]")
77                 else:
78                     print("\t" * indent + str(child.data))
79                     self.print_tree_helper(child, indent)
80             elif hasattr(child, "value"):
81                 print("\t" * indent + "[line: " + str(child.line) +
82                               ", ID: " + child.type +
83                               ", Value: " + str(child.value) + "]")
84             else:
85                 print("Error in print_tree_helper")
86                 print(child)
87                 return
88             # else:
89             # print("Failed")
90
91     def print_postordered_tree(self):
92         print("-" * 80 + "\n\t print post ordered tree called")
93         print(self.root.get_value())
94         self.post_order_tree_print(self.root)
95
96     def post_order_tree_print(self, node):
97         for child in node.children:
98             self.post_order_tree_print(child)
99             print("[line: " + str(child.data.line) + ", ID: " + child.data.
100 type + ", Value: " + str(

```

```

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

```

3.4 defines.py

```

__author__ = 'Drake'

2   files = []
4   options = []

6   if not 'current_token_index' in globals():
        current_token_index = 0

8
OPER_EQ = '='
10 OPER_ASSIGN = ':= '
OPER_ADD = '+'
12 OPER_SUB = '-'
OPER_DIV = '/'
14 OPER_MULT = '*'
OPER_LT = '<'
16 OPER_GT = '>'
OPER_LE = '<='
18 OPER_GE = '>='
OPER_NE = '!='
20 OPER_NOT = '!'
OPER_MOD = '%'
22 OPER_EXP = '^'
SEMI = ';'
24 L_PAREN = '('
R_PAREN = ')'
26 OPER_AND = 'and'
OPER_OR = 'or'
28 OPER_NOT = 'not'
OPER_SIN = 'sin'
30 OPER_TAN = 'tan'
OPER_COS = 'cos'
32 KEYWORD_STDOUT = 'stdout'
KEYWORD_LET = 'let'
34 KEYWORD_IF = 'if'
KEYWORD_WHILE = 'while'
36 KEYWORD_TRUE = "true"
KEYWORD_FALSE = "false"
38 TYPE_BOOL = 'bool'
TYPE_INT = 'int'
40 TYPE_REAL = 'float'
TYPE_STRING = 'string'
42 TYPE_ID = 'ID'

44
class Token:
46     type = ''
        value = ''
48     line = ''

```

defines.py

3.5 myparser.py

```
--author-- = 'drakebridgewater'
2 from lexer import *
  from defines import *
4

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

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

22    def get_child_at(self, index):
        return self.children[index]
24

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

30    def get_first_child_at_parent_level(self, obj, level):
32        if level == 0:
            return self.children[0]
34        else:
            if level >= 1:
36                if len(obj.children) > 0:
                    return obj.children[0]
38                else:
                    return self.children[0]
40            else:
                return self.children[0]
42

44    @staticmethod
    def get_parent_depth(obj):
        return obj.depth
46

48    def print_tree(self):
        print("--" * 40 + "\n\t print tree called")
        # print(self.data)
        self.print_tree_helper(self)
50        print("--" * 40)
52

    def print_tree_helper(self, node, indent=0):
```

```

54         indent += 1
55     for child in node.children:
56         # if child.get_child_count() > 0:
57         # if child.data is not None:
58         if isinstance(child, int):
59             print("\t" * indent + str(child))
60         elif isinstance(child, str):
61             print("\t" * indent + str(child))
62         elif hasattr(child, "data"):
63             if hasattr(child.data, "value"):
64                 print("\t" * indent + "[line: " + str(child.data.line) +
65                     ", ID: " + child.data.type +
66                     ", Value: " + str(child.data.value) + "]" )
67             else:
68                 print("\t" * indent + str(child.data))
69                 self.print_tree_helper(child, indent)
70         elif hasattr(child, "value"):
71             print("\t" * indent + "[line: " + str(child.line) +
72                 ", ID: " + child.type +
73                 ", Value: " + str(child.value) + "]" )
74         else:
75             print("Error in print_tree_helper")
76             print(child)
77             return
78         # else:
79         # print("Failed")
80
81     def print_postordered_tree(self):
82         print("_" * 80 + "\n\t print post ordered tree called")
83         self.post_order_tree_print(self)
84
85     def post_order_tree_print(self, node):
86         for child in node.children:
87             self.post_order_tree_print(child)
88             self.print_child(child)
89
90     def print_child(self, child, indent):
91         if isinstance(child, int):
92             print("\t" * indent + str(child))
93         elif isinstance(child, str):
94             print("\t" * indent + str(child))
95         elif hasattr(child, "data"):
96             if hasattr(child.data, "value"):
97                 print("\t" * indent + "[line: " + str(child.data.line) +
98                     ", ID: " + child.data.type +
99                     ", Value: " + str(child.data.value) + "]" )
100             else:
101                 print("\t" * indent + str(child.data))
102         elif hasattr(child, "value"):
103             print("\t" * indent + "[line: " + str(child.line) +
104                 ", ID: " + child.type +
105                 ", Value: " + str(child.value) + "]" )
106         else:
107             print("Error in print_tree_helper")
108             print(child)

```

```

        return False
110     return True

112
class MyParser(object):
114     def __init__(self, filename):
        temp_token = Node("EMPTY")
116         self.tree = Node(temp_token)
        self.lexer = Lexer(filename)
118         self.current_state = True
        self.tokens = []
120         self.line = 0
        self.epsilon_flag = 0

122
    def exit(self):
124         self.tree.print_tree()
        exit()

126
    def parse_error(self, msg=''):
128         pass
        # print("PARSE ERROR: [line: " + str(self.line) + "] " + msg)

130
    # Function Description:
    # will return a single token as the lexer may spit out multiple
132
    def get_token(self):
134         # if not self.tokens:
        new_token = self.lexer.get_token()
136         if new_token is not -1:
            self.tokens.append(new_token)
138         if self.tokens[len(self.tokens) - 1] == -1:
            return None
140         if len(self.tokens) <= globals()['current_token_index']:
            # if self.tokens[len(self.tokens) - 1] == -1:
142             self.current_state = False # Done reading file
            return None
144         else:
            self.line = self.tokens[globals()['current_token_index']].line
146             return self.tokens[globals()['current_token_index']]

148
    def print_tokens(self):
        try:
150             self.lexer.open_file()
            while self.get_token():
152                 print("[line: " + str(self.tokens.line) +
                        ", ID: " + self.tokens.type +
154                        ", Value: " + str(self.tokens.value) + "]")
            finally:
156                 self.lexer.close_file()

158
    def control(self):
        try:
160             self.lexer.open_file()
            print("-" * 30)
162             while self.current_state:
                self.tree.add_child(self.s())

```

```

164         # globals()[ 'current_token_index' ] += 1
165         self.tree.print_tree()
166         self.tree.print_postordered_tree()
167         if len(self.tokens) > globals()[ 'current_token_index' ]:
168             # if self.tokens[len(self.tokens) - 1] == -1:
169                 self.current_state = True # Done reading file
170     if len(self.tokens) == 0:
171         return None
172 finally:
173     self.lexer.close_file()
174
175 def is_type(self, token, compare):
176     if not self.current_state:
177         return None
178     if isinstance(token, int):
179         return None
180     if token.type == compare:
181         globals()[ 'current_token_index' ] += 1
182         return Node(token)
183     else:
184         return None
185
186 def is_value(self, token, compare):
187     if not self.current_state:
188         return None
189     if token is None:
190         return None
191     if token.value == compare:
192         globals()[ 'current_token_index' ] += 1
193         return Node(token)
194     else:
195         return None
196
197 def s(self):
198     if not self.current_state:
199         return None
200     # s -> expr S' | ( S"
201     new_node = Node("S")
202     save = globals()[ "current_token_index" ]
203     if new_node.add_child(self.expr()):
204         if new_node.add_child(self.s_prime()):
205             if self.epsilon_flag:
206                 self.epsilon_flag = 0
207                 return new_node
208             else:
209                 globals()[ "current_token_index" ] = save
210     elif new_node.add_child(self.is_value(self.get_token(), LPAREN)):
211         if new_node.add_child(self.s_double_prime()):
212             pass
213         else:
214             globals()[ "current_token_index" ] = save
215     else:
216         globals()[ "current_token_index" ] = save
217         print("ERROR")
218         self.current_state = True

```

```

220         return None
221     # if len(new_node.children) > 0:
222     #     return new_node
223     # else:
224     #     return None
225     return new_node

226 def s_prime(self):
227     if not self.current_state:
228         return None
229     # s' -> S S' | epsilon
230     new_node = Node("S'")
231     save = globals()["current_token_index"]
232     if new_node.add_child(self.s()):
233         if new_node.add_child(self.s_prime()):
234             pass
235         else:
236             globals()["current_token_index"] = save
237     else:
238         new_node.add_child("epsilon")
239         self.epsilon_flag = 1
240         # TODO Need a double return here so that it get back to s and
241         # starts a new s
242         self.current_state = False
243         return None
244     return new_node

245 def s_double_prime(self):
246     if not self.current_state:
247         return None
248     # S'' -> )S' | S)S'
249     new_node = Node('S''')
250     save = globals()["current_token_index"]
251     if new_node.add_child(self.is_value(self.get_token(), RPAREN)):
252         if new_node.add_child((self.s_prime())):
253             pass
254         else:
255             globals()["current_token_index"] = save
256     elif new_node.add_child((self.s())):
257         if new_node.add_child(self.is_value(self.get_token(), RPAREN)):
258             if new_node.add_child(self.s_prime()):
259                 pass
260             else:
261                 globals()["current_token_index"] = save
262         else:
263             globals()["current_token_index"] = save
264     else:
265         globals()["current_token_index"] = save
266         return None
267     return new_node

268 def expr(self):
269     if not self.current_state:
270         return None
271     # expr -> oper | stmts

```



```

324         globals()["current_token_index"] = l_paren_save
325     else:
326         globals()["current_token_index"] = l_paren_save
327
328     temp_node = new_node
329     if temp_node.add_child(self.unops()):
330         if temp_node.add_child(self.oper()):
331             if temp_node.add_child(self.is_value(self.get_token(),
RPAREN)):
332                 if temp_node.add_child(self.tokens[0]):
333                     new_node = temp_node
334                     return new_node
335                 else:
336                     globals()["current_token_index"] = l_paren_save
337             else:
338                 globals()["current_token_index"] = l_paren_save
339         else:
340             globals()["current_token_index"] = l_paren_save
341     else:
342         globals()["current_token_index"] = l_paren_save
343
344     elif new_node.add_child(self.constants()):
345         pass
346     elif new_node.add_child(self.name()):
347         pass
348     else:
349         self.parse_error("missing left paren constant or name")
350         globals()["current_token_index"] = save
351         return None
352     return new_node
353
354 def binops(self):
355     # binops -> + | - | * | / | % | ^ | = | > | >= | < | <= | != | or |
and
356     if not self.current_state:
357         return None
358     new_node = Node("binops")
359     save = globals()["current_token_index"]
360     if new_node.add_child(self.is_value(self.get_token(), OPER_ADD)):
361         pass
362     elif new_node.add_child(self.is_value(self.get_token(), OPER_SUB)):
363         pass
364     elif new_node.add_child(self.is_value(self.get_token(), OPER_MULT)):
365         pass
366     elif new_node.add_child(self.is_value(self.get_token(), OPER_DIV)):
367         pass
368     elif new_node.add_child(self.is_value(self.get_token(), OPER_MOD)):
369         pass
370     elif new_node.add_child(self.is_value(self.get_token(), OPER_EXP)):
371         pass
372     elif new_node.add_child(self.is_value(self.get_token(), OPER_EQ)):
373         pass
374     elif new_node.add_child(self.is_value(self.get_token(), OPER_LT)):
375         pass
376     elif new_node.add_child(self.is_value(self.get_token(), OPER_LE)):

```

```

378         pass
elif new_node.add_child(self.is_value(self.get_token(), OPER_GT)):
380         pass
elif new_node.add_child(self.is_value(self.get_token(), OPER_GE)):
382         pass
elif new_node.add_child(self.is_value(self.get_token(), OPER_NE)):
384         pass
elif new_node.add_child(self.is_value(self.get_token(), OPER_OR)):
386         pass
elif new_node.add_child(self.is_value(self.get_token(), OPER_AND)):
388         pass
else:
    self.parse_error("missing binop")
    globals()["current_token_index"] = save
    return None
392 return new_node

394 def unops(self):
    # unops -> - | not | sin | cos | tan
    if not self.current_state:
396         return None
    new_node = Node("unops")
    save = globals()["current_token_index"]
    if new_node.add_child(self.is_value(self.get_token(), OPER_NOT)):
400         pass
    elif new_node.add_child(self.is_value(self.get_token(), OPER_SIN)):
402         pass
    elif new_node.add_child(self.is_value(self.get_token(), OPER_COS)):
404         pass
    elif new_node.add_child(self.is_value(self.get_token(), OPER_TAN)):
406         pass
    else:
408         globals()["current_token_index"] = save
        self.parse_error("missing unop")
410         return None
412 return new_node

414 def constants(self):
    # constants -> string | ints | floats
    if not self.current_state:
416         return None
    new_node = Node("constant")
    save = globals()["current_token_index"]
    if new_node.add_child(self.strings()):
420         pass
    elif new_node.add_child(self.ints()):
422         pass
    elif new_node.add_child(self.floats()):
424         pass
    else:
426         globals()["current_token_index"] = save
        return None
428 return new_node
430
def strings(self):

```

```

432     # strings ->    reg-ex for str literal in C ( any alphanumeric )
433     # true | false
434     if not self.current_state:
435         return None
436     new_node = Node("string")
437     save = globals()[ "current_token_index" ]
438     if new_node.add_child( self.is_type( self.get_token() , TYPE_STRING ) ):
439         pass
440     elif new_node.add_child( self.is_type( self.get_token() , TYPE_BOOL ) ):
441         pass
442     else:
443         globals()[ "current_token_index" ] = save
444         return None
445     return new_node
446
447 def name( self ):
448     # name -> reg-ex for ids in C (any lower and upper char
449     # or underscore followed by any combination of lower,
450     # upper, digits, or underscores)
451     if not self.current_state:
452         return None
453     new_node = Node("name")
454     save = globals()[ "current_token_index" ]
455     if new_node.add_child( self.is_type( self.get_token() , TYPE_ID ) ):
456         pass
457     else:
458         globals()[ "current_token_index" ] = save
459         return None
460     return new_node
461
462 def ints( self ):
463     # ints -> reg ex for positive/negative ints in C
464     if not self.current_state:
465         return None
466     new_node = Node("int")
467     save = globals()[ "current_token_index" ]
468     if new_node.add_child( self.is_type( self.get_token() , TYPE_INT ) ):
469         pass
470     else:
471         globals()[ "current_token_index" ] = save
472         return None
473     return new_node
474
475 def floats( self ):
476     # floats -> reg ex for positive/negative doubles in C
477     if not self.current_state:
478         return None
479     new_node = Node("float")
480     save = globals()[ "current_token_index" ]
481     if new_node.add_child( self.is_type( self.get_token() , TYPE_REAL ) ):
482         pass
483     else:
484         globals()[ "current_token_index" ] = save
485         return None
486     return new_node

```

```

488 def stmts(self):
489     # stmts -> ifstmts | whilestmts | letstmts | printstmts
490     if not self.current_state:
491         return None
492     new_node = Node("stmts")
493     save = globals()["current_token_index"]
494     if new_node.add_child(self.ifstmts()):
495         pass
496     elif new_node.add_child(self.whilestmts()):
497         pass
498     elif new_node.add_child(self.letstmts()):
499         pass
500     elif new_node.add_child(self.printstmts()):
501         pass
502     else:
503         self.parse_error("missing if, while, let or print statment")
504         globals()["current_token_index"] = save
505         return None
506     return new_node

508 def printstmts(self):
509     # printstmts -> (stdout oper)
510     if not self.current_state:
511         return None
512     new_node = Node("printstmts")
513     save = globals()["current_token_index"]
514     if new_node.add_child(self.is_value(self.get_token(), LPAREN)):
515         if new_node.add_child(self.is_value(self.get_token(),
KEYWORD.STDOUT)):
516             if new_node.add_child(self.oper()):
517                 if new_node.add_child(self.is_value(self.get_token(),
RPAREN)):
518                     pass
519                 else:
520                     globals()["current_token_index"] = save
521             else:
522                 globals()["current_token_index"] = save
523         else:
524             globals()["current_token_index"] = save
525     else:
526         globals()["current_token_index"] = save
527         self.parse_error("missing left paren")
528         return None
529     return new_node

530 def ifstmts(self):
531     # ifstmts -> (if expr expr expr) | (if expr expr)
532     if not self.current_state:
533         return None
534     new_node = Node("ifstmts")
535     save = globals()["current_token_index"]
536     if new_node.add_child(self.is_value(self.get_token(), LPAREN)):
537         if new_node.add_child(self.expr()):
538             if new_node.add_child(self.is_value(self.get_token(), RPAREN)):

```

```

):
540         # (if expr expr)
541         pass
542     elif new_node.add_child(self.expr()):
543         if new_node.add_child(self.is_value(self.get_token(),
R_PAREN)):
544         # (if expr expr expr)
545         pass
546     else:
547         globals()["current_token_index"] = save
548     else:
549         globals()["current_token_index"] = save
550     else:
551         globals()["current_token_index"] = save
552     else:
553         globals()["current_token_index"] = save
554         self.parse_error("not an if statment")
555         return None
556     return new_node

def whilestmts(self):
558     # whilestmts -> (while expr exprlist)
559     if not self.current_state:
560         return None
561     new_node = Node("whilestmts")
562     save = globals()["current_token_index"]
563     if new_node.add_child(self.is_value(self.get_token(), L_PAREN)):
564         if new_node.add_child(self.is_value(self.get_token(),
KEYWORD_WHILE)):
565             if new_node.add_child(self.expr()):
566                 if new_node.add_child(self.exprlist()):
567                     if new_node.add_child(self.is_value(self.get_token(),
R_PAREN)):
568                     pass
569                 else:
570                     globals()["current_token_index"] = save
571                 else:
572                     globals()["current_token_index"] = save
573                 else:
574                     globals()["current_token_index"] = save
575             else:
576                 globals()["current_token_index"] = save
577         else:
578             globals()["current_token_index"] = save
579             self.parse_error("Not While stmts")
580             return None
581     return new_node

def exprlist(self):
584     # exprlist -> expr | expr exprlist
585     if not self.current_state:
586         return None
587     new_node = Node("exprlist")
588     save = globals()["current_token_index"]
589     if new_node.add_child(self.expr()):

```

```

592         if new_node.add_child(self.exprlist()):
593             pass
594         else:
595             globals()["current_token_index"] = save
596     else:
597         globals()["current_token_index"] = save
598         self.parse_error("not expression list")
599         return None
600     return new_node
601
602 def letstmts(self):
603     # letstmts -> (let (varlist))
604     if not self.current_state:
605         return None
606     new_node = Node("letstmts")
607     save = globals()["current_token_index"]
608     if new_node.add_child(self.is_value(self.get_token(), LPAREN)):
609         if new_node.add_child(self.is_value(self.get_token(), KEYWORDLET)
610 ):
611             if new_node.add_child(self.varlist()):
612                 if new_node.add_child(self.is_value(self.get_token(),
613 R_PAREN)):
614                     pass
615                 else:
616                     globals()["current_token_index"] = save
617             else:
618                 globals()["current_token_index"] = save
619         else:
620             globals()["current_token_index"] = save
621         self.parse_error("Checked if let statement")
622         return None
623     return new_node
624
625 def varlist(self):
626     # varlist -> (name type) | (name type) varlist
627     if not self.current_state:
628         return None
629     new_node = Node("varlist")
630     save = globals()["current_token_index"]
631     if new_node.add_child(self.is_value(self.get_token(), LPAREN)):
632         if new_node.add_child(self.is_type(self.get_token(), TYPE_ID)):
633             if new_node.add_child(self.type()):
634                 if new_node.add_child(self.is_value(self.get_token(),
635 R_PAREN)):
636                     if new_node.add_child(self.varlist()):
637                         return new_node
638                     # (name type)
639                     return new_node
640                 else:
641                     globals()["current_token_index"] = save
642             else:
643                 pass
644             # check of type failed

```

```

        else:
            globals()["current_token_index"] = save
644     else:
645         globals()["current_token_index"] = save
        self.parse_error("not varlist")
648     return None
    return new_node

650 def type(self):
651     # type -> bool | int | real | string
    if not self.current_state:
654         return None
    new_node = Node("type")
    save = globals()["current_token_index"]
    if new_node.add_child(self.is_value(self.get_token(), "bool")):
658         pass
    elif new_node.add_child(self.is_value(self.get_token(), "int")):
660         pass
    elif new_node.add_child(self.is_value(self.get_token(), "real")):
662         pass
    elif new_node.add_child(self.is_value(self.get_token(), "string")):
664         pass
    else:
666         globals()["current_token_index"] = save
        return None
    return new_node
668

```

myparser.py

3.6 lexer.py

```

__author__ = 'drakebridgewater'
2  import string

4  from defines import *

6
    class Lexer():
8        def __init__(self, filename):
            self.line = 1
10           self.filename = filename
            self.file = ''
12           self.current_char = ' '
            self.pointer = 0
14           self.token_list = []
            self.current_state = True # When false throw error
16           self.accepted_ops = ('=', '+', '-', '/', '*', '<', '>', '!', ';', ':',
'%', '(', ')', '^')
            # tokens is a dictionary where each token is a list
18           self.tokens = \
                {"keywords": [KEYWORD.STDOUT, KEYWORD.LET, KEYWORD.IF,
KEYWORD.WHILE,
20                        KEYWORD.TRUE, KEYWORD.FALSE, OPER.ASSIGN],
                "ops": [OPER.ASSIGN, OPER.ADD, OPER.SUB, OPER.DIV, OPER.MULT,
22                        OPER.LT, OPER.GT, OPER.NOT, OPER.MOD, OPER.EXP,
                        OPER.AND, OPER.OR, OPER.NOT, OPER.NE, R.PAREN, L.PAREN],

```



```

24         'type': [TYPE_BOOL, TYPE_INT, TYPE_REAL, TYPE_STRING]
25     }
26
27     def open_file(self):
28         self.file = open(self.filename, 'r')
29
30     def close_file(self):
31         self.file.close()
32
33     def has_token(self, value, key=''):
34         # if subgroup given check it first
35         if key != '':
36             if value in self.tokens[key]:
37                 return key
38
39         # if subgroup checking fails check all entries
40         for x in self.tokens:
41             if value in self.tokens[x]:
42                 return x
43         return -1
44
45     def get_next_char(self):
46         try:
47             self.current_char = self.file.read(1)
48         except EOFError:
49             print("Reached end of file")
50
51     def get_token(self):
52         self.get_next_char()
53         while True and self.current_state:
54             if not self.current_char:
55                 return -1
56             if self.current_char == ' ' or self.current_char == '\t':
57                 self.get_next_char()
58                 pass
59             elif self.current_char == '\n':
60                 self.get_next_char()
61                 self.line += 1
62             elif self.current_char in self.accepted_ops:
63                 return self.is_op()
64             elif self.is_letter():
65                 return self.identify_word() # identify the string and add to
the token list
66             elif self.is_digit():
67                 return self.is_number() # identify the number and add to the
token list
68             elif self.current_char == '"':
69                 return self.create_token(("ops", '"'))
70                 # self.parse_string() # parse a string
71             else:
72                 print("Line:ERROR: Could not identify on line: " + str(
self.line) + " near char: " + self.current_char + '"')
73                 return None
74
75         # TODO have all functions return to a state that has the next

```

```

char

78 # Function Description:
# General function to do something with the tokens once we have classified
  them.
80 def create_token(self, token):
    new_token = Token()
82     new_token.line = self.line
    new_token.type = token[0]
84     new_token.value = token[1]
    return new_token

86
def add_token(self, token):
88     new_token = Token()
    new_token.line = self.line
90     new_token.type = token[0]
    new_token.value = token[1]
92     self.token_list.append(new_token)

94
def print_tokens(self):
    for x in self.token_list:
96         print("[line: " + x.line + ", ID: " + x.type + ", Value: " + x.
value + "]")

98
def is_op(self):
    item = self.current_char
100    # If we see an op look to see if we see another. If we see another add
the previous
    # found op
102    if self.current_char is '+':
        self.get_next_char()
104        return self.create_token((self.has_token(item), item))
    elif self.current_char is '-':
106        self.get_next_char()
        # if self.current_char is '-':
108        # item += self.current_char # Seen — make new token
        # self.get_next_char()
110        return self.create_token((self.has_token(item), item))
    elif self.current_char in ('<', '>', '!'):
112        self.get_next_char()
        if self.current_char == '=':
114            item += self.current_char
            self.get_next_char()
116            return self.create_token((self.has_token(item), item))
    elif self.current_char in ':':
118        self.get_next_char()
        if self.current_char is '=':
120            item += self.current_char
            return self.create_token((self.has_token(item), item))
122        else:
            print("Lexer Error [Line: " + str(
124                self.line) + "] the " + self.current_char + " symbol not
recognized after colon [:] ")
        elif self.current_char in '=':
126            return self.create_token((self.has_token(item), item))

```

```

128         elif self.current_char in ('*', '/', '(', ')', '%', '^'):
129             self.get_next_char()
130             return self.create_token((self.has_token(item), item))
131         else:
132             print("Lexer Error: [Line: " + str(self.line) + "] could not
intemperate: " +
133                 self.current_char)
134             return -1
135
136     def parse_string(self):
137         accepted_chars = [ '"' ]
138         new_string = ''
139         self.get_next_char()
140         while self.current_char in accepted_chars:
141             new_string += self.current_char
142             self.get_next_char()
143         self.token_list.append(("string", new_string))
144
145     def identify_word(self):
146         accepted_chars = list(string.ascii_letters) + list(string.digits) +
list('_', '-')
147         acceptable_first_chars = list(string.ascii_letters)
148
149         word = ''
150         if self.current_char in acceptable_first_chars:
151             word += self.current_char
152             self.get_next_char()
153             while self.current_char in accepted_chars:
154                 word += self.current_char
155                 self.get_next_char()
156             token_value = word
157             token_type = self.has_token(token_value)
158             if token_type == -1:
159                 token_type = "ID"
160             return self.create_token((token_type, token_value))
161
162     # Function Description:
163     # This function should be called when a word identifier or keyword is
started
164     # and will return the full word upon seeing invalid characters.
165     def parse_word(self, accepted_chars, acceptable_first_chars=[]):
166         if self.current_char not in acceptable_first_chars:
167             return -1
168         else:
169             word = ''
170             while self.current_char in accepted_chars:
171                 word += self.current_char
172                 self.get_next_char()
173             return word
174
175     def is_int(self):
176         word = ''
177         while self.is_digit(exclude=['.', 'e']):
178             word += self.current_char
179             self.get_next_char()

```

```

180         return word

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

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

229             elif self.current_char is '-':
230                 self.get_next_char()
231                 exp = self.is_int()
232                 try:

```

```

230         self.get_next_char()
231         exp = int(exp)
232         word += 'e-'
233         word += str(exp)
234         try:
235             return self.create_token(("float", float(word)))
236         except ValueError:
237             print("Fatal parse error: [row: " +
238                   str(self.line) + "] when parsing char '" +
239                   str(self.current_char) + "' for: \n\t\t" +
240                   str(word))
241         except ValueError:
242             return [self.create_token(("int", word)),
243                     self.create_token(("ID", "e")),
244                     self.create_token((self.has_token("-"), "-"))]
245         else:
246             exp = self.is_int()
247             try:
248                 exp = int(exp)
249                 word += str(exp)
250                 return self.create_token(("float", float(word)))
251             except ValueError:
252                 print("Lexer Error: [row: " + str(self.line) + "]
253                 Unable to parse '" +
254                       str(self.current_char) + "' in: " + str(exp))
255             elif self.is_digit(other_accepted):
256                 word += self.current_char
257                 self.get_next_char()
258             else:
259                 break
260             if 'e' not in other_accepted:
261                 other_accepted.append('e')
262             if '.' in word or 'e' in word:
263                 try:
264                     return self.create_token(("float", float(word)))
265                 except ValueError:
266                     print("Lexer Error (line: " + str(self.line) +
267                           "): could not determine numerical token of: " + str(word))
268             else:
269                 try:
270                     return self.create_token(("int", int(word)))
271                 except ValueError:
272                     print("Lexer Error (line: " + str(self.line) +
273                           "): could not determine numerical token of: " + str(word))
274
275     # Function Description:
276     # checks to see if the current token in peek is a digit or '.'
277     # return true if it is
278     def is_digit(self, others=[], exclude=[]):
279         digits = ['.', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
280         for x in others:
281             if x not in digits:

```

```

        digits.append(x)
282     for x in exclude:
        if x in digits:
284         digits.remove(x)
        if self.current_char in digits:
286             return True
        return False
288
# Function Description:
# checks to see if the current token in peek is a letter
# return true if it is
292 def is_letter(self, others=[]):
    letters = list(string.ascii_letters)
294     for x in others:
        if x not in letters:
296         letters.append(x)
        if self.current_char in letters:
298             return True
    return False

```

lexer.py