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

1 Source Code Descriptions

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

1.1 node.py

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

1.2 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 planed
```

4 Source Code

4.1 main.py

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

from myparser import *
from codegen import *


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

    With no arguments this will run test1 by default
"""

files = []
options = []

global current_token_index
current_token_index = 0


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

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

    return content


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


def main():
    sys.setrecursionlimit(100)
    print("-"*80)
    if len(sys.argv) > 1:
        filename = sys.argv[len(sys.argv) - 1]
    else:
        filename = "test1"

    if '-help' in sys.argv or '-usage' in sys.argv or '-h' in sys.argv:
        print(usage)
        return 0
    if '-d' in sys.argv or '-debug' in sys.argv:
        globals()['DEBUG'] = 1
        globals()['OPTIONS'].append("DEBUG")
    else:
        globals()['DEBUG'] = 0
    if '-input' in sys.argv:
        print_verbose(filename)
    if '-tree' in sys.argv:
        globals()['OPTIONS'].append("tree")
    if '-postorder'in sys.argv:
        globals()['OPTIONS'].append("postorder")
    if '-print' in sys.argv:
        globals()['OPTIONS'].append("print")
    if '-lexer' in sys.argv:
        globals()['OPTIONS'].append("lexer")


    parser = MyParser(filename)
    parser.control()
    tree = parser.tree
    tree.print_postordered_tree()
    codegen = CodeGen(tree)
    gforth = codegen.control()

    outfile = filename.split(".")[0] + ".out" + ".fs"
    print("\n\n\ngforth code in:  " + outfile)
    with open(outfile, "w+") as file:
        for x in gforth:
            file.write(str(x) + " ")




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

main.py

4.2 codegen.py

```
__author__ = 'drakebridgewater'
from defines import *
from myparser import *


class Scope:
    paren = ''
    prev_was_string = False
    prev_was_real = False
```

```python


class CodeGen(object):
    def __init__(self, tree):
        self.error_flag = False
        self.tree = tree
        self.current_token = None
        self.stack = []
        self.scope_stack = []
        self.current_scope = []
        self.variables = []
        self.gforth = []
        self.index = 0
        self.pointer = 0
        self.other_tokens = {
            # KEYWORD_STDOUT: '.s',
            # KEYWORD_LET: 'let',
            KEYWORD_IF: 'if',
            KEYWORD_WHILE: 'while',
            KEYWORD_TRUE: "true",
            KEYWORD_FALSE: "false"
            # TYPE_BOOL: 'bool',
            # TYPE_INT: 'int',
            # TYPE_REAL: 'float',
            # TYPE_STRING: 'string'
        }
        self.conversions = {
            "%": "mod",
            "!=": "<>"
        }
        self.realConversions = {
            "+": "f+",
            "-": "f-",
            "*": "f*",
            "/": "f/",
            "mod": "fmod",
            "<": "f<",
            "<=": "f<=",
            ">": "f>",
            ">=": "f>=",
            "=": "f=",
            "<>": "f<>",
            "sin": "fsin",
            "cos": "fcos",
            "tan": "ftan",
        }

    def control(self):
        # TODO as we step through the tree convert and push element on to stack
        self.get_tokens_stack()
        if '-print' in globals()['OPTIONS']:
            print_title("CODE_GEN -- in progress")
        self.print_stack()
        self.write_out()
        self.gforth = self.rem_sll(self.gforth, L_PAREN)
        self.gforth = self.rem_sll(self.gforth, R_PAREN)

        if 'print' in globals()['OPTIONS']:
            print_title("gforth code")
            for x in self.gforth:
                print(x, end=" ")
        return self.gforth

    def rem_sll(self, L, item):
        answer = []
        for i in L:
            if i != item:
                answer.append(i)
        return answer

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

    def write_out(self):
        data = self.stack
        prev_was_string = False
        prev_was_real = False
        prev_was_int = False
        prev_was_assign = False
        prev_was_var = False
        prev_was_if = False
        prev_was_declare = False
        prev_was_stdout = False
        append_end = False
        oper_hold = []

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

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

                if data[self.pointer].type == 'ops':
                    if data[self.pointer].value is L_PAREN:
                        self.gforth.append(L_PAREN)
                        self.scope_stack.append(L_PAREN)
                    elif data[self.pointer].value is R_PAREN:
                        self.gforth.append(R_PAREN)
                        try:
                            if self.gforth[len(self.gforth) - 1] != '\n':
                                self.gforth.append('\n')
                            pop_value = self.scope_stack.pop()
                            if 'whileloop' in pop_value:
                                temp = pop_value.split("whileloop")
                                self.gforth.append("whileloop" + temp[len(temp) - 1] + "\n")
                            if 'ifloop' in pop_value:
                                temp = pop_value.split("ifloop")
                                self.gforth.append("ifloop" + temp[len(temp) - 1] + "\n")
                                prev_was_if = False
                            prev_was_string = False
                            prev_was_real = False
                            prev_was_int = False
                            prev_was_assign = False
                            prev_was_var = False
                            prev_was_declare = False
                            prev_was_stdout = False
                            append_end = False
                            oper_hold = []
                        except IndexError:
                            print_error("missing left paren [d]", error_type="code_gen")
                    elif data[self.pointer].value in ['+', '-', '/', '*', '<', '>', '!', ';', ':', '%', '^']:
                        self.pointer = self.is_math_expr(self.pointer)
                        # self.gforth.append(str(data[self.pointer].value))
                elif data[self.pointer].type == TYPE_ID:
                    if self.stack[self.pointer].value not in self.variables:
                        print_error("variable " + str(self.stack[self.pointer].value) + " not declared before use")
                    if data[self.pointer].value in self.current_scope:
                        pass
                    elif prev_was_declare:
                        if data[self.pointer].value in self.variables:
                            print_error("redecloration of variable " + str(data[self.pointer].value),
                                        error_type="codegen")
                        self.current_scope.append(data[self.pointer].value)
                        # oper_hold = str(data[self.pointer].value)
                    else:
                        print_error("unassigned variable " + str(data[self.pointer].value), error_type="codegen")
                    self.gforth.append(str(data[self.pointer].value))
                    if prev_was_assign:
                        self.gforth.append("!")
                        # oper_hold.append(str(data[self.pointer].value))
                        # oper_hold.append('!')
                        # self.variables.append(data[self.pointer].value)

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

```python
                    elif data[self.pointer].value == KEYWORD_LET:  # Declare
                        self.pointer = self.is_let(self.pointer)
                        # prev_was_declare = True
                        # self.gforth.append("variable")
                        pass
                    elif data[self.pointer].value == KEYWORD_IF:
                        self.pointer = self.is_ifstmt(self.pointer)
                        # if_stmts = "ifloop" + str(self.pointer)
                        # self.scope_stack.append(if_stmts)
                        # self.gforth.append(" : " + if_stmts)
                    elif data[self.pointer].value == KEYWORD_WHILE:
                        self.pointer = self.is_whilestmt(self.pointer)
                        # while_stmts = "whileloop" + str(self.pointer)
                        # self.scope_stack.append(while_stmts)
                        # self.gforth.append(" : " + while_stmts)
                elif data[self.pointer].type == TYPE_STRING:  # An actual string
                    self.gforth.append('s" ' + str(data[self.pointer].value) + '"')
                elif data[self.pointer].type == TYPE_INT:
                    prev_was_int = True
                    self.gforth.append(str(data[self.pointer].value))
                    pass
                elif data[self.pointer].type == TYPE_REAL:
                    prev_was_real = True
                    self.gforth.append(str(data[self.pointer].value) + "e0")
                    pass

                if append_end:
                    for op in oper_hold:
                        self.gforth.append(op)

            finally:
                self.pointer += 1
        if len(self.scope_stack) != 0:
            print_error("missing right paren [c]", error_type="code_gen")
            if globals()['DEBUG'] == 1:
                print_title("Scope stack should be empty but has")
                print(self.scope_stack)
        self.gforth.append("cr cr bye")

    def is_let(self, x):
        temp_scope = []
        if self.stack[x].value == KEYWORD_LET:
            x += 1
            if self.stack[x].value == L_PAREN:  # let *( ( x int) )
                temp_scope.append(L_PAREN)
                x +=1
                while self.stack[x].value == L_PAREN:
                    temp_scope.append(L_PAREN)
                    x += 1
                    self.gforth.append("variable")
                    if self.stack[x].type == TYPE_ID:
                        self.gforth.append(self.stack[x].value)
                        x +=1
                        if self.stack[x].value == TYPE_REAL:
                            self.variables.append(self.stack[x-1])
                            x +=1
                        elif self.stack[x].value == TYPE_INT:
                            self.variables.append(self.stack[x-1])
                            x +=1
                        elif self.stack[x].value == TYPE_STRING:
                            self.variables.append(self.stack[x-1])
                            x +=1
                        else:
                            print_error("let statement error", error_type="codegen")

                        if self.stack[x].value == R_PAREN:
                            x += 1
                            temp_scope.pop()
        if self.stack[x].value == R_PAREN:
            if len(temp_scope) > 0:
                temp_scope.pop()
            x+=1
        if len(temp_scope) != 0:
            print_error("expected to see final closing bracket on let stmt", error_type="codegen")
        return x

    def is_stdout(self, x, if_stmts=False):
        if self.stack[x].value == KEYWORD_STDOUT:
            x += 1
            if self.stack[x].value == L_PAREN:
                self.gforth.append(L_PAREN)
                x += 1
                x = self.is_math_expr(x)
                if not if_stmts:
                    self.gforth.append(".")
                x += 1
                if self.stack[x].value == R_PAREN:
                    self.gforth.append(R_PAREN)
            elif self.stack[x].type == TYPE_STRING:
                self.gforth.append('s" ' + self.stack[x].value + '"')
                if not if_stmts:
                    self.gforth.append(".s")
            elif self.stack[x].type == TYPE_INT:
                self.gforth.append(str(self.stack[x].value))
                if not if_stmts:
                    self.gforth.append(".s")
            elif self.stack[x].type == TYPE_REAL:
                self.gforth.append(str(self.stack[x].value) + "e0")
                if not if_stmts:
                    self.gforth.append("f.s")
        return x

    def is_assign(self, x):
        oper_hold = ''
        if self.stack[x].value == OPER_ASSIGN:
            x += 1
            if self.stack[x].type == TYPE_ID:
                x += 1
                if self.stack[x].value == L_PAREN:
                    oper_hold = self.stack[x-1].value
                    x += 1
                    x = self.is_math_expr(x)
                    if self.stack[x].value == R_PAREN:
                        x += 1
                    self.gforth.append(oper_hold)  # append ID
                elif self.stack[x].type == TYPE_STRING:
                    self.gforth.append('s" ' + self.stack[x - 1].value + '"')
                    self.gforth.append(self.stack[x - 1].value)
                elif self.stack[x].type == TYPE_INT:
                    self.gforth.append(str(self.stack[x].value))
                    self.gforth.append(self.stack[x - 1].value)
                elif self.stack[x].type == TYPE_REAL:
                    self.gforth.append(str(self.stack[x].value) + "e0")
                    self.gforth.append(self.stack[x - 1].value)
                elif self.stack[x].type == TYPE_ID:
                    self.gforth.append(str(self.stack[x].value))
                    self.gforth.append(self.stack[x - 1].value)
                else:
                    print_error("in assignment missing value", error_type="
codegen")

                self.gforth.append("!")
                if oper_hold != '':
                    self.gforth.append(oper_hold)
                    self.gforth.append("@")
        return x

    def is_math_expr(self, x):
        if self.stack[x].value in ['+', '-', '/', '*', '<', '>', '!', ';', ':'
, '%', '^']:
            x += 1
            if self.stack[x].type == TYPE_INT:
                x += 1
                if self.stack[x].type == TYPE_INT:  # int int
                    self.gforth.append(self.stack[x - 1].value)
                    self.gforth.append(self.stack[x].value)
                    self.gforth.append(self.stack[x - 2].value)
                elif self.stack[x].type == TYPE_REAL:  # int real
                    self.gforth.append(self.stack[x - 1].value)
                    self.gforth.append(str(self.stack[x].value) + "e0")
                    self.gforth.append("fswap")
                    self.gforth.append("s>f")
                    self.gforth.append(self.realConversions[self.stack[x - 2].
value])
                elif self.stack[x].type == TYPE_ID:
                    # TODO check if the variable exists
                    self.gforth.append(self.stack[x - 1].value)  # value
                    if self.stack[x].value not in self.variables:
                        print_error("variable " + str(self.stack[x].value) + "
 not declared before use")
                    self.gforth.append(self.stack[x].value)  # variable
                    self.gforth.append("@")
                    self.gforth.append(self.stack[x - 2].value)
                else:
                    print_error("expected another int/real/string", error_type="codegen")
            elif self.stack[x].type == TYPE_REAL:
                x += 1
                if self.stack[x].type == TYPE_INT:  # real int
                    self.gforth.append(str(self.stack[x - 1].value) + "e0")
                    self.gforth.append(self.stack[x].value)
                    self.gforth.append("fswap")
                    self.gforth.append("s>f")
                    self.gforth.append(self.realConversions[self.stack[x - 2].value])
                elif self.stack[x].type == TYPE_REAL:  # real real
                    self.gforth.append(str(self.stack[x - 1].value) + "e0")
                    self.gforth.append(str(self.stack[x].value) + "e0")
                    self.gforth.append(self.realConversions[self.stack[x - 2].value])
                elif self.stack[x].type == TYPE_ID:
                    # TODO check if the variable exists
                    self.gforth.append(self.stack[x - 1].value)  # value
                    if self.stack[x].value not in self.variables:
                        print_error("variable " + str(self.stack[x].value) + " not declared before use")
                    self.gforth.append(self.stack[x].value)  # variable
                    self.gforth.append("@")
                    self.gforth.append(self.stack[x - 2].value)
                else:
                    print_error("expected another int/real/string", error_type="codegen")
            elif self.stack[x].type == TYPE_STRING:
                x += 1
                if self.stack[x].type == TYPE_STRING:
                    self.gforth.append('s" ' + self.stack[x - 1].value + '"')
                    self.gforth.append('s" ' + self.stack[x].value + '"')
                    if self.stack[x-2].value == "+":
                        self.gforth.append("s" + self.stack[x - 2].value)
                    else:
                        print_error("only string concatenation is supported", error_type="codegen")
            elif self.stack[x].type == TYPE_ID:
                if self.stack[x].value not in self.variables:
                    print_error("variable " + str(self.stack[x].value) + " not declared before use")
                self.gforth.append(self.stack[x].value)
                self.gforth.append("@")
                x += 1
                if self.stack[x].type == TYPE_INT:
                    self.gforth.append(self.stack[x].value)
                elif self.stack[x].type == TYPE_REAL:
                    self.gforth.append(self.stack[x].value)
                x += 1
                self.gforth.append(self.stack[x-3].value)
        if self.stack[x].value == R_PAREN:
```

```python
            self.gforth.append(R_PAREN)
            self.scope_stack.pop()
        return x

    def is_whilestmt(self, x):
        while_stmts = ''
        if self.stack[x].value == KEYWORD_WHILE:
            x += 1
            if self.stack[x].value == L_PAREN:
                x += 1
                if self.stack[x].value in ['<', '>', '=<', '=>', '!', KEYWORD_TRUE, KEYWORD_FALSE]:
                    x += 1
                    if self.stack[x].type == TYPE_INT or self.stack[x].type == TYPE_REAL or \
                                    self.stack[x].type == TYPE_ID:
                        x += 1
                        if self.stack[x].type == TYPE_INT or self.stack[x].type == TYPE_REAL or \
                                        self.stack[x].type == TYPE_ID:
                            while_stmts = "whilestmts" + str(x - 4)
                            self.scope_stack.append(L_PAREN)
                            self.scope_stack.append(while_stmts)

                            temp_stack = []
                            # self.scope_stack.append(L_PAREN)
                            temp_stack.append(while_stmts)
                            temp_stack.append(L_PAREN)
                            self.scope_stack.append(L_PAREN)  # Because we enter this function at the 'if'

                            self.gforth.append(": " + while_stmts)
                            self.gforth.append(L_PAREN)
                            self.gforth.append("BEGIN")
                            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")
                            self.gforth.append(self.stack[x - 1].value)  # x
                            self.gforth.append("@")
                            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")
                            self.gforth.append(self.stack[x].value)  # 3
                            self.gforth.append(self.stack[x - 2].value)  # <
                            self.gforth.append("while")
                            x += 1

                            if self.stack[x].value == R_PAREN:
                                self.scope_stack.pop()
                                x += 1
                                stack_val = temp_stack.pop()
                                if stack_val != L_PAREN:
                                    print_error("incorrect token", error_type=
    "codegen")
                                else:
                                    self.gforth.append(stack_val)
                                    while self.stack[x].value != R_PAREN:
                                        temp = self.is_while_internals(x)
                                        if temp == -1:
                                            break
                                        else:
                                            x = temp
                                        self.gforth.append("TYPE ")
                                    self.scope_stack.pop()
                                    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")
        return x

    def is_while_internals(self, x):
        if self.stack[x].value == L_PAREN:
            x += 1
            x = self.is_stdout(x, True)
            x = self.is_math_expr(x)
            x = self.is_assign(x)

            x += 1
            if self.stack[x].value == R_PAREN:
                x += 1
        else:
            print_error("missing left paren [e]", error_type='codegen')
            return -1
        return x

    def is_ifstmt(self, x):
        if_stmts = ''
        if self.stack[x].value == KEYWORD_IF:
            x += 1
            if self.stack[x].value == L_PAREN:
                x += 1
                if self.stack[x].value in ['<', '>', '=<', '=>', '!',
    KEYWORD_TRUE, KEYWORD_FALSE]:
                    x += 1
                    if self.stack[x].type == TYPE_INT or self.stack[x].type ==
     TYPE_REAL or \
                                    self.stack[x].type == TYPE_ID:
                        x += 1
                        if self.stack[x].type == TYPE_INT or self.stack[x].
    type == TYPE_REAL or \
                                        self.stack[x].type == TYPE_ID:
                            if_stmts = "ifloop" + str(x - 4)
                            self.scope_stack.append(if_stmts)

                            temp_stack = []
```

```
                            # self.scope_stack.append(L_PAREN)
                            temp_stack.append(if_stmts)# Because we enter this function at the 'if'
                            temp_stack.append(L_PAREN)

                            self.gforth.append(": " + if_stmts)
                            self.gforth.append(L_PAREN)
                            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")
                            self.gforth.append(self.stack[x - 1].value)  # x
                            self.gforth.append("@")
                            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")
                            self.gforth.append(self.stack[x].value)  # 3
                            self.gforth.append(self.stack[x - 2].value)  # <
                            self.gforth.append("if")
                            x += 1

                            if self.stack[x].value == R_PAREN:
                                x += 1
                                stack_val = temp_stack.pop()
                                if stack_val != L_PAREN:
                                    print_error("incorrect token", error_type="codegen")
                                else:
                                    self.gforth.append(stack_val)
                                    while self.stack[x].value != R_PAREN:
                                        x = self.is_if_internals(x)
                                        self.gforth.append("TYPE else")
                                    self.scope_stack.pop()
            self.gforth.append("then")
            self.gforth.append(";")
            var = temp_stack.pop()
            self.gforth.append("\n" + var + "\n")
        return x

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

            x += 1
            if self.stack[x].value == R_PAREN:
                x += 1
                pass
            else:
                print_error("missing right paren [b]", error_type='codegen')
        else:
            print_error("missing left paren", error_type='codegen')
        return x
    def get_tokens_stack(self):
        self._get_token_stack(self.tree)
        self.print_stack()

    def _get_token_stack(self, node):
        for child in node.children:
            self._get_token_stack(child)
            temp_token = self.is_token(child)
            if temp_token:
                self.stack.append(temp_token)

    def is_token(self, child):
        if isinstance(child, int):
            return None
        elif isinstance(child, str):
            return None
        elif hasattr(child, "data"):
            if hasattr(child.data, "value"):
                return child.data
            else:
                return None
        elif hasattr(child, "value"):
            return child
        else:
            return None

    def print_stack(self):
        if globals()['DEBUG'] == 1:
            print_title("print stack called")
            for token in self.stack:
                print_token(token)
```

codegen.py

4.3 node.py

```
__author__ = 'Drake'


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

    def add_child(self, obj):
        if obj is None:
            globals()['current_token_index'] -= 1
            return obj
        self.children.append(obj)
        globals()['current_token_index'] += 1
        return True

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

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

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

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

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

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

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

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

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

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

node.py

4.4 defines.py

```
__author__ = 'Drake'

files = []
global OPTIONS
globals()["OPTIONS"] = []

global DEBUG
globals()["DEBUG"] = 0

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

OPER_EQ = '='
OPER_ASSIGN = ':='
OPER_ADD = '+'
OPER_SUB = '-'
OPER_DIV = '/'
OPER_MULT = '*'
OPER_LT = '<'
OPER_GT = '>'
OPER_LE = '<='
OPER_GE = '>='
OPER_NE = '!='
OPER_NOT = '!'
```

```
OPER_MOD = '%'
OPER_EXP = '^'
SEMI = ';'
L_PAREN = '('
R_PAREN = ')'
OPER_AND = 'and'
OPER_OR = 'or'
OPER_NOT = 'not'
OPER_SIN = 'sin'
OPER_TAN = 'tan'
OPER_COS = 'cos'
KEYWORD_STDOUT = 'stdout'
KEYWORD_LET = 'let'
KEYWORD_IF = 'if'
KEYWORD_WHILE = 'while'
KEYWORD_TRUE = "true"
KEYWORD_FALSE = "false"
KEYWORD = 'keywords'
TYPE_BOOL = 'bool'
TYPE_INT = 'int'
TYPE_REAL = 'real'
TYPE_STRING = 'string'
TYPE_ID = 'ID'


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


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


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


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


class Token:
    type = ''
    value = ''
    line = ''
    siblings = -1
```

defines.py

4.5 myparser.py

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


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

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

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

    def print_tree_helper(self, node, indent=0):
        indent += 1
        for child in node.children:
            if hasattr(child, "data"):
                if hasattr(child.data, "value"):
                    print_token(child.data, indent)
                else:
                    print("\t" * indent + str(child.data))
                self.print_tree_helper(child, indent)
            elif hasattr(child, "value"):
                print_token(child.data, indent)
            elif isinstance(child, int):
                print("\t" * indent + str(child))
            elif isinstance(child, str):
                print("\t" * indent + str(child))
            else:
                print("Error in print_tree_helper")
                print(child)
                return
               # else:
               # print("Failed")

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

    def post_order_tree_print(self, node):
        for child in node.children:
            self.print_child(child, 0)
            self.post_order_tree_print(child)

    def print_child(self, child, indent):
        if isinstance(child, int):
            print("\t" * indent + str(child))
        elif isinstance(child, str):
            print("\t" * indent + str(child))
        elif hasattr(child, "data"):
            if hasattr(child.data, "value"):
                print_token(child.data)
            else:
                print("\t" * indent + str(child.data))
        elif hasattr(child, "value"):
            print_token(child)
        else:
            print("Error in print_tree_helper")
            print(child)
            return False
        return True


class MyParser(object):
    def __init__(self, filename):
        temp_token = Node("EMPTY")
        self.tree = Node(temp_token)
        self.lexer = Lexer(filename)
        self.current_state = True
        self.tokens = []
        self.line = 1
        self.epsilon_flag = 0

    def parse_error(self, msg=''):
        if 'parse' in globals()['OPTIONS']:
            print_error(msg, self.line, "parse")

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

    def print_tokens(self):
        try:
            self.lexer.open_file()
            while self.get_token():
                print_token(self.tokens)
        finally:
            self.lexer.close_file()

    def control(self):
        try:
            self.lexer.open_file()
            if '-lexer' in globals()["OPTIONS"]:
                print_title("lexer output")
            # while self.current_state:
            self.tree.add_child(self.t())
            # globals()['current_token_index'] += 1
            self.tree.print_tree()
            if globals()['current_token_index'] > len(self.tokens):
                # if self.tokens[len(self.tokens) - 1] == -1:
                self.current_state = False  # Done reading file
            if len(self.tokens) == 0:
                return None
        finally:
            self.lexer.close_file()

    def is_type(self, token, compare):
        if not self.current_state:
            return None
        if isinstance(token, int):
            return None
        if token.type == compare:
            globals()['current_token_index'] += 1
            return Node(token)
        else:
            return None

    def is_value(self, token, compare):
        if not self.current_state:
            return None
        if token is None:
            return None
        if token.value == compare:
            globals()['current_token_index'] += 1
            return Node(token)
        else:
            return None

    def t(self):
        # T --> (T)
        new_node = Node("T")
        save = globals()["current_token_index"]
        if new_node.add_child(self.is_value(self.get_token(), L_PAREN)):
            while new_node.add_child(self.s()):
                pass
            if new_node.add_child(self.is_value(self.get_token(), R_PAREN)):
                pass
        else:
            self.parse_error("could not find grammar in T")
            globals()["current_token_index"] = save
            return None
        return new_node

    def s(self):
        # S --> [S' | Oper3 S | Oper3
        new_node = Node("S")
        save = globals()["current_token_index"]
        if new_node.add_child(self.is_value(self.get_token(), L_PAREN)) \
                and new_node.add_child(self.s_prime()):
            print_log("FOUND: (S' ")
        elif new_node.add_child(self.oper()) \
                and new_node.add_child(self.s()):
            print_log("FOUND: oper3 S")
        elif new_node.add_child(self.oper()):
            print_log("FOUND oper3")
        else:
            self.parse_error("could not find grammar in s")
            globals()["current_token_index"] = save
            return None
        return new_node

    def s_prime(self):
        # S' --> ] | S] | Expr2] | ]S
        new_node = Node("S")
        save = globals()["current_token_index"]
        if new_node.add_child(self.is_value(self.get_token(), R_PAREN)):
            print_log("FOUND: )")
        elif new_node.add_child(self.s()) \
                and new_node.add_child(self.is_value(self.get_token(), R_PAREN
    )):
            print_log("FOUND: S )")
        elif new_node.add_child(self.expr2()) \
                and new_node.add_child(self.is_value(self.get_token(), R_PAREN
    )):
            print_log("FOUND: expr2 )")
        elif new_node.add_child(self.is_value(self.get_token(), R_PAREN)) \
                and new_node.add_child(self.s()):
            print_log("FOUND: ) S")
        else:
            self.parse_error("could not find grammar in s")
            globals()["current_token_index"] = save
            return None
        return new_node

    def expr(self):
        if not self.current_state:
            return None
        # Expr --> [Expr2] | Oper3
        new_node = Node("expr")
        save = globals()["current_token_index"]
        if new_node.add_child(self.is_value(self.get_token(), L_PAREN)) \
                and new_node.add_child(self.expr2()) \
                and new_node.add_child(self.is_value(self.get_token(), R_PAREN
)):
            print_log("FOUND: ( expr2 )")
        elif new_node.add_child(self.oper3()):
            print_log("FOUND: oper3")
        else:
            globals()["current_token_index"] = save
            return None
        return new_node

    def expr2(self):
        if not self.current_state:
            return None
        # expr2 --> Stmt | Oper2
        new_node = Node("expr3")
        save = globals()["current_token_index"]
        if new_node.add_child(self.stmts()):
            print_log("FOUND: stmts")
        elif new_node.add_child((self.oper2())):
            print_log("FOUND: oper2")
        else:
            globals()["current_token_index"] = save
            return None
        return new_node

    def oper(self):
        # Oper --> [Oper2] | Oper3
        global current_token_index
        if not self.current_state:
            return None

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

    def oper2(self):
        # Oper2 --> := Name Oper
        # | Binop Oper Oper
        # | Unop Oper
        global current_token_index
        if not self.current_state:
            return None

        new_node = Node("oper2")
        saved_token_index = current_token_index
        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)
) \
                and new_node.add_child(self.oper()):
            print_log("FOUND: := Name Oper")
        elif new_node.add_child(self.binops()) \
                and new_node.add_child(self.oper()) \
                and new_node.add_child(self.oper()):
            print_log("FOUND: Binop Oper Oper")
        elif new_node.add_child(self.unops()) \
                and new_node.add_child(self.oper()):
            print_log("FOUND: Unop Oper")
        else:
            self.parse_error("missing oper2 constant or name")
            current_token_index = saved_token_index
            return None
        return new_node

    def oper3(self):
        # Oper3 --> Constant | Name
        global current_token_index
        if not self.current_state:
            return None
        new_node = Node("oper3")
        saved_token_index = current_token_index
        if new_node.add_child(self.constants()):
            print_log("FOUND: constants")
        elif new_node.add_child(self.name()):
            print_log("FOUND: name")
        else:
            self.parse_error("missing left paren constant or name")
            current_token_index = saved_token_index
            return None
        return new_node

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

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

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

    def strings(self):
        # strings ->    reg_ex for str literal in C (  any  alphanumeric )
        # true | false
        if not self.current_state:
            return None
        new_node = Node("string")
        save = globals()["current_token_index"]
        if new_node.add_child(self.is_type(self.get_token(), TYPE_STRING)):
            pass
        elif new_node.add_child(self.is_type(self.get_token(), TYPE_BOOL)):
            pass
        else:
            globals()["current_token_index"] = save
            return None
        return new_node

    def name(self):
        # name -> reg_ex for ids in C (any lower and upper char
        # or underscore followed by any combination of lower,
        # upper, digits, or underscores)
        if not self.current_state:
            return None
        new_node = Node("name")
        save = globals()["current_token_index"]
        if new_node.add_child(self.is_type(self.get_token(), TYPE_ID)):
            pass
        else:
            globals()["current_token_index"] = save
            return None
        return new_node

    def ints(self):
        # ints -> reg ex for positive/negative ints in C
        if not self.current_state:
            return None
        new_node = Node("int")
        save = globals()["current_token_index"]
        if new_node.add_child(self.is_type(self.get_token(), TYPE_INT)):
            pass
        else:
            globals()["current_token_index"] = save
            return None
        return new_node

    def floats(self):
        # floats -> reg ex for positive/negative doubles in C
        if not self.current_state:
            return None
        new_node = Node("float")
        save = globals()["current_token_index"]
        if new_node.add_child(self.is_type(self.get_token(), TYPE_REAL)):
            pass
        else:
            globals()["current_token_index"] = save
            return None
        return new_node

    def stmts(self):
        # stmts -> ifstmts | whilestmts | letstmts |printsmts
        if not self.current_state:
            return None
        new_node = Node("stmts")
        save = globals()["current_token_index"]
        if new_node.add_child(self.ifstmts()):
            print_log("FOUND: ifstmts")
        elif new_node.add_child(self.whilestmts()):
            print_log("FOUND: whilestmts")
        elif new_node.add_child(self.letstmts()):
            print_log("FOUND: letstmts")
        elif new_node.add_child(self.printstmts()):
            print_log("FOUND: printstmts")
        else:
            self.parse_error("missing if, while, let or print statment")
            globals()["current_token_index"] = save
            return None
        return new_node

    def printstmts(self):
        # printstmts -> (stdout oper)
        if not self.current_state:
            return None
        new_node = Node("printstmts")
        save = globals()["current_token_index"]
        if new_node.add_child(self.is_value(self.get_token(), KEYWORD_STDOUT))
 \
                and new_node.add_child(self.oper()):
            print_log("FOUND: stdout oper")
        else:
            globals()["current_token_index"] = save
            self.parse_error("missing print statement paren")
            return None
        return new_node

    def ifstmts(self):
        # ifstmts -> if Expr If2
        if not self.current_state:
            return None
        new_node = Node("ifstmts")
        save = globals()["current_token_index"]
        if new_node.add_child(self.is_value(self.get_token(), KEYWORD_IF)) \
                and new_node.add_child(self.expr()) \
                and new_node.add_child(self.ifstmts2()):
            print_log("FOUND: if expr if2")
        else:
            globals()["current_token_index"] = save
            self.parse_error("not an if statment")
            return None
        return new_node

    def ifstmts2(self):
        # ifstmts2 --> Expr | Expr Expr
        if not self.current_state:
            return None
        new_node = Node("ifstmts2")
        save = globals()["current_token_index"]
        if new_node.add_child(self.expr()):
            if new_node.add_child(self.expr()):
                pass
        else:
            globals()["current_token_index"] = save
            return None
        return new_node

    def whilestmts(self):
        # whilestmts -> (while expr exprlist)
        if not self.current_state:
            return None
        new_node = Node("whilestmts")
        save = globals()["current_token_index"]
        if new_node.add_child(self.is_value(self.get_token(), KEYWORD_WHILE)):
            if new_node.add_child(self.expr()):
                if new_node.add_child(self.exprlist()):
                    pass
                else:
                    globals()["current_token_index"] = save
                    return None
            else:
                globals()["current_token_index"] = save
                return None
        else:
            globals()["current_token_index"] = save
            self.parse_error("Not While stmts")
            return None
        return new_node

    def exprlist(self):
        # exprlist -> expr | expr exprlist
        if not self.current_state:
            return None
        new_node = Node("exprlist")
        save = globals()["current_token_index"]
        if new_node.add_child(self.expr()):
            if new_node.add_child(self.exprlist()):
                pass
        else:
            globals()["current_token_index"] = save
            self.parse_error("not expression list")
            return None
        return new_node

    def letstmts(self):
        # letstmts -> (let (varlist))
        if not self.current_state:
            return None
        new_node = Node("letstmts")
        save = globals()["current_token_index"]
        if new_node.add_child(self.is_value(self.get_token(), KEYWORD_LET)):
            if new_node.add_child(self.is_value(self.get_token(), L_PAREN)):
                if new_node.add_child(self.varlist()):
                    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)):
                    pass
                else:
                    globals()["current_token_index"] = save
                    print_error("missing right paren in let statement",
error_type="parser")
                    return None
            else:
                globals()["current_token_index"] = save
                print_error("missing opening paren after let statement")
                return None
        else:
            globals()["current_token_index"] = save
            self.parse_error("Checked if let statement")
            return None
        return new_node

    def varlist(self):
        # varlist -> (name type) | (name type) varlist
        if not self.current_state:
            return None
        new_node = Node("varlist")
        save = globals()["current_token_index"]
        if new_node.add_child(self.is_value(self.get_token(), L_PAREN)):
            if new_node.add_child(self.is_type(self.get_token(), TYPE_ID)):
                if new_node.add_child(self.type()):
                    if new_node.add_child(self.is_value(self.get_token(),
R_PAREN)):
                        if new_node.add_child(self.varlist()):
                            return new_node
                        # (name type)
                        return new_node
                    else:
                        globals()["current_token_index"] = save
                else:
                    globals()["current_token_index"] = save
                    return new_node
            elif new_node.add_child(self.varlist()):
                return new_node
        else:
            globals()["current_token_index"] = save
            self.parse_error("not varlist")
            return None
        return new_node

    def type(self):
        # type -> bool | int | real | string
        if not self.current_state:
            return None
        new_node = Node("type")
        save = globals()["current_token_index"]
        if new_node.add_child(self.is_value(self.get_token(), "bool")):
            pass
        elif new_node.add_child(self.is_value(self.get_token(), "int")):
            pass
        elif new_node.add_child(self.is_value(self.get_token(), "real")):
            pass
        elif new_node.add_child(self.is_value(self.get_token(), "string")):
            pass
        else:
            globals()["current_token_index"] = save
            return None
        return new_node
```

myparser.py

4.6 lexer.py

```
__author__ = 'drakebridgewater'
import string

from defines import *


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

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

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

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

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

    def get_token(self):
        self.get_next_char()
        while True and self.current_state:
            if not self.current_char:
                return −1
            if self.current_char == ' ' or self.current_char == '\t':
                self.get_next_char()
                pass
            elif self.current_char == '\n':
                self.get_next_char()
                self.line += 1
            elif self.current_char in self.accepted_ops:
                return self.is_op()
            elif self.is_letter():
                return self.identify_word()  # identify the string and add to
    the token list
            elif self.is_digit():
                return self.is_number()  # identify the number and add to the
    token list
            elif self.current_char == '"':
                return self.create_token(self.parse_string())
            else:
                print("Line:ERROR: Could not identify on line: " + str(
                    self.line) + " near char: '" + self.current_char + "'")
                return None

                # TODO have all functions return to a state that has the next
    char

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

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

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

    def is_op(self):
        item = self.current_char
        # If we see an op look to see if we see another. If we see another add
 the previous
        # found op
        if self.current_char is '+':
            self.get_next_char()
            return self.create_token((self.has_token(item), item))
        elif self.current_char is '-':
            self.get_next_char()
            # if self.current_char is '-':
            # item += self.current_char  # Seen -- make new token
            # self.get_next_char()
            return self.create_token((self.has_token(item), item))
        elif self.current_char in ('<', '>', '!'):
            self.get_next_char()
            if self.current_char == '=':
                item += self.current_char
                self.get_next_char()
            return self.create_token((self.has_token(item), item))
        elif self.current_char in ':':
            self.get_next_char()
            if self.current_char is '=':
                item += self.current_char
                return self.create_token((self.has_token(item), item))
            else:
                print("Lexer Error [Line: " + str(
                    self.line) + '] the "' + self.current_char + '" symbol not
 recognized after colon [:]')
        elif self.current_char in '=':
            return self.create_token((self.has_token(item), item))
        elif self.current_char in ('*', '/', '(', ')', '%', '^'):
            self.get_next_char()
            return self.create_token((self.has_token(item), item))
        else:
            print("Lexer Error: [Line: " + str(self.line) + "] could not
intemperate: " +
                  self.current_char)
            return -1

    def parse_string(self):
        accepted_chars = ['"']
        new_string = ''
        self.get_next_char()
        while self.current_char not in accepted_chars:
            new_string += self.current_char
            self.get_next_char()
        return "string", new_string

    def identify_word(self):
        accepted_chars = list(string.ascii_letters) + list(string.digits) +
    list('_')
        acceptable_first_chars = list(string.ascii_letters)

        word = ''
        if self.current_char in acceptable_first_chars:
            word += self.current_char
            self.get_next_char()
            while self.current_char in accepted_chars:
                word += self.current_char
                self.get_next_char()
        token_value = word
        token_type = self.has_token(token_value)
        if token_type == -1:
            token_type = "ID"
        return self.create_token((token_type, token_value))

    # Function Description:
    # This function should be called when a word identifier or keyword is
    started
    # and will return the full word upon seeing invalid characters.
    def parse_word(self, accepted_chars, acceptable_first_chars=[]):
        if self.current_char not in acceptable_first_chars:
            return -1
        else:
            word = ''
            while self.current_char in accepted_chars:
                word += self.current_char
                self.get_next_char()
            return word

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

        return word

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

```
    if value == '':
        word = self.current_char
    else:
        word = value
    self.get_next_char()

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

            elif self.current_char is '-':
                self.get_next_char()
                exp = self.is_int()
                try:
                    self.get_next_char()
                    exp = int(exp)
                    word += 'e-'
                    word += str(exp)
                    try:
                        return self.create_token(("float", float(word)))
                    except ValueError:
                        print("Fatal parse error: [row: " +
                              str(self.line) + "] when parsing char '" +
                              str(self.current_char) + "' for: \n\t\t" +
str(word))
                except ValueError:
                    return [self.create_token(("int", word)),
                            self.create_token(("ID", "e")),
                            self.create_token((self.has_token("-"), "-"))]
            else:
                exp = self.is_int()
                try:
                    exp = int(exp)
                    word += str(exp)
                    return self.create_token(("float", float(word)))
                except ValueError:
                    print_error("Unable to parse '" + str(self.
current_char) +
                                "' in: " + str(exp), self.line, 'lexer')
        elif self.is_digit(other_accepted):
            word += self.current_char
            self.get_next_char()
        else:
            break
        if 'e' not in other_accepted:
            other_accepted.append('e')

    if '.' in word or 'e' in word:
        try:
            return self.create_token(("float", float(word)))
        except ValueError:
            print_error("could not determine numerical token of: " +
                        str(word), self.line, 'lexer')
    else:
        try:
            return self.create_token(("int", int(word)))
        except ValueError:
            print_error("could not determine numerical token of: " +
                        str(word), self.line, 'lexer')

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

# Function Description:
# checks to see if the current token in peek is a letter
# return true if it is
def is_letter(self, others=[]):
    letters = list(string.ascii_letters)
    for x in others:
        if x not in letters:
            letters.append(x)
    if self.current_char in letters:
        return True
    return False
```

lexer.py