Oregon State University

CS 352 - TRANSLATORS

Winter 2015

Milestone 4: Constant Only Calculations

Author:
Drake Bridgewater

Professor:
Dr. Jennifer Parham-Mocello

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

Contents

1 Source Code Descriptions

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

3 Source Code

3.1 main.py

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

from myparser import *


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

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

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


def main():
    sys.setrecursionlimit(100)
    if '-d' in sys.argv:
        globals()['DEBUG'] = 1
    if len(sys.argv) > 1:
        filename = sys.argv[len(sys.argv) - 1]
    else:
        filename = "test1"
    parser = MyParser(filename)
    parser.control()
    tree = parser.tree
    tree.post_order_tree_print()


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

3.2 codegen.py

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


class codeGen():
    def __int__(self, tree):
        self.tree = tree
        self.current_token = None
        self.stack = []
        self.next_tree_item = False

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

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

    def do_something(self, data):
        if self.next_tree_item:
            return
        num_ops = self.oper_count(data)
        if num_ops == 1:
            if self.is_number(data):
                self.write_out()
                if self.next_tree_item:
                    return
            else:
                self.print_error("num_ops = 1 first value error")
                return
        if num_ops == 2:
            if self.write_out(self.is_number(data)):
                if self.write_out(self.is_number(data)):
                    self.write_out(data)
                else:
                    self.print_error("num_ops = 1 second value error")
            else:
                self.print_error("num_ops = 1 first value error")

        if self.is_number(data):
            if data.type is TYPE_INT:
                self.stack.append()
        if data == OPER_ADD:
            if self.is_number(data):
                self.write_out()
                if self.is_number(data):
                    self.stack.pop()

    # Function Description:
    # Only write out if data is actually data
    def write_out(self, data):
        if data.value is KEYWORD_STDOUT:
            print('.s')
        elif data.value is KEYWORD_STDOUT:
            print('.s')

        elif data.value is OPER_EQ:
            pass

        elif data.value is OPER_ASSIGN:
            pass

        elif data.value is OPER_ADD:
            pass

        elif data.value is OPER_SUB:
            pass

        elif data.value is OPER_DIV:
            pass

        elif data.value is OPER_MULT:
            pass

        elif data.value is OPER_LT:
            pass

        elif data.value is OPER_GT:
            pass

        elif data.value is OPER_LE:
            pass

        elif data.value is OPER_GE:
            pass

        elif data.value is OPER_NE:
            pass

        elif data.value is OPER_NOT:
            pass
            print(data)
        else:
            pass

    def compare(self, value1, value2):
        pass
    def is_number(self, value1):
        if hasattr(value1, "type") and value1.type in [TYPE_INT, TYPE_REAL]:
            self.next_tree_item = True
            return value1
        return False
```

codegen.py

3.3 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

3.4 defines.py

```
__author__ = 'Drake'

files = []
options = []

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"
TYPE_BOOL = 'bool'
TYPE_INT = 'int'
TYPE_REAL = 'float'
TYPE_STRING = 'string'
TYPE_ID = 'ID'


class Token:
    type = ''
    value = ''
    line = ''
```

defines.py

3.5 myparser.py

```
__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 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)
        print("-" * 40)

    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")
        self.post_order_tree_print(self)

    def post_order_tree_print(self, node):
        for child in node.children:
            self.post_order_tree_print(child)
            self.print_child(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("\t" * indent + "[line: " + str(child.data.line) +
                      ", ID: " + child.data.type +
                      ", Value: " + str(child.data.value) + "]")
            else:
                print("\t" * indent + str(child.data))
        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 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 = 0
        self.epsilon_flag = 0

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

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

    # Function Description:
    # will return a single token as the lexer may spit out multiple
    def get_token(self):
        # if not self.tokens:
        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("[line: " + str(self.tokens.line) +
                      ", ID: " + self.tokens.type +
                      ", Value: " + str(self.tokens.value) + "]")
        finally:
            self.lexer.close_file()

    def control(self):
        try:
            self.lexer.open_file()
            print("-" * 30)
            while self.current_state:
                self.tree.add_child(self.s())
                # globals()['current_token_index'] += 1
                self.tree.print_tree()
                self.tree.print_postordered_tree()
                if len(self.tokens) > globals()['current_token_index']:
                    # if self.tokens[len(self.tokens) - 1] == -1:
                    self.current_state = True  # 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 s(self):
        if not self.current_state:
            return None
        # s -> expr S' | ( S"
        new_node = Node("S")
        save = globals()["current_token_index"]
        if new_node.add_child(self.expr()):
            if new_node.add_child(self.s_prime()):
                if self.epsilon_flag:
                    self.epsilon_flag = 0
                    return new_node
            else:
                globals()["current_token_index"] = save
        elif new_node.add_child(self.is_value(self.get_token(), L_PAREN)):
            if new_node.add_child(self.s_double_prime()):
                pass
            else:
                globals()["current_token_index"] = save
        else:
            globals()["current_token_index"] = save
            print("ERROR")
            self.current_state = True
            return None
        # if len(new_node.children) > 0:
        # return new_node
        # else:
        # return None
        return new_node

    def s_prime(self):
        if not self.current_state:
            return None
        # s' -> S S' | epsilon
        new_node = Node("S'")
        save = globals()["current_token_index"]
        if new_node.add_child(self.s()):
            if new_node.add_child(self.s_prime()):
                pass
            else:
                globals()["current_token_index"] = save
        else:
            new_node.add_child("epsilon")
            self.epsilon_flag = 1
            # TODO Need a double return here so that it get back to s and
    starts a new s
            self.current_state = False
            return None
        return new_node

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

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

    def oper(self):
        if not self.current_state:
            return None
        # oper ->   ( := name oper )
        # ( binops oper oper )
        # ( unops oper )
        # constants
        # name
        new_node = Node("oper")
        save = globals()["current_token_index"]
        if new_node.add_child(self.is_value(self.get_token(), L_PAREN)):
            l_paren_save = globals()["current_token_index"]
            temp_node = new_node
            if temp_node.add_child(self.is_value(self.get_token(), OPER_ASSIGN
    )):
                if temp_node.add_child(self.is_type(self.get_token(), TYPE_ID)
    ):
                    if temp_node.add_child(self.oper()):
                        if temp_node.add_child(self.is_value(self.get_token(),
    R_PAREN)):
                            new_node = temp_node
                            return new_node
                        else:
                            globals()["current_token_index"] = l_paren_save
                    else:
                        globals()["current_token_index"] = l_paren_save
                else:
                    globals()["current_token_index"] = l_paren_save
            else:
                globals()["current_token_index"] = l_paren_save

            temp_node = new_node
            if temp_node.add_child(self.binops()):
                if temp_node.add_child(self.oper()):
                    if temp_node.add_child(self.oper()):
                        if temp_node.add_child(self.is_value(self.get_token(),
    R_PAREN)):
                            new_node = temp_node
                            return new_node
                        else:
                            globals()["current_token_index"] = l_paren_save
                    else:
                        globals()["current_token_index"] = l_paren_save
                else:
                    globals()["current_token_index"] = l_paren_save
            else:
                globals()["current_token_index"] = l_paren_save

            temp_node = new_node
            if temp_node.add_child(self.unops()):
                if temp_node.add_child(self.oper()):
                    if temp_node.add_child(self.is_value(self.get_token(),
    R_PAREN)):
                        if temp_node.add_child(self.tokens[0]):
                            new_node = temp_node
                            return new_node
                        else:
                            globals()["current_token_index"] = l_paren_save
                    else:
                        globals()["current_token_index"] = l_paren_save
                else:
                    globals()["current_token_index"] = l_paren_save
            else:
                globals()["current_token_index"] = l_paren_save

        elif new_node.add_child(self.constants()):
            pass
        elif new_node.add_child(self.name()):
            pass
        else:
            self.parse_error("missing left paren constant or name")
            globals()["current_token_index"] = save
            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()):
            pass
        elif new_node.add_child(self.whilestmts()):
            pass
        elif new_node.add_child(self.letstmts()):
            pass
        elif new_node.add_child(self.printstmts()):
            pass
        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(), L_PAREN)):
            if new_node.add_child(self.is_value(self.get_token(),
KEYWORD_STDOUT)):
                if new_node.add_child(self.oper()):
                    if new_node.add_child(self.is_value(self.get_token(),
R_PAREN)):
                        pass
                    else:
                        globals()["current_token_index"] = save
                else:
                    globals()["current_token_index"] = save
            else:
                globals()["current_token_index"] = save
        else:
            globals()["current_token_index"] = save
            self.parse_error("missing left paren")
            return None
        return new_node

    def ifstmts(self):
        # ifstmts -> (if expr expr expr) | (if expr expr)
        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(), L_PAREN)):
            if new_node.add_child(self.expr()):
                if new_node.add_child(self.is_value(self.get_token(), R_PAREN)
):
                    # (if expr expr)
                    pass
                elif new_node.add_child(self.expr()):
                    if new_node.add_child(self.is_value(self.get_token(),
R_PAREN)):
                        # (if expr expr expr)
                        pass
                    else:
                        globals()["current_token_index"] = save
                else:
                    globals()["current_token_index"] = save
            else:
                globals()["current_token_index"] = save
        else:
            globals()["current_token_index"] = save
            self.parse_error("not an if statment")
            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(), L_PAREN)):
            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()):
                        if new_node.add_child(self.is_value(self.get_token(),
R_PAREN)):
                            pass
                        else:
                            globals()["current_token_index"] = save
                    else:
                        globals()["current_token_index"] = save
                else:
                    globals()["current_token_index"] = save
            else:
                globals()["current_token_index"] = save
        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()):
```

```python
            if new_node.add_child(self.exprlist()):
                pass
            else:
                globals()["current_token_index"] = save
        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(), L_PAREN)):
            if new_node.add_child(self.is_value(self.get_token(), KEYWORD_LET)
):
                if new_node.add_child(self.varlist()):
                    if new_node.add_child(self.is_value(self.get_token(),
R_PAREN)):
                        pass
                    else:
                        globals()["current_token_index"] = save
                else:
                    globals()["current_token_index"] = save
            else:
                globals()["current_token_index"] = save
        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:
                    pass
                    # check of type failed
                else:
                    globals()["current_token_index"] = save
            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

3.6 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(("ops", '"'))
                # self.parse_string()                  # parse a 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))
```

```python
        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 in accepted_chars:
            new_string += self.current_char
            self.get_next_char()
        self.token_list.append(("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("Lexer Error: [row: " + str(self.line) + "]
Unable to parse '" +
                          str(self.current_char) + "' in: " + str(exp))
        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("Lexer Error (line: " + str(self.line) +
                  "): could not determine numerical token of: " + str(word
))
    else:
        try:
            return self.create_token(("int", int(word)))
        except ValueError:
            print("Lexer Error (line: " + str(self.line) +
                  "): could not determine numerical token of: " + str(word
))

# 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