Import the Python regex module

import re

Given a line of input, returns a list with the token, lexeme pair

def lex(line, num): # Strip comments out before continuing line = re.sub('!.*¡,”, line)

```
length = len(line)
position = 0
pairs = []

# Continue trying to find a matching state while there is still more to process
while (position < length):

    # Separators
    if (line[position] == " "):
        position = position + 1
    elif (line[position] == "{"):
        pairs.append(['separator', line[position], num])
        position = position + 1
    elif (line[position] == "}"):
        pairs.append(['separator', line[position], num])
        position = position + 1
    elif (line[position] == "("):
        pairs.append(['separator', line[position], num])
        position = position + 1
    elif (line[position] == ")"):
        pairs.append(['separator', line[position], num])
        position = position + 1
    elif (line[position] == ":"):
        pairs.append(['separator', line[position], num])
        position = position + 1
    elif (line[position] == ";"):
        pairs.append(['separator', line[position], num])
        position = position + 1
    elif (line[position] == "["):
        pairs.append(['separator', line[position], num])
        position = position + 1
    elif (line[position] == "]"):
        pairs.append(['separator', line[position], num])
        position = position + 1
```

```
elif (line[position] == "%"):
    pairs.append(['separator', line[position], num])
    position = position + 1
elif (line[position] == ","):
    pairs.append(['separator', line[position], num])
    position = position + 1

# Operators
elif (line[position:position+2] == "=="):
    pairs.append(['operator', line[position:position+2], num])
    position = position + 2
elif (line[position:position+2] == "<="):
    pairs.append(['operator', line[position:position+2], num])
    position = position + 2
elif (line[position:position+2] == ">="):
    pairs.append(['operator', line[position:position+2], num])
    position = position + 2
elif (line[position:position+2] == "^="):
    pairs.append(['operator', line[position:position+2], num])
    position = position + 2
elif (line[position] == "="):
    pairs.append(['operator', line[position], num])
    position = position + 1
elif (line[position] == ">"):
    pairs.append(['operator', line[position], num])
    position = position + 1
elif (line[position] == "<"):
    pairs.append(['operator', line[position], num])
    position = position + 1
elif (line[position] == "+"):
    pairs.append(['operator', line[position], num])
    position = position + 1
elif (line[position] == "-"):
    pairs.append(['operator', line[position], num])
    position = position + 1
elif (line[position] == "*"):
    pairs.append(['operator', line[position], num])
    position = position + 1
elif (line[position] == "/"):
    pairs.append(['operator', line[position], num])
    position = position + 1

# Keywords
elif (line[position:position+2] == "if"):
    pairs.append(['keyword', line[position:position+2], num])
    position = position + 3
```

```
elif (line[position:position+3] == "int"):
    pairs.append(['keyword', line[position:position+3], num])
    position = position + 3
elif (line[position:position+3] == "get"):
    pairs.append(['keyword', line[position:position+3], num])
    position = position + 3
elif (line[position:position+3] == "put"):
    pairs.append(['keyword', line[position:position+3], num])
    position = position + 3
elif (line[position:position+4] == "true"):
    pairs.append(['boolean', line[position:position+4], num])
    position = position + 4
elif (line[position:position+4] == "else"):
    pairs.append(['keyword', line[position:position+4], num])
    position = position + 4
elif (line[position:position+5] == "while"):
    pairs.append(['keyword', line[position:position+5], num])
    position = position + 5
elif (line[position:position+5] == "endif"):
    pairs.append(['keyword', line[position:position+5], num])
    position = position + 5
elif (line[position:position+5] == "false"):
    pairs.append(['boolean', line[position:position+5], num])
    position = position + 5
elif (line[position:position+6] == "return"):
    pairs.append(['keyword', line[position:position+6], num])
    position = position + 6
elif (line[position:position+7] == "boolean"):
    pairs.append(['keyword', line[position:position+7], num])
    position = position + 7
elif (line[position:position+8] == "function"):
    pairs.append(['keyword', line[position:position+8], num])
    position = position + 8
else:
    # Only Identifier, Integer and Real remain

    # Integer and Real
    if (re.match('[0-9]', line[position])):

        num_pos = position
        is_real = False

        while (num_pos < length):
            if (re.match('[0-9]', line[num_pos])):
                num_pos = num_pos + 1
            elif (line[num_pos] == "."):
                    is_real = True
                    num_pos = num_pos + 1
                else:
                    break

            if (is_real):
                pairs.append(['real', line[position:num_pos], num])
            else:
                pairs.append(['integer', line[position:num_pos], num])

            position = num_pos

        # Identifier
        elif (re.match('[a-zA-Z]', line[position])):

            id_pos = position

            while (id_pos < length):
                if (line[id_pos] == "$"):
                    id_pos = id_pos + 1
                    break
                elif (re.match('[a-zA-Z0-9]', line[id_pos])):
                    id_pos = id_pos + 1
                else:
                    break

            pairs.append(['identifier', line[position:id_pos], num])
            position = id_pos

        else:
            # Input is probably a whitespace or other non-printing char
            position = position + 1

if not pairs:
    return None
else:
    return pairs
```

from Lexer import *from Tables import*

rules = [“ERROR: Rules are not zero indexed! FIXME!”, “R1. ::= %% ”, “R2. ::= int | boolean”, “R3.

::= { }“,”R4. ::= | E“,”R5. ::= <Declaration List’>“,”R6. <Declaration List’> ::= | E“,”R7. ::= “,”R8. ::= <IDs’>“,”R9. <IDs’> ::= | E“,”R10. ::= <Statement List’>“,”R11. <Statement List’> ::= | E“,”R12. ::= | | | | | | “,”R13. ::= { } “,”R14. ::= = ;“,”R15. ::= if ( ) endif | if ( ) else endif

“,”R16. ::= return ; | return ;“,”R17. ::= put ( );“,”R18. ::= get ( );“,”R19. ::= while ( ) “,”R20. ::= “,”R21. ::= == | ˆ= | > | < | => | =< “,”R22. ::= <Expression’>“,”R23. <Expression’> ::= + <Expression’> | - <Expression’> | E“,”R24. ::= <Term’>“,”R25. <Term’> ::= * <Term’> | / <Term’> | E“,”R26. ::= - | “,”R27. ::= | | ( ) | ( ) | true | false"]

Create a list for the token lexeme pairs

pairs = []

This is our symbol table

symtable = SymbolTable()

index = 0

backtrack_stack = []

rule_stack = []

verbose = False debug = False

message = None

current_type = None current_identifier = None

Called when there is a safe index to backtrack back to

def btsafe(): global backtrack_stack global index backtrack_stack.append(index)

def backtrack(): global backtrack_stack global index old_index = index index = backtrack_stack.pop() print(“DEBUG: Backtracking from”, old_index, “(”, pairs[old_index][1], “)”, “to”, index, “(”, pairs[index][1], “)”)

def error(msg): global message global pairs global index

```
if index < len(pairs):
    message = "ERROR: " + str(msg) + "\t@ Token: " + str(pairs[index][0]) + "\tLexeme: " + s
```

def rule(num): global rules global rule_stack

```
rule_stack.append(rules[num])
```

def advance(): global pairs global index global debug

```
if (index - 1) >= (len(pairs)):
    return False
else:
    if debug:
        print("Accepted Token:", pairs[index][0], "\tLexeme:", pairs[index][1])
    index += 1
    return True
```

def accept(test, token = False): global index global backtrack_stack global debug

```
if token:
    offset = 0
else:
    offset = 1

if index == len(pairs):
    return False

if pairs[index][offset] == test:
    if advance():
        return True
    else:
        # FIXME: Figure out what to do here
        error("End of input")
        return False
else:
    if debug:
        print()
        print("DEBUG: failed \"" + str(pairs[index][offset]) + "\" != \"" + str(test) + "\""

    # If we can't accept the input and we're not at a safe index...
    if index not in backtrack_stack:
        # Backtrack back to the last safe index
        backtrack()

    return False
```

def p_ids_prime(): btsafe() if accept(“,”) and p_ids(): rule(9) return True

```
return True
```

def p_identifier(declare = False): global pairs global index global current_type global current_identifier

```
btsafe()
if accept("identifier", token = True):
    current_identifier = pairs[index - 1][1]
    add_symbol(pairs[index - 1][1], current_type)
    if declare:
        gen_instr("PUSHI", 0)

    gen_instr("PUSHM", get_address(current_identifier))
    return True
else:
    return False
```

def p_ids(declare=False): btsafe() if p_identifier(declare) and p_ids_prime(): rule(8) return True else: return False

def p_qualifier(): global pairs global index global current_type

```
btsafe()
if accept("int"):
    current_type = "int"
    rule(2)
    return True

btsafe()
if accept("boolean"):
    current_type = "int"
    rule(2)
    return True

error("Expected int or boolean qualifier")
return False
```

def p_declaration(): btsafe() if p_qualifier() and p_ids(declare=True): rule(7) return True else: return False

def p_decl_list_prime(): btsafe() if p_decl_list() and accept(“;”): rule(6) return True

```
rule(6)
return True
```

def p_decl_list(): btsafe() if p_declaration() and accept(“;”) and p_decl_list_prime(): rule(5) return True else: return False

def p_opt_decl_list(): btsafe() p_decl_list() rule(4) return True

def p_term_prime(): btsafe() if accept(“*“) and p_factor() and p_term_prime(): gen_instr(”MUL“,”nil“) rule(25) return True

```
btsafe()
if accept("/") and p_factor() and p_term_prime():
    gen_instr("DIV", "nil")
    rule(25)
    return True
return True
```

def p_primary(): global pairs global index

```
btsafe()
if p_identifier():
    rule(27)
    return True

btsafe()
if accept("integer", token=True):
    gen_instr("PUSHI", pairs[index - 1][1])
    rule(27)
    return True

btsafe()
if p_identifier() and accept("(") and p_ids() and accept(")"):
    rule(27)
    return True

btsafe()
if accept("(") and p_expression() and accept(")"):
    rule(27)
    return True

btsafe()
if accept("true") or accept("false"):
    rule(27)
    return True

return False
```

def p_factor(): btsafe() if accept(“-”) and p_primary(): rule(26) return True

```
btsafe()
if p_primary():
    rule(26)
    return True

return False
```

def p_term(): btsafe() if p_factor() and p_term_prime(): rule(24) return True else: return False

def p_expression_prime(): btsafe() if accept(“+”) and p_term() and p_expression_prime(): gen_instr(“ADD”, “nil”) rule(23) return True

```
btsafe()
if accept("-") and p_term() and p_expression_prime():
    gen_instr("SUB", "nil")
    rule(23)
    return True

return True
```

def p_expression(): btsafe() if p_term() and p_expression_prime(): rule(22) return True else: return False

def p_compound(): btsafe() if accept(“{”) and p_statement_list() and accept(“}”): rule(13) return True else: return False

def p_assign(): global current_identifier

```
btsafe()
if p_identifier() and accept("=") and p_expression() and accept(";"):
    if current_identifier is not None:
        gen_instr("POPM", get_address(current_identifier))
    rule(14)
    return True
else:
    return False
```

def p_relop(): global pairs global index

```
btsafe()
if accept("operator", token = True):
    if pairs[index - 1][1] == ">":
        gen_instr("GRT", "nil")
    else:
        gen_instr("LES", "nil")

    push_jumpstack(gen_instr("JUMPZ", "nil"))
    rule(21)
    return True

btsafe()
if accept("operator", token = True) and accept("operator", token = True):
    if (str(pairs[index - 2][1]) + str(pairs[index - 1][1])) == "==":
        gen_instr("EQU", "nil")
    elif (str(pairs[index - 2][1]) + str(pairs[index - 1][1])) == "^=":
        gen_instr("NEQ", "nil")
    elif (str(pairs[index - 2][1]) + str(pairs[index - 1][1])) == "=>":
        gen_instr("GEQ", "nil")
    else:
        gen_instr("LEQ", "nil")

    push_jumpstack(gen_instr("JUMPZ", "nil"))
    rule(21)
    return True

return False
```

def p_condition(): btsafe() if p_expression() and p_relop() and p_expression(): rule(20) return True return False

def p_if(): btsafe() if accept(“if”) and accept(“(”) and p_condition() and accept(“)”) and p_statement() and accept(“endif”): rule(15) backpatch() return True

```
btsafe()
if accept("if") and accept("(") and p_condition() and accept(")") and p_statement() and acce
    backpatch()
    if p_statement() and accept("endif"):
        rule(15)
        return True

return False
```

def p_return(): btsafe() if accept(“return”) and accept(“;”): rule(16) return True

```
btsafe()
if accept("return") and p_expression() and accept(";"):
    rule(16)
    return True

return False
```

def p_scan(): btsafe() if accept(“get”) and accept(“(”) and p_ids() and accept(“)”) and accept(“;”): gen_instr(“STDIN”, “nil”) rule(18) return True else: return False

def p_print(): btsafe() if accept(“put”) and accept(“(”) and p_expression() and accept(“)”) and accept(“;”): gen_instr(“STDOUT”, “nil”) rule(17) return True else: return False

def p_while(): btsafe() if accept(“while”): push_jumpstack(gen_instr(“LABEL”, “nil”)) if accept(“(”) and p_condition() and accept(“)”) and p_statement(): gen_instr(“JUMP”, “nil”) backpatch() rule(19) return True

```
error("Could not parse while loop")
return False
```

def p_statement(): btsafe() if p_compound(): rule(12) return True

```
btsafe()
if p_assign():
    rule(12)
    return True
```

```
btsafe()
if p_if():
    rule(12)
    return True

btsafe()
if p_return():
    rule(12)
    return True

btsafe()
if p_print():
    rule(12)
    return True

btsafe()
if p_scan():
    rule(12)
    return True

btsafe()
if p_while():
    rule(12)
    return True

error("Could not parse statement")
return False
```

def p_statement_list_prime(): btsafe() p_statement_list() rule(11) return True

def p_statement_list(): btsafe() if p_statement() and p_statement_list_prime(): rule(10) return True else: error("Could not parse statement list") return False

def p_rat18s(): if accept("%") and accept("%") and p_opt_decl_list() and p_statement_list(): rule(1) return True else: error("Could not parse Rat18s") return False

Given a list of token-lexeme pairs attempt to parse the syntax

def parse(): global pairs global message

```
if not p_rat18s():
    # If we can't parse rat18s, then print the last error message
    print(message)
```

Given a file to lex, print out the tokens and lexemes for each line

def parser(source_file, verbosity, debugging): with open(source_file, ‘r’) as file_in: global verbose global debug global rule_stack verbose = verbosity debug = debugging

```
    global pairs

    index = 0

    # Call lex() for each line and build our list of token, lexeme pairs
    for line in file_in:
        result = lex(line, index)
        index += 1

        if result:
            pairs = pairs + result

    parse()

    # Print the used rules if we are in verbose mode
    if verbose:
        rule_stack.reverse()
        for rule in rule_stack:
            print(rule)

    print("Symbol Table:")
    print_symbols()
    print()
    print("Instruction Table:")
    print_instructions()
```

#!/usr/bin/env python3

import argparse import os.path import sys

from Parser import *

Create our argument parser object

argparser = argparse.ArgumentParser(description=‘A Python3 compiler for the Rat18s language’)

Add our arguments to the parser

argparser.add_argument(“file”, help=“Path to the file to compile”) argparser.add_argument(“-v”, “–verbose”, help=“Prints extra information”, action=‘store_true’) argparser.add_argument(“-d”, “–debug”, help=“Turns on debug output”, action=‘store_true’)

Make sure we have enough arguments, if not print the help message

if len(sys.argv) < 1: argparser.print_help() sys.exit(1)

Actually parse the arguments

arguments = argparser.parse_args()

Check that the file actually exists before continuing

if not os.path.exists(arguments.file): print(arguments.file, “: No such file exists!”) sys.exit(1)

parser(arguments.file, arguments.verbose, arguments.debug)

class Symbol: def **init**(self, name, stype): self.name = name self.type = stype self.location = 0

```
def __str__(self):
    return str(str(self.name) + "\t" + str(self.type) + "\t" + str(self.location))
```

class SymbolTable: def **init**(self): self.last = 1999 self.symbols = []

```
def insert(self, sym):
    if self.lookup(sym.name) is None:
        sym.location = self.last + 1
        self.last += 1
        self.symbols.append(sym)

def lookup(self, name):
    for sym in self.symbols:
        if sym.name == name:
            return sym
    return None

def list(self):
    print("Name\tType\tMemory Location")
    for sym in self.symbols:
        print(sym)
```

class Instruction: def **init**(self, op, operand): self.address = 0 self.op = op self.operand = operand

```
def __str__(self):
    return str(str(self.address) + "\t" + str(self.op) + "\t" + str(self.operand))
```

class InstructionTable: def **init**(self): self.last = 0 self.instructions = []

```
def insert(self, inst):
    inst.address = self.last + 1
    self.last += 1
    self.instructions.append(inst)
    return inst

def set(self, addr, jump):
    for inst in self.instructions:
        if inst.address == addr:
            inst.operand = jump

def peek_end(self):
    return self.instructions[len(self.instructions) - 1]

def list(self):
    print("Address\tOp\tOperand")
    for inst in self.instructions:
        print(inst)
```

symbtable = SymbolTable()

insttable = InstructionTable()

jumpstack = []

def push_jumpstack(addr): global jumpstack jumpstack.append(addr)

def backpatch(): global jumpstack global insttable jump_addr = insttable.peek_end().address addr = jumpstack.pop() insttable.set(addr, jump_addr)

def gen_instr(op, operand): global insttable return insttable.insert(Instruction(op, operand)).address

def get_address(name): global symbtable return symbtable.lookup(name).location

def add_symbol(name, stype): global symbtable symbtable.insert(Symbol(name, stype))

def print_symbols(): global symbtable symbtable.list()

def print_instructions(): global insttable insttable.list()