

OREGON STATE UNIVERSITY

CS 352 - TRANSLATORS

WINTER 2015

---

## Milestone 5: Loops and Variables/Local Values

---

*Author:*  
Drake Bridgewater

*Professor:*  
Dr. Jennifer PARHAM-MOCELLO

DUE 03/15/15 (11:59pm)  
March 15, 2015

# Contents

<b>1</b>	<b>Source Code Descriptions</b>	<b>2</b>
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
<b>2</b>	<b>Report</b>	<b>4</b>
2.1	Develop a formal definition of the code generation algorithm . . . . .	4
2.2	Test the resulting program for correctness based on the formal definition . . . . .	4
<b>3</b>	<b>README</b>	<b>5</b>
<b>4</b>	<b>Source Code</b>	<b>6</b>
4.1	main.py . . . . .	6
4.2	codegen.py . . . . .	7
4.3	node.py . . . . .	18
4.4	defines.py . . . . .	20
4.5	myparser.py . . . . .	21
4.6	lexer.py . . . . .	33

# 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 harder 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 took some single statements and got them to parse all the way through. Once I had those working I was able to do multiple statements per file. **To solve** this problem I need to understand gforth and my code. Therefore I used my code to figure out how the gforth worked because gforth is difficult to grasp. As I was going I was able to see tokens that needed to be moved and because I was looking for full statements before converting to gforth I was able to reorganize easily.

### 2.1 Develop a formal definition of the code generation algorithm

How I implemented this was probably not the best way but I found it to work in many cases. I chose to implement this code generation by reading in the tokens from the tree as I was expecting them, post order traversal. This allowed me to see when there was an error as I only expect it to work this way. ie. if I have a let statement then it will be followed by the varlist. Once I saw enough to determine the output of that command I was able to produce the gforth code. Here the idea was to take the token I have seen and reorganize them so that they are in the post fix language, gforth.

### 2.2 Test the resulting program for correctness based on the formal definition

This is a very hard concept that I feel OSU needs to focus on more, but my idea was create test cases that contain single statements. Once a statement passed I moved to the next one. Having all the statements implemented before was helpful but I missed may corner cases because I did not spend enough time thinking about the corners and test cases. The next step was combing statements and testing them.

### 3 README

```
*****  
*****README*****  
*****
```

This assignment is written in python3 with the following usage:

```
python3 main.py [option] [files]  
-d or -debug      Show lots of debug stuff  
                  (development only)  
-lexer            Show lexer output (not helpful, but  
                  shows how nodes are created)  
-postorder        Show post order traversal of the tree  
-print            Show the gforth code  
-tree             Show the parse tree
```

With no arguments this will run test1 by default

```
*****USING THE MAKEFILE*****  
Just running make will create the pdf with latex, compile  
(but python does not need to be compiled), clean up  
unnecessary files and finally run stutest.
```

stutest contains three test cases to ensure that operations work as planned

## 4 Source Code

### 4.1 main.py

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

5  from myparser import *
   from codegen import *
7
9  usage = """
   Usage:
11  main.py [option] [files]
    -d or -debug\t\t Show lots of debug stuff (development only)
13  -tree \t\t\t Show the parse tree
    -postorder \t\t\t Show the post order traversal of the tree
15  -lexer \t\t\t Show the lexer output (not helpful, but shows how nodes are
    created)
    -print \t\t\t Show the gforth code
17
    With no arguments this will run test1 by default
19 """

21 files = []
    options = []
23
25 global current_token_index
    current_token_index = 0

27
29 def read_file(input_file):
    content = ""
    f = open(input_file)
31
    lines_raw = f.readlines()
33  for i in range(0, len(lines_raw)):
    content += lines_raw[i]
35
    return content
37

39 def print_verbose(selected_file):
    print_title("IBTL file: " + selected_file)
41  print('\n', "input: parsing " + str(selected_file))
    with open(selected_file, "r") as file:
43  text = file.read()
    print(text)
45

47 def main():
    sys.setrecursionlimit(100)
49  print("-"*80)
    if len(sys.argv) > 1:
```

```

51     filename = sys.argv[ len(sys.argv) - 1]
    else:
53         filename = "test1"

55     if '-help' in sys.argv or '-usage' in sys.argv or '-h' in sys.argv:
        print(usage)
        return 0
57     if '-d' in sys.argv or '-debug' in sys.argv:
        globals()[ 'DEBUG' ] = 1
        globals()[ 'OPTIONS' ].append("DEBUG")
59     else:
        globals()[ 'DEBUG' ] = 0
61     if '-input' in sys.argv:
        print_verbose( filename)
63     if '-tree' in sys.argv:
        globals()[ 'OPTIONS' ].append("tree")
65     if '-postorder' in sys.argv:
        globals()[ 'OPTIONS' ].append("postorder")
67     if '-print' in sys.argv:
        globals()[ 'OPTIONS' ].append("print")
69     if '-lexer' in sys.argv:
        globals()[ 'OPTIONS' ].append("lexer")
71
73
75     parser = MyParser(filename)
    parser.control()
77     tree = parser.tree
    tree.print_postordered_tree()
79     codegen = CodeGen(tree)
    gforth = codegen.control()
81
    outfile = filename.split(".")[0] + ".out" + ".fs"
83     print("\n\nngforth code in: " + outfile)
    with open(outfile, "w+") as file:
85         for x in gforth:
            file.write(str(x) + " ")
87
89
91 if __name__ == '__main__':
    main()

```

main.py

## 4.2 codegen.py

```

__author__ = 'drakebridgewater'
2 from defines import *
from myparser import *
4
6 class Scope:
    paren = ''
8     prev_was_string = False
    prev_was_real = False

```



```

10
12 class CodeGen(object):
13     def __init__(self, tree):
14         self.error_flag = False
15         self.tree = tree
16         self.current_token = None
17         self.stack = []
18         self.scope_stack = []
19         self.current_scope = []
20         self.variables = []
21         self.gforth = []
22         self.index = 0
23         self.pointer = 0
24         self.other_tokens = {
25             # KEYWORD_STDOUT: '.s',
26             # KEYWORD_LET: 'let',
27             KEYWORD_IF: 'if',
28             KEYWORD_WHILE: 'while',
29             KEYWORD_TRUE: "true",
30             KEYWORD_FALSE: "false",
31             # TYPE_BOOL: 'bool',
32             # TYPE_INT: 'int',
33             # TYPE_REAL: 'float',
34             # TYPE_STRING: 'string',
35         }
36         self.conversions = {
37             "%": "mod",
38             "!=": "<>",
39         }
40         self.realConversions = {
41             "+": "f+",
42             "-": "f-",
43             "*": "f*",
44             "/": "f/",
45             "mod": "fmod",
46             "<": "f<",
47             "<=": "f<=",
48             ">": "f>",
49             ">=": "f>=",
50             "=": "f=",
51             "<>": "f<>",
52             "sin": "fsin",
53             "cos": "fcos",
54             "tan": "ftan",
55         }
56
57     def control(self):
58         # TODO as we step through the tree convert and push element on to
59         # stack
60         self.get_tokens_stack()
61         if '-print' in globals()['OPTIONS']:
62             print_title("CODE_GEN — in progress")
63         self.print_stack()
64         self.write_out()

```

```

64     self.gforth = self.rem_sll(self.gforth, LPAREN)
        self.gforth = self.rem_sll(self.gforth, RPAREN)
66
        if 'print' in globals()['OPTIONS']:
68             print_title("gforth code")
            for x in self.gforth:
70                 print(x, end=" ")
            return self.gforth
72
def rem_sll(self, L, item):
74     answer = []
    for i in L:
76         if i != item:
            answer.append(i)
78     return answer

80 @staticmethod
def out(msg):
82     print(msg)

84 def write_out(self):
    data = self.stack
86     prev_was_string = False
    prev_was_real = False
88     prev_was_int = False
    prev_was_assign = False
90     prev_was_var = False
    prev_was_if = False
92     prev_was_declare = False
    prev_was_stdout = False
94     append_end = False
    oper_hold = []

96     while self.pointer <= len(self.stack) - 1:
98         try:
            convert = False

100
            if prev_was_real:
102                 if data[self.pointer].value in self.realConversions:
                    oper_hold.append(self.realConversions[data[self.
pointer].value])
104                 if data[self.pointer].type == TYPE_INT:
                    convert = True

106
            if data[self.pointer].type == 'ops':
108                 if data[self.pointer].value is LPAREN:
                    self.gforth.append(LPAREN)
                    self.scope_stack.append(LPAREN)
110                 elif data[self.pointer].value is RPAREN:
                    self.gforth.append(RPAREN)
112                 try:
                    if self.gforth[len(self.gforth) - 1] != '\n':
                        self.gforth.append('\n')
                    pop_value = self.scope_stack.pop()
                    if 'whileloop' in pop_value:

```

```

118         temp = pop_value.split("whileloop")
        self.gforth.append("whileloop" + temp[len(temp)
) - 1] + "\n")
120         if 'ifloop' in pop_value:
            temp = pop_value.split("ifloop")
122         self.gforth.append("ifloop" + temp[len(temp) -
1] + "\n")

        prev_was_if = False
124         prev_was_string = False
        prev_was_real = False
126         prev_was_int = False
        prev_was_assign = False
128         prev_was_var = False
        prev_was_declare = False
130         prev_was_stdout = False
        append_end = False
132         oper_hold = []
        except IndexError:
134             print_error("missing left paren [d]", error_type="
code-gen")
            elif data[self.pointer].value in ['+', '-', '/', '*', '<',
'>', '!', ';', ':', '%', '^']:
136                 self.pointer = self.is_math_expr(self.pointer)
                # self.gforth.append(str(data[self.pointer].value))
138             elif data[self.pointer].type == TYPE_ID:
                if self.stack[self.pointer].value not in self.variables:
140                     print_error("variable " + str(self.stack[self.pointer
].value) + " not declared before use")
                    if data[self.pointer].value in self.current_scope:
142                         pass
                    elif prev_was_declare:
144                         if data[self.pointer].value in self.variables:
                            print_error("redecloration of variable " + str(
data[self.pointer].value),
146                                     error_type="codegen")
                            self.current_scope.append(data[self.pointer].value)
                            # oper_hold = str(data[self.pointer].value)
148                         else:
                            print_error("unassigned variable " + str(data[self.
pointer].value), error_type="codegen")
                            self.gforth.append(str(data[self.pointer].value))
150                         if prev_was_assign:
                            self.gforth.append("!")
152                             # oper_hold.append(str(data[self.pointer].value))
                            # oper_hold.append('!')
154                             # self.variables.append(data[self.pointer].value)

156             elif data[self.pointer].type == 'keywords':
                if data[self.pointer].value == OPER_ASSIGN: # :=
158                     self.pointer = self.is_assign(self.pointer)
                    prev_was_assign = True
160                 elif data[self.pointer].value == KEYWORD_STDOUT:
                    self.pointer = self.is_stdout(self.pointer)
                    prev_was_stdout = True
162                 pass

```

```

166         elif data[self.pointer].value == KEYWORDLET: # Declare
167             self.pointer = self.is_let(self.pointer)
168             # prev_was_declare = True
169             # self.gforth.append(" variable")
170             pass
171         elif data[self.pointer].value == KEYWORDIF:
172             self.pointer = self.is_ifstmt(self.pointer)
173             # if_stmts = "ifloop" + str(self.pointer)
174             # self.scope_stack.append(if_stmts)
175             # self.gforth.append(" : " + if_stmts)
176         elif data[self.pointer].value == KEYWORDWHILE:
177             self.pointer = self.is_whilestmt(self.pointer)
178             # while_stmts = "whileloop" + str(self.pointer)
179             # self.scope_stack.append(while_stmts)
180             # self.gforth.append(" : " + while_stmts)
181         elif data[self.pointer].type == TYPESTRING: # An actual
string
182             self.gforth.append('s' + str(data[self.pointer].value) +
183                                ' ')
184         elif data[self.pointer].type == TYPEINT:
185             prev_was_int = True
186             self.gforth.append(str(data[self.pointer].value))
187             pass
188         elif data[self.pointer].type == TYPEREAL:
189             prev_was_real = True
190             self.gforth.append(str(data[self.pointer].value) + "e0")
191             pass
192
193         if append_end:
194             for op in oper_hold:
195                 self.gforth.append(op)
196
197         finally:
198             self.pointer += 1
199         if len(self.scope_stack) != 0:
200             print_error("missing right paren [c]", error_type="code_gen")
201             if globals()['DEBUG'] == 1:
202                 print_title("Scope stack should be empty but has")
203                 print(self.scope_stack)
204             self.gforth.append("cr cr bye")
205
206     def is_let(self, x):
207         temp_scope = []
208         if self.stack[x].value == KEYWORDLET:
209             x += 1
210             if self.stack[x].value == LPAREN: # let *( ( x int) )
211                 temp_scope.append(LPAREN)
212                 x += 1
213                 while self.stack[x].value == LPAREN:
214                     temp_scope.append(LPAREN)
215                     x += 1
216                 self.gforth.append(" variable")
217                 if self.stack[x].type == TYPEID:
218                     self.gforth.append(self.stack[x].value)
219                     x += 1

```

```

220         if self.stack[x].value == TYPE_REAL:
221             self.variables.append(self.stack[x-1])
222             x += 1
223         elif self.stack[x].value == TYPE_INT:
224             self.variables.append(self.stack[x-1])
225             x += 1
226         elif self.stack[x].value == TYPE_STRING:
227             self.variables.append(self.stack[x-1])
228             x += 1
229         else:
230             print_error("let statement error", error_type="
codegen")
231
232         if self.stack[x].value == R_PAREN:
233             x += 1
234             temp_scope.pop()
235         if self.stack[x].value == R_PAREN:
236             if len(temp_scope) > 0:
237                 temp_scope.pop()
238                 x += 1
239             if len(temp_scope) != 0:
240                 print_error("expected to see final closing bracket on let stmt",
error_type="codegen")
241             return x
242
243     def is_stdout(self, x, if_stmts=False):
244         if self.stack[x].value == KEYWORD_STDOUT:
245             x += 1
246             if self.stack[x].value == L_PAREN:
247                 self.gforth.append(L_PAREN)
248                 x += 1
249                 x = self.is_math_expr(x)
250                 if not if_stmts:
251                     self.gforth.append(".")
252                 x += 1
253             if self.stack[x].value == R_PAREN:
254                 self.gforth.append(R_PAREN)
255             elif self.stack[x].type == TYPE_STRING:
256                 self.gforth.append('s' + self.stack[x].value + '')
257                 if not if_stmts:
258                     self.gforth.append(".s")
259             elif self.stack[x].type == TYPE_INT:
260                 self.gforth.append(str(self.stack[x].value))
261                 if not if_stmts:
262                     self.gforth.append(".s")
263             elif self.stack[x].type == TYPE_REAL:
264                 self.gforth.append(str(self.stack[x].value) + "e0")
265                 if not if_stmts:
266                     self.gforth.append("f.s")
267             return x
268
269     def is_assign(self, x):
270         oper_hold = ''
271         if self.stack[x].value == OPER_ASSIGN:
272             x += 1

```

```

272         if self.stack[x].type == TYPE_ID:
273             x += 1
274             if self.stack[x].value == L_PAREN:
275                 oper_hold = self.stack[x-1].value
276                 x += 1
277                 x = self.is_math_expr(x)
278                 if self.stack[x].value == R_PAREN:
279                     x += 1
280                     self.gforth.append(oper_hold) # append ID
281             elif self.stack[x].type == TYPE_STRING:
282                 self.gforth.append('s' + self.stack[x-1].value + '')
283                 self.gforth.append(self.stack[x-1].value)
284             elif self.stack[x].type == TYPE_INT:
285                 self.gforth.append(str(self.stack[x].value))
286                 self.gforth.append(self.stack[x-1].value)
287             elif self.stack[x].type == TYPE_REAL:
288                 self.gforth.append(str(self.stack[x].value) + "e0")
289                 self.gforth.append(self.stack[x-1].value)
290             elif self.stack[x].type == TYPE_ID:
291                 self.gforth.append(str(self.stack[x].value))
292                 self.gforth.append(self.stack[x-1].value)
293             else:
294                 print_error("in assignment missing value", error_type="
codegen")

296         self.gforth.append("!")
297         if oper_hold != '':
298             self.gforth.append(oper_hold)
299             self.gforth.append("@")
300         return x

302     def is_math_expr(self, x):
303         if self.stack[x].value in ['+', '-', '/', '*', '<', '>', '!', ';', ':',
, '%', '^']:
304             x += 1
305             if self.stack[x].type == TYPE_INT:
306                 x += 1
307                 if self.stack[x].type == TYPE_INT: # int int
308                     self.gforth.append(self.stack[x-1].value)
309                     self.gforth.append(self.stack[x].value)
310                     self.gforth.append(self.stack[x-2].value)
311                 elif self.stack[x].type == TYPE_REAL: # int real
312                     self.gforth.append(self.stack[x-1].value)
313                     self.gforth.append(str(self.stack[x].value) + "e0")
314                     self.gforth.append("fswap")
315                     self.gforth.append("s>f")
316                     self.gforth.append(self.realConversions[self.stack[x-2].
value])
317             elif self.stack[x].type == TYPE_ID:
318                 # TODO check if the variable exists
319                 self.gforth.append(self.stack[x-1].value) # value
320                 if self.stack[x].value not in self.variables:
321                     print_error("variable " + str(self.stack[x].value) + "
not declared before use")
322                 self.gforth.append(self.stack[x].value) # variable

```

```

324         self.gforth.append("@")
325         self.gforth.append(self.stack[x - 2].value)
326     else:
327         print_error("expected another int/real/string", error_type
="codegen")
328     elif self.stack[x].type == TYPE_REAL:
329         x += 1
330         if self.stack[x].type == TYPE_INT: # real int
331             self.gforth.append(str(self.stack[x - 1].value) + "e0")
332             self.gforth.append(self.stack[x].value)
333             self.gforth.append("fswap")
334             self.gforth.append("s>f")
335             self.gforth.append(self.realConversions[self.stack[x - 2].
value])
336         elif self.stack[x].type == TYPE_REAL: # real real
337             self.gforth.append(str(self.stack[x - 1].value) + "e0")
338             self.gforth.append(str(self.stack[x].value) + "e0")
339             self.gforth.append(self.realConversions[self.stack[x - 2].
value])
340         elif self.stack[x].type == TYPE_ID:
341             # TODO check if the variable exists
342             self.gforth.append(self.stack[x - 1].value) # value
343             if self.stack[x].value not in self.variables:
344                 print_error("variable " + str(self.stack[x].value) + "
not declared before use")
345             self.gforth.append(self.stack[x].value) # variable
346             self.gforth.append("@")
347             self.gforth.append(self.stack[x - 2].value)
348         else:
349             print_error("expected another int/real/string", error_type
="codegen")
350     elif self.stack[x].type == TYPE_STRING:
351         x += 1
352         if self.stack[x].type == TYPE_STRING:
353             self.gforth.append('s' + self.stack[x - 1].value + "'")
354             self.gforth.append('s' + self.stack[x].value + "'")
355             if self.stack[x - 2].value == "+":
356                 self.gforth.append("s" + self.stack[x - 2].value)
357             else:
358                 print_error("only string concatenation is supported",
error_type="codegen")
359         elif self.stack[x].type == TYPE_ID:
360             if self.stack[x].value not in self.variables:
361                 print_error("variable " + str(self.stack[x].value) + " not
declared before use")
362             self.gforth.append(self.stack[x].value)
363             self.gforth.append("@")
364             x += 1
365             if self.stack[x].type == TYPE_INT:
366                 self.gforth.append(self.stack[x].value)
367             elif self.stack[x].type == TYPE_REAL:
368                 self.gforth.append(self.stack[x].value)
369             x += 1
370             self.gforth.append(self.stack[x - 3].value)
371         if self.stack[x].value == RPAREN:

```

```

372         self.gforth.append(RPAREN)
373         self.scope_stack.pop()
374     return x
375
376 def is_whilestmt(self, x):
377     while_stmts = ''
378     if self.stack[x].value == KEYWORD_WHILE:
379         x += 1
380         if self.stack[x].value == LPAREN:
381             x += 1
382             if self.stack[x].value in ['<', '>', '=<', '=>', '!',
KEYWORD_TRUE, KEYWORD_FALSE]:
383                 x += 1
384                 if self.stack[x].type == TYPE_INT or self.stack[x].type ==
TYPE_REAL or \
385                     self.stack[x].type == TYPE_ID:
386                     x += 1
387                     if self.stack[x].type == TYPE_INT or self.stack[x].
type == TYPE_REAL or \
388                         self.stack[x].type == TYPE_ID:
389                         while_stmts = "while_stmts" + str(x - 4)
390                         self.scope_stack.append(LPAREN)
391                         self.scope_stack.append(while_stmts)
392
393                         temp_stack = []
394                         # self.scope_stack.append(LPAREN)
395                         temp_stack.append(while_stmts)
396                         temp_stack.append(LPAREN)
397                         self.scope_stack.append(LPAREN) # Because we
enter this function at the 'if'
398
399                         self.gforth.append(": " + while_stmts)
400                         self.gforth.append(LPAREN)
401                         self.gforth.append("BEGIN")
402                         if self.stack[x-1].type == TYPE_ID and self.stack[
x-1].value not in self.variables:
403                             print_error("variable " + str(self.stack[x-1].
value) + " not declared before use")
404                         self.gforth.append(self.stack[x - 1].value) # x
self.gforth.append("@")
405                         if self.stack[x].type == TYPE_ID and self.stack[x
].value not in self.variables:
406                             print_error("variable " + str(self.stack[x].
value) + " not declared before use")
407                         self.gforth.append(self.stack[x].value) # 3
408                         self.gforth.append(self.stack[x - 2].value) # <
409                         self.gforth.append("while")
410                         x += 1
411
412                     if self.stack[x].value == RPAREN:
413                         self.scope_stack.pop()
414                         x += 1
415                         stack_val = temp_stack.pop()
416                         if stack_val != LPAREN:
417                             print_error("incorrect token", error_type=

```



```

"codegen")
418                                     else:
420                                         self.gforth.append(stack_val)
422                                         while self.stack[x].value != R_PAREN:
424                                             temp = self.is_while_internals(x)
426                                             if temp == -1:
428                                                 break
429                                             else:
430                                                 x = temp
431                                                 self.gforth.append("TYPE ")
432                                         self.scope_stack.pop()
433                                         self.gforth.pop() # remove the last 'type
' from gforth

    self.gforth.append("REPEAT")
    self.gforth.append(";")
    var = temp_stack.pop()
    self.gforth.append("\n" + var + "\n")
434 return x

def is_while_internals(self, x):
436     if self.stack[x].value == L_PAREN:
438         x += 1
439         x = self.is_stdout(x, True)
440         x = self.is_math_expr(x)
441         x = self.is_assign(x)
442
443         x += 1
444         if self.stack[x].value == R_PAREN:
445             x += 1
446     else:
447         print_error("missing left paren [e]", error_type='codegen')
448         return -1
449     return x

def is_ifstmt(self, x):
450     if_stmts = ''
451     if self.stack[x].value == KEYWORD_IF:
452         x += 1
453         if self.stack[x].value == L_PAREN:
454             x += 1
455             if self.stack[x].value in ['<', '>', '=<', '=>', '!'],
KEYWORD_TRUE, KEYWORD_FALSE]:
456                 x += 1
457                 if self.stack[x].type == TYPE_INT or self.stack[x].type ==
TYPE_REAL or \
458                     self.stack[x].type == TYPE_ID:
459                     x += 1
460                     if self.stack[x].type == TYPE_INT or self.stack[x].
type == TYPE_REAL or \
461                         self.stack[x].type == TYPE_ID:
462                         if_stmts = "ifloop" + str(x - 4)
463                         self.scope_stack.append(if_stmts)
464
465     temp_stack = []

```

```

468         # self.scope_stack.append(LPAREN)
         temp_stack.append(if_stmts)# Because we enter this
function at the 'if'
470         temp_stack.append(LPAREN)

472         self.gforth.append(": " + if_stmts)
         self.gforth.append(LPAREN)
474         if self.stack[x-1].type == TYPE_ID and self.stack[
x-1].value not in self.variables:
             print_error("variable " + str(self.stack[x-1].
value) + " not declared before use")
476         self.gforth.append(self.stack[x - 1].value) # x
         self.gforth.append("@")
478         if self.stack[x].type == TYPE_ID and self.stack[x
].value not in self.variables:
             print_error("variable " + str(self.stack[x].
value) + " not declared before use")
480         self.gforth.append(self.stack[x].value) # 3
         self.gforth.append(self.stack[x - 2].value) # <
482         self.gforth.append("if")
         x += 1

484         if self.stack[x].value == RPAREN:
486             x += 1
             stack_val = temp_stack.pop()
488             if stack_val != LPAREN:
                 print_error("incorrect token", error_type=
"codegen")

490             else:
                 self.gforth.append(stack_val)
492                 while self.stack[x].value != RPAREN:
                     x = self.is_if_internals(x)
494                     self.gforth.append("TYPE else")
                     self.scope_stack.pop()

496                 self.gforth.append("then")
                 self.gforth.append(";")
498                 var = temp_stack.pop()
                 self.gforth.append("\n" + var + "\n")
500         return x

502 def is_if_internals(self, x):
         if self.stack[x].value ==LPAREN:
504             x += 1
             x = self.is_stdout(x, True)
506             x = self.is_math_expr(x)

508             x += 1
             if self.stack[x].value == RPAREN:
510                 x += 1
                 pass
512             else:
                 print_error("missing right paren [b]", error_type='codegen')
514         else:
             print_error("missing left paren", error_type='codegen')
516         return x

```

```

518     def get_tokens_stack(self):
519         self._get_token_stack(self.tree)
520         self.print_stack()

522     def _get_token_stack(self, node):
523         for child in node.children:
524             self._get_token_stack(child)
525             temp_token = self.is_token(child)
526             if temp_token:
527                 self.stack.append(temp_token)

528     def is_token(self, child):
529         if isinstance(child, int):
530             return None
531         elif isinstance(child, str):
532             return None
533         elif hasattr(child, "data"):
534             if hasattr(child.data, "value"):
535                 return child.data
536             else:
537                 return None
538         elif hasattr(child, "value"):
539             return child
540         else:
541             return None

542     def print_stack(self):
543         if globals()["DEBUG"] == 1:
544             print_title("print stack called")
545             for token in self.stack:
546                 print_token(token)

```

codegen.py

### 4.3 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)

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) +

```

```

76         ", ID: " + child.data.type +
        ", Value: " + str(child.data.value) + "]"
78     else:
79         print("\t" * indent + str(child.data))
80         self.print_tree_helper(child, indent)
81     elif hasattr(child, "value"):
82         print("\t" * indent + "[line: " + str(child.line) +
        ", ID: " + child.type +
        ", 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.
type + ", Value: " + str(
100             child.data.value) + "]")
node.py

```

## 4.4 defines.py

```

__author__ = 'Drake'
2
files = []
4 global OPTIONS
globals()["OPTIONS"] = []
6
global DEBUG
8 globals()["DEBUG"] = 0
10
11 if not 'current_token_index' in globals():
12     current_token_index = 0
13
14 OPER_EQ = '='
15 OPER_ASSIGN = ':= '
16 OPER_ADD = '+'
17 OPER_SUB = '-'
18 OPER_DIV = '/'
19 OPER_MULT = '*'
20 OPER_LT = '<'
21 OPER_GT = '>'
22 OPER_LE = '<='
23 OPER_GE = '>='
24 OPER_NE = '!='
OPER_NOT = '!'

```

```

OPER_MOD = '%'
26 OPER_EXP = '^'
SEMI = ';'
28 L_PAREN = '('
R_PAREN = ')'
30 OPER_AND = 'and'
OPER_OR = 'or'
32 OPER_NOT = 'not'
OPER_SIN = 'sin'
34 OPER_TAN = 'tan'
OPER_COS = 'cos'
36 KEYWORD_STDOUT = 'stdout'
KEYWORD_LET = 'let'
38 KEYWORD_IF = 'if'
KEYWORD_WHILE = 'while'
40 KEYWORD_TRUE = "true"
KEYWORD_FALSE = "false"
42 KEYWORD = 'keywords'
TYPE_BOOL = 'bool'
44 TYPE_INT = 'int'
TYPE_REAL = 'real'
46 TYPE_STRING = 'string'
TYPE_ID = 'ID'
48

def print_error(msg, line='NA', error_type='general'):
    print(error_type.upper() + " ERROR: [line: " + str(line) + "]" + msg)
52

def print_title(msg):
    print("-" * 40 + "\n" + msg.upper() + "\n" + "-" * 40)
56

def print_token(token, indent=0):
    print("\t" * indent + "[line: " + str(token.line) +
60         "\t ID: " + token.type +
        "\t Value: " + str(token.value) +
62         "# \t Siblings: " + str(token.siblings) +
        "]" )
64

def print_log(msg):
    if 'lexer' in globals()[ 'OPTIONS' ]:
68         print(msg)

70
class Token:
72     type = ''
    value = ''
74     line = ''
    siblings = -1

```

defines.py

## 4.5 myparser.py

```

1  __author__ = 'drakebridgewater'
   from lexer import *
3  from defines import *

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

15
           def add_child(self, obj):
17               if obj is None:
                   return obj
19               self.children.append(obj)
               return True

21
           def print_tree(self):
23               if globals()['DEBUG'] == 1 or 'tree' in globals()['OPTIONS']:
                   print_title("print tree")
25                   self.print_tree_helper(self)

27
           def print_tree_helper(self, node, indent=0):
               indent += 1
29               for child in node.children:
                   if hasattr(child, "data"):
31                       if hasattr(child.data, "value"):
                           print_token(child.data, indent)
33                       else:
                           print("\t" * indent + str(child.data))
35                           self.print_tree_helper(child, indent)
                   elif hasattr(child, "value"):
37                       print_token(child.data, indent)
                   elif isinstance(child, int):
39                       print("\t" * indent + str(child))
                   elif isinstance(child, str):
41                       print("\t" * indent + str(child))
                   else:
43                       print("Error in print_tree_helper")
                           print(child)
45                       return
                           # else:
47                       # print("Failed")

49
           def print_postordered_tree(self):
               if globals()['DEBUG'] == 1 or 'postorder' in globals()['OPTIONS']:
51                   print_title("post ordered tree ")
                           self.post_order_tree_print(self)

53
           def post_order_tree_print(self, node):
55               for child in node.children:

```

```

57         self.print_child(child, 0)
58         self.post_order_tree_print(child)
59
60     def print_child(self, child, indent):
61         if isinstance(child, int):
62             print("\t" * indent + str(child))
63         elif isinstance(child, str):
64             print("\t" * indent + str(child))
65         elif hasattr(child, "data"):
66             if hasattr(child.data, "value"):
67                 print_token(child.data)
68             else:
69                 print("\t" * indent + str(child.data))
70         elif hasattr(child, "value"):
71             print_token(child)
72         else:
73             print("Error in print_tree_helper")
74             print(child)
75             return False
76         return True
77
78 class MyParser(object):
79     def __init__(self, filename):
80         temp_token = Node("EMPTY")
81         self.tree = Node(temp_token)
82         self.lexer = Lexer(filename)
83         self.current_state = True
84         self.tokens = []
85         self.line = 1
86         self.epsilon_flag = 0
87
88     def parse_error(self, msg=''):
89         if 'parse' in globals()['OPTIONS']:
90             print_error(msg, self.line, "parse")
91
92     # Function Description:
93     # will return a single token as the lexer may spit out multiple
94     # will return a single token as the lexer may spit out multiple
95     def get_token(self):
96         # if not self.tokens:
97             new_token = self.lexer.get_token()
98             if new_token is not -1:
99                 self.tokens.append(new_token)
100             if self.tokens[len(self.tokens) - 1] == -1:
101                 return None
102             if len(self.tokens) <= globals()['current_token_index']:
103                 # if self.tokens[len(self.tokens) - 1] == -1:
104                     self.current_state = False # Done reading file
105                 return None
106             else:
107                 self.line = self.tokens[globals()['current_token_index']].line
108                 return self.tokens[globals()['current_token_index']]
109
110     def print_tokens(self):

```



```

111         try:
112             self.lexer.open_file()
113             while self.get_token():
114                 print_token(self.tokens)
115         finally:
116             self.lexer.close_file()
117
118     def control(self):
119         try:
120             self.lexer.open_file()
121             if '-lexer' in globals()[ "OPTIONS" ]:
122                 print_title("lexer output")
123             # while self.current_state:
124             self.tree.add_child(self.t())
125             # globals()[ 'current_token_index' ] += 1
126             self.tree.print_tree()
127             if globals()[ 'current_token_index' ] > len(self.tokens):
128                 # if self.tokens[len(self.tokens) - 1] == -1:
129                 self.current_state = False # Done reading file
130             if len(self.tokens) == 0:
131                 return None
132         finally:
133             self.lexer.close_file()
134
135     def is_type(self, token, compare):
136         if not self.current_state:
137             return None
138         if isinstance(token, int):
139             return None
140         if token.type == compare:
141             globals()[ 'current_token_index' ] += 1
142             return Node(token)
143         else:
144             return None
145
146     def is_value(self, token, compare):
147         if not self.current_state:
148             return None
149         if token is None:
150             return None
151         if token.value == compare:
152             globals()[ 'current_token_index' ] += 1
153             return Node(token)
154         else:
155             return None
156
157     def t(self):
158         # T → (T)
159         new_node = Node("T")
160         save = globals()[ "current_token_index" ]
161         if new_node.add_child(self.is_value(self.get_token(), LPAREN)):
162             while new_node.add_child(self.s()):
163                 pass
164             if new_node.add_child(self.is_value(self.get_token(), RPAREN)):
165                 pass

```

```

167         else:
168             self.parse_error("could not find grammar in T")
169             globals()["current_token_index"] = save
170             return None
171         return new_node
172
173     def s(self):
174         # S → [S' | Oper3 S | Oper3
175         new_node = Node("S")
176         save = globals()["current_token_index"]
177         if new_node.add_child(self.is_value(self.get_token(), LPAREN)) \
178             and new_node.add_child(self.s_prime()):
179             print_log("FOUND: (S' ")
180         elif new_node.add_child(self.oper()) \
181             and new_node.add_child(self.s()):
182             print_log("FOUND: oper3 S")
183         elif new_node.add_child(self.oper()):
184             print_log("FOUND oper3")
185         else:
186             self.parse_error("could not find grammar in s")
187             globals()["current_token_index"] = save
188             return None
189         return new_node
190
191     def s_prime(self):
192         # S' → ] | S] | Expr2] | ]S
193         new_node = Node("S'")
194         save = globals()["current_token_index"]
195         if new_node.add_child(self.is_value(self.get_token(), RPAREN)):
196             print_log("FOUND: )")
197         elif new_node.add_child(self.s()) \
198             and new_node.add_child(self.is_value(self.get_token(), RPAREN
199         )):
200             print_log("FOUND: S )")
201         elif new_node.add_child(self.expr2()) \
202             and new_node.add_child(self.is_value(self.get_token(), RPAREN
203         )):
204             print_log("FOUND: expr2 )")
205         elif new_node.add_child(self.is_value(self.get_token(), RPAREN)) \
206             and new_node.add_child(self.s()):
207             print_log("FOUND: ) S")
208         else:
209             self.parse_error("could not find grammar in s'")
210             globals()["current_token_index"] = save
211             return None
212         return new_node
213
214     def expr(self):
215         if not self.current_state:
216             return None
217         # Expr → [Expr2] | Oper3
218         new_node = Node("expr")
219         save = globals()["current_token_index"]
220         if new_node.add_child(self.is_value(self.get_token(), LPAREN)) \
221             and new_node.add_child(self.expr2()) \

```

```

219         and new_node.add_child(self.is_value(self.get_token()), R_PAREN
    ):
        print_log("FOUND: ( expr2 )")
221     elif new_node.add_child(self.oper3()):
        print_log("FOUND: oper3")
223     else:
        globals()["current_token_index"] = save
225         return None
        return new_node
227
def expr2(self):
229     if not self.current_state:
        return None
231     # expr2 → Stmt | Oper2
    new_node = Node("expr3")
233     save = globals()["current_token_index"]
    if new_node.add_child(self.stmts()):
235         print_log("FOUND: stmts")
    elif new_node.add_child((self.oper2())):
237         print_log("FOUND: oper2")
    else:
239         globals()["current_token_index"] = save
        return None
241     return new_node

243 def oper(self):
    # Oper → [Oper2] | Oper3
245     global current_token_index
    if not self.current_state:
247         return None

    new_node = Node("oper")
    saved_token_index = current_token_index
251     if new_node.add_child(self.is_value(self.get_token()), L_PAREN)) \
        and new_node.add_child(self.oper2()) \
253         and new_node.add_child(self.is_value(self.get_token()), R_PAREN
    ):
        print_log("FOUND: (oper2)")
255     elif new_node.add_child(self.oper3()):
        print_log("FOUND: oper3")
257     else:
        self.parse_error("missing oper constant or name")
259         current_token_index = saved_token_index
        # new_node.print_tree()
        return None
261     return new_node

263
def oper2(self):
265     # Oper2 → := Name Oper
    # | Binop Oper Oper
267     # | Unop Oper
    global current_token_index
269     if not self.current_state:
        return None
271

```

```

273     new_node = Node("oper2")
274     saved_token_index = current_token_index
275     if new_node.add_child(self.is_value(self.get_token(), OPER_ASSIGN)) \
        and new_node.add_child(self.is_type(self.get_token(), TYPE_ID))
276     ) \
        and new_node.add_child(self.oper()):
277         print_log("FOUND: := Name Oper")
278     elif new_node.add_child(self.binops()) \
        and new_node.add_child(self.oper()) \
279         and new_node.add_child(self.oper()):
280         print_log("FOUND: Binop Oper Oper")
281     elif new_node.add_child(self.unops()) \
282         and new_node.add_child(self.oper()):
283         print_log("FOUND: Unop Oper")
284     else:
285         self.parse_error("missing oper2 constant or name")
286         current_token_index = saved_token_index
287         return None
288     return new_node
289
290 def oper3(self):
291     # Oper3 → Constant | Name
292     global current_token_index
293     if not self.current_state:
294         return None
295     new_node = Node("oper3")
296     saved_token_index = current_token_index
297     if new_node.add_child(self.constants()):
298         print_log("FOUND: constants")
299     elif new_node.add_child(self.name()):
300         print_log("FOUND: name")
301     else:
302         self.parse_error("missing left paren constant or name")
303         current_token_index = saved_token_index
304         return None
305     return new_node
306
307 def binops(self):
308     # binops → + | - | * | / | % | ^ | = | > | >= | < | <= | != | or |
309     and
310     if not self.current_state:
311         return None
312     new_node = Node("binops")
313     save = globals()["current_token_index"]
314     if new_node.add_child(self.is_value(self.get_token(), OPER_ADD)):
315         pass
316     elif new_node.add_child(self.is_value(self.get_token(), OPER_SUB)):
317         pass
318     elif new_node.add_child(self.is_value(self.get_token(), OPER_MULT)):
319         pass
320     elif new_node.add_child(self.is_value(self.get_token(), OPER_DIV)):
321         pass
322     elif new_node.add_child(self.is_value(self.get_token(), OPER_MOD)):
323         pass
324     elif new_node.add_child(self.is_value(self.get_token(), OPER_EXP)):

```

```

325         pass
326     elif new_node.add_child(self.is_value(self.get_token(), OPER_EQ)):
327         pass
328     elif new_node.add_child(self.is_value(self.get_token(), OPER_LT)):
329         pass
330     elif new_node.add_child(self.is_value(self.get_token(), OPER_LE)):
331         pass
332     elif new_node.add_child(self.is_value(self.get_token(), OPER_GT)):
333         pass
334     elif new_node.add_child(self.is_value(self.get_token(), OPER_GE)):
335         pass
336     elif new_node.add_child(self.is_value(self.get_token(), OPER_NE)):
337         pass
338     elif new_node.add_child(self.is_value(self.get_token(), OPER_OR)):
339         pass
340     elif new_node.add_child(self.is_value(self.get_token(), OPER_AND)):
341         pass
342     else:
343         self.parse_error("missing binop")
344         globals()["current_token_index"] = save
345         return None
346     return new_node
347
348 def unops(self):
349     # unops -> - | not | sin | cos | tan
350     if not self.current_state:
351         return None
352     new_node = Node("unops")
353     save = globals()["current_token_index"]
354     if new_node.add_child(self.is_value(self.get_token(), OPER_NOT)):
355         pass
356     elif new_node.add_child(self.is_value(self.get_token(), OPER_SIN)):
357         pass
358     elif new_node.add_child(self.is_value(self.get_token(), OPER_COS)):
359         pass
360     elif new_node.add_child(self.is_value(self.get_token(), OPER_TAN)):
361         pass
362     else:
363         globals()["current_token_index"] = save
364         self.parse_error("missing unop")
365         return None
366     return new_node
367
368 def constants(self):
369     # constants -> string | ints | floats
370     if not self.current_state:
371         return None
372     new_node = Node("constant")
373     save = globals()["current_token_index"]
374     if new_node.add_child(self.strings()):
375         pass
376     elif new_node.add_child(self.ints()):
377         pass
378     elif new_node.add_child(self.floats()):
379         pass

```

```

381         else:
382             globals()[ "current_token_index" ] = save
383             return None
384         return new_node
385
386     def strings(self):
387         # strings -> reg-ex for str literal in C ( any alphanumeric )
388         # true | false
389         if not self.current_state:
390             return None
391         new_node = Node("string")
392         save = globals()[ "current_token_index" ]
393         if new_node.add_child(self.is_type(self.get_token(), TYPE_STRING)):
394             pass
395         elif new_node.add_child(self.is_type(self.get_token(), TYPE_BOOL)):
396             pass
397         else:
398             globals()[ "current_token_index" ] = save
399             return None
400         return new_node
401
402     def name(self):
403         # name -> reg-ex for ids in C (any lower and upper char
404         # or underscore followed by any combination of lower,
405         # upper, digits, or underscores)
406         if not self.current_state:
407             return None
408         new_node = Node("name")
409         save = globals()[ "current_token_index" ]
410         if new_node.add_child(self.is_type(self.get_token(), TYPE_ID)):
411             pass
412         else:
413             globals()[ "current_token_index" ] = save
414             return None
415         return new_node
416
417     def ints(self):
418         # ints -> reg ex for positive/negative ints in C
419         if not self.current_state:
420             return None
421         new_node = Node("int")
422         save = globals()[ "current_token_index" ]
423         if new_node.add_child(self.is_type(self.get_token(), TYPE_INT)):
424             pass
425         else:
426             globals()[ "current_token_index" ] = save
427             return None
428         return new_node
429
430     def floats(self):
431         # floats -> reg ex for positive/negative doubles in C
432         if not self.current_state:
433             return None
434         new_node = Node("float")
435         save = globals()[ "current_token_index" ]

```

```

435         if new_node.add_child(self.is_type(self.get_token(), TYPE_REAL)):
436             pass
437         else:
438             globals()["current_token_index"] = save
439             return None
440         return new_node
441
442     def stmts(self):
443         # stmts -> ifstmts | whilestmts | letstmts | printstmts
444         if not self.current_state:
445             return None
446         new_node = Node("stmts")
447         save = globals()["current_token_index"]
448         if new_node.add_child(self.ifstmts()):
449             print_log("FOUND: ifstmts")
450         elif new_node.add_child(self.whilestmts()):
451             print_log("FOUND: whilestmts")
452         elif new_node.add_child(self.letstmts()):
453             print_log("FOUND: letstmts")
454         elif new_node.add_child(self.printstmts()):
455             print_log("FOUND: printstmts")
456         else:
457             self.parse_error("missing if, while, let or print statment")
458             globals()["current_token_index"] = save
459             return None
460         return new_node
461
462     def printstmts(self):
463         # printstmts -> (stdout oper)
464         if not self.current_state:
465             return None
466         new_node = Node("printstmts")
467         save = globals()["current_token_index"]
468         if new_node.add_child(self.is_value(self.get_token(), KEYWORD_STDOUT)) \
469             and new_node.add_child(self.oper()):
470             print_log("FOUND: stdout oper")
471         else:
472             globals()["current_token_index"] = save
473             self.parse_error("missing print statement paren")
474             return None
475         return new_node
476
477     def ifstmts(self):
478         # ifstmts -> if Expr If2
479         if not self.current_state:
480             return None
481         new_node = Node("ifstmts")
482         save = globals()["current_token_index"]
483         if new_node.add_child(self.is_value(self.get_token(), KEYWORD_IF)) \
484             and new_node.add_child(self.expr()) \
485             and new_node.add_child(self.ifstmts2()):
486             print_log("FOUND: if expr if2")
487         else:
488             globals()["current_token_index"] = save

```

```

489         self.parse_error("not an if statment")
490         return None
491     return new_node

493 def ifstmts2(self):
494     # ifstmts2 -> Expr | Expr Expr
495     if not self.current_state:
496         return None
497     new_node = Node("ifstmts2")
498     save = globals()["current_token_index"]
499     if new_node.add_child(self.expr()):
500         if new_node.add_child(self.expr()):
501             pass
502     else:
503         globals()["current_token_index"] = save
504         return None
505     return new_node

507 def whilestmts(self):
508     # whilestmts -> (while expr exprlist)
509     if not self.current_state:
510         return None
511     new_node = Node("whilestmts")
512     save = globals()["current_token_index"]
513     if new_node.add_child(self.is_value(self.get_token(), KEYWORD_WHILE)):
514         if new_node.add_child(self.expr()):
515             if new_node.add_child(self.exprlist()):
516                 pass
517             else:
518                 globals()["current_token_index"] = save
519                 return None
520         else:
521             globals()["current_token_index"] = save
522             return None
523     else:
524         globals()["current_token_index"] = save
525         self.parse_error("Not While stmts")
526         return None
527     return new_node

529 def exprlist(self):
530     # exprlist -> expr | expr exprlist
531     if not self.current_state:
532         return None
533     new_node = Node("exprlist")
534     save = globals()["current_token_index"]
535     if new_node.add_child(self.expr()):
536         if new_node.add_child(self.exprlist()):
537             pass
538     else:
539         globals()["current_token_index"] = save
540         self.parse_error("not expression list")
541         return None
542     return new_node
543

```



```

def letstmts(self):
545     # letstmts -> (let (varlist))
    if not self.current_state:
547         return None
    new_node = Node("letstmts")
549     save = globals()["current_token_index"]
    if new_node.add_child(self.is_value(self.get_token(), KEYWORDLET)):
551         if new_node.add_child(self.is_value(self.get_token(), LPAREN)):
            if new_node.add_child(self.varlist()):
553                 new_node.add_child((self.is_value(self.get_token(),
R_PAREN)))
            elif new_node.add_child(self.is_value(self.get_token(),
R_PAREN)):
555                 pass
            else:
557                 globals()["current_token_index"] = save
                print_error("missing right paren in let statement",
error_type="parser")
559                 return None
            else:
561                 globals()["current_token_index"] = save
                print_error("missing opening paren after let statement")
563                 return None
        else:
565             globals()["current_token_index"] = save
                self.parse_error("Checked if let statement")
567             return None
    return new_node

569 def varlist(self):
    # varlist -> (name type) | (name type) varlist
    if not self.current_state:
571         return None
    new_node = Node("varlist")
573     save = globals()["current_token_index"]
    if new_node.add_child(self.is_value(self.get_token(), LPAREN)):
575         if new_node.add_child(self.is_type(self.get_token(), TYPE_ID)):
577             if new_node.add_child(self.type()):
579                 if new_node.add_child(self.is_value(self.get_token(),
R_PAREN)):
                    if new_node.add_child(self.varlist()):
581                         return new_node
                    # (name type)
583                     return new_node
                else:
585                     globals()["current_token_index"] = save
                else:
587                     globals()["current_token_index"] = save
                        return new_node
                    elif new_node.add_child(self.varlist()):
589                         return new_node
                else:
591                     globals()["current_token_index"] = save
                        self.parse_error("not varlist")
593                     return None

```

```

595         return new_node

597     def type(self):
598         # type -> bool | int | real | string
599         if not self.current_state:
600             return None
601         new_node = Node("type")
602         save = globals()["current_token_index"]
603         if new_node.add_child(self.is_value(self.get_token(), "bool")):
604             pass
605         elif new_node.add_child(self.is_value(self.get_token(), "int")):
606             pass
607         elif new_node.add_child(self.is_value(self.get_token(), "real")):
608             pass
609         elif new_node.add_child(self.is_value(self.get_token(), "string")):
610             pass
611         else:
612             globals()["current_token_index"] = save
613         return new_node

```

myparser.py

## 4.6 lexer.py

```

__author__ = 'drakebridgewater'
2 import string

4 from defines import *

6
8 class Lexer():
9     def __init__(self, filename):
10         self.line = 1
11         self.filename = filename
12         self.file = ''
13         self.current_char = ' '
14         self.pointer = 0
15         self.token_list = []
16         self.current_state = True # When false throw error
17         self.accepted_ops = ('=', '+', '-', '/', '*', '<', '>', '!', ';', ':',
18             '%', '(', ')', '^')
19         # tokens is a dictionary where each token is a list
20         self.tokens = \
21             {"keywords": [KEYWORD_STDOUT, KEYWORD_LET, KEYWORD_IF,
22                 KEYWORD_TRUE, KEYWORD_FALSE, OPER_ASSIGN],
23              "ops": [OPER_ASSIGN, OPER_ADD, OPER_SUB, OPER_DIV, OPER_MULT,
24                  OPER_LT, OPER_GT, OPER_NOT, OPER_MOD, OPER_EXP,
25                  OPER_AND, OPER_OR, OPER_NOT, OPER_NE, R_PAREN, L_PAREN],
26              'type': [TYPE_BOOL, TYPE_INT, TYPE_REAL, TYPE_STRING]
27             }
28
29     def open_file(self):
30         self.file = open(self.filename, 'r')

```

```

30 def close_file(self):
    self.file.close()
32
33 def has_token(self, value, key=''):
34     # if subgroup given check it first
    if key != '':
35         if value in self.tokens[key]:
36             return key
37
38     # if subgroup checking fails check all entries
    for x in self.tokens:
39         if value in self.tokens[x]:
40             return x
41     return -1
42
43 def get_next_char(self):
44     try:
45         self.current_char = self.file.read(1)
46     except EOFError:
47         print("Reached end of file")
48
49 def get_token(self):
50     self.get_next_char()
51     while True and self.current_state:
52         if not self.current_char:
53             return -1
54         if self.current_char == ' ' or self.current_char == '\t':
55             self.get_next_char()
56             pass
57         elif self.current_char == '\n':
58             self.get_next_char()
59             self.line += 1
60         elif self.current_char in self.accepted_ops:
61             return self.is_op()
62         elif self.is_letter():
63             return self.identify_word() # identify the string and add to
the token list
64         elif self.is_digit():
65             return self.is_number() # identify the number and add to the
token list
66         elif self.current_char == '"':
67             return self.create_token(self.parse_string())
68         else:
69             print("Line:ERROR: Could not identify on line: " + str(
70                 self.line) + " near char: '" + self.current_char + "'")
71             return None
72
73     # TODO have all functions return to a state that has the next
char
74
75 # Function Description:
76 # General function to do something with the tokens once we have classified
them.
77 def create_token(self, token):
78     new_token = Token()
79
80

```

```

82     new_token.line = self.line
83     new_token.type = token[0]
84     new_token.value = token[1]
85     return new_token

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

92
93 def print_tokens(self):
94     for x in self.token_list:
95         print("[line: " + x.line + ", ID: " + x.type + ", Value: " + x.
value + "]")
96
97 def is_op(self):
98     item = self.current_char
99     # If we see an op look to see if we see another. If we see another add
the previous
100    # found op
101    if self.current_char is '+':
102        self.get_next_char()
103        return self.create_token((self.has_token(item), item))
104    elif self.current_char is '-':
105        self.get_next_char()
106        # if self.current_char is '-':
107        # item += self.current_char # Seen -- make new token
108        # self.get_next_char()
109        return self.create_token((self.has_token(item), item))
110    elif self.current_char in ('<', '>', '!'):
111        self.get_next_char()
112        if self.current_char == '=':
113            item += self.current_char
114            self.get_next_char()
115            return self.create_token((self.has_token(item), item))
116    elif self.current_char in ':':
117        self.get_next_char()
118        if self.current_char is '=':
119            item += self.current_char
120            return self.create_token((self.has_token(item), item))
121    else:
122        print("Lexer Error [Line: " + str(
self.line) + "] the " + self.current_char + " symbol not
recognized after colon [:] ")
123    elif self.current_char in '=':
124        return self.create_token((self.has_token(item), item))
125    elif self.current_char in ('*', '/', '(', ')', '%', '^'):
126        self.get_next_char()
127        return self.create_token((self.has_token(item), item))
128    else:
129        print("Lexer Error: [Line: " + str(self.line) + "] could not
intemperate: " +
self.current_char)

```

```

132         return -1

134     def parse_string(self):
135         accepted_chars = [ '" ' ]
136         new_string = ''
137         self.get_next_char()
138         while self.current_char not in accepted_chars:
139             new_string += self.current_char
140             self.get_next_char()
141         return "string", new_string

142
143     def identify_word(self):
144         accepted_chars = list(string.ascii_letters) + list(string.digits) +
list(' _ ')
145         acceptable_first_chars = list(string.ascii_letters)
146
147         word = ''
148         if self.current_char in acceptable_first_chars:
149             word += self.current_char
150             self.get_next_char()
151             while self.current_char in accepted_chars:
152                 word += self.current_char
153                 self.get_next_char()
154         token_value = word
155         token_type = self.has_token(token_value)
156         if token_type == -1:
157             token_type = "ID"
158         return self.create_token((token_type, token_value))

160     # Function Description:
161     # This function should be called when a word identifier or keyword is
162     # started
163     # and will return the full word upon seeing invalid characters.
164     def parse_word(self, accepted_chars, acceptable_first_chars=[]):
165         if self.current_char not in acceptable_first_chars:
166             return -1
167         else:
168             word = ''
169             while self.current_char in accepted_chars:
170                 word += self.current_char
171                 self.get_next_char()
172             return word

173
174     def is_int(self):
175         word = ''
176         while self.is_digit(exclude=['.', 'e']):
177             word += self.current_char
178             self.get_next_char()
179
180         return word

181     # Function Description:
182     # This function should be called after seeing the start of a number
183     # If a period is present the number is converted to a float and returned
184     def is_number(self, value=''):

```

```

186         if value == '':
187             word = self.current_char
188         else:
189             word = value
190         self.get_next_char()
191
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
230         elif self.current_char is '-':
231             self.get_next_char()
232             exp = self.is_int()
233             try:
234                 self.get_next_char()
235                 exp = int(exp)
236                 word += 'e-'
237                 word += str(exp)
238             try:
239                 return self.create_token(("float", float(word)))
240             except ValueError:

```

```

236         print("Fatal parse error: [row: " +
238               str(self.line) + "]" when parsing char " +
               str(self.current_char) + " for: \n\t\t" +
str(word))
               except ValueError:
240                 return [self.create_token(("int", word)),
                           self.create_token(("ID", "e")),
                           self.create_token((self.has_token("-"), "-"))]
242             else:
244                 exp = self.is_int()
246                 try:
248                     exp = int(exp)
249                     word += str(exp)
250                     return self.create_token(("float", float(word)))
251                 except ValueError:
252                     print_error("Unable to parse " + str(self.
current_char) +
                               "' in: " + str(exp), self.line, 'lexer')
253             elif self.is_digit(other_accepted):
254                 word += self.current_char
255                 self.get_next_char()
256             else:
257                 break
258             if 'e' not in other_accepted:
259                 other_accepted.append('e')
260
261             if '.' in word or 'e' in word:
262                 try:
263                     return self.create_token(("float", float(word)))
264                 except ValueError:
265                     print_error("could not determine numerical token of: " +
str(word), self.line, 'lexer')
266             else:
267                 try:
268                     return self.create_token(("int", int(word)))
269                 except ValueError:
270                     print_error("could not determine numerical token of: " +
str(word), self.line, 'lexer')
271
272     # Function Description:
273     # checks to see if the current token in peek is a digit or '.'
274     # return true if it is
275     def is_digit(self, others=[], exclude=[]):
276         digits = ['.', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
277         for x in others:
278             if x not in digits:
279                 digits.append(x)
280         for x in exclude:
281             if x in digits:
282                 digits.remove(x)
283         if self.current_char in digits:
284             return True
285         return False
286
287     # Function Description:

```

```
290 # checks to see if the current token in peek is a letter
291 # return true if it is
292 def is_letter(self, others=[]):
293     letters = list(string.ascii_letters)
294     for x in others:
295         if x not in letters:
296             letters.append(x)
297     if self.current_char in letters:
298         return True
299     return False
```

lexer.py